**[控制反转(Ioc)和依赖注入(DI)](http://www.cnblogs.com/Dlonghow/p/5415421.html)**

控制反转IOC, 全称 “Inversion of Control”。依赖注入DI, 全称 “Dependency Injection”。

面向的问题：软件开发中，为了降低模块间、类间的耦合度，提倡基于接口的开发，那么在实现中必须面临最终是有“谁”提供实体类的问题。（将各层的对象以松耦合的方式组织起来，各层对象的调用面向接口。）

当一个类的实例需要另一个类的实例协助时，在传统的程序设计过程中，通常有调用者来创建被调用者的实例。

然后，采用依赖注入原则，创建被调用者的实例的工作不再由调用者完成，而是由IOC容器来完成，这就是“控制反转”的意思，然后，将其注入调用者，因此也称为 “依赖注入”。

Martin Fowler，在其著名的文章《Inversion of Control Containers and the Dependency Injection pattern》中将具体依赖注入划分为三种形式，即构造器注入、属性（设置）注入和接口注入。

习惯将其划分为一种（类型）匹配和三种注入：

* 类型匹配（Type Matching）：虽然我们通过接口（或者抽象类）来进行服务调用，但是服务本身还是实现在某个具体的服务类型中，这就需要某个类型注册机制来解决服务接口和服务类型之间的匹配关系；
* 构造器注入（Constructor Injection）：IoC容器会智能地选择选择和调用适合的构造函数以创建依赖的对象。如果被选择的构造函数具有相应的参数，IoC容器在调用构造函数之前解析注册的依赖关系并自行获得相应参数对象；
* 属性注入（Property Injection）：如果需要使用到被依赖对象的某个属性，在被依赖对象被创建之后，IoC容器会自动初始化该属性；
* 方法注入（Method Injection）：如果被依赖对象需要调用某个方法进行相应的初始化，在该对象创建之后，IoC容器会自动调用该方法

创建一个控制台程序，定义如下几个接口（IA、IB、IC和ID）和它们的实现类（A、B、C、D）。在类型A中定义了三个属性B、C和D，其参数类型分别为IB、IC和ID。

其中，

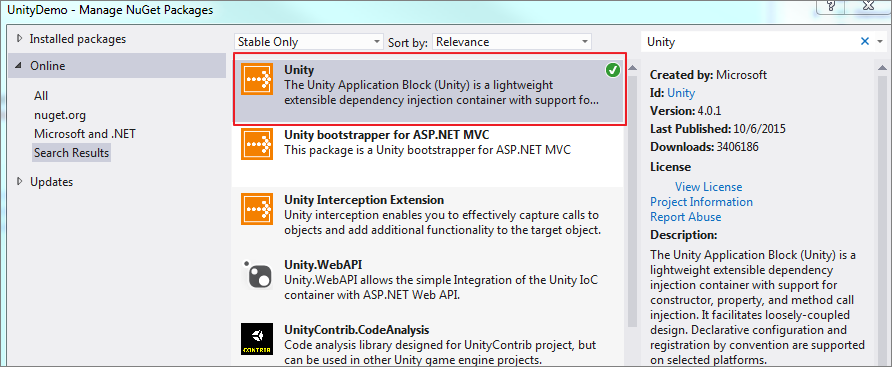
属性B作为构函数的参数，认为它会以**构造器注入**的方式被初始化 （？？）；

属性C应用了DependencyAttribute特性，意味着这是一个需要以**属性注入**方式被初始化的依赖属性；

属性D则通过方法Initialize初始化，该方法上应用了特性InjectionMethodAttribute, 意味着这是一个**方法注入，**在A对象被Ioc容器创建的时候，D会被自动调用。

Microsoft有一个轻量级的IoC框架Unity, 支持构造器注入，属性注入，方法注入。对于C#语言，由于语法元素上本身较其他语言丰富许多，如何实施注入还有些技巧和特色之处。

下面介绍如下：



测试类：

[复制代码](javascript:void(0);)

namespace UnityDemo

{

public interface IA { }

public interface IB { }

public interface IC { }

public interface ID { }

public class A : IA

{

public IB B { get; set; }

[Dependency]

public IC C { get; set; }

public ID D { get; set; }

public A(IB b)

{

this.B = b;

}

[InjectionMethod]

public void Initialize(ID d)

{

this.D = d;

}

}

public class B : IB { }

public class C : IC { }

public class D : ID { }

}

[复制代码](javascript:void(0);)

配置注册：

[复制代码](javascript:void(0);)

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

<configSections>

<section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection, Microsoft.Practices.Unity.Configuration"/>

</configSections>

<unity>

<containers>

<container name="defaultContainer">

<register type="UnityDemo.IA, UnityDemo" mapTo="UnityDemo.A, UnityDemo"/>

<register type="UnityDemo.IB, UnityDemo" mapTo="UnityDemo.B, UnityDemo"/>

<register type="UnityDemo.IC, UnityDemo" mapTo="UnityDemo.C, UnityDemo"/>

<register type="UnityDemo.ID, UnityDemo" mapTo="UnityDemo.D, UnityDemo"/>

</container>

</containers>

</unity>

<startup>

<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />

</startup>

</configuration>

[复制代码](javascript:void(0);)

Main方法中，创建一个Ioc容器的UnityContainer对象，并加载配置信息对其初始化，然后调用它的泛型的Resolve方法创建一个实现了泛型接口IA的对象。

最后将返回对象转换成类型A, 并逐一检验B，C和D属性是否为空，即初始化情况。

[复制代码](javascript:void(0);)

namespace UnityDemo

{

class Program

{

static void Main(string[] args)

{

var container = new UnityContainer();

var configuration = ConfigurationManager.GetSection(UnityConfigurationSection.SectionName) as UnityConfigurationSection;

configuration.Configure(container, "defaultContainer");

A a = container.Resolve<IA>() as A;

if (null != a)

{

Console.WriteLine("a.B==null? {0}", a.B == null ? "Yes" : "No");

Console.WriteLine("a.C==null? {0}", a.C == null ? "Yes" : "No");

Console.WriteLine("a.D==null? {0}", a.D == null ? "Yes" : "No");

}

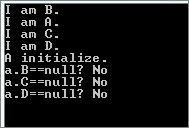
}

}

}

[复制代码](javascript:void(0);)

**执行结果：**



分别体现了接口注入、构造器注入（属性B）、属性注入（属性C）和方法注入（属性D）。

JACK D. @ NJ USA

# [IoC模式（依赖、依赖倒置、依赖注入、控制反转）](http://www.cnblogs.com/fuchongjundream/p/3873073.html)

# 

# 1.依赖

依赖就是有联系，有地方使用到它就是有依赖它，一个系统不可能完全避免依赖。如果你的一个类或者模块在项目中没有用到它，恭喜你，可以从项目中剔除它或者排除它了，因为没有一个地方会依赖它。下面看一个简单的示例：

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

/// <summary>

/// 用户播放媒体文件

/// </summary>

public class OperationMain

{  
 public void PlayMedia()

{

MediaFile \_mtype = new MediaFile();

Player \_player = new Player();

\_player.Play(\_mtype);

}

}

/// <summary>

/// 播放器

/// </summary>

public class Player

{

public void Play(MediaFile file)

{

Console.WriteLine(file.FilePath);

}

}

/// <summary>

/// 媒体文件

/// </summary>

public class MediaFile

{

public string FilePath { get; set; }

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

上面是一个用户用播放器播放文件简单示例，用户操作是OperationMain类中的PlayMedia方法，打开一个播放器，选择一个文件来播放。先看看他们之间的依赖关系，可以简单找到有3个依赖

1. **Player依赖MediaFile**
2. **OperationMain依赖Player**
3. **OperationMain依赖MediaFile**

# 2.依赖倒置

需求增加了，要用不同的播放器，播放不同的文件，我们要抽象出来，减少耦合。

耦合关系就是依赖关系，如果依赖关系相当繁杂，牵一发而动全身，很难维护；依赖关系越少，耦合关系就越低，系统就越稳定，所以我们要减少依赖。

幸亏Robert Martin大师提出了面向对象设计原则----依赖倒置原则：

* **A. 上层模块不应该依赖于下层模块，它们共同依赖于一个抽象。**
* **B. 抽象不能依赖于具象，具象依赖于抽象。**

理解：A.上层是使用者，下层是被使用者，这就导致的结果是上层依赖下层了，下层变动了，自然就会影响到上层了，导致系统不稳定，甚至是牵一发而动全身。那怎么减少依赖呢？就是上层和下层都去依赖另一个抽象，这个抽象比较稳定，整个就来说就比较稳定了。

B.面向对象编程时面向抽象或者面向借口编程，抽象一般比较稳定，实现抽象的具体肯定是要依赖抽象的，抽象不应该去依赖别的具体，应该依赖抽象。

上面播放器的示例中，我们已经找到依赖关系了，现在我们要按照依赖倒置原则，来进行优化。

根据原则如下改动：

* **Player依赖MediaFile，好办，让Player和MediaFile都去依赖一个抽象IMediaFile**
* **OperationMain依赖Player，好办，让OperationMain和Player都依赖一个抽象IPlayer**
* **OperationMain依赖MediaFile，好办，让OperationMain和MediaFile都依赖一个抽象IMediaFile**
* **IPlayer不能依赖具体MediaFile，应该依赖于具体MediaFile的抽象IMediaFile**

结构很简单，于是代码大致如下：

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

/// <summary>

/// 用户播放媒体文件

/// </summary>

public class OperationMain

{  
 public void PlayMedia()

{

IMediaFile \_mtype = new MediaFile();

IPlayer \_player = new Player();

\_player.Play(\_mtype);

}

}

/// <summary>

/// 播放器

/// </summary>

public interface IPlayer

{

void Play(IMediaFile file);

}

/// <summary>

/// 默认播放器

/// </summary>

public class Player : IPlayer

{

public void Play(IMediaFile file)

{

Console.WriteLine(file.FilePath);

}

}

/// <summary>

/// 媒体文件

/// </summary>

public interface IMediaFile

{

string FilePath { get; set; }

}

/// <summary>

/// 默认媒体文件

/// </summary>

public class MediaFile : IMediaFile

{

public string FilePath { get; set; }

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

上面代码进行了抽象，可以看到，目的是减少了依赖，但是看上去依赖关系增加了，如用户PlayMedia方法，依赖还增加了依赖接口和具体的实现，但是接口是稳定的，可以不考虑，具体的实现才是变动的，这个依赖还是要的，要播放文件，必定要用到具体的播放器和具体文件。

# 3.控制反转(IoC)

现实生活中，是具体的播放器和具体的媒体文件没有关系，你给它一个Mp3文件他可以播放，给它一个Mp4文件它也可以播放，你删掉你的媒体文件，播放器照样在，具体什么播放器，播放什么文件，控制权全部是我们用户自己。

上面的示例中基本实现了隔离，具体的播放器跟具体的媒体隔离了，具体的播放器只跟媒体接口和播放器接口有关。但是PlayMedia的方法里面的具体对象，写死了，控制权非常小，如果我想用百度影音播放呢，我想换一首音乐呢，只能重新改代码，那控制怎么进行转移呢？

我们可以通过反射来创建，把具体的文件名写在配置文件里，这时候客户端代码也不用变了，只需要改配置文件就好了，稳定性又有了提高，如下：

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

public void PlayMedia()

{

IMediaFile \_mtype = Assembly.Load(ConfigurationManager.AppSettings["AssemName"]).CreateInstance(ConfigurationManager.AppSettings["MediaName"]);

IPlayer \_player = Assembly.Load(ConfigurationManager.AppSettings["AssemName"]).CreateInstance(ConfigurationManager.AppSettings["PlayerName"]);

\_player.Play(\_mtype);

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

这个具对象是哪一个，全由配置文件来控制了，这个具体对象的控制权交给了配置文件了，这也是人们常说的控制反转。

控制反转IoC是Inversion of Control的缩写，是说对象的控制权进行转移，转移到第三方，比如转移交给了IoC容器，它就是一个创建工厂，你要什么对象，它就给你什么对象，有了IoC容器，依赖关系就变了，原先的依赖关系就没了，它们都依赖IoC容器了，通过IoC容器来建立它们之间的关系。

# 4.依赖注入(DI)

上面说到控制反转，是一个思想概念，但是也要具体实现的，上面的配置文件也是一种实现方式。依赖注入提出了具体的思想。

依赖注入DI是Dependency Injection缩写，它提出了“哪些东东的控制权被反转了，被转移了？”，它也给出了答案：“依赖对象的创建获得被反转”。

所谓依赖注入，就是由IoC容器在运行期间，动态地将某种依赖关系注入到对象之中。

上面的示例中，哪些要依赖注入，依赖对象需要获得实例的地方，即 PlayMedia方法，需要IPlayer具体对象和IMediaFile的具体对象，找到了地方就从这里下手，为了灵活的控制这两个对象，必须是外面能够控制着两个对象的实例化，提供对外的操作是必要的，可以是属性，可以是方法，可以是构造函数，总之别的地方可以控制它，下面将会使用Unity来注入，使用的是构造函数注入，代码如下：

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

/// <summary>

/// 用户播放媒体文件

/// </summary>

public class OperationMain

{

IMediaFile \_mtype;

IPlayer \_player;

public OperationMain(IPlayer player, IMediaFile mtype)

{

\_player = player;

\_mtype = mtype;

}

public void PlayMedia()

{

\_player.Play(\_mtype);

}

}

/// <summary>

/// 播放器

/// </summary>

public interface IPlayer

{

void Play(IMediaFile file);

}

/// <summary>

/// 默认播放器

/// </summary>

public class Player : IPlayer

{

public void Play(IMediaFile file)

{

Console.WriteLine(file.FilePath);

}

}

/// <summary>

/// 媒体文件

/// </summary>

public interface IMediaFile

{

string FilePath { get; set; }

}

/// <summary>

/// 默认媒体文件

/// </summary>

public class MediaFile : IMediaFile

{

public string FilePath { get; set; }

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

给 OperationMain类一个构造函数，因为Unity有一个构造函数注入，调用代码如下：

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

static UnityContainer container = new UnityContainer();

static void init()

{

container.RegisterType<IPlayer, Player>();

container.RegisterType<IMediaFile, MediaFile>();

}

static void Main(string[] args)

{

init();

OperationMain op1 = container.Resolve<OperationMain>();

op1.PlayMedia();

OperationMain op3 = container.Resolve<OperationMain>();

op3.PlayMedia();

//普通方式

OperationMain op2 = new OperationMain(new Player(), new MediaFile());

op2.PlayMedia();

Console.Read();

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

看出来吧，Unity的功能远不止这些，你可以初始化时注册N多，以后直接使用，而不用使用new，还有实例周期的控制、配置文件等灵活控制，具体可以看看Unity（具体不是本节的范畴）的说明。

# 5.小结

通过一个小例子由浅入深地进行优化，已加深对IoC模式的理解，我想复杂的结构也是从这种简单的架构累加起来的。

最近在看相关文章，很多都太专业化了没怎么看懂，这是自己现在对IoC的一些理解，记录下来，要不然时间一久，也就忘了。

自己对IoC模式理解还很浅，希望得到各位的指点。

学习过**[spring](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "_blank)**框架的人一定都会听过Spring的IoC(控制反转) 、DI(依赖注入)这两个概念，对于初学Spring的人来说，总觉得IoC 、DI这两个概念是模糊不清的，是很难理解的，今天和大家分享网上的一些技术大牛们对Spring框架的IOC的理解以及谈谈我对Spring Ioc的理解。

## 一、分享Iteye的开涛对Ioc的精彩讲解

　　首先要分享的是Iteye的开涛这位技术牛人对Spring框架的IOC的理解，写得非常通俗易懂，以下内容全部来自原文，原文地址：http://jinnianshilongnian.iteye.com/blog/1413846

### 1.1、IoC是什么

**Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。**在[**Java**](http://lib.csdn.net/base/java)开发中，**Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。**如何理解好Ioc呢？理解好Ioc的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

　　●谁控制谁，控制什么：传统[**Java**](http://lib.csdn.net/base/java)SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对 象的创建；**谁控制谁？当然是IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。**

　　●为何是反转，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？**因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。**

　　用图例说明一下，传统程序设计如图2-1，都是主动去创建相关对象然后再组合起来：

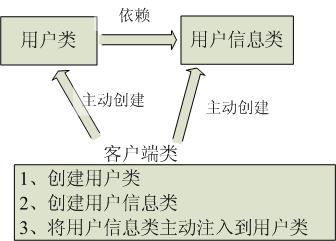


图1-1 传统应用程序示意图

　　当有了IoC/DI的容器后，在客户端类中不再主动去创建这些对象了，如图2-2所示:

图1-2有IoC/DI容器后程序结构示意图

### 1.2、IoC能做什么

　　IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于[**测试**](http://lib.csdn.net/base/softwaretest)；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是 松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

　　其实**IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。**

**IoC很好的体现了面向对象设计法则之一—— 好莱坞法则：“别找我们，我们找你”；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。**

### 1.3、IoC和DI

**DI—Dependency Injection，即“依赖注入”**：**组件之间依赖关系**由容器在运行期决定，形象的说，即**由容器动态的将某个依赖关系注入到组件之中**。**依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。**通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

　　理解DI的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

　　●谁依赖于谁：当然是**应用程序依赖于IoC容器**；

　　●为什么需要依赖：**应用程序需要IoC容器来提供对象需要的外部资源**；

　　●谁注入谁：很明显是**IoC容器注入应用程序某个对象，应用程序依赖的对象**；

　　●注入了什么：就是**注入某个对象所需要的外部资源（包括对象、资源、常量数据）**。

**IoC和DI**由什么**关系**呢？其实它们**是同一个概念的不同角度描述**，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物Martin Fowler又给出了一个新的名字：“依赖注入”，相对IoC 而言，“依赖注入”明确描述了“被注入对象依赖IoC容器配置依赖对象”。

　　看过很多对Spring的Ioc理解的文章，好多人对Ioc和DI的解释都晦涩难懂，反正就是一种说不清，道不明的感觉，读完之后依然是一头雾水，感觉就是开涛这位技术牛人写得特别通俗易懂，他清楚地解释了IoC(控制反转) 和DI(依赖注入)中的每一个字，读完之后给人一种豁然开朗的感觉。我相信对于初学Spring框架的人对Ioc的理解应该是有很大帮助的。

## 二、分享Bromon的blog上对IoC与DI浅显易懂的讲解

### 2.1、IoC(控制反转)

　　首先想说说**IoC（Inversion of Control，控制反转）**。这是**spring的核心**，贯穿始终。**所谓IoC，对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系。**这是什么意思呢，举个简单的例子，我们是如何找女朋友的？常见的情况是，我们到处去看哪里有长得漂亮身材又好的mm，然后打听她们的兴趣爱好、qq号、电话号、ip号、iq号………，想办法认识她们，投其所好送其所要，然后嘿嘿……这个过程是复杂深奥的，我们必须自己设计和面对每个环节。传统的程序开发也是如此，在一个对象中，如果要使用另外的对象，就必须得到它（自己new一个，或者从JNDI中查询一个），使用完之后还要将对象销毁（比如Connection等），对象始终会和其他的接口或类藕合起来。

　　那么IoC是如何做的呢？有点像通过婚介找女朋友，在我和女朋友之间引入了一个第三者：婚姻介绍所。婚介管理了很多男男女女的资料，我可以向婚介提出一个列表，告诉它我想找个什么样的女朋友，比如长得像李嘉欣，身材像林熙雷，唱歌像周杰伦，速度像卡洛斯，技术像齐达内之类的，然后婚介就会按照我们的要求，提供一个mm，我们只需要去和她谈恋爱、结婚就行了。简单明了，如果婚介给我们的人选不符合要求，我们就会抛出异常。整个过程不再由我自己控制，而是有婚介这样一个类似容器的机构来控制。**Spring所倡导的开发方式**就是如此，**所有的类都会在spring容器中登记，告诉spring你是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由 spring来控制，也就是说控制对象生存周期的不再是引用它的对象，而是spring。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转。**

### 2.2、DI(依赖注入)

**IoC的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的**。比如对象A需要操作**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "_blank)**，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了 spring我们就只需要告诉spring，A中需要一个Connection，至于这个Connection怎么构造，何时构造，A不需要知道。在系统运行时，spring会在适当的时候制造一个Connection，然后像打针一样，注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connection才能正常运行，而这个Connection是由spring注入到A中的，依赖注入的名字就这么来的。那么DI是如何实现的呢？ Java 1.3之后一个重要特征是反射（reflection），它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring就是通过反射来实现注入的。

　　理解了IoC和DI的概念后，一切都将变得简单明了，剩下的工作只是在spring的框架中堆积木而已。

## 三、我对IoC****(控制反转)****和DI****(依赖注入)****的理解

　　在平时的java应用开发中，我们要实现某一个功能或者说是完成某个业务逻辑时至少需要两个或以上的对象来协作完成，在没有使用Spring的时候，每个对象在需要使用他的合作对象时，自己均要使用像new object() 这样的语法来将合作对象创建出来，这个合作对象是由自己主动创建出来的，创建合作对象的主动权在自己手上，自己需要哪个合作对象，就主动去创建，创建合作对象的主动权和创建时机是由自己把控的，而这样就会使得对象间的耦合度高了，A对象需要使用合作对象B来共同完成一件事，A要使用B，那么A就对B产生了依赖，也就是A和B之间存在一种耦合关系，并且是紧密耦合在一起，而使用了Spring之后就不一样了，创建合作对象B的工作是由Spring来做的，Spring创建好B对象，然后存储到一个容器里面，当A对象需要使用B对象时，Spring就从存放对象的那个容器里面取出A要使用的那个B对象，然后交给A对象使用，至于Spring是如何创建那个对象，以及什么时候创建好对象的，A对象不需要关心这些细节问题(你是什么时候生的，怎么生出来的我可不关心，能帮我干活就行)，A得到Spring给我们的对象之后，两个人一起协作完成要完成的工作即可。

　　所以**控制反转IoC(Inversion of Control)是说创建对象的控制权进行转移，以前创建对象的主动权和创建时机是由自己把控的，而现在这种权力转移到第三方**，比如转移交给了IoC容器，它就是一个专门用来创建对象的工厂，你要什么对象，它就给你什么对象，有了 IoC容器，依赖关系就变了，原先的依赖关系就没了，它们都依赖IoC容器了，通过IoC容器来建立它们之间的关系。

　　这是我对Spring的IoC**(控制反转)**的理解。DI**(依赖注入)**其实就是IOC的另外一种说法，DI是由Martin Fowler 在2004年初的一篇论文中首次提出的。他总结：**控制的什么被反转了？就是：获得依赖对象的方式反转了。**

## 四、小结

　　对于Spring Ioc这个核心概念，我相信每一个学习Spring的人都会有自己的理解。这种概念上的理解没有绝对的标准答案，仁者见仁智者见智。如果有理解不到位或者理解错的地方，欢迎广大园友指正！