技术实现方案

详细的功能说明参见:[E:\Workspace\vowei\spec\functional spec.docx](file:///E:\Workspace\vowei\spec\functional%20spec.docx)。

[技术实现方案 1](#_Toc278746147)

[系统的拓扑图 1](#_Toc278746148)

[采用软件 3](#_Toc278746149)

[各组件的实现方式 4](#_Toc278746150)

[数据库设计 4](#_Toc278746151)

[配置管理系统 4](#_Toc278746152)

[与配置管理系统的集成 4](#_Toc278746153)

[在Mercurial和SVN客户端实现检索任务 4](#_Toc278746154)

[全文搜索的实现方式 4](#_Toc278746155)

[身份验证 5](#_Toc278746156)

[用户组和权限管理组件 5](#_Toc278746157)

[内容管理组件 5](#_Toc278746158)

[日程安排 6](#_Toc278746159)

[生成OLAP报表 6](#_Toc278746160)

[定期处理提醒 6](#_Toc278746161)

[网站支持的客户端 7](#_Toc278746162)

[系统接口设计 7](#_Toc278746163)

[vowei.data 10](#_Toc278746164)

[vowei.Service 21](#_Toc278746165)

[查询数据 21](#_Toc278746166)

[编辑数据 25](#_Toc278746167)

[安全的实现 28](#_Toc278746168)

# 系统的拓扑图

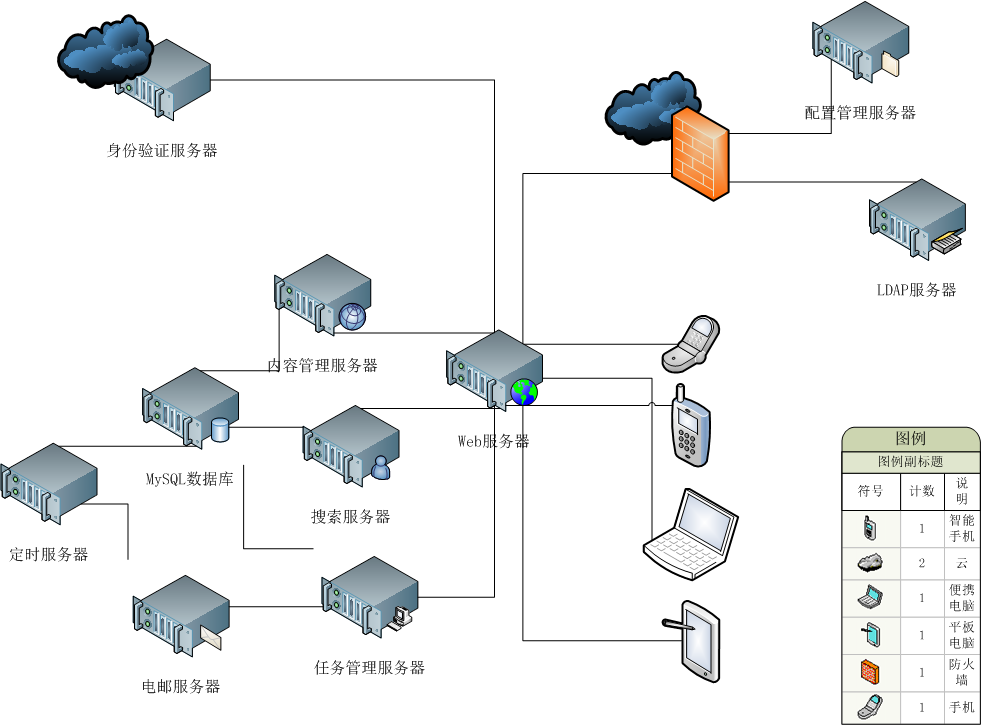
这个系统可以支持三种部署方案：

1. 公有部署，即整个系统放在Internet服务器上，多个团队或组织共享服务器，当然他们的项目数据是分开的。
2. 私有部署，即一个组织在自己的组织内部部署整个系统。
3. 混合部署，由于有些组织不希望自己的机密数据，例如项目产出物，功能说明以及联系人信息存放在公有的系统里，然而他们希望通过这个系统整合他们现存的系统，也可以支持（当然难度非常的大）。

系统应该由如下几个部分组成：

1. 用户组和权限管理组件
   1. 这个组件必须在我们的系统里实现，否则无法有效控制权限了。
   2. 但是最佳情况下，我们只需要实现权限的定义就可以了，因为我们可以复用现有的LDAP, Windows域活动目录提供的现有功能实现用户组管理，和权限管理。
2. 需求管理组件（同时包含任务管理系统）
   1. 这个组件是我们系统的核心组件。
3. 用户身份验证组件
   1. 这个系统可以委托给openid, 或者Windows Live, Google Talk实现。
   2. 当然我们也需要实现一个自己的用户管理组件。
4. 配置管理组件
   1. 可以支持SVN,、Mercurial、考虑如何支持SharePoint Server。
5. 内容管理组件
   1. 即系统中所有创建的内容都属于内容管理的范畴，是否可以用orchard实现？
6. 生成OLAP报表
7. 全文检索功能。
8. 定期处理提醒、执行数据仓库处理的定时处理组件。
9. 发送邮件组件

整个系统的拓扑结构如下图，



## 采用软件

为了节省最终的成本，除了Web服务器、内容管理服务器和任务管理服务器（其实也就是Web中间层）使用ASP.NET实现，需要使用到Windows Server服务器以外，其他的服务都应该可以运行在 Linux机器上，这也就意味着我们采用的软件不能使用只能运行在Windows平台下的软件。

# 各组件的实现方式

## 数据库设计

我们采用ADO .NET Entity Framework 4 CTP来设计数据库，因为它的特性就是code first，而且我已经试过了，效果不错。采用ADO.NET EF 4 CTP设计好数据库结构，在SQL Server CE上生成表，等最后产品完成之前，再将SQL语句导出并在MySql里生成对应的数据。

## 配置管理系统

### 与配置管理系统的集成

当前的配置管理系统只支持Mercurial和SVN系统，这是因为Mercurial和SVN系统都自带有Web界面，支持SSH访问协议，以及它们都支持插件模式。即可以在指定的事件，例如commit, changegroup等事件发生的时候调用插件实现自定义操作。

Mercurial和SVN都已经有现成的插件，支持bugzilla的插件，我们完全可以复用这个插件，只要实现类似bugzilla接收从Mercurial或者SVN过来的commit消息就可以了。

详情请参考下面链接里的（Bundled hooks\bugzilla这一节）：

<http://hgbook.red-bean.com/read/handling-repository-events-with-hooks.html>

另外，Mercurial也支持TFS的shelveset的概念，请参考它的Mercurial Queue的概念：

<http://hgbook.red-bean.com/read/managing-change-with-mercurial-queues.html>

### 在Mercurial和SVN客户端实现检索任务

SVN和Mercurial的客户端，TortoiseSVN和TortoiseHg都是通过调用原始自带的客户端向服务器发送命令的方式实现的。其中，Mercurial是采用Python实现的，因此我们有可能使用IronPython，甚至是Python实现自己的客户端—添加一个对话框，以便用户选择他自己的工作任务。

## 全文搜索的实现方式

使用Lucene.NET实现，参考链接：

1. <http://lucene.apache.org/java/docs/index.html>
2. <http://lucene.apache.org/lucene.net/>

## 身份验证

在第一期里，实现由系统直接管理用户权限的功能。

在之后的版本中，我们需要实现代理用户身份验证。我们可以采用AppFabric Access Control List来实现代理身份验证，或者直接调用Google Talk, Yahoo ID, Windows Live ID的身份验证Web服务来实现。

请参考以下这些文档：

AppFabric Access Control: <http://msdn.microsoft.com/en-us/library/ee732536.aspx>

Windows Live ID SDK: <http://msdn.microsoft.com/en-us/library/bb404787.aspx>

Windows Live Messenger Connect: <http://msdn.microsoft.com/en-us/library/ff752216.aspx>

Getting Start with Messenger Connect: <http://msdn.microsoft.com/en-us/library/ff752317.aspx>

## 用户组和权限管理组件

用户组和Windows用户组的概念类似，用户组不仅可以包括用户，也可以包括其他用户组。另外，权限的设计也类似于.NET代码安全的权限设计，既权限分为单个权限和权限集，权限集可以包含其它权限集或者权限。对用户组分配权限是依据权限集分配的，默认有如下几个权限集：

1. 完全控制权限
2. 只读权限
3. 读写权限
4. 访问者权限

在第一期中，可以做的简单些：

1. 用户组只能包含用户，不能包含其它用户组。
2. 硬编码好几个权限集，并将权限集的权限列表硬编码。

另外，如果用户希望使用自己的域活动目录管理服务器来执行权限验证，即权限验证也是放在用户的服务器上，这个要求以后考虑如何实现，可以参考Windows AppFabric的实现方式：

<http://msdn.microsoft.com/en-us/library/ee732536.aspx>

## 内容管理组件

可配置项需要执行版本控制，为了最大限度的集成现有的客户端软件，系统管理的所有元素的列表，元素的详细信息都可以通过RSS和Atom的方式浏览。我希望各种客户端链接到网站的方式都是一样的，比如说，无论是多种浏览器，智能手机，或者Outlook、Excel等客户端软件，都是通过同一个URL链接到网站收取内容。

1. 在第一期中，对于各个客户端暂时采用不同的客户端。
2. 在以后的版本中，再实现统一URL的方式。

在.NET中创建RSS和Atom的方法：

<http://msdn.microsoft.com/en-us/library/system.servicemodel.syndication.syndicationfeed.aspx>

另外，为了能够让用户使用Excel链接到网站收取数据，执行自定义的统计分析，网站需要支持REST和OData协议。请参看下面的链接：

**ODATA（ODATA实际上基于REST协议的）**

<http://www.odata.org/developers/odata-sdk>

<http://msdn.microsoft.com/en-us/library/cc668792.aspx>

**RESTful**

RESTful Services with ASP.NET MVC: <http://msdn.microsoft.com/en-us/magazine/dd943053.aspx>。

Creating RESTful Web Services with WCF: <http://www.developer.com/net/article.php/3695436>

## 日程安排

为了方便用户日程安排在各个软件之间的传输，所有的日程安排都采用iCalendar实现，关于iCalendar的资料，参见：

<http://en.wikipedia.org/wiki/ICalendar>

在.NET里处理iCalendar格式的程序请参看：

<http://sourceforge.net/projects/dday-ical/>

## 生成OLAP报表

我们不会按需生成OLAP报表，而是每天定期在机器不忙的时候，后台执行统计过程，将数据保存到另外一份报表里，通过这个设计，应该可以提升整体网站的性能。

## 定期处理提醒

实现定期处理，使用Linux平台下的cron程序，这个程序定时调用指定的一个脚本或者程序。因此只需要编写个脚本来定期执行OLAP统计，发送邮件提醒等工作。

## 网站支持的客户端

* IE 6, 9
* Firefox 3
* Google Chrome
* Outlook
* Excel
* Open office系列
* iPhone
* 黑莓
* Windows Phone 7

# 系统接口设计

下面是系统的层次:：

1. Vowei.web — 这是整个系统的前台界面，由于我们的系统是以提供服务为主要目的，因此这个界面并不是一个很重要的组件。
2. Vowei.service — 使用WCF Data Service向客户提供数据服务，并且处理工作流相关的操作。Vowei.web，跟其它客户程序一样，调用该服务提供的接口获取数据。
3. Vowei.data — 整个系统的数据层，负责解决系统与各个数据库之间的操作。

Log

Security

Vowei.web

vowei.service

Vowei.data

在系统各个组件之间通过传递接口来实现组件交互，在后续的版本里，获取某个接口的具体实现由Ioc容器完成，当前版本还是采用传统的new的方式实现。为了方便组件之间共享数据和类型定义，vowei.core项目里定义了系统中通用的接口和类型实例。下面是所有的接口定义：

|  |
| --- |
| /// <summary>     /// 表示数据库里面的结构     /// </summary>     public interface IVoweiContext     {         IQueryable<Ticket> Tickets { get; set; }           IQueryable<Task> Tasks { get; set; }           IQueryable<MeetingRequest> MeetingRequests { get; set; }           IQueryable<Category> Categories { get; set; }           IQueryable<Bug> Bugs { get; set; }           IQueryable<Risk> Risks { get; set; }           IQueryable<CategoryColor> CategoryColors { get; set; }           IQueryable<TaskCategory> TaskCategories { get; set; }           IQueryable<Project> Projects { get; set; }           IQueryable<Milestone> Milestones { get; set; }           IQueryable<Requirement> Requirements { get; set; }           IQueryable<Comment> Comments { get; set; }     }       /// <summary>     /// 所有的表都需要支持这个接口     /// </summary>     public interface ITable     {         /// <summary>         /// 主键         /// </summary>         int ID { get; set; }           string Title { get; set; }     }       public interface IRecursive     {         /// <summary>         /// 是否是重复任务？         /// </summary>         bool IsRecursive { get; set; }           /// <summary>         /// 重复任务的终止时间         /// </summary>         DateTime? RecursiveEndDate { get; set; }           /// <summary>         /// 重复任务的设置         /// </summary>         RecurrenceSettings Settings { get; set; }     }       /// <summary>     /// 标记一个类型是否可以被评论     /// </summary>     public interface ITagable     {         /// <summary>         /// 获取Tag列表         /// </summary>         string[] Tags { get; set; }           /// <summary>         /// 添加一个Tag，如果添加的Tag是一个重复的Tag，默认什么事情都不做         /// </summary>         /// <param name="tag"></param>         void AddTag(string tag);     }       public interface ISupportDefaultProperties : ITable     {         DateTime? ClosedDate { get; set; }         DateTime OpenDate { get; set; }         string Reporter { get; set; }     }       public interface IMilestone : ISupportDefaultProperties     {         DateTime? DueDate { get; set; }           // TODO: 想办法看看能不能把下面的属性加进来         // IList<IMilestone> Timeline { get; }     }       public interface ITicket : ICommentable, ISupportDefaultProperties, ITable     {         string Description { get; set; }           string AssignedTo { get; set; }           IMilestone Milestone { get; set; }     }       public interface IContainer<T>     {         List<T> Children { get; }     }          /// <summary>     /// 在数据库中表示用户     /// </summary>     public interface IUser     {         /// <summary>         /// 用户的Email         /// </summary>         string Email { get; set; }                  /// <summary>         /// 用户的显示名称         /// </summary          string DisplayName { get; set; }     } |

## vowei.data

由于系统是可以同时面对大量的项目，因此在数据库中，为了便于达到同步性，对于每一个项目，我们都为其创建一个独立的数据库结构。比如说，在程序的项目模板里，可能有下面这些表：tblProject, tblMilestone, tblTask等表。当用户要创建一个新的项目时，系统以模板里已有的几个表结构为模板，为新项目复制这几个数据表，也就是说，新项目的表里，可能是这样的：tblProject\_1, tbl\_Milestone\_1, tblTask\_1。而这些事情需要在vowei.data这一层就处理好了，其他的层次不应该处理这种问题。

在数据层里，所有的表都应该被映射成类型，所有的查询返回的结果，都应该是IQueryable<T>接口的实例。下面是数据层各个表的定义（使用POCO定义，通过Entity Framework 4.0 CTP的code first方式，通过类型在数据库中定义好表，然后从SQL CE或者SQL Server提取表的SQL定义）：

|  |
| --- |
| /// <summary>     /// 表示一个里程碑     /// </summary>     public class Milestone : IMilestone     {         public Milestone()         {             Timeline = new List<Milestone>();             OpenDate = DateTime.Now;         }           /// <summary>         /// 主键         /// </summary>         public int ID { get; set;  }           /// <summary>         /// 里程碑的名称         /// </summary>         [Required]         [Display(Name = "名称")]         public string Title { get; set; }           /// <summary>         /// 计划中里程碑结束的时间         /// </summary>         [Display(Name = "结束日期")]         public DateTime? DueDate { get; set; }           /// <summary>         /// 实际上里程碑结束的时间         /// </summary>         /// <remarks>         /// 这个应该是根据里程碑里任务完成的数目而实现的         /// </remarks>         [Display(Name = "完成日期")]         public DateTime? ClosedDate { get; set; }           /// <summary>         /// 项目的具体日程表         /// </summary>         public List<Milestone> Timeline { get; private set; }           #region ISupportDefaultProperties Members         [Display(Name = "创建日期")]         public DateTime OpenDate         {             get;             set;         }           [Display(Name = "创建人")]         public string Reporter         {             get;             set;         }           #endregion     }       /// <summary>     /// 表示一个项目     /// </summary>     public class Project : Milestone     {         public class SupportedScm         {             public const string Mercurial = "Mercurial";             public const string Svn = "Svn";             public const string Tfs = "Tfs";         }           public Project()         {             StartDate = DateTime.Now;         }           /// <summary>         /// 项目开始的日期         /// </summary>         [Required]         [Display(Name = "开始日期")]         public DateTime StartDate { get; set; }           /// <summary>         /// 负责整个项目的人员，也就是在项目管理中Accountable的那个人。         /// </summary>         [Required]         [Display(Name = "负责人")]         public string Owner { get; set; }           /// <summary>         /// 获取配置管理系统的路径         /// </summary>         /// <remarks>         /// 这个路径可以是在因特网上,大家都可以访问到的,也可以是在用户企业内部的路径         /// </remarks>         public string ScmUrl { get; set; }           /// <summary>         /// 设置和获取配置管理系统的类型         /// </summary>         /// <remarks>         /// 关于网站支持的配置系统列表，请参考:<seealso cref="Project.SupportedScm"/>          /// </remarks>         public string ScmType { get; set; }     }      /// <summary>     /// 系统中所有的任务相关，与人相关的行为，都是一个Ticket     ///      /// 例如，一个任务是一个Ticket, 一个会议邀请是一个Ticket，只要这个事情需要有人参加，     /// 那么，就应该发一个Ticket给这个人。     /// </summary>     public class Ticket : ITicket     {         public Ticket()         {             OpenDate = DateTime.Now;             ClosedDate = null;             Comments = new List<Comment>();         }           /// <summary>         /// 主键         /// </summary>         public int ID { get; set; }           /// <summary>         /// Ticket的标题         /// </summary>         [Required]         [Display(Name="标题")]         public string Title { get; set; }           /// <summary>         /// Ticket的详细说明         /// </summary>         [Display(Name="详细说明")]         public string Description { get; set; }           /// <summary>         /// Ticket创建的日期         /// </summary>         [Required]         [Display(Name="创建日期")]         public DateTime OpenDate { get; set; }           /// <summary>         /// Ticket关闭的日期         /// </summary>         /// <remarks>         /// 这个是一个Nullable类型，如果ClosedDate为空，说明这个Ticket还没有关闭         /// 或者就是根本没有关闭日期的，例如一个消防系统，是不应该有关闭日期的         /// </remarks>         [Display(Name="关闭日期")]         public DateTime? ClosedDate { get; set; }           /// <summary>         /// 打开Ticket的用户，必须是用户的email         /// </summary>         /// <remarks>         /// 之所以使用email来做Ticket与用户系统的外键联系，主要是考虑到：         ///          /// 1、有可能开这个Ticket的用户没有在系统中注册，例如一个使用开源软件的用户，         /// 发现了一个bug，他可以开一个bug，但我们并不要求他在系统中有注册用户。         /// 2、这样可以整合其他的系统，例如可以跟MSN、facebook等系统的用户联系         /// MSN Messenger自己就是用email做用户名，对于facebook, qq之类的非email格式的         /// 用户名，我们可以采取[user@siteurl]的形式实现，例如1234567@qq.com。         /// </remarks>         [Required]         [Display(Name="报告者")]         public string Reporter { get; set; }           /// <summary>         /// Ticket指派给谁         /// </summary>         [Required]         [Display(Name="指派给")]         public string AssignedTo { get; set; }           /// <summary>         /// 该Ticket需要在哪一个Milestone完成?         /// </summary>         /// <remarks>         /// 如果值为null,则说明这个Ticket不属于任何一个Milestone的关键任务         /// </remarks>         public virtual IMilestone Milestone { get; set; }           public virtual List<Comment> Comments         {             get;             set;         }     }          /// <summary>     /// 表示一个可以包含其它Ticket的Ticket     /// </summary>     public class ContainerTicket : Ticket, IContainer<ITicket>     {         public ContainerTicket()         {             // Children = new List<Ticket>();         }           public virtual List<ITicket> Children { get; private set; }     }       /// <summary>     /// 表示一个客户的需求     /// </summary>     public class Requirement : ContainerTicket     {         /// <summary>         /// 实现这个客户需求能够获得的商业价值         /// </summary>         [Display(Name="商业价值")]         public float BusinessValue { get; set; }     }       /// <summary>     /// 代表一个日常工作任务     /// </summary>     /// <remarks>     /// 在Outlook里面的Calendar, FollowUp都可以使用Task模拟     ///      /// 日常工作任务的状态是有版本控制的，因为用户可以设置一个基线（baseline）     /// 这样后面实际工作时，由负责任务的人员更新任务的开始和结束执行日期     /// </remarks>     public class Task : Ticket, IRecursive     {         public Task()         {             StartDate = DateTime.Now;         }           /// <summary>         /// 任务开始执行的日期         /// </summary>         [Required]         [Display(Name="开始日期")]         public DateTime StartDate { get; set; }           /// <summary>         /// 任务完成得日期         /// </summary>         [Display(Name="计划结束日期")]         public DateTime? DueDate { get; set; }           /// <summary>         /// 是否是重复任务？         /// </summary>         /// <remarks>         /// 当用户创建一个重复任务时，系统会在数据库里创建根据重复设置创建重复备份         /// 之所以这样做，是因为用户在后期有可能会临时更改重复任务的某一次或几次任务的发生日期         /// </remarks>         [Display(Name="重复任务")]         public bool IsRecursive { get; set;  }           /// <summary>         /// 重复任务的终止时间         /// </summary>         [Display(Name="重复终止时间")]         public DateTime? RecursiveEndDate { get; set; }           /// <summary>         /// 任务的重要程度，使用用户自己熟悉的种类方式定义，会最终映射到时间管理的优先级         /// </summary>         [Display(Name="重要程度")]         public Category Category { get; set;  }           /// <summary>         /// 任务当前完成的状态         /// </summary>         [Display(Name = "进度")]         public float Progress { get; set; }           #region IRecursive Members             public RecurrenceSettings Settings         {             get;             set;         }           #endregion     }       /// <summary>     /// 代表一个软件缺陷，需要人去修复     /// </summary>     public class Bug : Ticket     {         [Display(Name = "重现步骤")]         public string ReproStep { get; set; }     }       /// <summary>     /// 代表一个会议邀请，对于Outlook里面的apointment，我们使用Task模拟     /// </summary>     // [IgnoreProperties("Attendees")]     public class MeetingRequest : Ticket, IRecursive     {         private List<string> \_Attendees = new List<string>();           /// <summary>         /// 与会者的邮件地址         /// </summary>         [Display(Name = "与会者")]         public List<string> Attendees { get { return \_Attendees; } }           /// <summary>         /// 会议记录         /// </summary>         [Display(Name="会议记录")]         public string MeetingNote { get; set; }           /// <summary>         /// 会议主持人         /// </summary>         [Display(Name="主持人")]         public string Host { get { return AssignedTo; } set { AssignedTo = value; } }           #region IRecursive Members         public bool IsRecursive         {             get;             set;         }           public DateTime? RecursiveEndDate         {             get;             set;         }         #endregion           #region IRecursive Members             public RecurrenceSettings Settings         {             get;             set;         }           #endregion     }       /// <summary>     /// 代表项目管理里面的风险管理     /// </summary>     public class Risk : Task     {         /// <summary>         /// 风险发生的可能性         /// </summary>         [Required]         [Display(Name="可能性")]         public int Possibility { get; set; }           /// <summary>         /// 风险发生的严重后果         /// </summary>         [Required]         [Display(Name="严重程度")]         public int Impact { get; set; }           // TODO: 考虑明白应急计划、预防措施在系统中的表现形式     }       /// <summary>     /// 代表时间管理里面的任务分类     /// </summary>     public class TaskCategory : ITable     {         public const int ImportantUrgent = 1;         public const int ImportantYetNotUrgent = 2;         public const int UrgentYetNotImportant = 3;         public const int NotImportantNotUrgent = 4;           public int ID { get; set; }           [Required]         public string Title { get; set; }     }       /// <summary>     /// 用户可以将自己的工作任务类型与<seealso cref="TaskCategory"/>对应起来     /// </summary>     /// <remarks>     /// 例如一个用户可以创建这样一个Category:老板的事情，然后将TaskCategory指定为ImportantUrgent，     /// 因为他认为老板的事情永远都是紧急且重要的。     ///      /// 之所以这样做，是为了帮助用户在接到新任务的时候，能够快速地明确自己的工作优先级     /// </remarks>     public class Category : ITable     {         /// <summary>         /// 主键         /// </summary>         public int ID { get; set; }               /// <summary>         /// 用户自定义的工作任务类型         /// </summary>         [Display(Name="种类名称")]         public string Title { get; set; }           /// <summary>         /// 映射到时间管理中的任务优先级。         /// </summary>         [Display(Name="优先级")]         public TaskCategory TaskCategory { get; set; }     } } |

备注：由于在前期实现过程中，我们并不考虑很大的用户并发性，因此系统实现采用比较简单的方式，先只实现所有项目共享同一套表（即不急于表结构模板为每个项目创建独立表的方式）。

## vowei.Service

系统的服务层，可以接受JSON和XML RPC以及SOAP格式的输入数据，并且可以根据客户端的不同，响应不同的数据。比如说，如果客户端是一个JavaScript程序，那么服务层可以接受JSON输入数据，并且返回JSON格式的结果数据。

### 查询数据

下面是查询方面的类型定义，

|  |
| --- |
| public static class DataLayer     {         /// <summary>         /// 根据ID从数据库中查找某个类型的实例         /// </summary>         /// <typeparam name="T">需要查询的数据类型，比如任务、项目等</typeparam>         /// <param name="id">数据的主键</param>         /// <returns>数据的实例</returns>         public static T Find<T>(int id) where T : class         {             var table = FindTableByType<T>();               return table.Find(id);         }           public static DbSet<T> FindTableByType<T>() where T : class         {             if (!VoweiContext.TableMap.ContainsKey(typeof(T)))                 throw new InvalidOperationException(string.Format("类型 {0} 并没有相关的表对应!", typeof(T)));               var propertyInfo = typeof(VoweiContext).GetProperty(VoweiContext.TableMap[typeof(T)]);             Debug.Assert(propertyInfo != null);             var propertyValue = propertyInfo.GetValue(VoweiContext.Instance, EditingHelpers.EmptyParams);             DbSet<T> table = propertyValue as DbSet<T>;             return table;         }                  public static void SaveChangesToDatabase()         {             VoweiContext.Instance.SaveChanges();         }           // TODO: 为了实现Chainability,性能是一个很大的问题         /// <summary>         /// 返回用户的任务         /// </summary>         /// <typeparam name="T">从Ticket类型继承下来的类型列表</typeparam>         /// <param name="table">任务数据库</param>         /// <returns>所有用户的任务</returns>         public static IQueryable<T> My<T>(this IQueryable<T> table) where T : Ticket         {             return table.Where(t => t.AssignedTo ==                                      ContextContainer.Instance.User.Email);         }           /// <summary>         /// 返回属于用户负责的项目列表         /// </summary>         /// <param name="table">项目列表数据库</param>         /// <returns>用户负责的所有项目</returns>         public static IQueryable<Project> My(this IQueryable<Project> table)         {             return table.Where(t => t.Owner ==                                     ContextContainer.Instance.User.Email);         }           /// <summary>         /// 返回指定表里面处于激活状态的记录         /// </summary>         /// <typeparam name="T">需要查询的数据类型，比如任务、项目等</typeparam>         /// <param name="table">任务、项目等数据表</param>         /// <returns>返回所有激活状态的数据</returns>         public static IQueryable<T> Active<T>(this IQueryable<T> table) where T : ISupportDefaultProperties         {             return table.Where(t => t.ClosedDate == null);         }           /// <summary>         /// 返回指定表里面处于关闭状态的记录         /// </summary>         /// <typeparam name="T">需要查询的数据类型，比如任务、项目等</typeparam>         /// <param name="table">任务、项目等数据表</param>         /// <returns>返回所有关闭状态的数据</returns>         public static IQueryable<T> Closed<T>(this IQueryable<T> table) where T : ISupportDefaultProperties         {             return table.Where(t => t.ClosedDate.HasValue);         }           /// <summary>         /// 返回跟指定集合关联的数据记录         /// </summary>         /// <typeparam name="T">需要查询的数据类型，比如任务、项目等</typeparam>         /// <typeparam name="U">关联集合的类型</typeparam>         /// <param name="table">任务、项目等数据表</param>         /// <param name="container">关联集合</param>         /// <returns>返回跟关联集合相关的数据记录</returns>         public static IQueryable<T> BelongsTo<T, U>(this IQueryable<T> table,              U container)              where T : ITable             where U : IContainer<T>         {             return table.Where(t => container.Children.Any(c => c.ID == t.ID));         }           public static IQueryable<T> BelongsTo<T, U>(this IQueryable<T> table,             IQueryable<U> colContainer)             where T : ITable             where U : IContainer<T>         {             return table.Join<T, U, ICollection<T>, T>(colContainer,                 t => new T[] { t },                 c => c.Children,                 (t, c) => c.Children.First(a => a.ID == t.ID));         }           /// <summary>         /// 返回某个结果集的前面几条记录         /// </summary>         /// <typeparam name="T">需要查询的数据类型，比如任务、项目等</typeparam>         /// <param name="table">任务、项目等数据表</param>         /// <param name="number">返回记录的条目数</param>         /// <returns>指定结果集的前几条记录</returns>         public static IQueryable<T> Top<T>(this IQueryable<T> table, int number)         {             if (number < 0)                 throw new ArgumentOutOfRangeException("number should greater than or equal to zero");               throw new NotImplementedException();         } |

这样设计的目的是让客户端可以流畅地组合多个函数来完成更复杂的任务。比如说，要获取到属于当前登录用户的，处于激活状态的任务列表，可以使用下面的方式完成：

var tasks = FindTableByType<Task>().My().Active();

然而，这种实现方式的一个缺点是，如果处理不善，组合多个函数的工作会导致系统从数据库中加载多次数据集，然后在这些数据集之间完成过滤操作。我之所以推荐上门的设计，主要是因为编码过程简单易懂，对于这种问题的缺点，主要有以下两个考虑：

1. ADO.NET Entity Framework在执行数据库查询之前，会将针对同一结果集的多个过滤lamba表达式合并到最终的查询SQL语句上去。
2. 即时上一条说的假设不成立，我们可以研究使用.NET 4.0里面的Lazy<T>这个类型来实现类似的操作，即IQueryable接口里面包含的T是从Lazy<T>继承下来的，一直等到在客户端使用到结果数据集的数据的时候，才执行查询操作。

### 编辑数据

由于数据库中的表都已经映射到.NET里面的具体POCO对象，因此编辑的数据实际上是通过反射对象的属性列表，将属性值的变化合并到对象里，然后整体将所有的改动提交到数据库中。这个过程，请参考下面代码里面的ApplyEditing函数：

|  |
| --- |
| public static class EditingHelpers     {         public const int CreatingObjectPrimaryKey = 0;         public const string JsonObjectKey = "JsonObject";         public const string JsonObjectName = "server\_json\_object";           internal static object[] EmptyParams = new object[] { };           public static void ApplyEditing<T>(this T tobeEdit, T other)         {             if (object.Equals(other, default(T)))                 return;               if (tobeEdit == null)                 throw new ArgumentException("tobeEdit");             if (other == null)                 throw new ArgumentException("other");               var properties = typeof(T).GetProperties();             foreach (var property in properties)             {                 var propEditing = property.GetValue(tobeEdit, EmptyParams);                 var propOther = property.GetValue(other, EmptyParams);                   if (property.PropertyType.IsAssignableFrom(typeof(IList)))                 {                     ApplyCollectionEditing(propEditing as IList, propOther as IList);                 }                   if (property.CanWrite)                     property.SetValue(tobeEdit, propOther, EmptyParams);             }         }           public static void CreatingFillProperties<T>(this T tobeEditing) where T : ISupportDefaultProperties         {             if (tobeEditing == null)                 throw new ArgumentException("tobeEdit");               tobeEditing.OpenDate = DateTime.Now;             // TODO: 当实现权限控制后,将这一行替换成获取用户名的代码             tobeEditing.Reporter = "yimin\_shi@hotmail.com";         }           public static T AddJsonSupport<T>(this T view)             where T : ViewResult         {             view.ViewData[JsonObjectKey] = JsonConvert.SerializeObject(view.ViewData.Model);             return view;         }           public static U GenericEndEdit<U>(U obj)              where U : class, ISupportDefaultProperties         {             if (obj == null)                 throw new ArgumentNullException("obj");               obj.CreatingFillProperties();             DbSet<U> table = DataLayer.FindTableByType<U>();               if (obj.ID == CreatingObjectPrimaryKey)                 table.Add(obj);             else             {                 var objEditing = table.Find(obj.ID);                 objEditing.ApplyEditing(obj);                 obj = objEditing;             }               DataLayer.SaveChangesToDatabase();               return obj;         }                  private static void SaveChangesToDatabase()         {             VoweiContext.Instance.SaveChanges();         }           public static T GenericBeginEdit<T>(int id) where T : class, new()         {             T objEditing = default(T);             if (id == CreatingObjectPrimaryKey)                 objEditing = new T();             else             {                 DbSet<T> table = DataLayer.FindTableByType<T>();                 objEditing = table.Find(id);                 if (objEditing == null)                     throw new ArgumentException(string.Format("找不到ID为 {0} 的数据!", id));             }               return objEditing;         }           private static void ApplyCollectionEditing(IList original, IList newer)         {             // if newer is not null, means that no changes made             if (newer == null)                 return;               original.Clear();             foreach (var item in newer)             {                 original.Add(item);             }             }     } |

通过将所有的更新操作集中到一个函数上处理，我们可以通过在这个函数里添加日志记录功能，例如保存哪些属性在什么时候，被谁变动了之类的信息。

# 安全的实现

为了保证权限验证的顺利实施，我们通过在vowei.service一端，验证用户是否有权限访问某个指定的URL的方式来实现。因为vowei.service才是实际我们向终端用户提供服务的接口，而所有的资源（比如所有的任务，我的任务，更新任务等）都是通过URL的方式访问的，所以URL除了作为指明服务的唯一路径以外，还可以让我们执行权限验证。这种方式，也便于我们在后续版本里，将权限控制集成到http modules里的想法。