[C#墨攻IOC](http://www.cnblogs.com/zhangchenliang/archive/2013/01/08/2850975.html)

原文叫看《墨攻》理解IOC概念

2006年多部贺岁大片以让人应接不暇的频率纷至沓来，其中张之亮的《墨攻》算是比较出彩的一部，讲述了战国时期墨家人革离帮助梁

国反抗赵国侵略的个人英雄主义故事，恢宏壮阔，浑雄凝重的历史场面相当震撼。其中有一个场景：当刘德华所饰的墨者革离到达梁国都城

下，城上梁国守军问：“来者何人？”，刘德华回答：“墨者革离！”，我们不妨用C#（原文是java，我修改）对这段“城门问对”的场景进行编剧并借由这个例子来理解IoC的内涵。

剧本和饰演者耦合

MoAttack代表《墨攻》的剧本，cityGetAsk()代表“城门问对”这段剧情，LiuDeHua是具体饰演者刘德华：

**代码清单1**

public class MoAttack {

     public MoAttack() {}

     public void cityGateAsk(){

        LiuDeHua ldh = new LiuDeHua(); ① 演员直接侵入剧本

        ldh.responseAsk("墨者革离！");

   }

}

我们会发现以上剧本在①处，作为具体饰演者的刘德华直接侵入到剧本中，使剧本和演员直接耦合在一起：

http://www.uml.org.cn/c++/images/201111101.gif

**图(1)剧本与演员直接耦合**

一个明智的编剧在剧情创作时应围绕故事的角色进行，而不应考虑角色的具体饰演者，这样才可能在剧本投拍时自由地选择任何适合的演员，而非绑定在刘德华一人身上。通过以上的分析，我们知道需要为该剧本主人公革离定义一个接口，以角色进行剧情安排，饰演者实现角色的接口：

**代码清单2 MoAttack：引入剧本角色**

public class MoAttack{

    public MoAttack() {}

    public void cityGateAsk()

    {

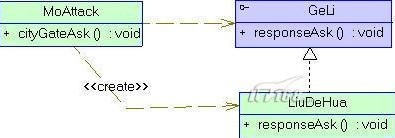
         GeLi geli = new LiuDeHua(); ① 引入革离角色接口

         geli.responseAsk("墨者革离！"); ② 通过接口开展剧情

     }

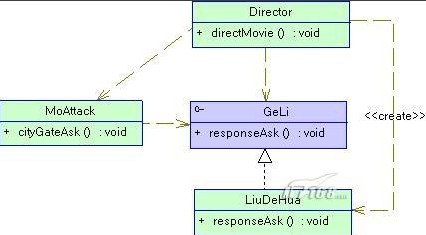
}

在①处引入了剧本的角色——革离，剧本的情节通过角色展开，在拍摄时角色的事迹由演员表现，如②处所示。因此剧本、革离、刘德华三者的类图关系如图2所示：



**图2剧本、革离、刘德华三者的类图关系**

我们希望剧本和演员无关，可是，在图2中，我们看到MoAttack同时依赖于GeLi接口和LiuDeHua类，并没有达到我们所期望的剧本仅依赖于角色的目的。可是角色最终又必须通过具体的演员才能完成拍摄，如何将让LiuDeHua和剧本无关而又能完成GeLi的具体动作呢？当然是在影片投拍时，导演将LiuDeHua安排在GeLi的角色上，通过导演之手将剧本、角色、饰演者装配起来。



**图3剧本和饰演者解耦了**

通过引入导演，剧本和具体的饰演者解耦了，对应到软件中，导演象是一个装配器，将具体的饰演者赋给了剧本的角色。

现在我们可以反过来讲解IOC的概念了。IOC（Inverse of Control）的字面意思是控制反转，它包括两个层面的内容：其一是“控制”，其二是“反转”，到底是什么东西的控制被反转了呢？对应到前面的例子， “控制”是指GeLi角色扮演者的选择控制权，“反转”是指这种选择控制权从《墨攻》剧本中移除，转交到导演的手中。对于程序来说，即是某一接口具体实现类的选择控制权从客户类中移除，转交给第三方来确定，客户类不知道是哪个具体的实现类，它通过接口方法对实现类进行调用。

因为IOC确实不够开门见山，因此业界曾进行了广泛的讨论，最终软件界的泰斗级人物Martin Fowler提出了DI（依赖注入：Dependency Injection）的概念，即将客户类对接口实现类的依赖关系由第三方（容器或协作类）注入，以移除客户类对具体接口实现类的依赖。“依赖注入”的概念显然比“控制反转”直接达意，易于理解。

IOC的三种类型

从注入方法上看，主要可以划分为三种的注入类型，分别是构造函数注入、属性注入和接口注入，Spring.Net支持构造函数注入和属性注入。下面我们继续使用以上的例子说明这三种注入方法的区别。

构造函数注入

我们通过客户类的构造函数，将接口实现类通过接口变量传入，如代码清单3所示：

**代码清单3 MoAttack：通过构造函数注入革离扮演者**

public class MoAttack{

    public MoAttack(){}

    private GeLi geli;

    public MoAttack(GeLi geli){ ① 注入革离的具体扮演者

        this.geli = geli;

     }

     public void cityGateAsk()

     {

        geli.responseAsk(“墨者革离！”);

    }

}

MoAttack的构造函数不关心具体是谁扮演革离这个角色，只要在①处传入的扮演者按剧本要求完成角色功能即可。

角色的具体扮演者由导演来安排，如代码清单4所示：

**代码清单 4 Director：通过构造函数注入革离扮演者**

public class Director {

     public void direct(){

        GeLi geli = new LiuDeHua(); ① 指定角色的扮演者

        MoAttack moAttack = new MoAttack(geli); ② 注入具体扮演者到剧本中

        moAttack.cityGateAsk();

    }

}

属性注入

有时，导演会发现，虽然革离是影片《墨攻》的第一主人公，但并非每场戏都需要革离的出现，通过构造函数方式注入显得很不妥当，在这种情况下，可以使用属性注入进行改造。属性注入通过通.Net 属性完成客户类所需依赖的注入，更灵活，更方便。

**代码清单5 MoAttack：通.Net 属性器注入革离扮演者**

public class MoAttack{

     private GeLi gelii;

    public GeLi Gelii{ ① 属性注入方法

         set{ gelii = value; }

    }

    public void cityGateAsk() ...{

         geli.responseAsk("墨者革离");

    }

}

MoAttack在①处为geli 字段提供一个属性，以便让导演在拍需要革离的戏时才将注入geli的具体扮演者，而不需要刘德华从头到尾跟着墨攻剧组跑。

**代码清单 6 Director：通过属性注入革离扮演者**

public class Director{

    public void direct(){

        GeLi geli = new LiuDeHua();

         MoAttack moAttack = new MoAttack();

        moAttack.Gelii = geli; ① 调用属性注入

        moAttack.cityGateAsk();

    }

}

和通过构造函数注入革离扮演者不同，在实例化MoAttack时，并未指定任何扮演者，而是在实例化MoAttack后，调用其属性注入扮演者。按照类似的方式，我们还可以为剧本中其他如巷淹中，梁王等角色分别提供注入的属性，导演即可以根据所拍剧段的不同，注入所需要的角色了。

接口注入

将客户类所有注入的方法抽取到一个接口中，客户类通过实现这一接口提供注入的方法。为了采取接口注入的方式，需要声明一个额外的接口：

public interface IActorArrangable{

     void injectGeli(GeLi geli);

}

然后，MoAttack实现这个接口并实现接口中的方法：

**代码清单7 MoAttack：通过接口方法注入革离扮演者**

public class MoAttack : IActorArrangable{

    private GeLi geli;

    public void injectGeli (GeLi geli) { ① 实现接口方法

        this.geli = geli;

    }

     public void cityGateAsk() ...{

        geli.responseAsk("墨者革离");

     }

}

Director通过IActorArrangable接口的injectGeli()方法完成扮演者的注入工作。

**代码清单 8 Director：通过接口方法注入革离扮演者**

public class Director{

     public void direct(){

        GeLi geli = new LiuDeHua();

        MoAttack moAttack = new MoAttack();

        moAttack.injectGeli (geli);

        moAttack.cityGateAsk();

    }

}

由于通过接口注入需要额外声明一个接口，增加了类的数目，而且它的效果和属性注入并无本质区别，因此我们不提倡这种方式。

通过容器完成依赖关系的建立

虽然MoAttack和LiuDeHua实现了解耦，无需关注实现类的实例化工作，但这些工作在代码中依然存在，只是转移到Director中而已，导致导演的权力非常大，潜规则不断滋生。假设某一制片人想改变这一局面，在相中某个剧本后，通过一个“海选”或者第三公正中介来选择导演、演员，让他们各司其职，那剧本、导演、演员就都实现解耦了。

所谓媒体“海选”和中介机构在程序领域即是一个第三方容器，它帮助我们完成类的初始化和装配工作，让我们从这些底层的实现类实例化，依赖关系的装配中脱离出来，专注于更有意思的业务代码的编写工作，那确实是挺惬意的事情。Spring.Net就是这样一个容器，它通过配置文件描述类之间的依赖关系，下面是Spring.Net配置文件的对以上实例进行配置的样式代码：

<objects>

    <object id="geli" type="com.baobaotao.LiuDeHua"></object>

    <object id="moAttack" type=" com.baobaotao.MoAttack">

        <property name="geli"><ref="geli"/></property>

    </object>

</objects>