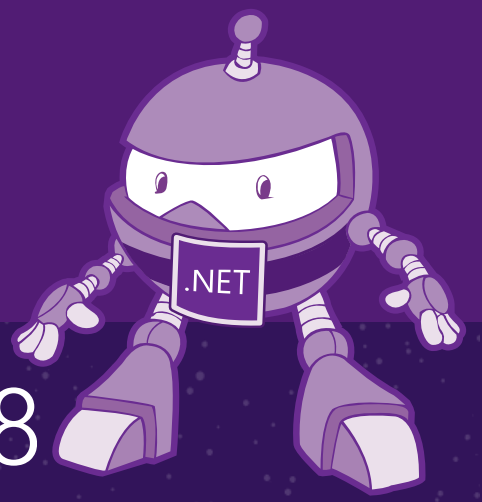

.NET Conf 2018
Discover the world of .NET
.NET
dotnet new
www.dotnetconf.net
STUDY4.TW
為 學 習 而 生

使用 Dependency Injection 撰寫簡潔 C# 程式碼原來這麼簡單

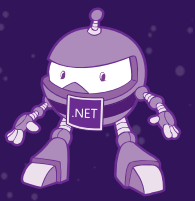
Poy Chang



.NET Conf 2018

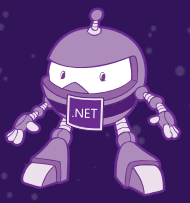
STUDY4.TW
為學習而生

為什麼要講這個主題？



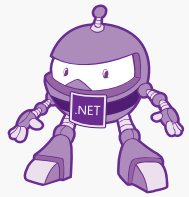
對我來說這一切起源於

關注點分離

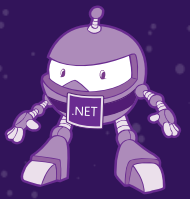


我希望我能做到...

- 降低程式碼複雜程度
- 提升程式碼可重複利用性
- 程式碼具有較佳的可讀性
- 讓模組具有高內聚力、低耦合力
- 需求變更時減少破壞既有功能
- ...



使用 DI 之後
程式碼好像從此變簡單了

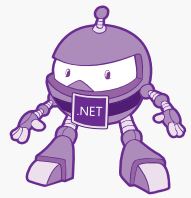


Key Takeaway

Dependency Injection 的背景知識

DI 與生命週期

如何使用 .NET 內建的 DI 框架



Poy Chang

目前任職於全美 100 大私人企業，負責企業內部 IT 解決方案設計與開發，專注於 Angular、ASP.NET Core、Azure 等技術研究

- ✓ Angular Taiwan 社群核心成員
- ✓ Microsoft MVP Developer Technologies
- ✓ Global Azure Bootcamp@北京 講師

STUDY4.TW 為學習而生

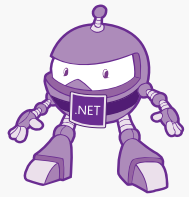
.NET Conf 2018

S O L I D

物件導向設計原則

STUDY4.TW
為 學 習 而 生

.NET Conf 2018



單一責任原則
SRP

開放封閉原則
OCP

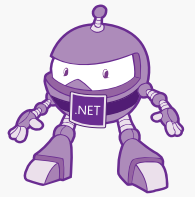
里氏替換原則
LSP

介面隔離原則
ISP

依賴反轉原則
DIP

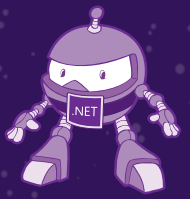
SOLID

物件導向設計原則

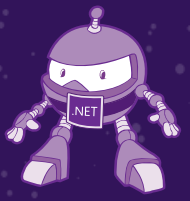


教條般的 SOLID 很重要

只是該怎麼運用在實作呢？



Dependency Injection
一次實現 5 種原則

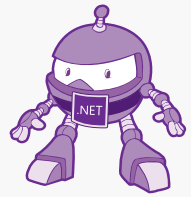


首先，先介紹一種設計模式

IOC 控制反轉

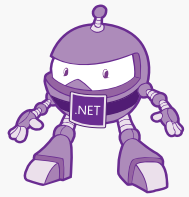
IOC 控制反轉

- 物件導向程式設計中的一種設計原則
- 用來減低程式碼之間的耦合度
- 最常見的方式叫做依賴注入（Dependency Injection，簡稱 DI）
- 另一種叫做依賴尋找（Dependency Lookup）



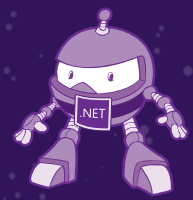
IOC 控制反轉

- 通過控制反轉，物件在被建立時，由一個調控系統內所有物件的外界實體，將其所依賴的物件的參照傳遞給它
- 聽說這又叫做好萊塢原則



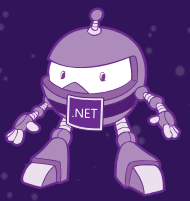
Hollywood Princle

.NET



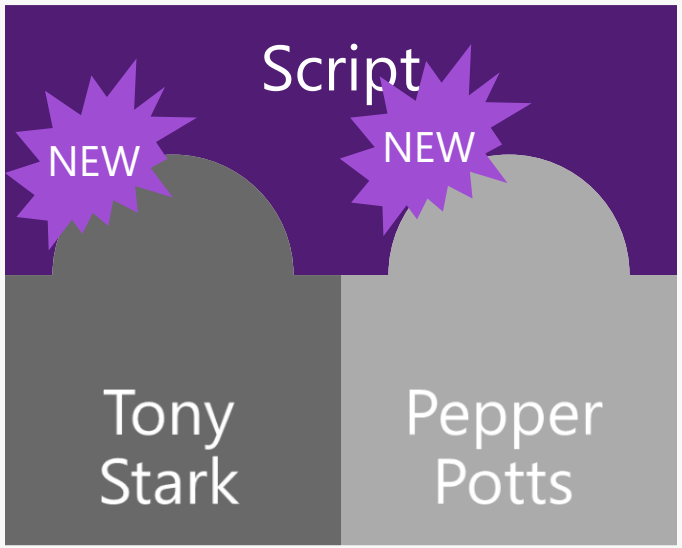
How IOC works ?

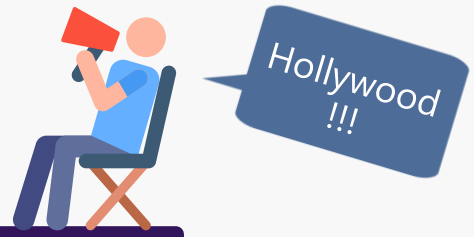
Don’t call me, I call you.



Main

IronManClass





Main

IronManClass

Script

DI Container

NEW

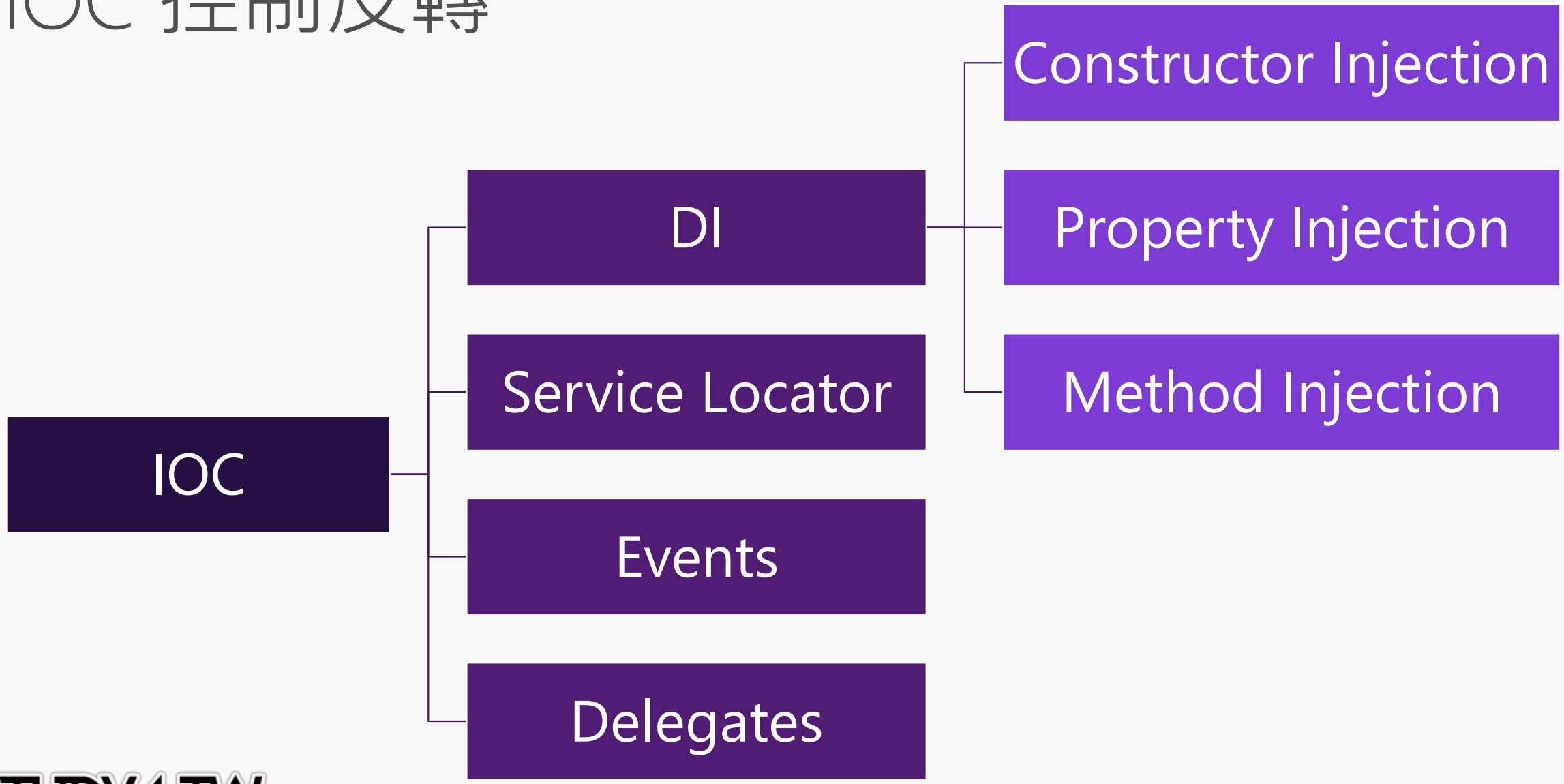
NEW

Tony Stark

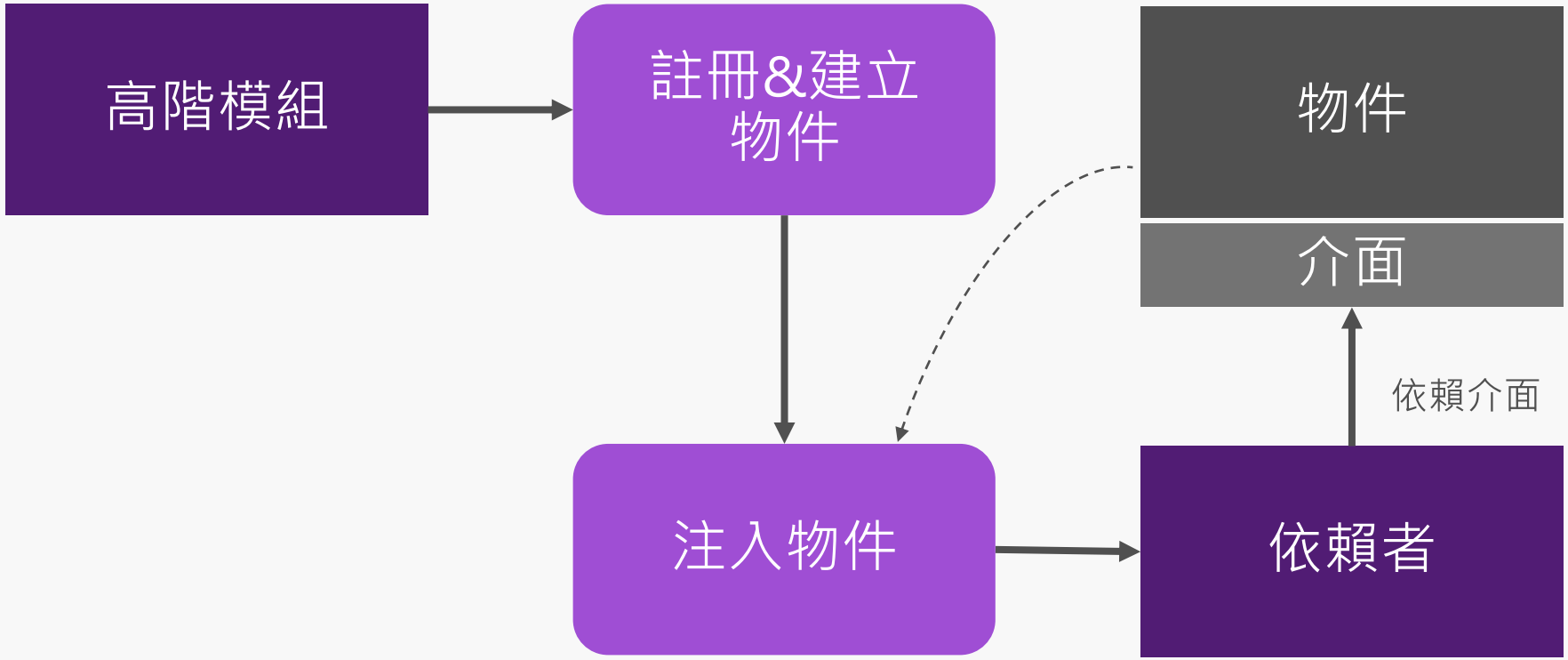
Pepper Potts

Call by Reference

IOC 控制反轉



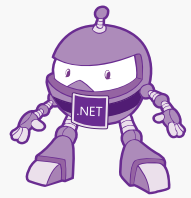
DI 框架運作方式



先停一下

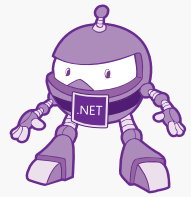
很多名詞，他們彼此之間到底是甚麼關係

- ✓ DIP 是架構設計的原則
- ✓ IOC 是反轉依賴、控制的設計模式
- ✓ DI 是 IOC 的實現方式
- ✓ IOC/DI 容器或 DI 框架指的是 DI 的實作



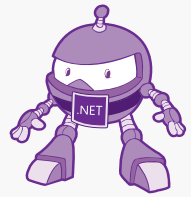
第三方 DI 框架

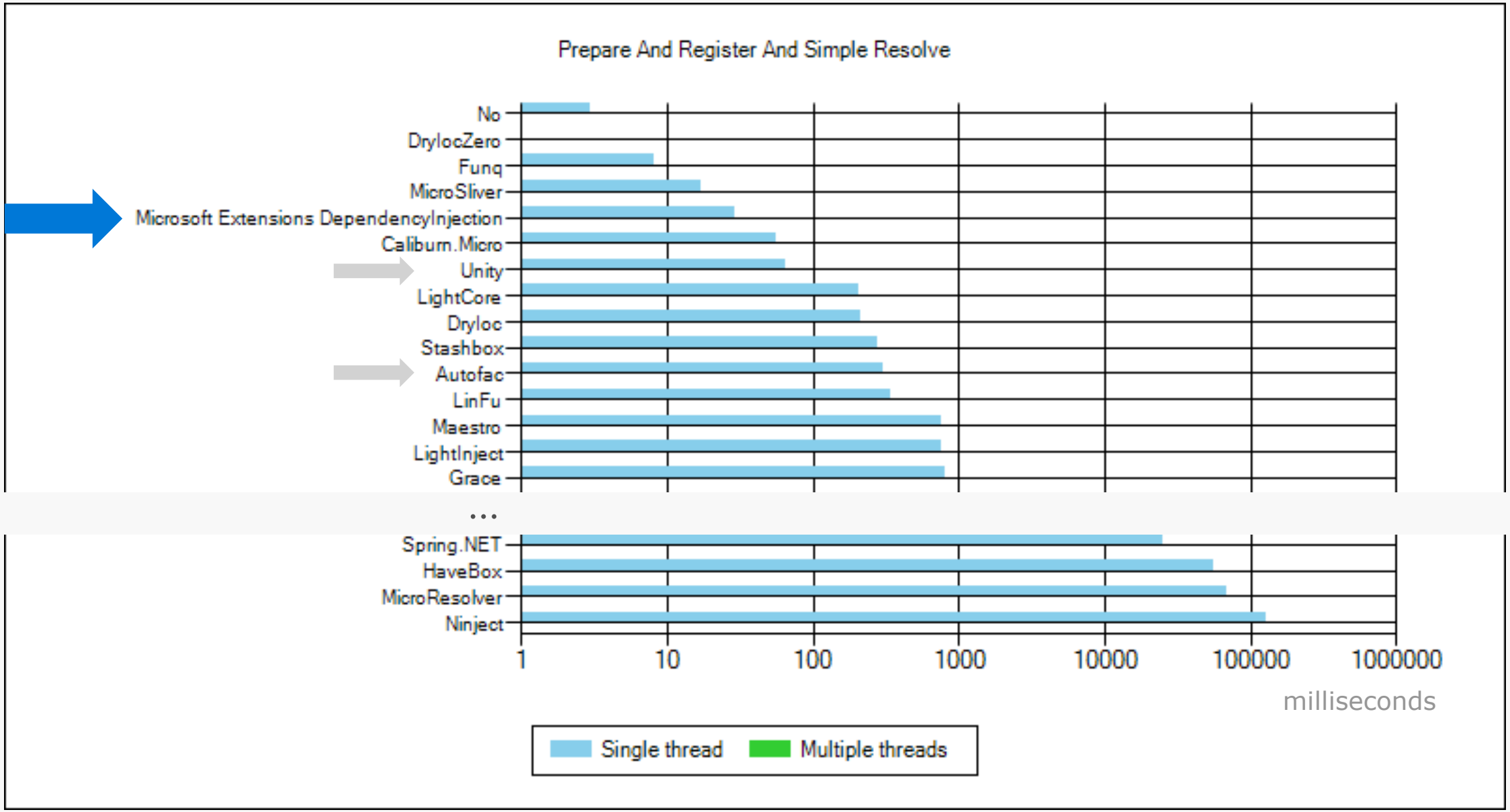
AutoFac
Caliburn.Micro
Catel
DryIoc
Dynamo
fFastInjector
Funq
Grace
Griffin
HaveBox
IfInjector
LightCore
LightInject
LinFu
Maestro
MEF
MEF2
MicroSliver
Mugen
Munq
Ninject
Rezolver
SimpleInjector
Spring.NET
Stashbox
StructureMap
StyleMVVM
Unity
Windsor



Microsoft.Extensions.DependencyInjection

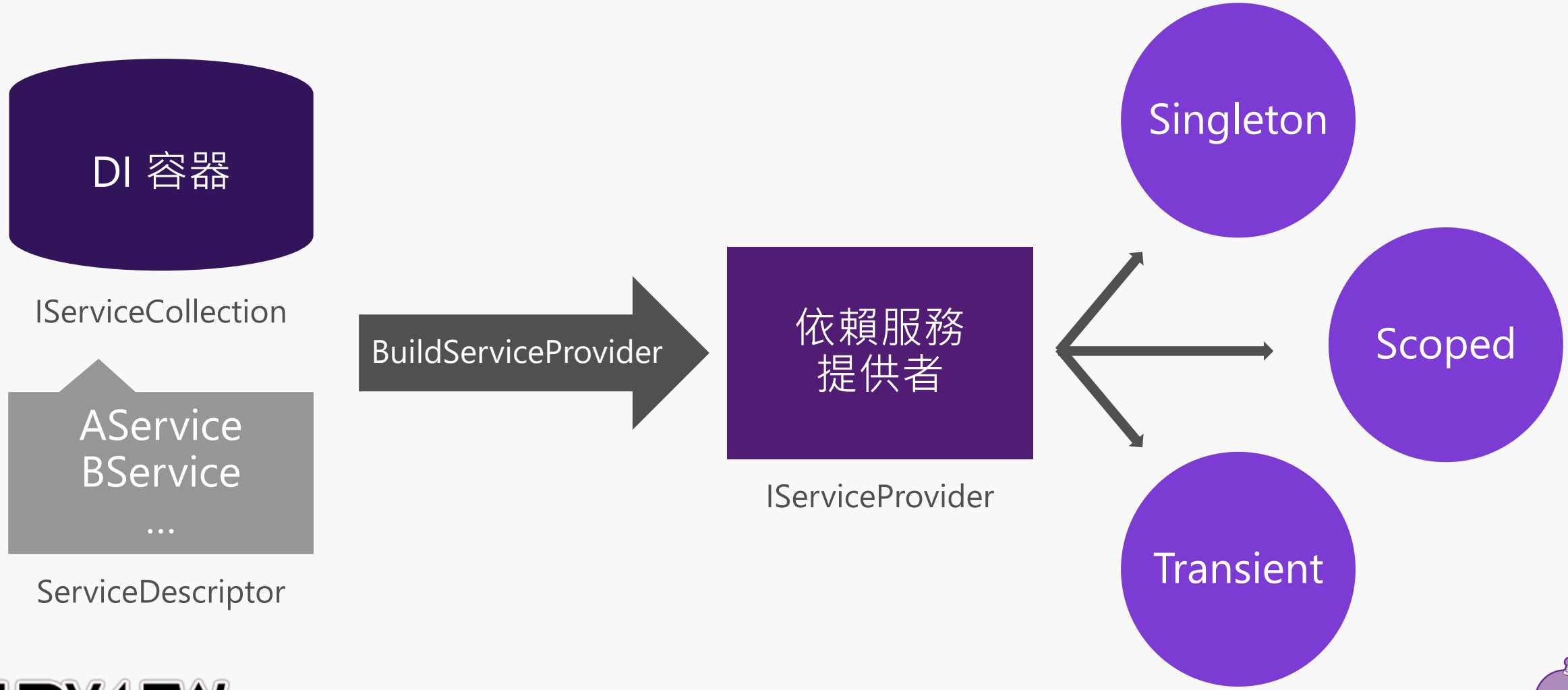
- ✓ .NET Core 原生內建
- ✓ ASP .NET Core 大量使用
- ✓ 足以處理大多數的情境
- ✓ 效能快速
- ✓ 簡單易用





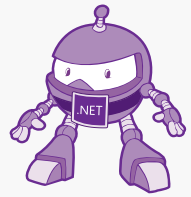
Daniel Palme, [IOC Container Benchmark - Performance comparison]

Microsoft.Extensions.DependencyInjection



ASP .NET Core

- 整個框架建置在 DI 之上
- 透過 DI 提供以下應用服務
  - ASP.NET Core MVC
  - Entity Framework DbContext
  - Authorization/Authentication
  - Cookie/Session
  - more

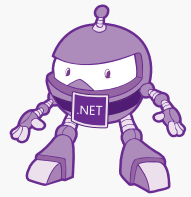


DEMO

ASP .NET Core 中的 DI 機制

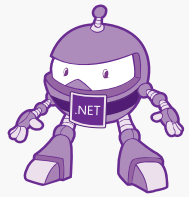
生命週期

- Singleton
  - ✓被實例化後就不會消失，程式運行期間只會有一個實例
- Transient
  - ✓每次注入時，都重新建立一個新的實例
- Scope
  - ✓每個 Scope 都會重新建立一個新的實例
  - ✓對 ASP .NET Core 來說，一個 Request 就是一個 Scope，同一個 Request 不管經過多少個 Pipeline 都是用同一個實例



生命週期 - 自動解析

- Singleton
  - ✓ services.AddSingleton<IService, Service>();
- Transient
  - ✓ services.AddTransient<IService, Service>();
- Scope
  - ✓ services.AddScoped<IService, Service>();
- 使用 ServiceDescriptor
  - ✓ services.Add(new ServiceDescriptor(typeof(IService),
    typeof(Service),
    ServiceLifetime.Transient));



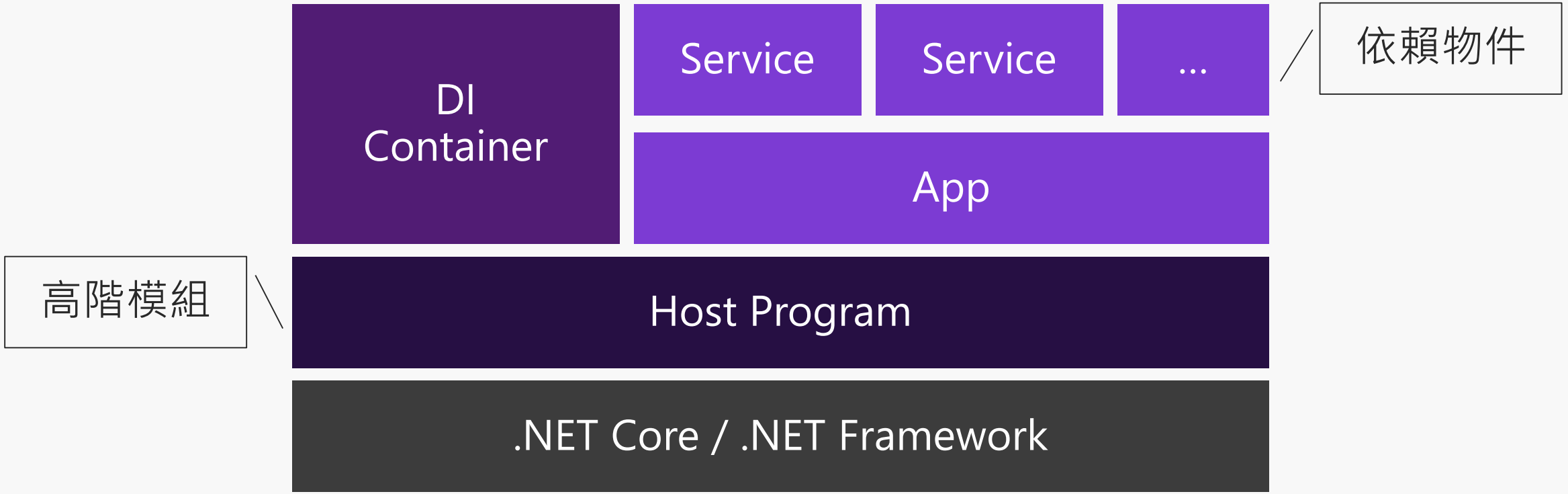
生命週期 - 明確實作

- Singleton
  - ✓ services.AddSingleton<IService>(new Service());
  - ✓ services.AddSingleton<IService>(
    Func<IServiceProvider, Service> ImplementationFactory);
- Transient
  - ✓ services.AddTransient<IService>(
    Func<IServiceProvider, Service> ImplementationFactory);
- Scope
  - ✓ services.AddScope<IService>(
    Func<IServiceProvider, Service> ImplementationFactory);

DEMO

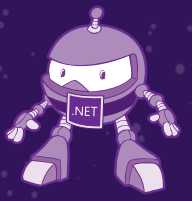
在 Service Collection 中
服務的 DI 生命週期

搭建具 DI 功能的架構



DEMO

使用原生 DI 容器 Service Collection
搭建主控台程式架構

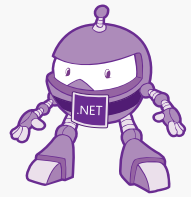


DI 不是萬靈丹

- IOC 最大的問題在於大量使用黑魔法
- 黑魔法 = Relection 反射機制
- 反射機制可能造成效能問題

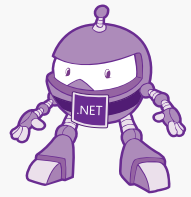
DI 不是萬靈丹 - Interface 滿天飛

```
1 reference | 0 changes | 0 authors, 0 changes
public class FooService
{
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public FooService(
        IBarService barService,
        ICarRepository carRepository,
        ICustomerRepository customerRepository,
        IUserRepository userRepository,
        IUserPermissionRepository userPermissionRepository,
        IEndpointRepository endpointRepository,
        ICallRepository callRepository,
        ITemplateService templateService,
        IUserProfileService userProfileService,
        IACMEUserRepository acmeUserRepository,
        IAccessControlService accessControlService,
        ITemplateRepository templateRepository)
    {
        
    }
}
```

DI 不是萬靈丹 - 註冊到要要的

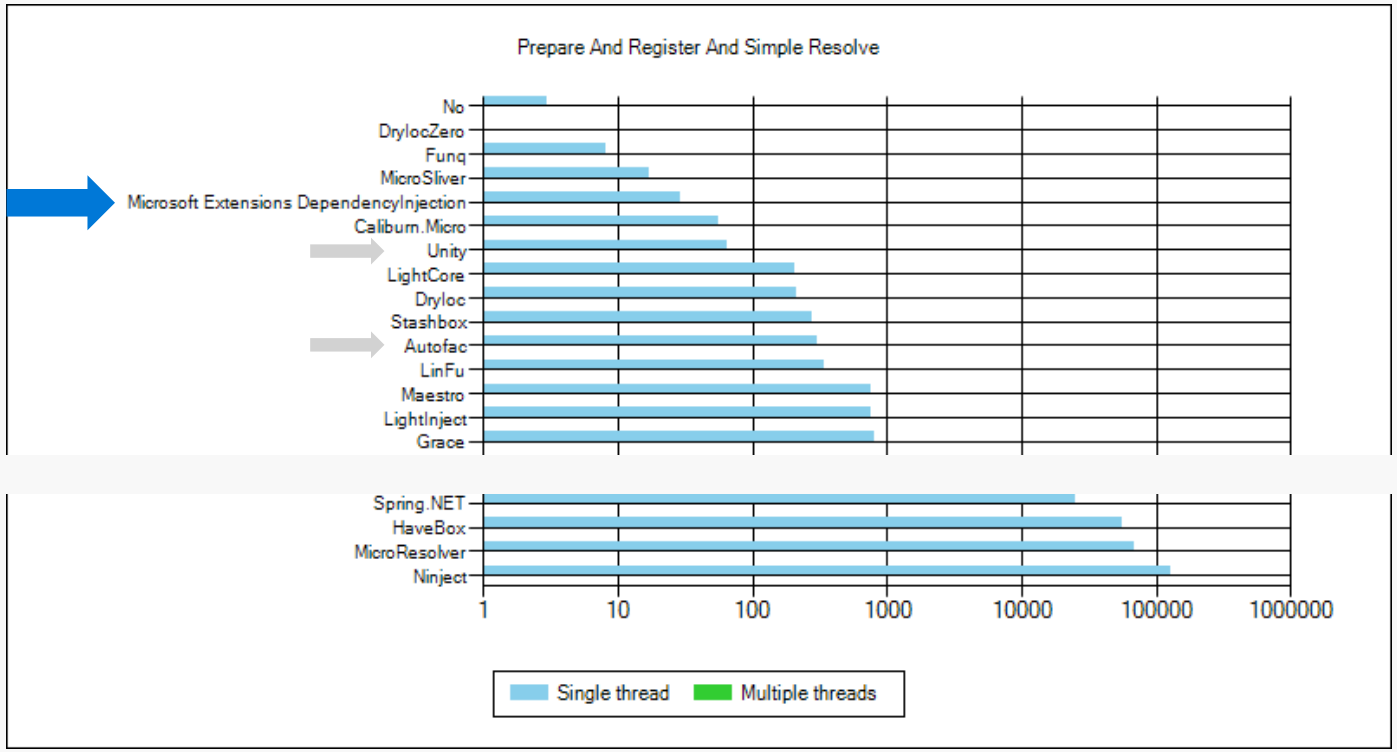
```
0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IBarService>();
    services.AddSingleton<ICarRepository>();
    services.AddSingleton<ICustomerRepository>();
    services.AddSingleton<IUserRepository>();
    services.AddSingleton<IUserPermissionRepository>();
    services.AddSingleton<IEndpointRepository>();
    services.AddSingleton<ICallRepository>();
    services.AddSingleton<ITemplateService>();
    services.AddSingleton<IUserProfileService>();
    services.AddSingleton<IACMEUserRepository>();
    services.AddSingleton<IAccessControlService>();
    services.AddSingleton<ITemplateRepository>();
    services.AddMvc();
}
```

DI 不是萬靈丹 - 解法

✓ 效能問題我解不了

=>那就挑一個又快又好用的 DI 框架



DI 不是萬靈丹 - 解法

- ✓ Interface 滿天飛

  =>God Service 讓上帝幫你實現所有願望

  =>很方便，但就失去了關注點分離的意義

  =>適度的規劃你的服務

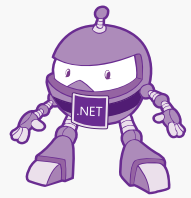
```
1 reference | 0 changes | 0 authors, 0 changes
public class FooService
{
    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public FooService(
        IBarService barService,
        ICarRepository carRepository,
        ICustomerRepository customerRepository,
        IUserRepository userRepository,
        IUserPermissionRepository userPermissionRepository,
        IEndpointRepository endpointRepository,
        ICallRepository callRepository,
        ITemplateService templateService,
        IUserProfileService userProfileService,
        IACMEUserRepository acmeUserRepository,
        IAccessControlService accessControlService,
        ITemplateRepository templateRepository)
    {
        ...
    }
}
```

DI 不是萬靈丹 - 解法

✓ 註冊到要要的

=>用習慣取代註冊

```
0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IBarService>();
    services.AddSingleton<ICarRepository>();
    services.AddSingleton<ICustomerRepository>();
    services.AddSingleton<IUserRepository>();
    services.AddSingleton<IUserPermissionRepository>();
    services.AddSingleton<IEndpointRepository>();
    services.AddSingleton<ICallRepository>();
    services.AddSingleton<ITemplateService>();
    services.AddSingleton<IUserProfileService>();
    services.AddSingleton<IACMEUserRepository>();
    services.AddSingleton<IAccessControlService>();
    services.AddSingleton<ITemplateRepository>();
    services.AddMvc();
}
```

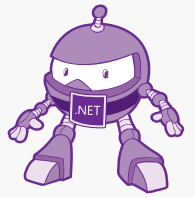



DEMO

用習慣來註冊 DI 服務

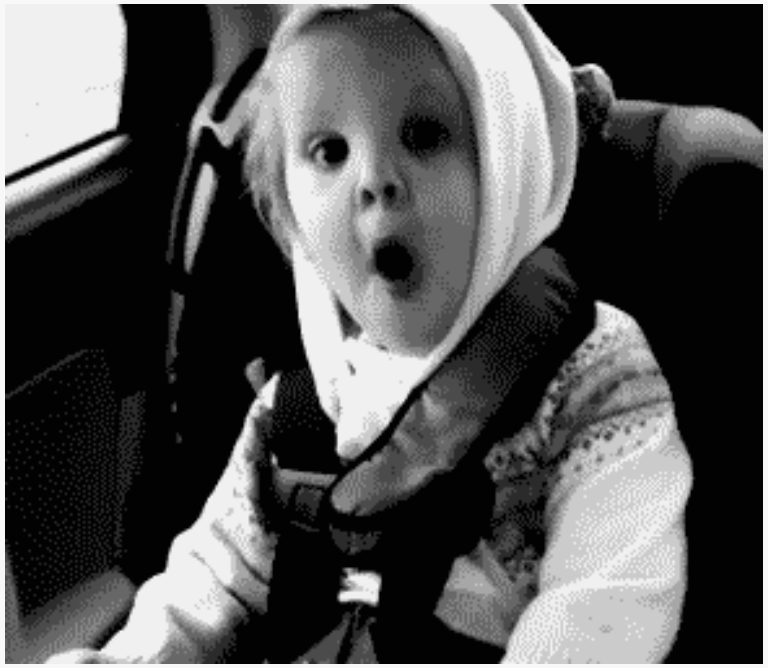
RECAP

- 什麼是依賴注入
  - 當一個類別需要另一個類別來處理工作時，便產生了依賴
  - 透過 IOC，讓依賴的類別不是由使用的類別建立，而是由調用者傳遞，這就是注入
- 為什麼要學依賴注入
  - 降低因為商業邏輯變更而變動程式碼的幅度
  - 關注點分離、友善的測試架構
  - ASP.NET Core 的架構基於 DI 之上
- 打造有依賴注入的程式碼架構
  - 統一管理系統中所有的依賴的類別
  - DEMO Code：https://github.com/poychang/NetConf2018TW.SimpleCodeWithDI



SOLID 原則或者 DI 機制能吸引你嗎？

對我來說，能讓程式碼變得簡單易讀、職責分離，光這樣就夠吸引我了



再也不用 new 了！DI 自動組合依賴的物件

規劃好服務的生命週期，只要註冊好，之後注入就可以用，超方便

DI 實現我關注點分離的願望 ☺

服務是一個個的商業邏輯，撰寫程式時能更專注，之後切換也簡單

特別感謝

STUDY4.TW
為 學 習 而 生

.NET Conf 2018