**Web后台快速开发框架使用说明书**

**Coldairarrow**

# 目录

目录

[第1章 目录 1](#_Toc524429040)

[第2章 简介 3](#_Toc524429041)

[第3章 基础准备 4](#_Toc524429042)

[3.1 开发环境要求 4](#_Toc524429043)

[3.2 基础数据库构建 4](#_Toc524429044)

[3.3 运行 5](#_Toc524429045)

[第4章 详细教程 6](#_Toc524429046)

[4.1 代码架构 6](#_Toc524429047)

[4.1.1 总体架构 6](#_Toc524429048)

[4.1.2 基础设施层 7](#_Toc524429049)

[4.1.3 数据仓储层 9](#_Toc524429050)

[4.1.4 数据实体层 13](#_Toc524429051)

[4.1.5 业务逻辑层 14](#_Toc524429052)

[4.1.6 应用展示层 16](#_Toc524429053)

[4.2 功能架构 17](#_Toc524429054)

[4.2.1 全局配置 17](#_Toc524429055)

[4.2.2 快速开发 17](#_Toc524429056)

[4.2.3 管理员登录 23](#_Toc524429057)

[4.2.4 系统用户管理 24](#_Toc524429058)

[4.2.5 系统角色管理 25](#_Toc524429059)

[4.2.6 权限管理 26](#_Toc524429060)

[4.2.7 接口秘钥管理 28](#_Toc524429061)

[4.2.8 系统日志 28](#_Toc524429062)

[第5章 结语 29](#_Toc524429063)

# 简介

本框架旨在为.NET开发人员提供一个Web后台快速开发框架，采用本框架，能够极大的提高项目开发效率。

整个框架包括三个版本：

.NET新版，采用.NET452,GitHub地址为：<https://github.com/Coldairarrow/Coldairarrow.Fx.Net.Easyui.GitHub>

.NET40版，采用.NET40,GitHub地址为：<https://github.com/Coldairarrow/Coldairarrow.Fx.Net40.Easyui.GitHub>

.NET Core版，采用.NET Core2.1, GitHub地址为：<https://github.com/Coldairarrow/Coldairarrow.Fx.Core.Easyui.GitHub>

以上三个版本中，.NET新版，主要支持最新的技术方案，作为主要生产版本；.NET40版是为了兼容Windows 2003服务器而降级的版本，功能正常。.NET Core 版本是未来的发展方向，能够跨平台，并且涉及Linux、Docker、Nginx、微服务等概念，已完成移植。

# 基础准备

## 开发环境要求

操作系统：Windows 10

开发工具：Visual Studio 2017

SDK：安装.NET4.52及以上

数据库：SQLServer2008 R2及以上

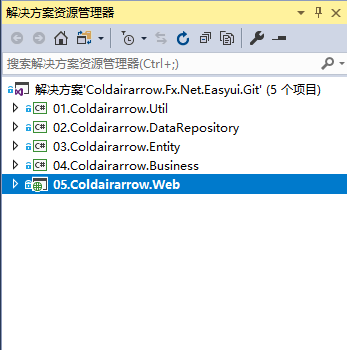
## 基础数据库构建

使用本框架需要构建基础数据库，具体步骤如下：

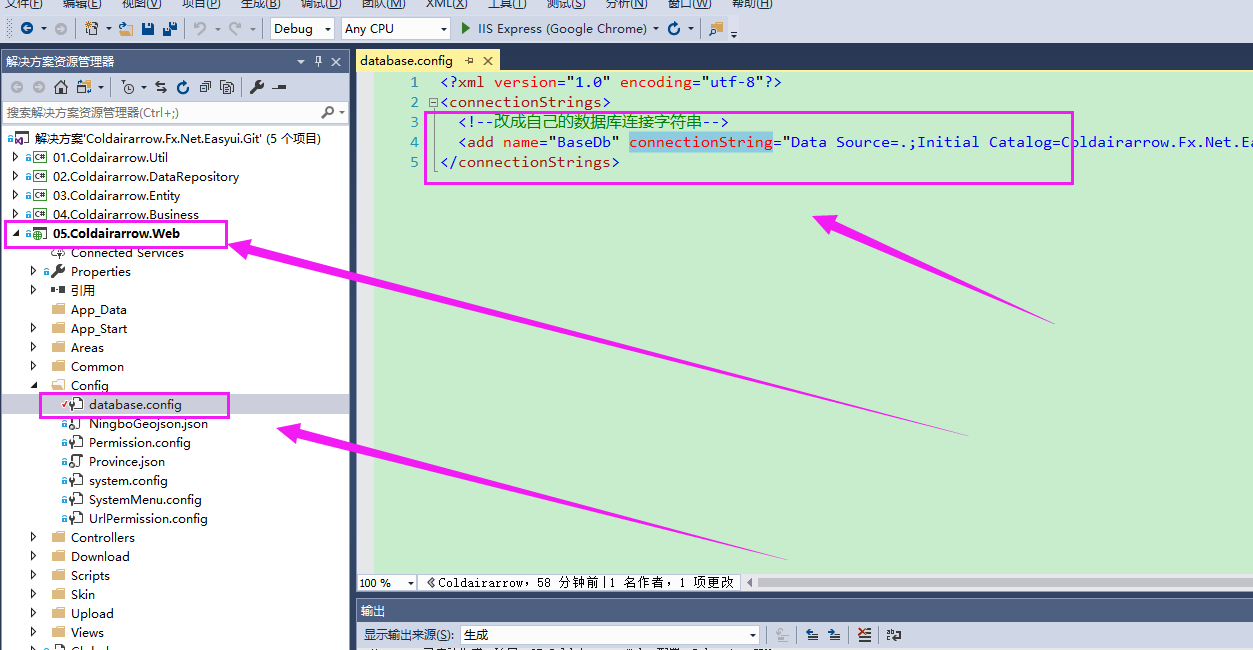
创建基础数据库的Sql脚本文件在：/docs/初始化文件/db.sql，在数据库中运行db.sql脚本即可创建数据库:Coldairarrow.Fx.Net.Easyui.Git

若Sql运行出错，请直接使用同目录下的Coldairarrow.Fx.Net.Easyui.GitHub.bak还原数据库

打开src目录下Coldairarrow.Fx.Net.Easyui.Git.sln的解决方案，如下图



如下图所示依次展开05.Coldairarrow.Web=>Config=>database.config，配置数据库连接字符串，name不用修改，connectionString改为上述创建的数据库（若不清楚数据库连接字符串请自行百度搜索教程）



自此基础数据库配置完成。

## 运行

请先还原Nuget包

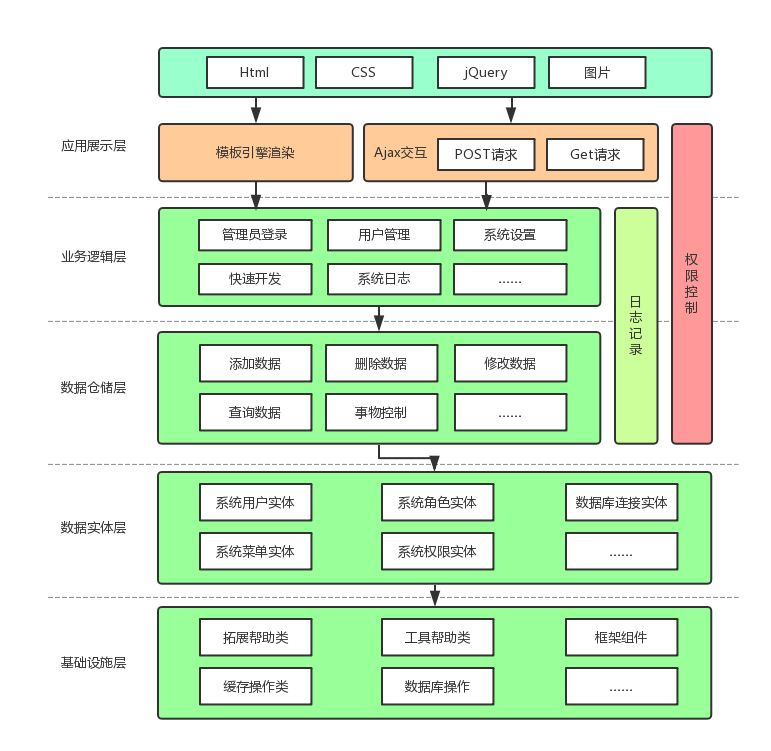
然后将05.Coldairarrow.Web设为启动项目，成功运行即可进入以下页面



# 详细教程

## 代码架构

### 总体架构



框架组成结构一共分为5层，如上图所示，分别如下：

基础设施层：此层为最底层，可以为其余所有层服务。主要提供了项目开发所需的各种帮助类：数据库访问帮助类、文件操作帮助类、二维码生成帮助类、分拣压缩帮助类等等其余帮助类；拓展类：字符串拓展类、集合操作拓展类、表达式树拓展类等等其余拓展类；还集成了个人编写的Socket通讯框架，WCF拓展使用框架，Windows服务容器。这些丰富的类库都是为开发人员提供了开发中常用的功能，为快速开发提供强有力的保障。

数据仓储层：这层主要为对数据库操作CRUD的简单封装，以EntityFramework为核心，采用简单工厂、抽象工厂、工厂方法、三个工厂设计模式，使开发人员进行CRUD只需要极为简单的代码即可完成。本层还提供了数据库事务的支持，更是提供了分布式事务支持，为数据库操作提供必备的保障。使用本层提供的接口，无需关心具体的数据库类型，比如是采用SQLServer数据库或者MySQL数据库，开发人员只需要关心具体的业务逻辑实现，哪怕更换数据库，也无需更改业务逻辑代码，只需要更改简单的数据库配置即可。总之，本层为开发人员对数据库的操作提供了简单高效的操作接口，可以极大的提高开发效率。

实体层：这层主要为ORM框架数据库表对应的实体类，为业务逻辑层和应用层服务。

业务逻辑层：本层是开发人员主要编写层，通过调用数据仓储层操作数据库，并为应用层提供所需的接口，处理具体操作的业务逻辑，可以说是最为复杂的一层。

应用层：本层在本框架中表现形式为ASP.NET MVC网站，其实也可以泛指其余的表现形式：控制台程序、Windows服务程序、WinForm程序、WPF程序等等。本层为具体的应用，负责系统功能的实现。

### 基础设施层

此层为最底层，可以为其余所有层服务。主要提供了项目开发所需的各种类库，主要为以下几种类库：

1. 拓展帮助类

利用C#的语法糖（是由英国计算机科学家彼得·约翰·兰达（Peter J. Landin）发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用），可以在类上拓展自定义方法，这样开发人员在使用的时候就可以十分的方便，既能减少重复的代码又能加快开发效率。具体包含但不限于以下类：

Object拓展：Object是所有类的基类。现在前后端数据交互中，采用JSON是选择，因此对象JSON序列化与反序列化的使用就十分的频繁，通过给Object拓展ToJson方法，就可以将数据对象直接转为对应的JSON字符串，能够极大的简化JSON序列化所需要的代码。

Byte拓展：Byte，即字节，一个字节是8个比特位，十进制数值范围在0-255。由于Byte与二进制是直接对应的，而计算机中一切数据都是二进制，所以关于Byte与其他数据类型之间的转换就显得尤为重要，编写Byte相关的拓展方法能够方便Byte的使用。例如，拓展将Byte数组转为16进制字符串的方法，能够将字节数组直接转为对应的16进制字符串，现在物联网的发展也十分的迅速，在物联网开发中，关键是建立与硬件之间的通信，但是由于硬件的配置一般都不高，因此与硬件之间的通信大多以字节为单位，这种情况下，使用该拓展方法就可以提高开发效率。

Expression拓展：Expression，即表达式树，在LINQ查询中经常使用。在日常使用中,Where筛选估计是用得最多的，但是大多数开发人员都是使用最原始的Where筛选，当筛选条件增多，筛选条件变复杂的时候，原始的Where虽然也可以胜任，但是就会导致代码重复，不够简洁雅观，此时可以为Expression<Func<T, bool>>类拓展And与Or方法，其中And是“与”操作，Or是“或”操作，这样就可以将多个筛选条件拼接在一起，十分的实用。

IQueryable<T>拓展：IQueryable<T>是提供针对特定数据源（其中数据类型未未知）评估查询的功能接口，其在EntityFramework中发挥着重要的作用。例如，在后台管理系统中，数据表格的使用最为普遍，前端向后台请求表格数据时，不外乎需要以下几个参数：当前请求页码、每页记录数、排序列与排序类型，这些参数在每个请求表格数据请求中都需要处理，这时，通过给IQueryable<T>拓展GetPagination<T>方法，只需要传入分页参数，就可以获取所需的数据表格数据，十分的方便高效。

1. 工具帮助类

主要提供了常用的一些帮助类，包含但不限于：

Office办公文档导入导出帮助类：当今，随着人们环保意识的增强，以及各行业对办公模式需求的不断升级，现代化、信息化建设步伐的加快，无纸化办公已经由概念逐渐应用到多个行业领域中，办公中，各种办公文件，尤其是Excel文件与Word文件经常需要导入与导出，但是C#操作Excel与Word文件并不简单，经常困扰着开发者，由此，本框架提供了Office文档操作帮助类，封装简化了对办公文档的常用操作，能够降低开发难度，提高开发效率。

HTTP请求模拟帮助类：在传统网站开发中，一般都是前端浏览器向后台发起请求，但是，现在的系统与系统之间的合作越来越紧密，经常需要后端向后端发起请求，即需要后端模拟HTTP请求，但编写一个完整的HTTP请求并不是那么容易，因此本框架提供HTTP请求帮助类，开发人员只需要传入需要请求的URL地址与参数即可完成HTTP请求操作，使用起来简单高效，能够极大的提高开发效率。

1. 缓存操作帮助类

在现代化系统开发中，随着业务量的增大，系统性能就难以满足要求，要达到性能要求，一方面可以采用更好的硬件，但是成本较高，而另一方面就是使用缓存，有效使用缓存能够提高吞吐量与并发量，所需成本较低，是绝大多数用户的首选。

.NET Framework框架提供了系统缓存，虽然使用简单方便，但是不支持分布式，因此大多选择诸如Redis和Memcached缓存，但是不同的缓存为开发者提供的接口不一样，当使用不同的缓存时开发人员又需要去学习别的缓存操作接口，十分的麻烦，为解决这个问题，本框架提供了缓存操作帮助类。缓存操作帮助类将缓存核心操作抽象定义成操作接口：添加缓存、删除缓存、设置缓存过期时间，然后再用Redis与系统缓存实现抽象接口，当使用缓存时可以使用同样的操作接口操作不同的缓存，能够降低开发人员学习成本，提高开发效率。

1. 数据库操作帮助类

无论什么网站，只要需要对数据进行操作，那么大多离不开数据库。数据库目前使用最多的就是三大关系型数据库：SQLServer、MySQL与Oracle，访问数据库需要一系列的程序：首先需要创建数据库连接对象，紧接着打开数据库，其次传入数据库操作命令，然后执行命令，最后得到数据结果，若每次访问数据库都要写一遍这些流程，那么将会极大的阻碍开发效率，因此，本框架提供了数据库操作帮助类，将数据库操作所需要的流程封装，开发人员只需要关心具体的SQL语句的编写即可，并且支持三大关系型数据库，适合不同的开发人员使用。

### 数据仓储层

在后端开发中，数据库操作是最频繁的，每一个后端开发人员或多或少都会接触，甚至不少开发人员每天的工作就是与数据库打交道，所以可见数据库操作是多么的重要。在现在开发的过程中，绝大多数开发人员只是以编写SQL语句的方式操作数据库，这种方式是操作数据库最基本最原始的方式，简单高效，但是在编写SQL语句的过程中，极容易因马虎大意而编写错误，就会出现一个现象：开发人员面对着一堆SQL语句在DEBUG，而且每次都需要开发人员自己去手写SQL语句，其开发效率极低。哪怕开发人员足够出色，能够保证编写的SQL语句较低的出错率，但是，不同的数据库所需要的SQL语句还是有差异的，这就需要开发人员学习不同的数据库SQL语法，添加学习成本。而且在项目开发中难免会遇到更换数据库的情况，这时还需要花费大量的精力去进行修改SQL语句。

在本框架的数据仓储层中，上述问题即可迎刃而解。数据仓储层，不同于传统三层架构中的数据层，其核心继承关系图如下图所示（图6-1）：

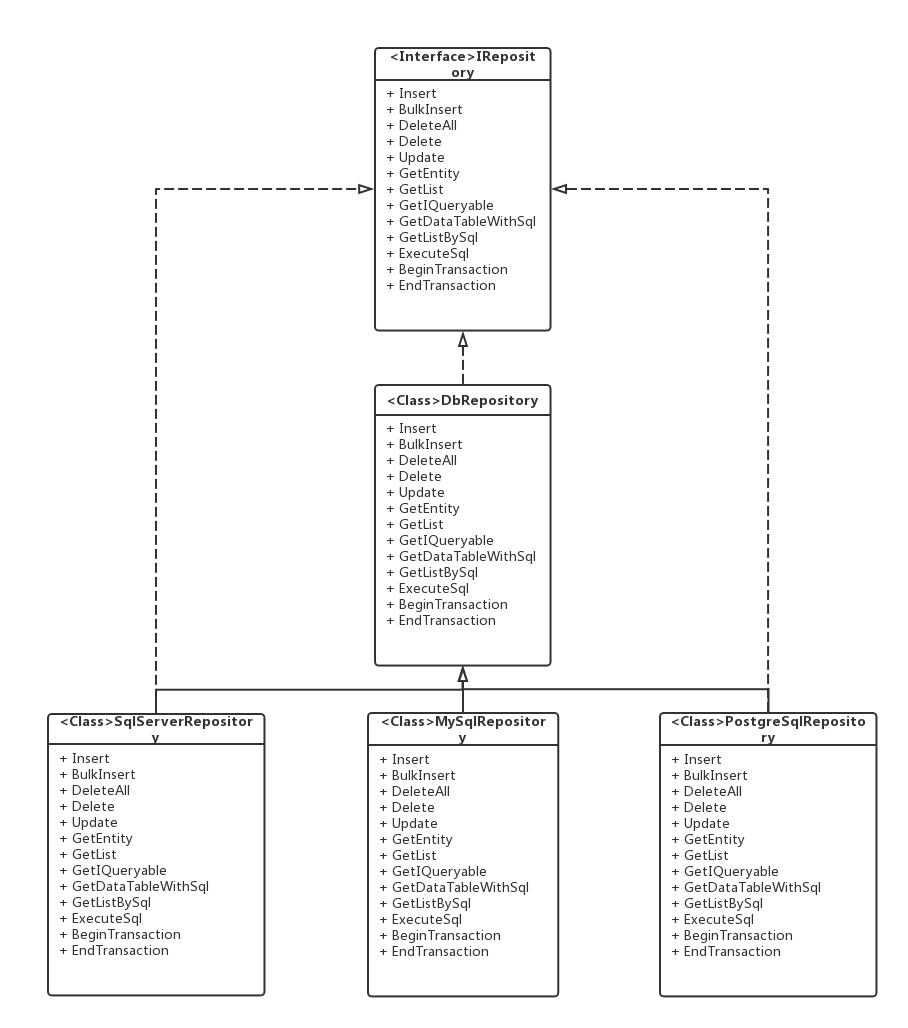


图6-1 数据仓储类图

如上图所示，首先定义了数据操作接口IRepository,该接口包含了增、删、改、查、事物控制等数据库常用核心操作，能够满足对数据库的常用操作，DbRepository类实现了IRepository接口，主要以ORM框架Entity Framework为基础，封装实现了大部分IRepository所需的操作，SqlServerRepository、MySqlRepository和PostgreSqlRepository分别实现具体数据库对应的数据仓储，而且继承自同一个数据操作接口IRepository，因此在具体的使用上，可以实现以同一个操作方法访问不同的数据库，当遇到需要更换数据库的情况时，采用本框架开发的系统能够不改代码而正常运行，这一点能够极大的降低软件开发成本。并且以Entity Framework为核心，不需要编写SQL语句就能够完成绝大部分的数据库操作，再加上简洁的LINQ配合，彻底将开发人员从SQL语句中解放出来，让开发人员能够更加专注于业务逻辑的实现，能够极大的提高软件开发效率。

IRepository代码如下：

using System;

using System.Collections.Generic;

using System.Data;

using System.Data.Common;

using System.Data.Entity;

using System.Linq;

using System.Linq.Expressions;

namespace Coldairarrow.DataRepository

{

public interface IRepository

{

#region 数据库连接相关方法

DbContext GetDbContext();

#endregion

#region 事物提交

/// <summary>

/// 开始单库事物

/// 注意:若要使用跨库事务,请使用DistributedTransaction

/// </summary>

void BeginTransaction();

bool EndTransaction();

#endregion

#region 增加数据

void Insert<T>(T entity) where T : class, new();

void Insert<T>(List<T> entities) where T : class, new();

void BulkInsert<T>(List<T> entities) where T : class, new();

#endregion

#region 删除数据

void DeleteAll<T>() where T : class, new();

void Delete<T>(string key) where T : class, new();

void Delete<T>(List<string> keys) where T : class, new();

void Delete<T>(T entity) where T : class, new();

void Delete<T>(List<T> entities) where T : class, new();

void Delete<T>(Expression<Func<T, bool>> condition) where T : class, new();

#endregion

#region 更新数据

void Update<T>(T entity) where T : class, new();

void Update<T>(List<T> entities) where T : class, new();

void UpdateAny<T>(T entity, List<string> properties) where T : class, new();

void UpdateAny<T>(List<T> entities, List<string> properties) where T : class, new();

#endregion

#region 查询数据

T GetEntity<T>(object keyValue) where T : class, new();

List<T> GetList<T>() where T : class, new();

IQueryable<T> GetIQueryable<T>() where T : class, new();

DataTable GetDataTableWithSql(string sql);

DataTable GetDataTableWithSql(string sql, List<DbParameter> parameters);

List<T> GetListBySql<T>(string sqlStr) where T : class, new();

List<T> GetListBySql<T>(string sqlStr, List<DbParameter> parameters) where T : class, new();

#endregion

#region 执行Sql语句

void ExecuteSql(string sql);

void ExecuteSql(string sql, List<DbParameter> parameters);

#endregion

}

}

### 数据实体层

由于框架主要采用了Entity Framework作为ORM框架，这其中数据库实体映射必不可少，需要将数据库中每张表映射到类中，并且一张表一个类。这些实体类即能够作为数据库操作中的实体，还能够作为DTO(Data Transfer Object)，将这些实体类划分为独立的一层，能够方便对实体的管理，易于开发与维护。

### 业务逻辑层

在整个后端开发中，业务逻辑的处理是最复杂的，因为从技术角度来讲，很多技术都能够实现代码复用，即无需重复造轮子（重造轮子是重复创造一个已经存在的基本方法或者被其他人优化），而且只要会使用了就能够快速投入生产中，虽然技术可以从设计上实现代码重用，降低学习成本，但是不同的系统其业务逻辑通常是不可复制的，因此开发人员可以不关心具体数据仓储的实现技术，但是不得不关心具体业务逻辑的实现，既然业务逻辑无法避免又那么复杂，那么设计出合理的业务逻辑架构来加快开发效率就显得尤为重要。

本框架将业务逻辑独立一层，其核心继承关系如下图（图6-2）所示：

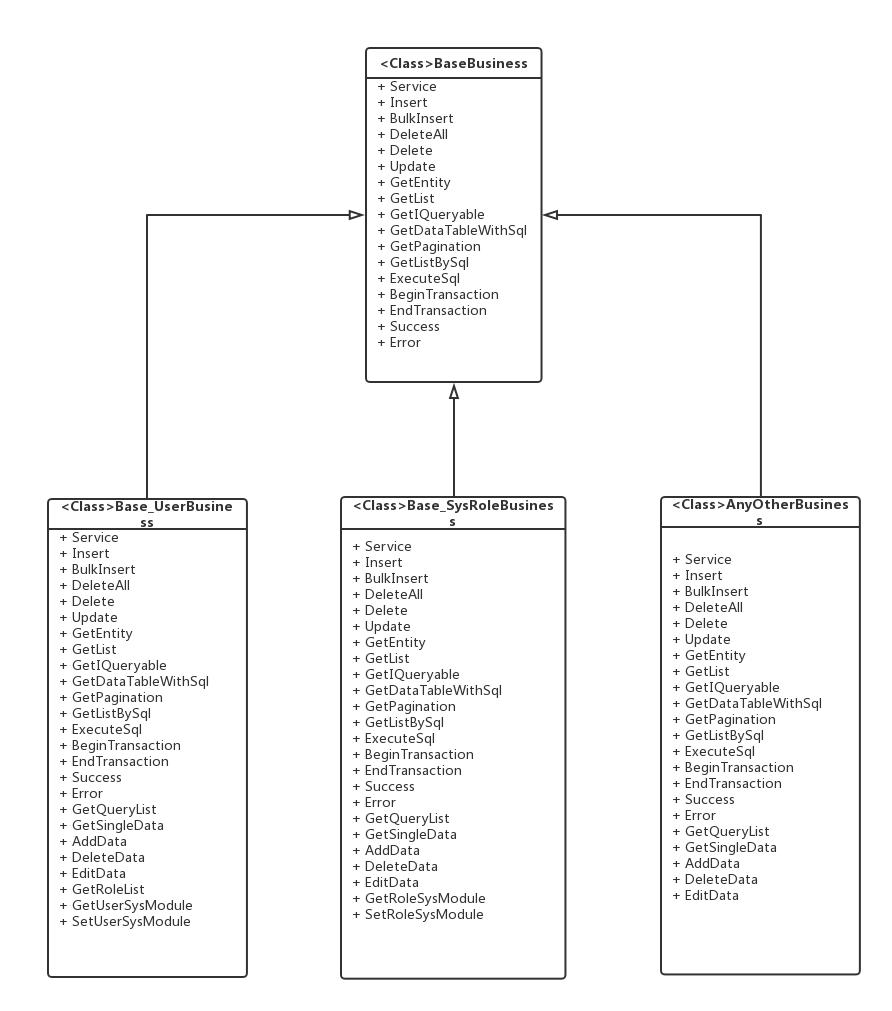


图6-2 业务逻层次图

如上图所示：首先定义了一个业务逻辑基类BaseBusiness<T>，该基类实现了所有调用数据仓储的方法，其余所有业务处理类都需要继承自基类BaseBusiness<T>，其中泛型T为数据库实体，一般默认操作但张表，但在实际运用中连表操作十分的普遍，因此在BaseBusiness中提供了Service属性，该属性为IRepository类型，因此可以操作泛型接口，传入不同的实体类型即可对应到所需表的操作，故而支持联表操作，方便开发人员操作数据库。基类还提供了通用业务逻辑返回数据，Success和Error分别对应成功请求返回和失败请求返回，统一数据格式，方便前后端开发对接。

### 应用展示层

这层也可以称之为应用层，其余的层表现形式都是类库，而这一层负责具体项目应用的实施，比如可以使用控制台程序、Windows服务程序、WinForm程序、WCF程序等等,在本项目中使用了ASP.NET MVC网站项目负责具体功能的实现。

AOP（Aspect Oriented Programming）使用：AOP，即面向切片编程，利用AOP能够将系统各个部分进行隔离，从而降低模块之间的耦合度，提高程序可用性，同时提高开发效率。本框架中主要使用了以下AOP：

管理员登录校验CheckLoginAttribute：在后台管理系统中，管理员只有登录后才能够进行相关操作，一般通过Session来记录管理员登录信息，最简单直接就是在每一个请求中都需要判断一遍管理员是否登录，这无疑将会导致很多的重复代码，此时，通过将登陆校验作为一个特性，只需要在需要登录的控制器或方法上添加该特性即可完成管理员登录校验，这样就能够减少大量的重复代码，加快开发效率。

签名校验CheckSignAttribute：当后端接口需要给外部系统调用时，若不对接口访问进行限制，那么就会面临恶意请求攻击的风险，轻则影响系统性能，重则导致系统瘫痪，数据被恶意串改，此时，保证接口的安全性就十分关键。保证接口的安全性，主要就是按照一定的签名算法，对请求者传入的参数进行签名校验，只有通过才能够正常访问，原始做法就是在每个请求中去进行签名校验，这同样会导致大量的重复代码，这时通过引入签名校验特性，只需要在需要签名校验的控制器或方法中加入特性即可，使用简单方便，开发效率也高。

应用层还在视图中使用了通用布局模板，并统一了代码规范，再集成了JQuery以及一些其它常用的JS类库，以Layui为主要前端UI框架，界面风格统一，开发效率高。

## 功能架构

### 全局配置

在01.Coldairarrow.Util中的GlobalSwitch类中，设置了各个参数，其中RunModel需要重点关注一下，若RunModel==RunModel.LocalTest，则系统会直接跳过登录，默认使用Admin超级管理员登录，其它参数请看注释。

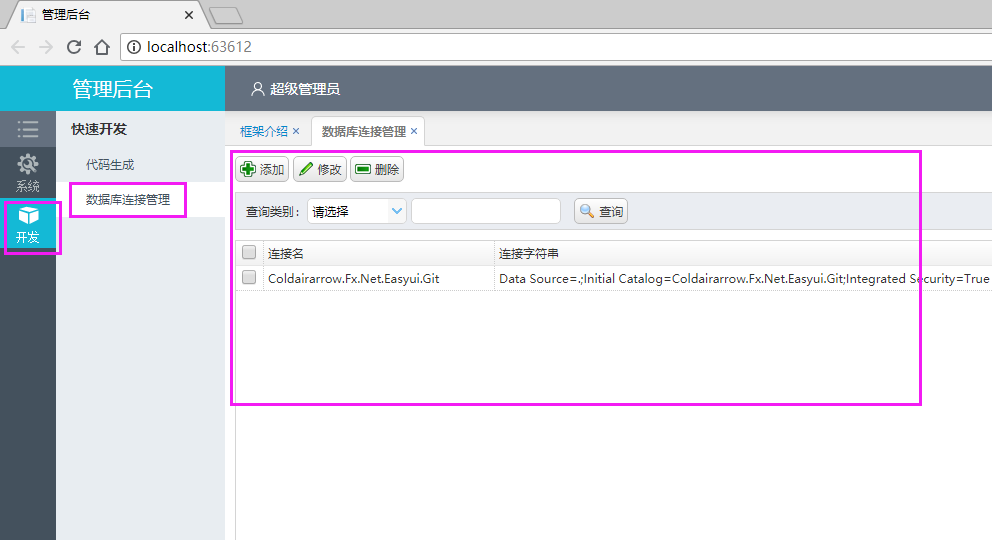
### 快速开发

使用此功能请确保GlobalSwitch.RunModel= RunModel.LocalTest

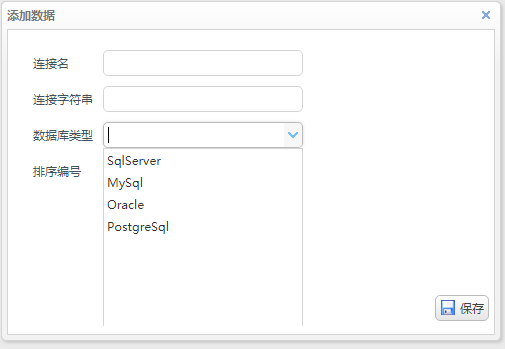
此功能为本框架的核心功能，能够自动生产完整的可运行代码，具体使用如下：

首选需要有数据库源，因为代码生成是根据数据库表来生成的。

菜单：开发=>快速开发=>数据库连接管理



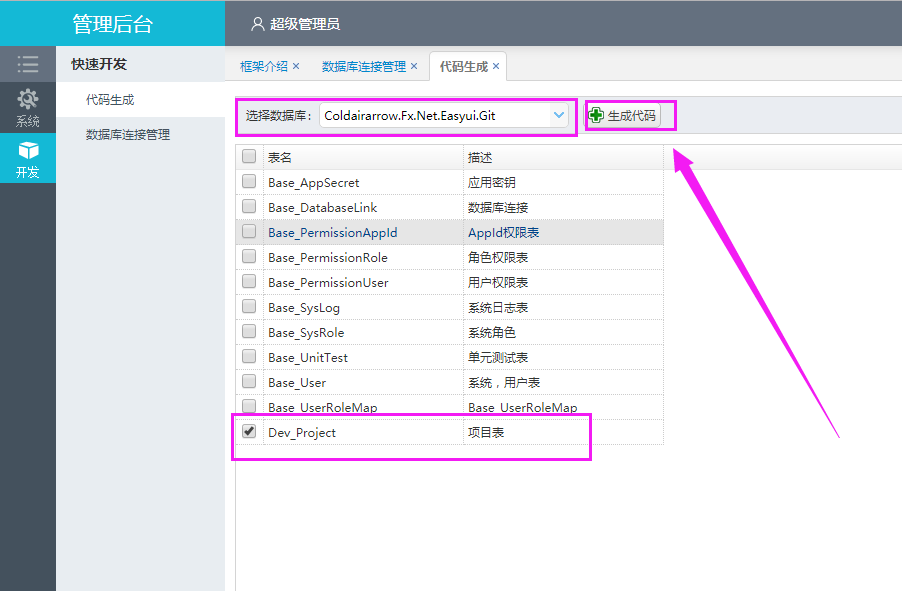
若列表中没有目标数据源，则添加数据库连接



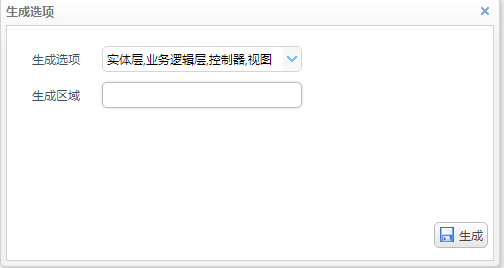
数据连接名、连接字符串、数据库类型（目前不支持Oracle，有空余时间再研究）即可。添加完成后即可看到连接字符串信息。

有了数据库连接之后，即可进行代码生成。

菜单：开发=>快速开发=>代码生成

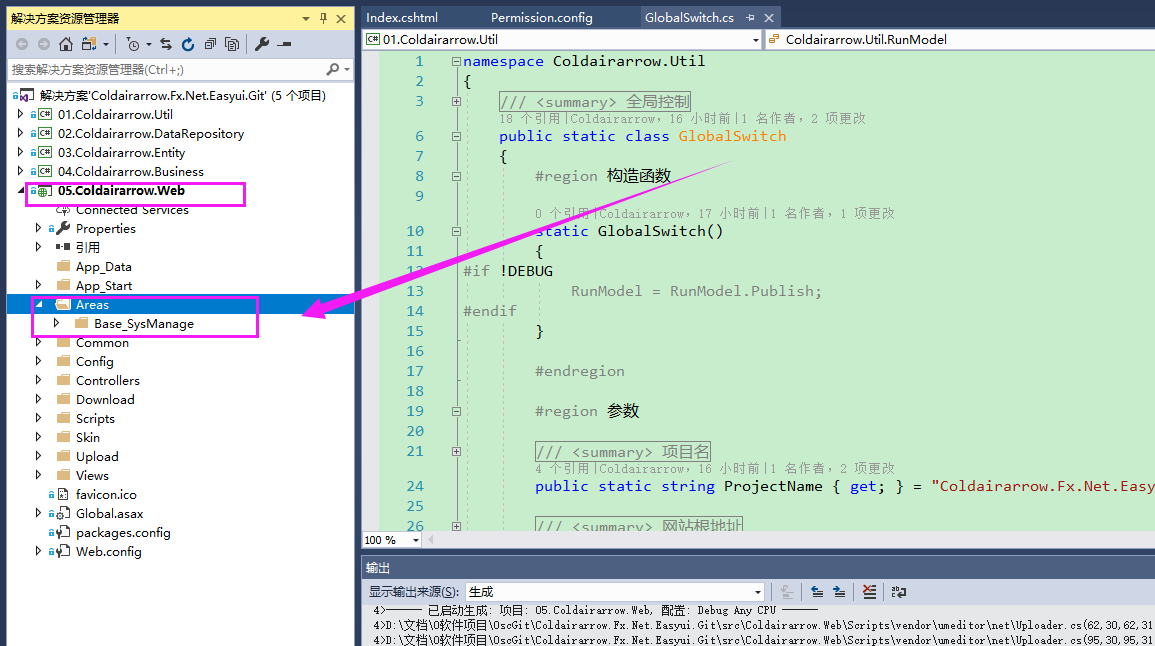


选择数据库，然后勾选需要生成代码的数据库表，点击生成代码会弹出生成选项（这里暂时只能勾选Dev\_Project，其余表全是系统基础表，不要勾选，否则会被覆盖，导致异常，请勾选自己的业务表进行生成）：

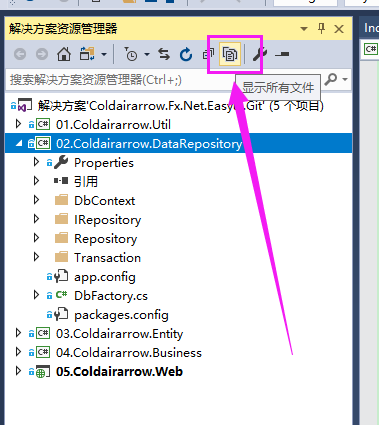


生成选项中可以选择需要生成的类型，可以默认生成实体层、业务逻辑层、控制器和视图。

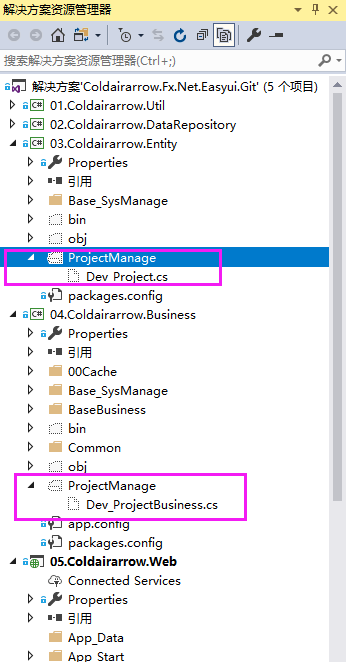
生成区域对应MVC中的Areas，请按需填入（必填）

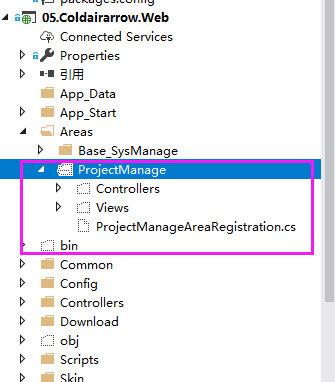


这里示例填写ProjectManage，点击生成按钮，即可完成代码生成。生成后的代码在项目解决方案中，将代码文件包括进入项目



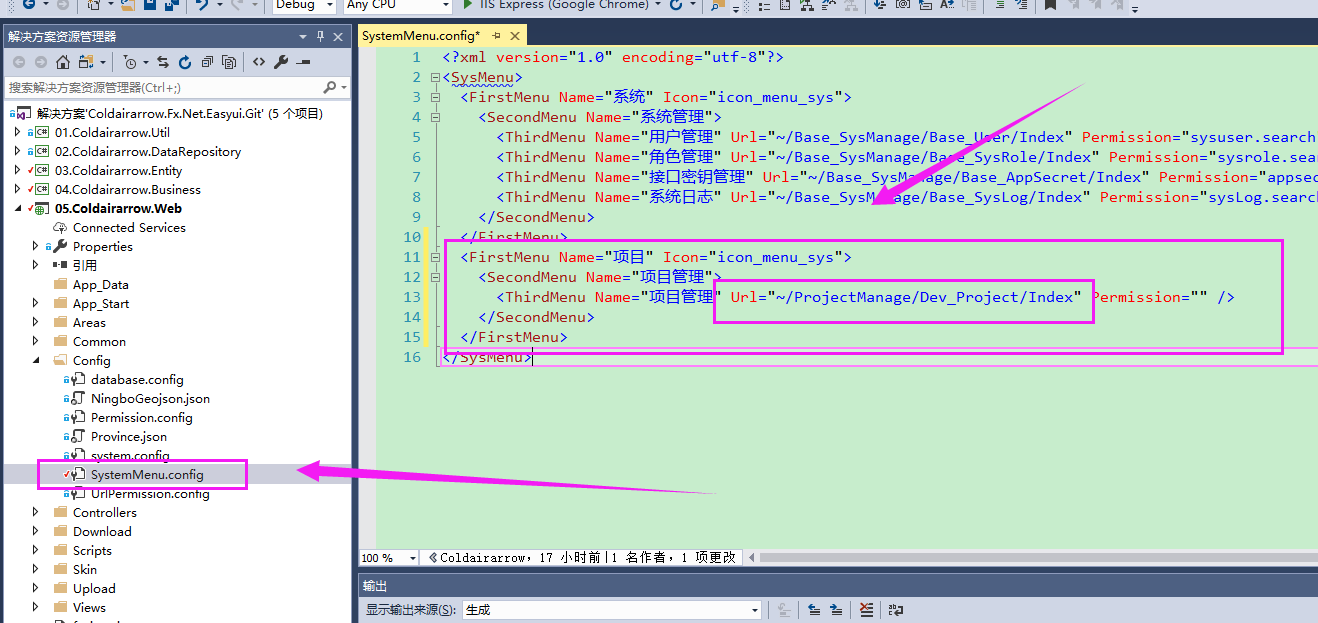
默认生成后的文件是隐藏的，需要点击显示所有文件按钮，即可看到生成后的新文件



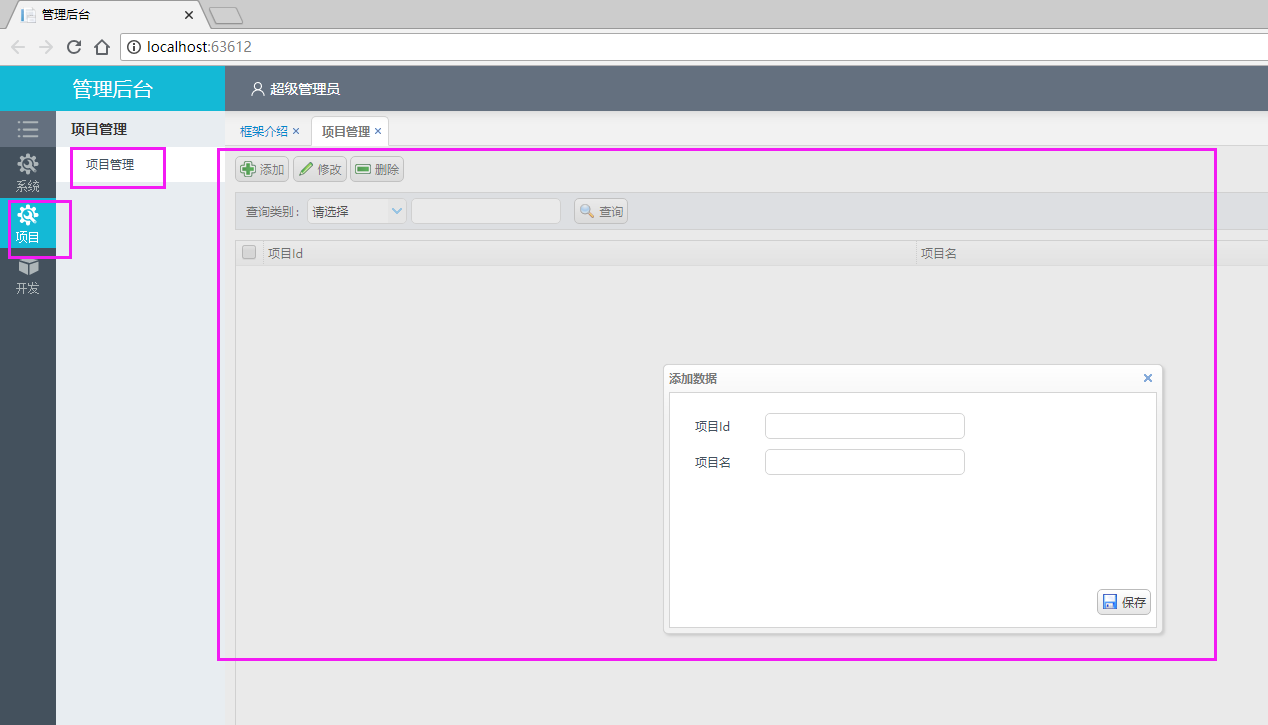


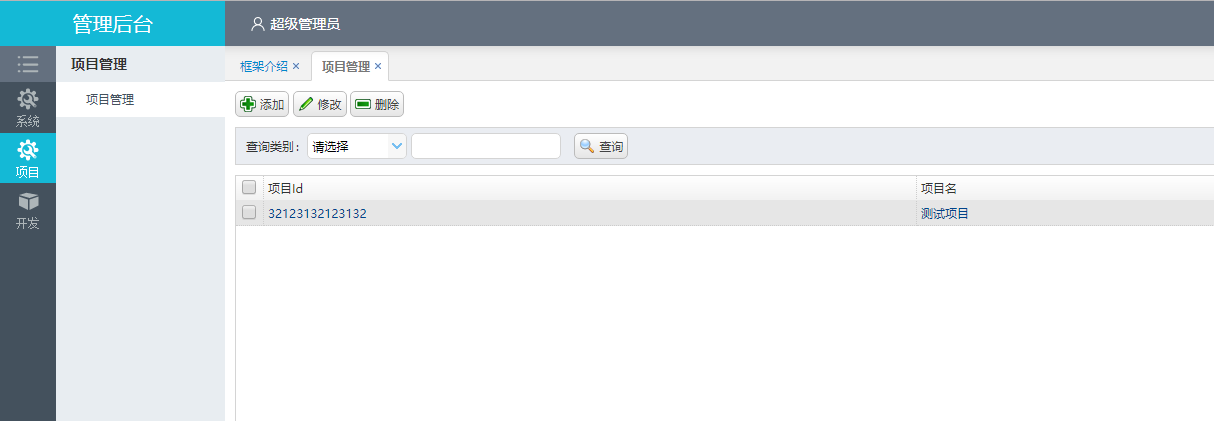
右键新的文件夹，包括在项目中

由于是新生成的代码，所以才配置新的菜单



如上图，在Web项目中的Config/SystemMenu.config中配置菜单，模仿原有菜单即可，其中Url是指页面的路径，Permission是指若需要访问此菜单需要的权限（对应权限配置），若没有此权限，则菜单也中不会显示此菜单，修改完成后重新编译生成（权限相关模块进行了缓存，重新生成会清缓存），重新运行即可看到新的项目菜单如下：

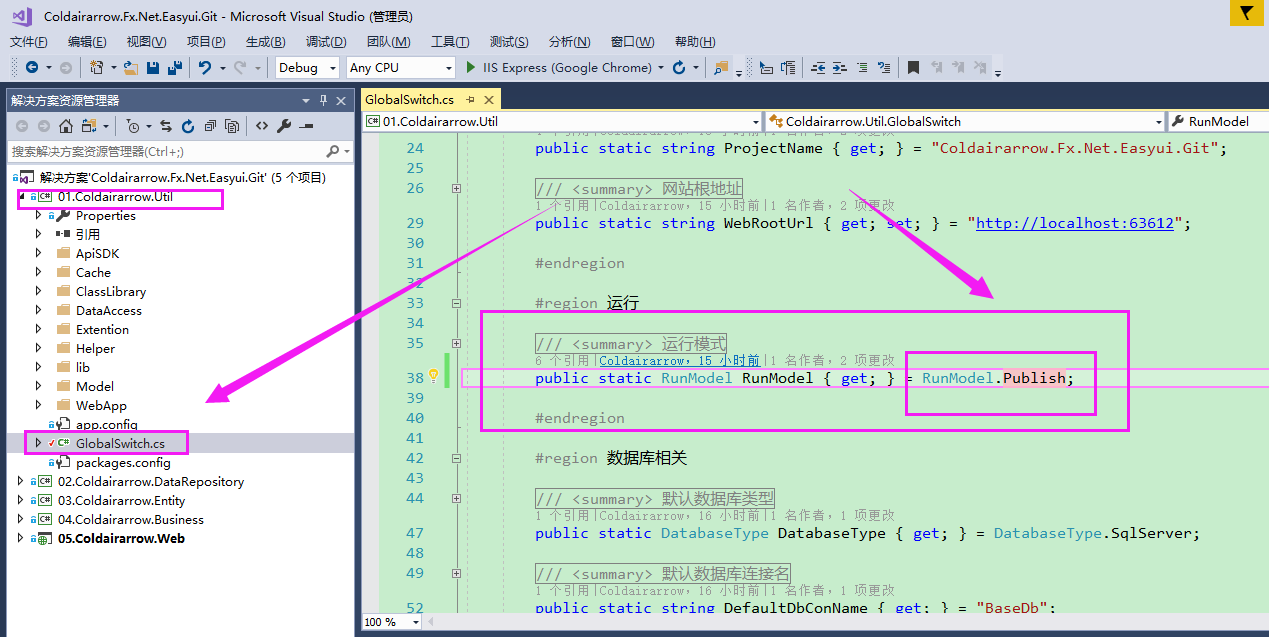




整个代码生成过程，无需编写代码即可完成一张表的CRUD，当然需要根据具体业务中进行相应的修改，本次示例中字段比较少，但是当一张表的字段很多时，那么此功能能够将开发效率提高几个档次。

### 管理员登录

若要使用登录功能，请将GlobalSwitch中的RunModel改为RunModel.Publish





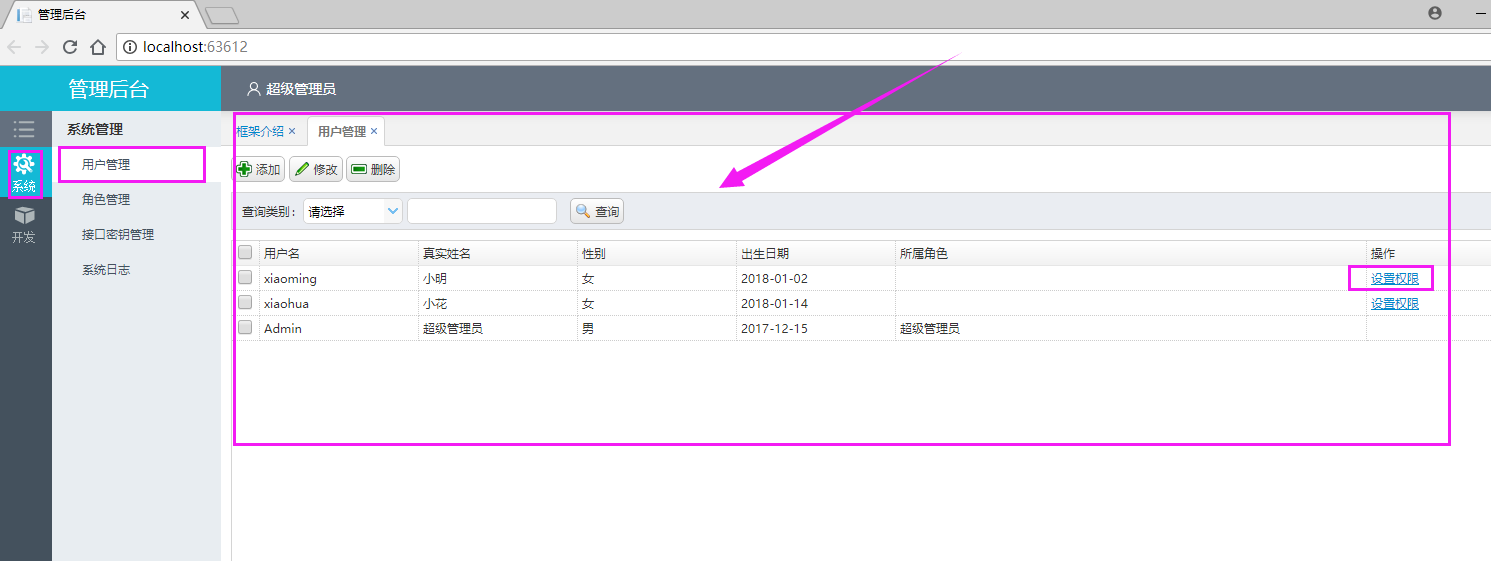
默认超级管理员账号为：Admin

密码为：123456

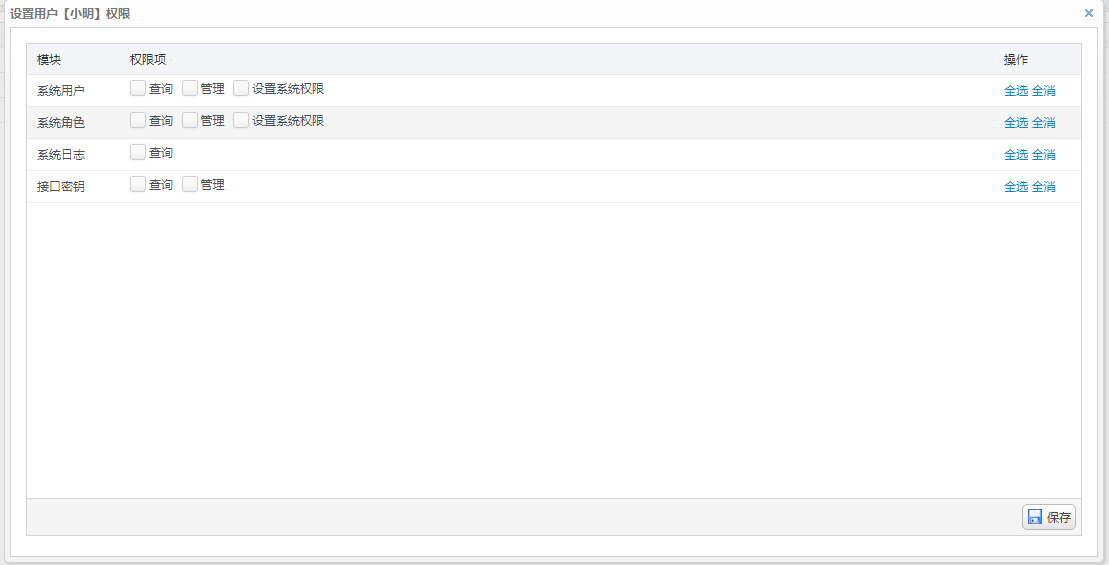
### 系统用户管理

管理系统登录的用户

菜单：系统=>系统管理=>用户管理，如下页面



点击右侧设置权限，可以设置用户权限，详情见<权限管理>模块

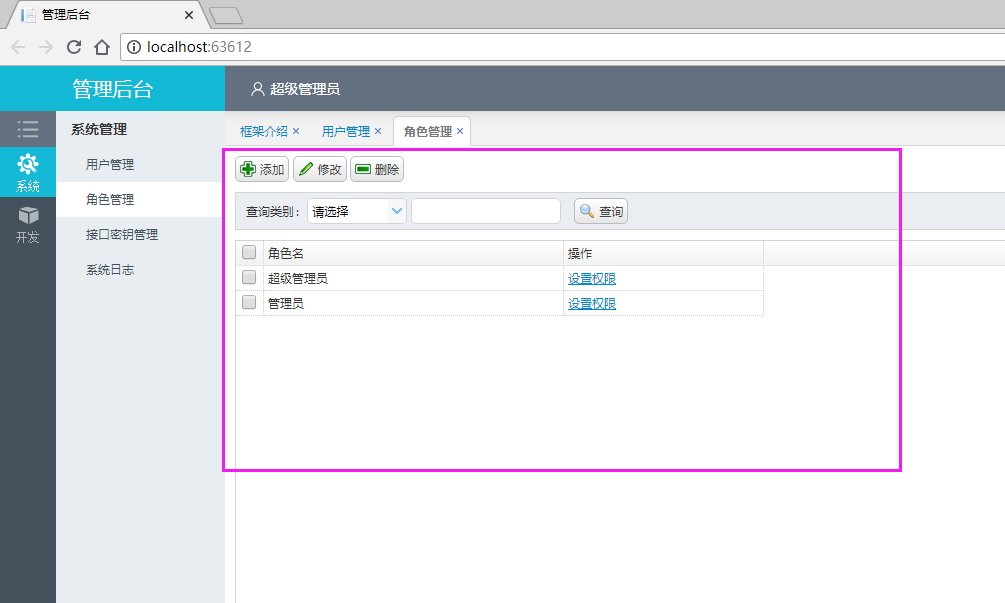


具体权限相关配置见权限管理模块

### 系统角色管理

管理系统角色，角色是权限的载体，合理分配角色有利于权限管理

菜单：系统=>系统管理=>角色管理



操作中可以设置角色的权限，详情见<权限管理>模块

### 权限管理

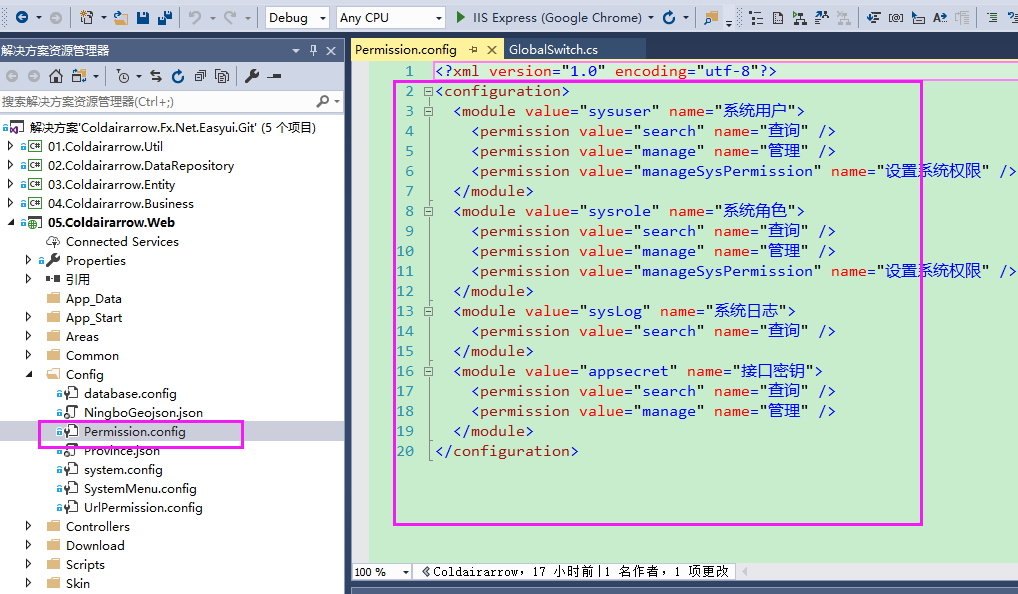
一般情况下，后台管理系统多少会涉及权限管理，因此本框架提供了一个灵活、高效、简洁的权限管理系统。

首先，权限分为两种，即操作权限和数据权限，其中操作权限报货系统用户权限和AppId权限，系统用户权限就是指操作用户具备哪些权限，而当对外提供Api接口时，为了保证接口的安全性（若不在意可忽略），通常会提供接口签名算法，其中AppId和AppSecret是必备的，通过对AppId设置权限，即可控制接口的权限。数据权限比较复杂，若采用纯SQL方式，那么会更加复杂，本框架全程采用EF作为ORM框架，通过对IQueryable<T>进行过滤，即可完成数据权限控制。

用户权限：若对每个用户都设置对应的权限，那么工作量无疑是十分巨大的，因此引入了角色的概念，角色是权限的集合载体，那么属于此角色的用户就继承了角色的权限，当然某些特殊用户需要拥有自己的不属于角色的特殊权限，因此最终用户拥有的权限就是自己的权限和所属角色权限的并集。

权限使用：

权限定义：

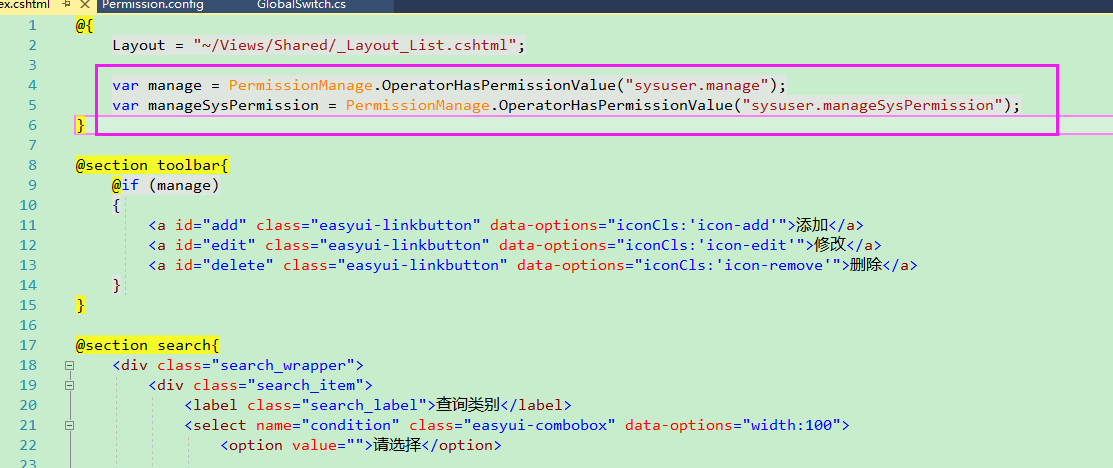


如上图，在Permission.config中定义了各个权限

权限配置：

在系统用户管理和系统角色管理中可以设置用户和角色的权限，把需要的权限勾选即可。

权限使用：



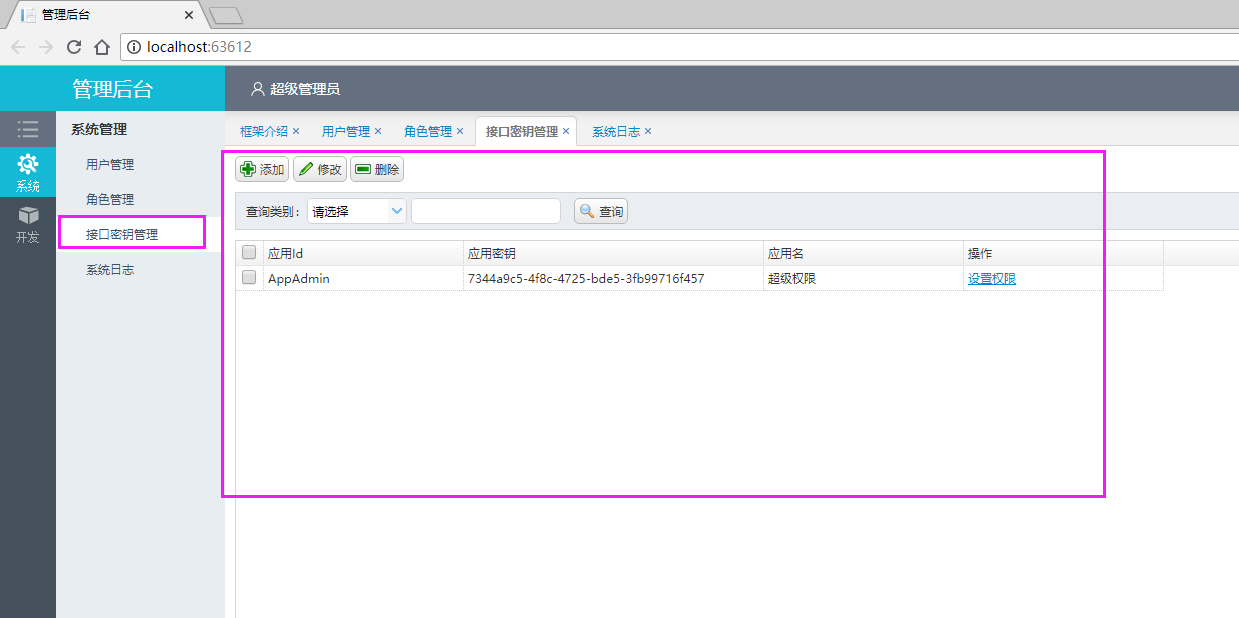
如上图所示，在需要控制权限的页面中，调用方法：PermissionManage.OperatorHasPermissionValue("sysuser.manage")

这个方法是判断操作者用户是否含有sysuser.manage权限值，其中sysuser是指Permission.config中定义的module的value属性，manage是指permission中的value属性，用.连接即是最终权限值。

更详细的使用方式，请参考源代码。

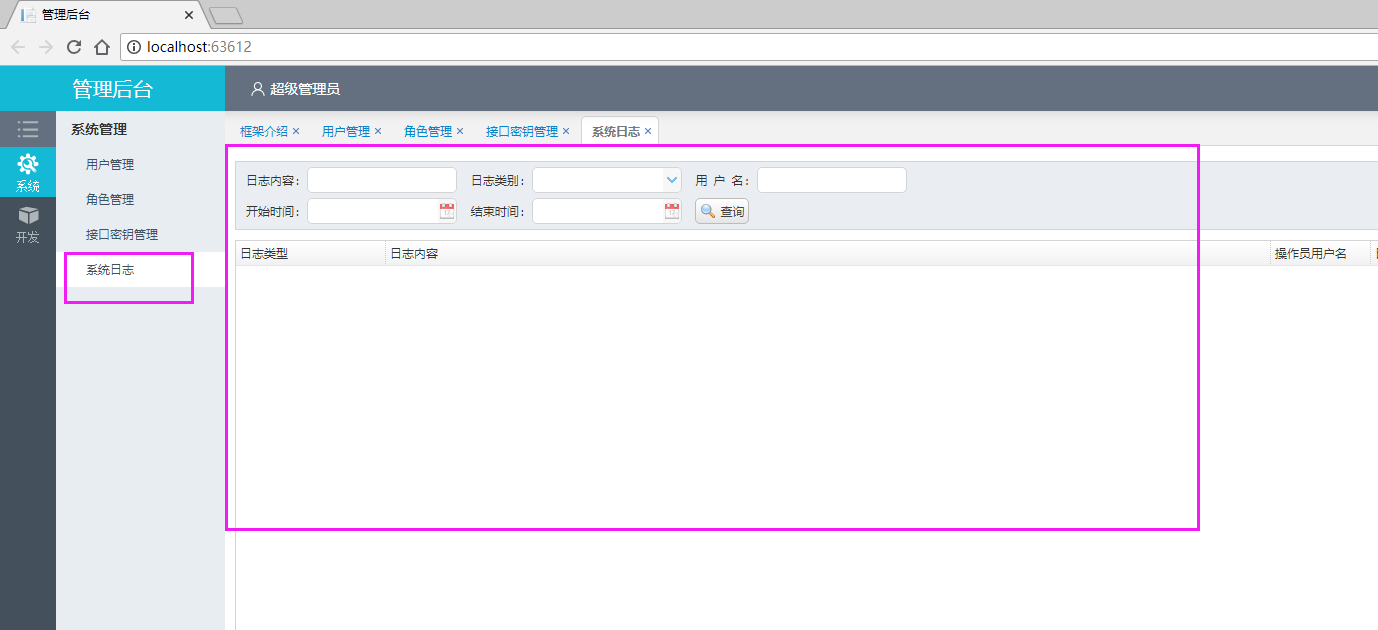
### 接口秘钥管理

菜单：系统=>系统管理=>接口秘钥管理

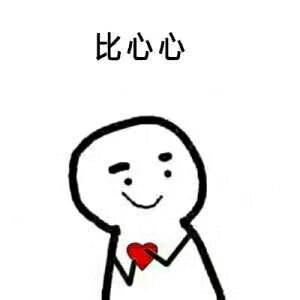


### 系统日志

菜单：系统=>系统管理=>系统日志



# 结语

欢迎使用本框架，若觉得不错，请比心

Github:<https://github.com/Coldairarrow>，请Statrt

博客园：<https://www.cnblogs.com/coldairarrow/>

QQ群：373144077

本人将会对这个快速开发框架不断完善与维护，希望能够帮助到各位

若遇到任何问题或需要技术支持，请联系我。

---------------------学习永无止境，技术永无上限，代码就是艺术-----------------------