数据格式

为了统一，后端响应的数据类型都是 `TData` 或者泛型 `TData<T>`类型的 json 数据，T 为泛型，可以是 List 类型或是 Object 类型的数据。

- 如果是简单数据的状态返回，可以使用 `TData` 类型，有 Tag、Message、Description 这 3 个属性。

```
public class TData
{
    /// <summary>
    /// 操作结果，Tag 为 1 代表成功，0 代表失败，其他的验证返回结果，可根据需要设置
    /// </summary>
    public int Tag { get; set; }

    /// <summary>
    /// 提示信息或异常信息
    /// </summary>
    public string Message { get; set; }

    /// <summary>
    /// 扩展 Message
    /// </summary>
    public string Description { get; set; }
}
```

- 如果是复杂数据的状态返回，可以使用泛型类型的 `TData<T>`，它继承自 TData。 当需要返回一条记录的时候，TotalCount 属性不使用，当需要返回分页数据的时候，TotalCount 属性为总记录数，Result 属性存放需要返回的数据。

```
public class TData<T> : TData
{
    /// <summary>
    /// 列表的记录数
    /// </summary>
    public int TotalCount { get; set; }

    /// <summary>
    /// 数据
    /// </summary>
    public T Result { get; set; }
```

```
    }
```

注意看类属性上面的注释，对各个属性的作用都有解释，比如属性 Tag 成功为 1，失败为 0（int 型默认就是 0，失败时可以不用赋值，成功时才要赋值）， 这是一种默认的约定，实际开发可以按照这种约定，方便统一控制。

分页实现

1. 前端通过组件 `ysTable` 来显示表格

```
   $(function () {
       initGrid();
    });

    function initGrid() {
       var queryUrl =
'@Url.Content("~/OrganizationManage/Position/GetPageListJson")';
       $('#gridTable').ysTable({
           url: queryUrl,
           sortName: 'PositionSort',
           sortOrder: 'Asc',
           columns: [
               { checkbox: true, visible: true },
               { field: 'Id', title: 'Id', visible: false },
               { field: 'PositionName', title: '职位名称', width: "15%",
sortable: true },
               { field: 'PositionSort', title: '显示顺序', width: "15%",
sortable: true },
               {
                   field: 'BaseModifyTime', title: '创建时间', formatter:
function (value, row, index) {
                       return ys.formatDate(value, "yyyy-MM-dd HH:mm:ss");
                   }
               }
           ],
           queryParams: function (params) {
               // 获取分页参数
               var pagination = $('#gridTable').ysTable('getPagination',
params);
```

```
                // 获取界面搜索条件
                var queryString =
$("#searchDiv").getWebControls(pagination);
                return queryString;
            }
        });
    }
```

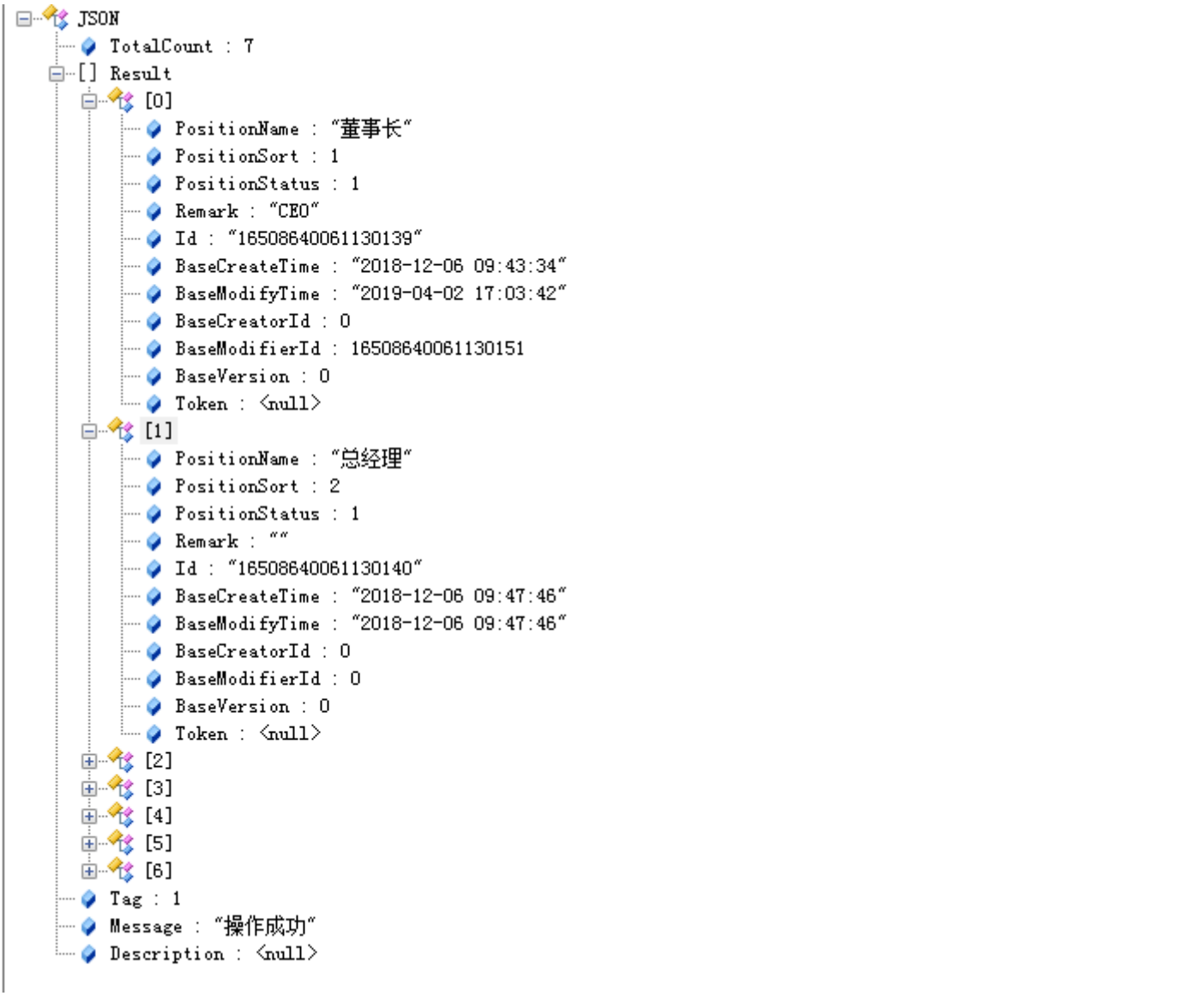
`searchDiv` 是一个大的 div，里面是搜索条件控件。

2. 后端 Controller

```
        [HttpGet]
        [AuthorizeFilter("organization:position:search")]
        public async Task<IActionResult> GetPageListJson(PositionListParam
param, Pagination pagination)
        {
            TData<List<PositionEntity>> obj = await
sysPositionBLL.GetPageList(param, pagination);
            return Json(obj);
        }
```

- `AuthorizeFilter` 继承自 `ActionFilterAttribute`，作用是对 Action 进行权限校验，`organization:position:search` 为 `GetPageListJson` 这个 Action 对应的权限标识。
- `GetPageListJson` 方法的第一个参数 `PositionListParam` 会自动映射传入的搜索条件参数，第二参数 `PositionListParam` 会自动映射传入的分页条件。

- 响应的数据为 Json 格式：



- 搜索条件映射成 Sql 语句有 2 种写法，第一种是把输入条件参数转成 Expression 表达式树，适合简单的 Sql 查询，第二种是拼接原生 Sql，适合复杂的 Sql 查询。

  1）第一种转成 Expression 表达式树写法

```
     private Expression<Func<PositionEntity, bool>>
ListFilter(PositionListParam param)
     {
         var expression = LinqExtensions.True<PositionEntity>();
         if (param != null)
         {
             if (!string.IsNullOrEmpty(param.PositionName))
             {
                 expression = expression.And(t =>
t.PositionName.Contains(param.PositionName));
             }
```

```
            if (!string.IsNullOrEmpty(param.PositionIds))
            {
                long[] positionIdArr =
CommonHelper.SplitToArray<long>(param.PositionIds, ',');
                expression = expression.And(t =>
positionIdArr.Contains(t.Id.Value));
            }
        }
        return expression;
    }
```

调用方法如下：

```
    public async Task<List<PositionEntity>>
GetPageList(PositionListParam param, Pagination pagination)
    {
        var expression = ListFilter(param);
        var list = await this.BaseRepository().FindList(expression,
pagination);
        return list.ToList();
    }
```

2）第二种拼接原生 Sql 写法

```
    private List<DbParameter> ListFilter(LogApiListParam param,
StringBuilder strSql)
    {
        strSql.Append(@"SELECT  a.id as Id,
                               a.base_modify_time as BaseModifyTime,
                               a.base_modifier_id as BaseModifierId,
                               a.log_status as LogStatus,
                       FROM    sys_log_api a
                               LEFT JOIN sys_user b ON
a.base_modifier_id = b.id
                               LEFT JOIN sys_department c ON
b.department_id = c.id
                       WHERE   1 = 1");
        var parameter = new List<DbParameter>();
        if (param != null)
        {
            if (!string.IsNullOrEmpty(param.UserName))
            {
                strSql.Append(" AND b.user_name like @UserName");
```

```
parameter.Add(DbParameterExtension.CreateDbParameter("@UserName",
'%' + param.UserName + '%'));
            }
        }
        return parameter;
    }
```

调用方法如下：

```
    public async Task<List<LogApiEntity>>
GetPageList(LogApiListParam param, Pagination pagination)
    {
        var strSql = new StringBuilder();
        List<DbParameter> filter = ListFilter(param, strSql);
        var list = await
this.BaseRepository().FindList<LogApiEntity>(strSql.ToString(),
filter.ToArray(), pagination);
        return list.ToList();
    }
```

Excel 导入导出

Excel 导入导出使用开源项目 `NPOI` 进行处理，以下把用户导入和用户导出作为示例。

Excel 导入

Excel 导入分两步，第一步把需要导入的文件通过 `FileController` 上传到后台，第二步把第一步返回的文件相对地址传给导入处理方法。

1. 前端写法

```
<input id="importFile" type="file">


// 把需要导入的文件通过 FileController 上传到后台
var filePath = undefined;
$("#importFile").fileinput({
    language: 'zh',
    'uploadUrl': '@Url.Content("~/File/UploadFile")' +
'?fileModule=@UploadFileType.Import.ParseToInt()',
    showPreview: false,
    allowedFileExtensions: ['xls', 'xlsx']
}).on("fileuploaded", function (event, data) {
    var obj = data.response;
    if (obj.Tag == 1) {
```

```
        filePath = obj.Result;
    }
    else {
        filePath = '';
    }
});

// 把第一步返回的文件相对地址传给导入处理方法
function saveForm(index) {
    if (!filePath) {
        ys.alertError('文件未上传或者上传失败');
        return;
    }

    var postData =$("#form").getWebControls();
    postData.FilePath = filePath;
    ys.ajax({
        url: '@Url.Content("~/OrganizationManage/User/ImportUserJson")',
        type: "post",
        data: postData,
        success: function (obj) {
            if (obj.Tag == 1) {
                ys.msgSuccess('导入成功');
                parent.searchGrid();
                parent.layer.close(index);
            }
            else {
                ys.msgError(obj.Message);
            }
        }
    });
}
```

2. 后端写法

```
[HttpPost]
public async Task<IActionResult> ImportUserJson(ImportParam param)
{
    List<UserEntity> list = new
ExcelHelper<UserEntity>().ImportFromExcel(param.FilePath);
    TData obj = await userBLL.ImportUser(param, list);
    return Json(obj);
```

```
}
```

方法 ImportFromExcel 会根据传入的 Excel 文件相对路径，找到文件的绝对路径。Excel 的第一行是标题，第二行是列名，列名可以是实体类的属性名称或者实体类属性名称的 Description 描述。

Excel 导出

Excel 导出可以根据界面的搜索条件，后台设置需要导出的列，导出所有符合条件的记录。

1. 前端写法

```
<a id="btnExport" class="btn btn-warning" onclick="exportForm()"><i
class="fa fa-download"></i> 导出</a>

function exportForm() {
    var url = '@Url.Content("~/OrganizationManage/User/ExportUserJson")';
    var postData = $("#searchDiv").getWebControls();
    ys.exportExcel(url, postData);
}
```

2. 后端写法

```
[HttpPost]
public async Task<IActionResult> ExportUserJson(UserListParam param)
{
    TData<string> obj = new TData<string>();
    TData<List<UserEntity>> userObj = await userBLL.GetList(param);
    if (userObj.Tag == 1)
    {
        string file = new ExcelHelper<UserEntity>().ExportToExcel("用户列
表.xls",
                                                                  "用户列表",
                                                                  userObj.Result,
                                                                  new string[]
{ "UserName", "RealName", "Gender", "Mobile", "Email" });
        obj.Result = file;
        obj.Tag = 1;
    }
    return Json(obj);
}
```

方法 ExportToExcel 的第一个参数是 Excel 文件名，第二个参数是 Excel 的表头，第三个参数是数据源，第四个参数是导出的列（列名会自动映射为实体类上面的 Description 属性）

文件或图片上传

管理后台上传的图片或文件可能也需要在 Api 项目能够访问，考虑到文件的统一管理，后台的文件默认是上传到 Api 项目里的。
YiSha.Admin.Web 项目里的配置文件 appsetting.json 中的 ApiSite 节点就是配置管理后台使用的 Api 地址，所以如果管理后台和 Api 不在一个域名下，就涉及到跨域上传。当然，Api 项目通过配置是支持跨域请求的， 在 Api 项目的配置文件 appsetting.json 中的 AllowCorsSite 节点就是配置允许的跨域站点来请求 Api。
使用流程

1. 前端图片上传

```
function uploadThumbImage(file, callback) {
 var formdata = new FormData();
 formdata.append("fileList", file);
 ys.ajaxUploadFile({
     url: '@GlobalContext.SystemConfig.ApiSite' +
  '/File/UploadFile?fileModule=@UploadFileType.News.ParseToInt()',
     data: formdata,
     success: function (obj) {
         if (obj.Tag == 1) {
             if (callback) {
                 callback('@GlobalContext.SystemConfig.ApiSite' +
  obj.Result);
             }
         }
         else {
             ys.msgError(obj.Message);
         }
     }
 })
}
```

完成代码，可以参考 News/NewsForm.cshtml 页面，方法 uploadThumbImage 的参数 file 就是 <input type='file' /> 控件的属性 files 数组的值。

2. 后端上传处理

上面代码里 url 中的/File/UploadFile 就是后端处理上传的 Controller，代码如下，参数 fileModule 表示上传到哪个模块，就是按模块存放文件。

```
[HttpPost]
public async Task<TData<string>> UploadFile(int fileModule, IFormCollection
fileList)
```

```
{
    TData<string> obj = await FileHelper.UploadFile(fileModule,
fileList.Files);
    return obj;
}
```

日志和异常处理

日志分三类，登录日志、操作日志、Api 日志。

日志记录采用异步的方式 Insert 到日志表中，每个调用的耗时也会记录下来，方便查看哪些调用比较耗时。 如果调用异常了，会插入一条失败的日志记录，并且异常信息会记录下来，方便 debug ^_^

1. 登录日志：用户登录或退出系统，都会插入一条日志记录，对应的表为 `sys_log_login`。
2. 操作日志：用户在后台查看、新增、修改、删除数据都会插入一条日志记录，对应的表为 `sys_log_operate`。
3. Api 日志：Api 调用每个方法的调用都会插入一条日志记录，对应的表为 `sys_log_api`。

权限控制

系统是由页面组成的，每个页面对应一个权限标识，页面包含按钮，每个按钮同样对应一个权限标识，下面以 `OrganizationMange` 模块下的 `News` 页面为例。

1. 页面权限

页面 `OrganizationManage/News/NewsIndex`，页面可见对应的权限标识为 `organization:news:view`。

2. 按钮权限

页面包含增删改查按钮，查询对应的权限标识为 `organization:news:search`，增加对应的权限标识为 `organization:news:add`， 修改对应的权限标识为 `organization:news:edit` 删除对应的权限标识为 `organization:news:delete`。
注意事项

1. 前台权限限制

当用户进入到系统的主页面`/Home/Index` 时，通过 `yisha-data.js` 会把当前用户所有的权限加载进来，针对 `toolbar` 类型的按钮， 如果没有权限，按钮会被自动移除掉。按钮的命名需要和权限标识对应起来，例如权限标识 `organization:news:add` 对应在页面的按钮是 `btnAdd`， 遵循这样的命名，框架会根据按钮的 id 找权限，找不到权限，按钮会被自动移除掉。

这是前台的权限限制，一般来说，在前台做这种限制，可以限制在系统操作的正常用户，如果有不友好的朋友，模拟抓包请求没有权限的接口， 所以后台也需要做权限限制。

```
<div class="btn-group-sm hidden-xs" id="toolbar">
     <a id="btnAdd" class="btn btn-success"
onclick="showSaveForm(true)"><i class="fa fa-plus"></i> 新增</a>
     <a id="btnEdit" class="btn btn-primary disabled"
onclick="showSaveForm(false)"><i class="fa fa-edit"></i> 修改</a>
     <a id="btnDelete" class="btn btn-danger disabled"
onclick="deleteForm()"><i class="fa fa-remove"></i> 删除</a>
</div>
```

1. 后台权限限制

在每个 `Action` 方法上加 `AuthorizeFilter("")` 这个 Attribute，多个权限标识字符串用逗号隔开。

```
[HttpPost]
[AuthorizeFilter("organization:news:add,organization:news:edit")]
public async Task<IActionResult> SaveFormJson(NewsEntity entity)
{
    TData<string> obj = await newsBLL.SaveForm(entity);
    return Json(obj);
}
```

定时任务

在实际开发 Web 项目时，有一类不可缺少的，那就是定时任务。定时任务的场景可以说非常广泛，比如某些视频网站，购买会员后，每天会给会员送成长值， 每月会给会员送一些电影券；比如定时备份数据库、发送邮件、清理数据等。所以我们提供方便友好的 Web 界面，实现动态管理任务，可以达到控制定时任务 启动、暂停、重启、删除、添加、修改等操作，极大地方便了开发过程。

使用流程

1. 新建定时任务数据库记录

   在 系统管理-> 定时任务 中新建一个任务，`任务名称` 用来唯一的区分一个任务。

2. 新建定时任务类

   在 YiSha.Business.AutoJob 项目的 Job 文件夹，建一个任务类，继承自接口 `IJobTask`，在任务类里面写业务代码。

3. 加入到 `JobExecute`

   在 `JobExecute` 类的 `Execute` 方法，加一个 case 分支，case 的名称就是第一步建的`任务名称`。

```
switch (context.JobDetail.Key.Name)
{
 case "数据库备份":
    obj = await new DatabasesBackupJob().Start();
    break;

 case "任务名称":
    // do job
    break;
}
```

Api 接口

Api 对应的项目为 `YiSha.Admin.WebApi`，使用 `Swagger` 来查看文档及测试。

1. 在控制层 `Controller` 中添加特定的 Attribute 来描述接口信息或是验证。

```
[Route("[controller]/[action]")]
[ApiController]
[AuthorizeFilter]
 public class NewsController : ControllerBase
```

`AuthorizeFilter` 是自定义的 Attribute，用来验证调用者是否有整个 Controller 的权限。

5. 在 `Action` 中配置方法的 Request Method

```
[HttpGet]
public async Task<TData<List<NewsEntity>>> GetPageList([FromQuery] NewsListParam param, [FromQuery] Pagination pagination)

[HttpPost]
public async Task<TData<string>> SaveForm([FromBody]NewsEntity entity)
```

10. 在 系统管理-> 系统 Api 中查看 Api 文档及测试接口。

数据库表

为了方便处理，数据表和实体之间的命名保持一致，采用帕斯卡命名法（也叫大驼峰命名法 Upper Camel Case）。 数据库表采用 模块名+表名 的规则命名，比如系统角色表，SysRole，如果你有自己的业务模块，客户管理（Crm），客户表就可以这样命名 CrmCustomer。 模块可以和 MVC 的 `Areas` 不是完全对应的，一切以自己的业务需要为基准。

数据库表字段的命名和表名的命名类似。还是以系统角色表为例，比如字段 RoleName，就代表角色名，比如字段 Remark，就表示备注。

每个表根据需要继承自不同的实体，目前可供继承的实体有 `BaseEntity`、`BaseCreateEntity`、`BaseModifyEntity`、`BaseExtensionEntity`。如果你的表里只有 Id 字段，可以继承 `BaseEntity`；如果你的表里需要创建时间 `BaseCreateTime` 字段和创建人 `BaseCreatorId` 字段，可以继承 `BaseCreateEntity`，其他以此类推。 继承合适的实体在于框架会给这些实体自动赋值，这样就不用在自己的业务模块里去操作了。

下面以 MySql 数据库为准，框架支持的表基础字段。

字段名	类型	描述
Id	BIGINT	主键，非自增
BaseIsDelete	INT	删除字段标记，1 表示删除，默认是 0
BaseCreateTime	DATETIME	创建时间
BaseModifyTime	DATETIME	修改时间
BaseCreatorId	BIGINT	创建人
BaseModifierId	BIGINT	修改人
BaseVersion	INT	数据更新版本，控制并发

代码生成

大部分项目里其实有很多代码是重复的，几乎每个基础模块的代码都有增删改查（CRUD）的功能，而这些功能都是大同小异的， 每个功能对应的 View、Controller、Business、Service、Model 文件结构也是一致的。 如果这些功能都要自己动手写，将会大大浪费我们的精力降低效率，所以这种重复性的代码和创造文件的工作可以使用代码生成。

注意事项

代码生成只生成单表的增删改查，如果需要复杂的多表操作，可以在生成的单表代码基础上修改。

使用流程

1. 新建数据库表结构

   手动创建或是用脚本创建表，创建之后，可以在 系统工具-> 代码生成 页面看到创建的表。

2. 生成代码基本配置

   选择刚刚创建的表，点击生成，会弹出一个生成代码界面，控件的 * 表示为必填项。

   类名前缀 如果没有特殊需求，可以默认，不用修改，它是根据数据库表名生成的。

输出到所在模块 此下拉框里面的值，是读取 `YiSha.Admin.Web` 项目的 `Areas` 目录里面的文件夹，如果要生成到一个新模块，可以添加一个新文件夹，注意模块名的后缀是 Manage。

3. 列表页编辑

   在这个页面，可以选择列表页是否需要搜索、是否需要工具栏、列表需要显示哪些字段、列表是否需要分页等，设置好之后，代码生成工具会按需生成。

4. 表单页编辑

   在这个页面，可以选择表单页需要显示哪些字段、显示方式等，设置好之后，代码生成工具会按需生成。

5. 代码预览

   在这个页面，可以看到即将生成的所有代码，代码会自动生成到对应模块的目录下，如果文件已经存在，就不产生新文件。
   菜单会生成在菜单根目录，在实际使用时，需要移动菜单的位置，在 系统管理-> 菜单管理 里面选择父级菜单即可。