🍹Yotech新框架使用文档

配置中心

关系型数据库

- 使用步骤和注意事项
- ⚡使用权限仓储过滤数据
- ⚡权限初始化
- 仓储封装常用方法
- ⚡注意事项

身份验证

注册发现

消息队列

短信邮件

全局用户信息

附件

MongoDB

- 切库

缓存

定时任务

性能日志

日志

工作流

- ⚡使用新框架必执行⚡
- 用法
- 签核活动
  - 会签
- 抄送活动
- 无人审核自动通过

人脸识别

OCR

转PDF
初始化数据库和工作流
网关
用法1单纯转发
用法2使用注册发现中心
网关转发失败原因自查
自动发布
模型校验
请求传参校验
其他情况
NLP

2023.4.10	第一版
2023.4.14	第二版：新增MongoDB，Redis
2023.4.20	第三版：新增Job，性能日志，缓存优化
2023.5.10	第四版：新增工作流，Meta初始化，逻辑删除 nuget版本1.0.4
2023.5.24	第五版: 新增人脸识别,OCR, 转Pdf,[GenerateTable],nuget版本1.0.9
2023.7.18	第六版：支持连接字符串和不同类型数据库切换（目前支持MySQL和SqlServer）,nuget版本1.2.7
2023.10.20	第七版：新增OData权限特性及EFCore权限仓储,nuget版本1.3.5
2023.11.10	第八版：新增初始化数据库和工作流,nuget版本1.3.6
2023.11.18	第九版：工作流事务和mongodb优化,nuget版本1.3.11
2024.01.15	第十版：更新自动发布、网关及注册发现等内容,nuget版本1.3.17
2024.03.05	第十一版：加模型校验等,nuget版本1.4.4
2024.03.19	第十二版：加mongo切库,最新nuget版本1.4.15
2024.05.14	第十三版：加NLP和附件是否hash验证的配置,最新nuget版本1.4.16-pre
2024.06.18	第十四版：底层工作流加会签、无人审核自动通过、抄送活动。最新nuget版本1.5.4

目前配置项和注入都初始化过了，后续可以根据需要配置。

program.cs必加 C#

```
var app = builder.Build();
ServiceProviderHelper.Instance = app.Services;  //这句话直接加在app下面，一些单例注入，比如配置或者缓存需要用到，建议都先写上
```

配置中心

开发环境地址http://172.20.20.29:8848/nacos/#/login 登录账户密码都是nacos

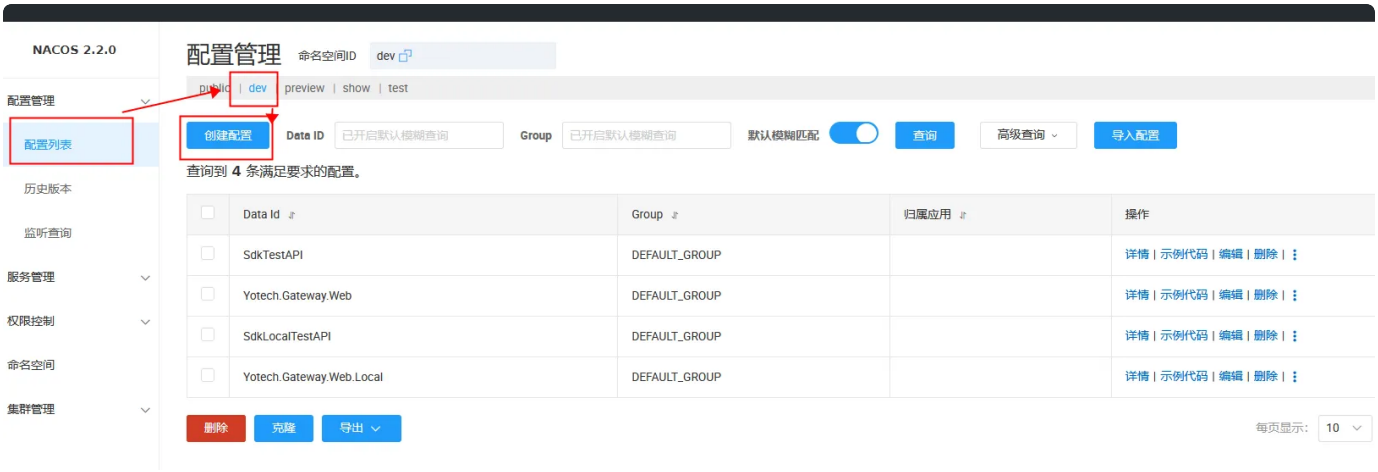
1. 引用 Yotech.Sdk.ConfigCenter
2. program.cs

```
builder.Configuration.AddConifgCenter(builder.Configuration.GetSection("ConfigCenter"));
```

3. appsettings.json

（除此之外的配置可以放到配置中心，只有这个是本地必须有的）

```
"ConfigCenter": {
    "Listeners": [
      {
        "Optional": false,
        "DataId": "SdkLocalTestAPI", //跟nacos创建的DataId一样就行
        "Group": "DEFAULT_GROUP"
      }
    ],
    "Namespace": "dev", //根据使用的环境配置
    "ServerAddresses": [ "http://218.28.18.12:8848/" ],
    "UserName": "nacos",
    "Password": "nacos"
  },
```



关系型数据库

引用 Yotech.Sdk.Data Yotech.Sdk.Data.Linq

```
1  //使用普通仓储
2  builder.Services.AddDb<YotechDbContext>(builder.Configuration.GetSection("C
   onnectionStrings"));
3
4  //使用权限仓储（会在增删改查自动查询权限）
5  builder.Services.AddPermissionDb<YotechDbContext>(builder.Configuration.Get
   Section("ConnectionStrings"));
6
```

```
//单库
  "ConnectionStrings": {
    "DbType": 0,
    "Master": "Server=172.20.20.21; database=Test; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;"
    }



//单主多从
"ConnectionStrings": {
    "DbType": 0,//0 sqlserver 1 mysql
    "Master": "Server=172.20.20.21; database=Test; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;",
    "Slaves": [
      "Server=172.20.20.21; database=Test_Slave; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;",
      "Server=172.20.20.21; database=Test_Slave2; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;"
    ]
  },


//多主多从
  "ConnectionStrings": {
    "DbType": 0, //0 sqlserver 1 mysql
    "Master": "Server=172.20.20.21; database=Test; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;",
    "Slaves": [
      "Server=172.20.20.21; database=Test_Slave; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;",
      "Server=172.20.20.21; database=Test_Slave2; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;"
    ],
    "OtherMasters": {
      "TestDb": {
        "DbType": 0,
        "Master": "Server=172.20.20.21; database=Test; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;",
        "Slaves": [
          "Server=172.20.20.21; database=Test_Slave; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;",
          "Server=172.20.20.21; database=Test_Slave2; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;"
        ]
      }
    }
```

```
38
39      }
40
```

使用步骤和注意事项

1. 添加实体，
    a. 初始化的表必须添加[GenerateTable]
    b. 不需要初始化但是需要注入到EFCore的需要加[UseDbContext]（有那种需要用别的子系统的表但是又不能初始化到当前库的情况）
    c. 必须继承PermissionEntityBase
    d. 非必须继承EntityBase<TKey>

```
/// <summary>
/// 实体基类
/// </summary>
public class EntityBase<TKey>: PermissionEntityBase
{
    /// <summary>
    /// 主键
    /// </summary>
    [Key]
    [MaxLength(50)]
    public TKey Id { get; set; }
    /// <summary>
    /// 名称
    /// </summary>
    [MaxLength(200)]
    [PrimaryName]
    public string? Name { get; set; }
    /// <summary>
    /// 创建人
    /// </summary>
    [MaxLength(50)]
    public TKey? CreateUserId { get; set; }
    /// <summary>
    /// 更新人
    /// </summary>
    [MaxLength(50)]
    public TKey? UpdateUserId { get; set; }
    /// <summary>
    /// 创建时间
    /// </summary>
    public DateTime? CreateTime { get; set; }
    /// <summary>
    /// 更新时间
    /// </summary>
    public DateTime? UpdateTime { get; set; }
    /// <summary>
    /// 版本号
    /// </summary>
    [Timestamp]
    public byte[]? LockVersion { get; set; }

}
//第七版权限新增
public class PermissionEntityBase
{
    /// <summary>
    /// 所属人Id
```

```
        /// </summary>
        [MaxLength(50)]
        public string? PossessorId { get; set; }
        /// <summary>
        /// 部门Id
        /// </summary>
        [MaxLength(50)]
        public string? DepartmentId { get; set; }
        /// <summary>
        /// 机构Id
        /// </summary>
        [MaxLength(50)]
        public string? TradingOrgId { get; set; }
    }
```

2. 创建ICourseRepository （这些都可以参照目前的代码）

```
public interface ICourseRepository : IBaseRepository<Course, string>
{
}
```

3. 创建CourseRepository，继承ICourseRepository

4. 创建ICourseService 业务逻辑层接口

```
public interface ICourseService: IBaseService
{

    bool Insert();
    ...
}
```

5. 实现CourseService 继承ICourseService

```
public class CourseService : ICourseService
{
    private readonly IUnitOfWork Db;
    private readonly ICourseRepository _courseRepository;
    //private readonly IBaseRepository<Course,string> _courseRepository;//注意这个写法跟上面一样可以注入Course仓储，且不用建仓储类和接口，很方便，可以自己选择
    private readonly IStudentRepository _studentRepository;
    public CourseService(IUnitOfWork db, ICourseRepository courseRepository, IStudentRepository studentRepository)
    {
        Db = db;
        _courseRepository = courseRepository;
        _studentRepository = studentRepository;
    }
    public Course FindById(string Id)
    {
        var course = _courseRepository.GetByKey(Id);
        return course;

    }
}
```

6. 控制器 继承BaseApiController，可以注入ICourseService，IUserIdentity（用户信息）

```
[Authorize]
[ApiController]
[Route("api/[controller]/[action]")]
public class CourseController : BaseApiController
{
    private readonly ICourseService _courseService;
    private readonly IUserIdentity _userIdentity;
    private static IConfiguration _configuration { get; set; }
    public CourseController(ICourseService courseService, IConfiguration configuration, IUserIdentity userIdentity)
    {
        _courseService = courseService;
        _configuration = configuration;
        _userIdentity = userIdentity;
    }
```

7. 注意 接口和实现类名称只能差一个“I”才能自动注入。
8. 增删改查方法和原来的一致，唯一不同就是现在事务和直接执行sql语句需要用IUnitOfWork，其他单表增删改查用自己创建的IRepository，具体可以看 CourseService相关实现
9. 主从切换举例（事务中不要切换）

```
//刚开始初始化是配置的主库
var course = _courseRepository.GetByKey("主键Id");
//切换配置的其他主库，因为会取配置项里的数据库类型，可以不传第二个参数
Db.ReadWrite(DbChangeTypeEnum.Config,null,"TestDb");
var course2 = _courseRepository.GetByKey("主键Id");
//通过连接字符串切换mysql数据库
Db.Read(DbChangeTypeEnum.ConnectString,  DbTypeEnum.MySql, "Server=172.20.20.99; port=13306;database=Test; Uid=root; Pwd=yotechmysql;Allow User Variables=True; sslMode=None;");
var course3 = _courseRepository.GetByKey("主键Id");
//通过连接字符串切换sqlserver数据库
Db.Read(DbChangeTypeEnum.ConnectString, DbTypeEnum.SqlServer, "Server=172.20.20.21; database=Test_Slave3; uid=queryuser; pwd=1qaz2wsx~;TrustServerCertificate=true;");
var course4 = _courseRepository.GetByKey("主键Id");
```

```
//不管是主从还是不同数据库切换，不要在同一个事务进行操作，但是在不同事务是可以的
try
{
    Db.BeginTransaction();
    _courseRepository.Insert(new Course { Id = "1031", Credits = 201, Title = "数学" });
    Db.Commit();
}
catch (Exception e)
{
    Db.Rollback();
}

Db.ReadWrite(DbChangeTypeEnum.ConnectString, DbTypeEnum.MySql, "Server=172.20.20.99; port=13306;database=Test; Uid=root; Pwd=yotechmysql;Allow User Variables=True; sslMode=None;");
try
{

    Db.BeginTransaction();
    Db.Execute("INSERT INTO Course(Id, Title, Credits) VALUES ('1033', '数学', 201);", null, Db.GetContextTransaction());
    Db.Commit();
}
catch (Exception e)
{
    Db.Rollback();
}
```

⚡使用权限仓储过滤数据

1. 在program.cs中添加

```
builder.Services.AddPermissionDb<YotechDbContext>(builder.Configuration.GetSection("ConnectionStrings"));//和原来的AddDb<YotechDbContext>区分，其他都不用改

builder.Services.AddScoped<IPermissionOptions, EFCorePermission>();
builder.Services.AddScoped<NoPermissionVerifyFilter>();

builder.Services.Configure<RedisConnectionOptions>(builder.Configuration.GetSection("RedisConfig"));
services.AddSingleton<ICacheHandler,RedisHandler>(); //注意：这里要注入缓存 并且用单例模式
```

2. 如果表没继承EntityBase，则必须继承PermissionEntityBase，记得初始化表，所有的表会加三个字段

```
/// <summary>
/// 所属人Id
/// </summary>
[MaxLength(50)]
public string? PossessorId { get; set; }
/// <summary>
/// 部门Id
/// </summary>
[MaxLength(50)]
public string? DepartmentId { get; set; }
/// <summary>
/// 机构Id
/// </summary>
[MaxLength(50)]
public string? TradingOrgId { get; set; }
```

3. 其他正常写代码，在增删改查时会自动查询用户角色的表增删改查权限
4. 如果有匿名方法不需要使用权限，可以在方法或者控制器上加 [ServiceFilter(typeof(NoPermissionVerifyFilter))]，下面是示例

```
[ServiceFilter(typeof(NoPermissionVerifyFilter))]
public class ArticleController : BaseApiController
{
    private readonly IArticleService _articleService;
    private readonly IUserIdentity _userIdentity;
    public ArticleController(IArticleService articleService, IUserIdentity
 userIdentity)
    {
        _articleService = articleService;
        _userIdentity = userIdentity;
    }
    /// <summary>
    /// 文章分页列表
    /// </summary>
    /// <param name="param">查询model</param>
    /// <returns></returns>
    [HttpPost]

    public PageModel<T_CMS_Article> QueryPage()
    {
        var result = _articleService.QueryPage();
        return result;
    }
}
```

5. 或者在不需要走权限的实体加[NoPermissionVerify]特性，一般公用的表比如字典表可以加这个特性不走权限校验，如下示例【一般子表都可以加[NoPermissionVerify]】

```
/// <summary>
/// 附件表
/// </summary>
[GenerateTable]
[NoPermissionVerify]
public  class SysAttach  :PermissionEntityBase
{
    /// <summary>
    /// 附件id.
    /// </summary>
    [Key]
    [MaxLength(50)]
    public string AttachId { get; set; }

    ...
}
```

6. 不管控制器还是实体，只要加不走权限的标记，就不会在增删该查的时候进行权限过滤

7. ⚡多表联查特别注意⚡多表联查以主表最大权限为主【如图】
   a. 【主表】权限查询可以用private readonly IBaseRepository<T_CMS_Article, string> _articleRepository注入后的Table进行查询（图中红框）
   b. 如果是【从表】则要用b.Context.Set<T_CMS_Column>()（图中蓝框）
   c. 为什么要这样写呢，因为Table会走权限筛选，但是Db.Set不会💅，方便我们查询按需求使用权限过滤

```
public PageModel<T_CMS_Article> QueryPage()
{
    var data = from a in _articleRepository.Table
               join c in Db.Context.Set<T_CMS_Column>() on a.ColumnId equals c.Id
               into c_temp from c in c_temp.DefaultIfEmpty()
               select a;
    var result = data.ToPageList(1,5);
    return result;

}
```

8. 查询注意：如果header不加TradingOrgId和SystemKey走权限是查不出数据的！因为权限要根据当前系统和机构id去查用户权限

⚡权限初始化

   a. 【基础是最新的就不用执行】初始化基础的数据库表
      i. 初始化数据库：/api/InitDb/InitDb 【⚡注意：字段类型，如果相比原来的长度变长会改，变短不会改】
   b. 【当前系统第一次上权限必执行】初始化当前系统的数据库表和工作流
      i. 初始化数据库：/api/InitDb/InitDb
      ii. 初始化工作流： /api/InitDb/InitMetadata?subsystemId=子系统Id 【⚡注意：子系统Id需要与基础数据库中配置的管理端子系统Id一致，权限也会用到这个】
   c. 【当前系统第一次上权限必执行】基础子系统表更新 `update SysSubSystem SET IsOpenPermission = 1 where id = '子系统id'`
   d. 【基础是最新的就不用执行】同步部门缓存 baseapi/api/test/InitDepartmentCache，数据库4，key前缀Dept_
   e. 【基础是最新的就不用执行】同步机构用户的缓存baseapi/api/Job/SyncUserCache?userType=20，数据库0，key前缀User_
   f. 【基础是最新的就不用执行】初始化角色权限

baseapi/api/Permission/InitRolePermissionCache 数据库4，key前缀Role_

g. 【实物3.0.4新增，只有这个版本需要初始化，后续只有新增机构才用再次初始化】同步机构及下级机构缓存 baseapi/api/TradingOrg/SyncTradingOrgCache，数据库1，key前缀TradingOrgTree_

h. 【实物3.0.4新增，只有这个版本需要初始化】工作流用，同步角色及包含用户缓存 baseapi/api/Role/SyncRoleCache ，数据库1，key前缀TradingOrgRole_

仓储封装常用方法

1. 只要注入继承IBaseRepository的类就可以使用的方法

C#

```
/// <summary>
/// 主键
/// </summary>
/// <returns></returns>
string GetNextIdentity();
#region Select/Get/Query
/// <summary>
/// GetByKey
/// </summary>
/// <param name="id">主键</param>
/// <returns>Entity</returns>
TEntity? GetByKey(TKey id);
/// <summary>
/// GetByKeyAsync
/// </summary>
/// <param name="id">主键</param>
/// <returns>Entity</returns>
Task<TEntity?> GetByKeyAsync(TKey id);
/// <summary>
/// LockAndGet
/// </summary>
/// <param name="id">主键</param>
/// <returns></returns>
TLockEntity? LockAndGet<TLockEntity>(TKey id) where TLockEntity : EntityBase<TKey>;
 /// <summary>
 /// Table
 /// </summary>
 IQueryable<TEntity> Table { get; }

/// <summary>
/// TableNoTracking(只限定不需要操作的数据，不然不会成功)
/// </summary>
IQueryable<TEntity> TableNoTracking { get; }

#endregion

#region Insert
/// <summary>
/// Insert
/// </summary>
/// <param name="entity">Inserted entity</param>
void Insert(TEntity entity);

/// <summary>
```

```
45  /// BatchInsert
46  /// </summary>
47  /// <param name="entities">Inserted entity</param>
48  void BatchInsert(IEnumerable<TEntity> entities);
49
50
51  /// <summary>
52  /// InsertAsync
53  /// </summary>
54  /// <param name="entity">Inserted entity</param>
55  Task InsertAsync(TEntity entity);
56
57  /// <summary>
58  /// BatchInsertAsync
59  /// </summary>
60  /// <param name="entities">Inserted entity</param>
61  Task BatchInsertAsync(IEnumerable<TEntity> entities);
62  #endregion
63
64  #region Update
65  /// <summary>
66  /// Update
67  /// </summary>
68  /// <param name="entity">Entity</param>
69  void Update(TEntity entity);
70  /// <summary>
71  /// UpdateAsync
72  /// </summary>
73  /// <param name="entity">Entity</param>
74  Task UpdateAsync(TEntity entity);
75  /// <summary>
76  /// BatchUpdate
77  /// </summary>
78  /// <param name="entities"></param>
79  void BatchUpdate(IEnumerable<TEntity> entities);
80  /// <summary>
81  /// BatchUpdateAsync
82  /// </summary>
83  /// <param name="entities"></param>
84  Task BatchUpdateAsync(IEnumerable<TEntity> entities);
85  /// <summary>
86  /// UpdatePartial
87  /// </summary>
88  /// <param name="entity"></param>
89  /// <param name="fields"></param>
90  void UpdatePartial(TEntity entity, List<string> fields);
91  /// <summary>
92  /// UpdatePartialAsync
    /// </summary>
```

```
93  /// <param name="entity"></param>
94  /// <param name="fields"></param>
95  Task UpdatePartialAsync(TEntity entity, List<string> fields);
96
97
98  #endregion
99
100 #region Delete
101 /// <summary>
102 /// 物理删除
103 /// </summary>
104 /// <param name="entity">Entity to be deleted</param>
105 void Delete(TEntity entity);
106 /// <summary>
107 /// 物理删除
108 /// </summary>
109 /// <param name="entity">Entity to be deleted</param>
110 Task DeleteAsync(TEntity entity);
111 /// <summary>
112 /// 物理删除
113 /// </summary>
114 /// <param name="entities"></param>
115 void BatchDelete(IEnumerable<TEntity> entities);
116 /// <summary>
117 /// 物理删除
118 /// </summary>
119 /// <param name="entities"></param>
120 Task BatchDeleteAsync(IEnumerable<TEntity> entities);
121
122 /// <summary>
123 /// 物理删除
124 /// </summary>
125 /// <param name="predicate">A condition to filter entities</param>
126 void BatchDelete(Expression<Func<TEntity, bool>> predicate);
127 /// <summary>
128 /// 物理删除
129 /// </summary>
130 /// <param name="predicate">A condition to filter entities</param>
131 Task BatchDeleteAsync(Expression<Func<TEntity, bool>> predicate);
132 /// <summary>
133 /// 逻辑删除
134 /// </summary>
135 /// <param name="id">主键</param>
136 /// <param name="userId">修改的用户Id，可以为空</param>
137 /// <param name="fieldName">状态字段，可以为空，默认为state</param>
138 /// <returns></returns>
    Task LogicDeleteAsync(TKey id, string userId = null, string fieldName = n
    ull);
139 /// <summary>
```

```
140     /// 逻辑删除
141     /// </summary>
142     /// <param name="ids">主键</param>
143     /// <param name="userId">修改的用户Id，可以为空</param>
144     /// <param name="fieldName">状态字段，可以为空，默认为state</param>
145     /// <returns></returns>
146     Task LogicBatchDeleteAsync(List<TKey> ids, string userId = null, string f
        ieldName = null);
147     #endregion
```

2. 只要注入IUnitOfWork就可以使用的方法（主从切换，事务，dapper查询从这里做）

C#

```
#region 异步
/// <summary>
/// 查询
/// </summary>
/// <typeparam name="TEntity"></typeparam>
/// <param name="sql"></param>
/// <param name="param"></param>
/// <param name="trans"></param>
/// <returns></returns>
Task<IEnumerable<TEntity>> QueryAsync<TEntity>(string sql, object param = null, IDbContextTransaction trans = null) where TEntity : class;
/// <summary>
/// 执行语句，返回数量
/// </summary>
/// <param name="sql"></param>
/// <param name="param"></param>
/// <param name="trans"></param>
/// <returns></returns>
Task<int> ExecuteAsync(string sql, object param, IDbContextTransaction trans = null);
/// <summary>
/// 查询语句，返回泛型
/// </summary>
/// <typeparam name="TEntity"></typeparam>
/// <param name="sql"></param>
/// <param name="param"></param>
/// <param name="trans"></param>
/// <returns></returns>
Task<TEntity> ExecuteScalarAsync<TEntity>(string sql, object param = null, IDbContextTransaction trans = null);
#endregion
IEnumerable<TEntity> Query<TEntity>(string sql, object param = null, IDbContextTransaction trans = null) where TEntity : class;
int Execute(string sql, object param = null, IDbContextTransaction trans = null);
TEntity ExecuteScalar<TEntity>(string sql, object param = null, IDbContextTransaction trans = null);
/// <summary>
/// 开启事务
/// </summary>
void BeginTransaction();
/// <summary>
/// 提交
/// </summary>
void Commit();
```

```
40  /// <summary>
41  /// 回滚
42  /// </summary>
43  void Rollback();
44  IDbConnection GetConnection();
45  /// <summary>
46  /// 切换主库
47  /// </summary>
48  /// <param name="type">0取配置 1自定义连接字符串</param>
49  /// <param name="dbType">0Sqlserver 1MySQL</param>
50  /// <param name="keyOrConnectString">配置主库只用传type，配置其他主库需要传配置k
    ey，自定义连接串需要传数据库连接字符串</param>
51  void ReadWrite(DbChangeTypeEnum type, DbTypeEnum? dbType = null, string? k
    eyOrConnectString = null);
52  /// <summary>
53  /// 切换从库
54  /// </summary>
55  /// <param name="type">0取配置 1自定义连接字符串</param>
56  /// <param name="dbType">0Sqlserver 1MySQL</param>
57  /// <param name="keyOrConnectString">配置主库只用传type，配置其他主库需要传配置k
    ey，自定义连接串需要传数据库连接字符串</param>
58  void Read(DbChangeTypeEnum type, DbTypeEnum? dbType = null, string? keyOrC
    onnectString = null);
```

⚡注意事项

1. EFCore .ToListAsync()需要引用using Microsoft.EntityFrameworkCore;
2. 在代码中，多次使用仓储IBaseRepository中EFCore的GetBykey方法获取到的是最新数据。

```
//整个过程很丝滑~
//正常获取
var course = _courseRepository.GetByKey("1");
course.Name = "第1次";
_courseRepository.Update(course);

//再获取
var course2 = _courseRepository.GetByKey("1");  //name :第1次
course2.Name = "第2次";
_courseRepository.Update(course);

//赋值更新
var course3 = course; //name :第2次
course3.Name = "第3次";
_courseRepository.Update(course3);

//再再获取
course3 = _courseRepository.GetByKey("1");//name :第3次
course3.Name = "第4次";
_courseRepository.Update(course3);

var course4 = _courseRepository.GetByKey("1"); //name:第4次
```

3. 使用IUnitOfWork中的更新和查询方法，和仓储EFCore中的数据不互通。

```
//efcore与efcore互通 sql与sql互通

var course = _courseRepository.GetByKey("1");
//efcore更新
course.Name = "第1次";
_courseRepository.Update(course);

//sql更新
Db.Execute("update course set name = '第二次' where id = '1'");

var course2 = _courseRepository.GetByKey("1"); // name:第1次  展示了efcore更新的值
var sqlname = Db.Query<string>("select name from course where id = '1'");// name:第二次 展示了sql更新的值
```

4. 使用efcore时，已获取到某id的实体对象，再new一个id一样的新对象（引用地址变了），更新时efcore会拦截报错

```
var course = _courseRepository.GetByKey("1");
var course2 = new Course
{
    Id = "1",
    Name = "测试"
    };
_courseRepository.Update(course2); //报错
```

5. 如果用不是AsNoTracking的方法，数据会放在DbContext的上下文中。这些数据更改字段值后，就算没有调用本实体Update方法，但是后面进行了其他包含Savechanges的操作（比如其他表进行增删改）就会更新。解决方法，如果想在代码里改数据的值而且不更新数据库：
   a. 这个数据值改变后不进行其他增删改的动作
   b. 在获取数据的时候用AsNoTracking
6. 如果同一张表EFCore和Dapper Sql语句混搭更新或删除的时候，就会报 The database operation was expected to affect 1 row(s), but actually affected 0 row(s); 的错误。【Mysql 不会出现这种情况，因为EFCore的并发令牌只针对Sqlserver】
   a. 报错情况1：

```
Db.BeginTransaction();
var course = _courseRepository.GetByKey("1");
Db.Execute("update course set name = '哈哈哈' where id = '1' ", null, Db.GetContextTransaction());
course.Title = "嘿嘿嘿";
_courseRepository.Update(course); //报错The database operation was expected to affect 1 row(s), but actually affected 0 row(s); data may have been modified or deleted since entities were loaded
Db.Commit();
```

   b. 报错情况2：

```
 Db.BeginTransaction();
 var course = _courseRepository.GetByKey("1");
 var enrollment = _enrollmentRepository.GetByKey("1");

 Db.Execute("update enrollment set name = '哈' where id = '1' ", null, Db.GetContextTransaction());
 enrollment.Name = "呀哈";

 course.Title = "嘿嘿嘿";
 _courseRepository.Update(course); //报错The database operation was expected to affect 1 row(s), but actually affected 0 row(s); data may have been modified or deleted since entities were loaded
 //如果用不是AsNoTracking的方法，数据会放在DbContext的上下文中。这些数据更改字段值后，就算没有调用本实体Update方法，但是后面进行了其他包含Savechanges的操作（比如其他表进行增删改）就会更新
 Db.Commit();
```

c. 解决办法1示例：

```
Db.BeginTransaction();
var course = _courseRepository.GetByKey("1");
Db.Execute("update course set name = '哈哈哈1' where id = '1' ", null, Db.GetContextTransaction());

Db.Context.Entry<Course>(course).Reload(); //调用此方法后，实体将处于 Unchanged 状态，除非实体不存在于数据库中，在这种情况下，实体将为 Detached。 最后，对数据库中不存在的 Added 实体调用 Reload 是一种不操作。 但请注意，已添加实体可能尚未创建其永久键值。

course.Title = "嘿嘿嘿1";
_courseRepository.Update(course);  //不报错
Db.Commit();
```

d. 解决办法2示例:

```
 Db.BeginTransaction();
 var course = _courseRepository.GetByKey("1");
 Db.Execute("update course set name = '哈哈哈2' where id = '1' ", null, Db.GetContextTransaction());

 Db.Context.LocalDetach<Course>(x => x.Id == "1");
 course = _courseRepository.GetByKey("1"); //需要重新获取下

 course.Title = "嘿嘿嘿2";
 _courseRepository.Update(course);  //不报错
 Db.Commit();
```

7. 占位

身份验证

引用 Microsoft.IdentityModel.Tokens Yotech.Sdk.Auth.Validate

```
builder.Services.AddAuth(opt => {
    opt.Authority = builder.Configuration.GetSection("SSO").Value;
    opt.RequireHttpsMetadata = false;
    opt.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateAudience = true,
        ValidAudiences = builder.Configuration.GetSection("SSOValidAudiences").Get<List<string>>(),
        ValidateLifetime = true,
        RequireExpirationTime = true,
        ClockSkew = TimeSpan.FromMinutes(5)
    };
});
```

```
  "SSO": "http://dev-sso.hnyotech.com/adfs",
  "SSOValidAudiences": ["https://test3-gateway.hnyotech.com.cn/recycleapi", "https://test3-gateway.hnyotech.com.cn/recycleapi/"]
```

需要token验证的时候请求对应环境的sso，然后把获取到的token放到header里就可以了

注册发现

引用 Yotech.Sdk.ServiceRegistry

```
builder.Services.AddServiceRegistry(builder.Configuration.GetSection("ServiceRegistry"));
```

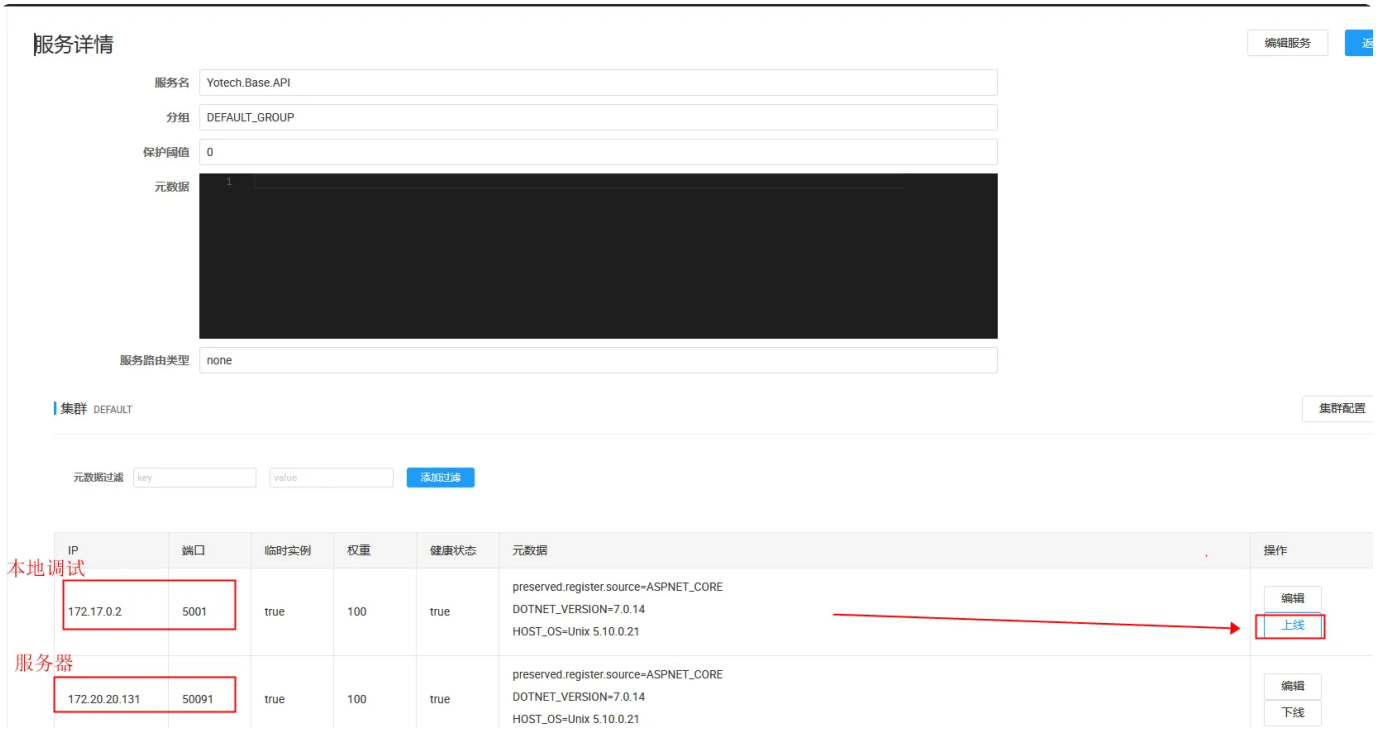
```
"ServiceRegistry": {
    "ServerAddresses": [ "http://172.20.10.204:58848/" ],//218.28.18.12
    "DefaultTimeOut": 15000,
    "Namespace": "dev",
    "ListenInterval": 1000,
    "ServiceName": "SdkTestAPI",
    "GroupName": "DEFAULT_GROUP",
    "ClusterName": "DEFAULT",
    "Weight": 100,
    "RegisterEnabled": true,
    "InstanceEnabled": true,
    "Ephemeral": true,
    "Secure": false,
    "UserName": "nacos",
    "Password": "nacos",
    "ConfigUseRpc": true,
    "NamingUseRpc": true,
    "LBStrategy": "WeightRandom" //WeightRandom WeightRoundRobin
  },
```

注意：

1. 成功注册后，会在列表展示一个实例，两个就是证明有两个实例都注册进来



2. 注册发现的ServiceName就是Nacos里的服务名（DataId），比如基础Api可以叫做Yotech.Base.API
3. 注册发现和网关的自动发现功能一起使用的时候，会根据ServiceName名字进行查询【具体看网关部分】
    a. **有时候需要在本地连其他环境的配置调试代码，但是可能会同时注册到对应环境的注册发现中，这个时候我们需要检查ip和端口，对本地的服务进行下线操作，下线后会显示上线按钮，这个时候就不参与网关的自动发现负载，不然会影响测试，当负载到本地ip时访问数据失败**



消息队列

引用Yotech.Sdk.MessageQueue

```
builder.Services.AddMessageQueue<RabbitMQHandler>(builder.Configuration.GetSection("RabbitMQ"));
```

```
"RabbitMQ": {
  "Host": "127.0.0.1",
  "Port": "5672",
  "UserName": "guest",
  "Password": "guest"
},
```

地址用原来的，配置里是我本地地址。

使用：

```C#
//通过控制器注入
private readonly IMessageQueue _mq;
public RabbitMQController(IMessageQueue mq)
{
    _mq = mq;
}

public bool Test()
{
    _mq.SetMessage("消息", "队列名称");
    return true;
}

```

短信邮件

引用 Yotech.Sdk.Message

```
builder.Services.Configure<SmsConfigOptions>(builder.Configuration.GetSection("SmsConfig"));
builder.Services.AddScoped<ISmsSender, SmsSender>();
builder.Services.Configure<EmailConfigOptions>(builder.Configuration.GetSection("EmailConfig"));
builder.Services.AddScoped<IEmailSender, EmailSender>();
```

```
"SmsConfig": {
    "IsSend": true,
    "Sn": " ",
    "Encode": "gb2312",
    "Pwd": " ",
    "ApiUrl": "https://sdks.entinfo.cn/webservice.asmx/mt"
  },
  "EmailConfig": {
    "IsSend": true,
    "SendPort": 465,
    "SendHost": "smtp.exmail.qq.com",
    "IsHtml": true,
    "SendUserName": "admin@hnprec.com",
    "SendUserPassword": "27n38aiY8JMbmv6P",
    "IsEnableSsl": true
  }
```

使用：

```C#
private readonly ISmsSender _sms;
private readonly IEmailSender _email;
public RabbitMQController(ISmsSender sms,IEmailSender email)
{
    _sms = sms;
    _email = email;
}

public bool Test()
{
    //接收人、邮件标题、内容
    _email.Send(new string[] { "watercolours@qq.com" }, "底层邮件封装测试", "内容" );
    //接收人、内容，后面不变
    _sms.Send(new string[] { "15537179630" }, "内容" ,null,"2");
}
```

全局用户信息

引用Yotech.Sdk.Auth.Validate

```
builder.Services.AddScoped<IUserIdentity,UserIdentity>();
builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
HttpContextHelper.ServiceCollection = builder.Services;
builder.Services.AddControllers(options =>
{
    options.Filters.Add<UserIdentityFilter>();
});
```

不用配置其他

使用（只要注入即可使用）：

```
private readonly IUserIdentity _userIdentity;
public CourseController(IUserIdentity userIdentity)
{
    _userIdentity = userIdentity;
}

 public Course FindUser()
{
    var auth = _userIdentity.UserId;
    var tradingOrgId= _userIdentity.TradingOrgId;

}
```

附件

1. 引用 Yotech.Sdk.Attach
2. 配置：

```
  "AttachOptions": {  //附件
    "SignExpiredSecond": 30,
    "MaxAttachSize": 0,
    "FFmpeg_Url": "",
    "IsSaveThumbPicture": true,
    "UploadDirectory": "C:\\Astec_Upload",
    "IsOpenWhitelist":false,//是否开启文件名后缀白名单(默认否
    "IsVerifyHash":false,//是否校验Hash重复（默认是
  },
  "QiniuConfigOptions": {  //七牛
    "AccessKey": "NVtpOzRswEhLx3E-esyKs85LrQDj00-st7tY_BYu",
    "SecretKey": "lujkTtk17LYbOBzUPPmb9yPWZyzBAAmUJqt0_6Oa",
    "Bucket": "yotech-test",
    "Expires": 7200
  },
```

3. 注入

```
builder.Services.Configure<AttachOptions>(builder.Configuration.GetSection("AttachOptions"));
builder.Services.Configure<QiniuConfigOptions>(builder.Configuration.GetSection("QiniuConfigOptions"));
builder.Services.AddScoped<IOSSHandler,QiniuHandler>();
builder.Services.AddScoped<IPdfHandler,MyPdfHandler>();
```

控制器从原来搬过来了，主要给前端调用，如果需要调整要跟前端沟通

MongoDB

1. 引用 Yotech.Sdk.NoSQL.MongoDB
2. 启动项

```
builder.Services.Configure<MongoConnectionOptions>(builder.Configuration.GetSection("MongoDBConfig"));
builder.Services.AddScoped(typeof(IMongoRepository<>), typeof(MongoRepository<>));
```

```
 "MongoDBConfig": {
    "Connection": "mongodb://172.20.20.16:27017",
    "DatabaseName": "test"
  }
```

3. 需要用的地方注入【如果像cms那样，多个表未定义实体但是又需要mongo进行操作的，可以用原来的IMongoRepository<BsonDocument> _mongo形式注入，初始化仓储时自动调用SetCollection】

```
private readonly IMongoRepository<Course> _mongo;
public CourseController(IMongoRepository<Course> mongo)
{
    _mongo = mongo;
}

```

4. 使用

```
//设置当前使用的表 【现在会在初始化仓储时自动调用这个，除非换库不然不需要了】
_mongo.SetCollection("表名");

//单条插入
await _mongo.InsertAsync(new Course { Id = "1", Credits = 101, Title = "语文" });
//如果是用bson的话可以这么写（cms有用到）
var mongoData = BsonDocument.Parse(mongoJson);
await _mongo.InsertAsync(mongoData);


//批量插入
var courses = new List<Course> { new Course { Id = "3", Credits = 301, Title = "科学" }, new Course { Id = "2", Credits = 201, Title = "数学" } };
await _mongo.BatchInsertAsync(courses);
//单条查询
var result = await _mongo.GetByKeyAsync("1");

//列表查询
var result = await _mongo.GetListAsync(x => x.Title.Length > 0);
var result = await _mongo.GetListAsync();

//单条更新
var result = await _mongo.UpdateAsync(new Course { Id = "1", Credits = 1022, Title = "chinese" });

//单条更新部分字段
var result = await _mongo.UpdatePartialAsync(new Course { Id = "2", Credits = 202, Title = "math" }, new List<string> { "Title" });

//获取数量
var result = await _mongo.CountAsync(x => x.UpdateTime == null);
var result = await _mongo.CountAsync();

//单条删除
var result = await _mongo.DeleteAsync(x => x.Id == "1");
//如果是用bson的话可以这么写（cms有用到）
var filter = Builders<BsonDocument>.Filter.Eq("Credits", 301);
await _mongo.DeleteAsync(filter);

//批量删除
 var result = await _mongo.BatchDeleteAsync(x => x.UpdateTime == null);
//如果是用bson的话可以这么写（cms有用到）
var filter = Builders<BsonDocument>.Filter.Eq("Credits", 301);
await _mongo.BatchDeleteAsync(filter);


```

```
44
45
```

切库

切库

`_mongo.SetDatabase("库名");`

切表

`_mongo.SetCollection("表名");`

切库的时候带上要查的表

`_mongo.Set(databaseName: "库名", tableName: "表名");`

示例

```
1   var data = await _stumongo.GetByKeyAsync("id1");
2
3   _mongo.SetDatabase("库1"); //切库1
4   _mongo.SetCollection("表1"); //库1的表1
5   var data = await _mongo.GetByKeyAsync("id2");
6
7   _mongo.Set(databaseName: "库2", tableName: "表2"); //相当于上面切库切表
8   var data = await _stumongo.GetByKeyAsync("id3");
```

缓存

注意：库用枚举Yotech.Sdk.Common.Enum.CacheKey_Enum区分

1. 引用 Yotech.Sdk.NoSQL.Cache
2. 启动项

```
1   builder.Services.Configure<RedisConnectionOptions>(builder.Configuration.Ge
    tSection("RedisConfig"));
2   services.AddScoped<ICacheHandler,RedisHandler>();
```

```
1   "RedisConfig": {
2     "Connection": "172.20.20.15:36301,allowadmin=true,password=1qaz2wsx~"
3    }
```

3. 需要用的地方注入

```
1  private readonly ICacheHandler _cache;
2  public CourseController(ICacheHandler cache)
3  {
4      _cache = cache;
5  }
6
```

4. 使用

```
#region string
/// <summary>
/// Cache添加key
/// </summary>
/// <param name="key">key</param>
/// <param name="value">值</param>
/// <param name="dbNum">缓存类型</param>
/// <param name="expiry">过期时间</param>
/// <returns></returns>
bool CacheSet<T>(string key, T value, int dbNum, TimeSpan? expiry = default(TimeSpan?));

/// <summary>
/// Cache添加key
/// </summary>
/// <param name="key">key</param>
/// <param name="value">值</param>
/// <param name="dbNum">缓存类型</param>
/// <param name="expiry">过期时间</param>
/// <returns></returns>
Task<bool> CacheSetAsync<T>(string key, T value, int dbNum, TimeSpan? expiry = default(TimeSpan?));

/// <summary>
/// 根据key获取value
/// </summary>
/// <typeparam name="T">泛型</typeparam>
/// <param name="key">key</param>
/// <param name="dbNum">数据库id</param>
/// <returns></returns>
T CacheGet<T>(string key, int dbNum);

/// <summary>
/// 根据key获取value
/// </summary>
/// <typeparam name="T"></typeparam>
/// <param name="key">key</param>
/// <param name="dbNum">数据库id</param>
/// <returns></returns>
Task<T> CacheGetAsync<T>(string key, int dbNum);

/// <summary>
/// 根据key集合获取value集合
/// </summary>
/// <typeparam name="T"></typeparam>
/// <param name="listKey">key集合</param>
/// <param name="dbNum">数据库id</param>
```

```
46  /// <returns></returns>
47  List<T> CacheGet<T>(List<string> listKey, int dbNum);
48  /// <summary>
49  /// 根据key集合获取value集合
50  /// </summary>
51  /// <typeparam name="T"></typeparam>
52  /// <param name="listKey">key集合</param>
53  /// <param name="dbNum">数据库id</param>
54  /// <returns></returns>
55  Task<List<T>> CacheGetAsync<T>(List<string> listKey, int dbNum);
56
57
58  #endregion
59
60  #region Hash
61  /// <summary>
62  /// Hash添加Key
63  /// </summary>
64  /// <typeparam name="T"></typeparam>
65  /// <param name="key">Key</param>
66  /// <param name="value">实体</param>
67  /// <param name="dbNum">数据库id</param>
68  /// <param name="expiry">过期时间</param>
69  /// <returns></returns>
70  Task<bool> HashSetAsync<T>(string key, T value, int dbNum, TimeSpan? expi
    ry = default(TimeSpan?));
71  /// <summary>
72  /// Hash根据key获取实体
73  /// </summary>
74  /// <typeparam name="T"></typeparam>
75  /// <param name="key">Key</param>
76  /// <param name="dbNum">数据库id</param>
77  /// <returns></returns>
78  Task<T> HashGetAllAsync<T>(string key, int dbNum);
79  /// <summary>
80  /// Hash根据key和field(字段名)获取value
81  /// </summary>
82  /// <param name="key">Key</param>
83  /// <param name="dbNum">数据库id</param>
84  /// <returns></returns>
85  Task<T> HashGetAsync<T>(string key, string field, int dbNum);
86
87  #endregion
88
89  #region List
90  /// <summary>
91  /// List 在指定索引处,添加一个值
92  /// </summary>
```

```
93  /// <typeparam name="T"></typeparam>
94  /// <param name="key">key</param>
95  /// <param name="value">值</param>
96  /// <param name="dbNum">数据库id</param>
97  /// <param name="index">索引</param>
98  /// <returns></returns>
99  Task<bool> ListSetByIndexAsync(string key, RedisValue value, long index,
    int dbNum, TimeSpan? expiry = default(TimeSpan?));
100
101 /// <summary>
102 /// List 向列表的右边追加值
103 /// </summary>
104 /// <typeparam name="T"></typeparam>
105 /// <param name="key">key</param>
106 /// <param name="value">值</param>
107 /// <param name="dbNum">数据库id</param>
108 /// <returns></returns>
109 Task<bool> ListRightPushAsync(string key, RedisValue value, int dbNum);
110
111 /// <summary>
112 /// List 获取列表最左边的值
113 /// </summary>
114 /// <typeparam name="T"></typeparam>
115 /// <param name="key">key</param>
116 /// <param name="dbNum">数据库id</param>
117 /// <returns></returns>
118 Task<RedisValue> ListLeftPopAsync(string key, int dbNum);
119 #endregion
120
121 #region Set
122 /// <summary>
123 /// Set 添加key值
124 /// </summary>
125 /// <param name="key"></param>
126 /// <param name="value"></param>
127 /// <param name="dbNum"></param>
128 /// <param name="expiry"></param>
129 /// <returns></returns>
130 Task<bool> SetAddAsync(string key, RedisValue value, int dbNum, TimeSpan?
     expiry = default(TimeSpan?));
131 /// <summary>
132 /// Set 根据key获取value
133 /// </summary>
134 /// <param name="key"></param>
135 /// <param name="dbNum">类型获取数据库id</param>
136 /// <returns></returns>
137 Task<RedisValue[]> GetSetMembersAsync(string key, int dbNum);
138
```

```
#endregion

#region Sorted Set

/// <summary>
/// Sorted Set  异步带权重的向有序列表插入单个元素
/// </summary>
/// <param name="key">key</param>
/// <param name="value">值</param>
/// <param name="dbNum">数据库id</param>
/// <param name="score">权重</param>
/// <param name="expiry">过期时间</param>
/// <returns></returns>
Task<bool> SortedSetAddAsync(string key, RedisValue value, double score,
int dbNum, TimeSpan? expiry = default(TimeSpan?));

/// <summary>
/// Sorted Set 获取集合所有元素，默认的排序时按权重从低到高
/// </summary>
/// <param name="key">key</param>
/// <param name="dbNum">数据库id</param>
/// <param name="order">排序</param>
/// <param name="start">权重上限</param>
/// <param name="stop">权重下限</param>
/// <returns></returns>
Task<SortedSetEntry[]> SortedSetRangeByScoreAsync(string key, int dbNum,
Order order, double start = double.NegativeInfinity, double stop = double
.PositiveInfinity);

/// <summary>
/// Sorted Set 异步删除指定键下的指定值
/// </summary>
/// <param name="key"></param>
/// <param name="value"></param>
/// <returns></returns>
Task<bool> SortedSetRemoveAsync(string key, RedisValue value, int dbNum);
#endregion

#region 删除 通用
/// <summary>
/// 根据Key清理指定的内容
/// </summary>
/// <param name="key">key</param>
/// <param name="dbNum">数据库id</param>
/// <returns></returns>
bool CacheRemove(string key, int dbNum);
/// <summary>
/// 根据Key清理指定的内容
```

```
184     /// </summary>
185     /// <param name="key">key</param>
186     /// <param name="dbNum">数据库id</param>
187     /// <returns></returns>
188     Task<bool> CacheRemoveAsync(string key, int dbNum);
189     /// <summary>
190     /// 根据Key,模糊删除
191     /// </summary>
192     /// <param name="key">key</param>
193     /// <param name="dbNum">数据库id</param>
194     /// <returns></returns>
195     bool CacheFuzzyRemove(string key, int dbNum);
196
197     /// <summary>
198     /// 删除某个Db Σ(っ°Д°;)  慎用！！
199     /// </summary>
200     /// <param name="dbNum">数据库id</param>
201     /// <returns></returns>
202     bool DeleteDb(int dbNum);
203     #endregion
```

定时任务

1. 引入Yotech.Sdk.TaskScheduler
2. 启动项

C#

```
1   services.AddTaskScheduler(hostcontext.Configuration.GetSection("JobConfig"
    ));
```

3. 配置

```
"JobConfig": {
    "DbType": 0,
    "Connection": "Server=172.20.20.21; database=Test; uid=sa; pwd=1qaz2wsx~;TrustServerCertificate=true;",
    "SsoUrl": "http://dev-sso.hnyotech.com/adfs",
    "ClientId": "11",
    "ApiUrl": "https://test-api.hnyotech.com/baseapi/",
    "ThreadCount": 10,
    "JobItems": [
      {
        "JobName": "同步服务",
        "JobGroup": "基础组",
        "AssemblyName": "Yotech.Sdk.Service.dll",
        "ClassName": "Yotech.Sdk.Service.WorkerTest",
        "CronExpression": "*/5 * * * * ?",
        "CronDescription": "5s执行一次",
        "JobDescription": "测试同步服务",
        "IsDeleted": false,
        "ConfigItems": [
          { "e": "r" }, //自定义配置项
          { "a": "b" }
        ]
      }

    ]

  }
```

内部方法应该基本无需修改，做了适配

性能日志

1. 启动项

```
builder.Services.AddControllers(options =>
{
    //性能统计
    if(builder.Configuration.GetValue("PerformanceOptions:IsOpen", false))
        options.Filters.Add<PerformanceFilter>();
});
builder.Services.Configure<PerformanceOptions>(builder.Configuration.GetSection("PerformanceOptions"));
```

2. 在appsettings.cs 里配置

```
  "PerformanceOptions": {
    "IsOpen": false, //是否开启
    "IsShowArgs": false, //是否展示入参
    "IsShowResult": false //是否展示出参
  },
```

日志

本次没有改动哦~出于性能（现在的写法比注入的性能高）和升级工作量的考虑，跟原来一样使用LogHelper即可。

工作流

⚡使用新框架必执行⚡

```
ALTER TABLE SysApproveActivityData ALTER COLUMN ApproveTime datetime2;
```

注意♨️：用1.5.0以后版本都需要初始化库，因为工作流新加的抄送活动加了区分抄送和审核的字段

用法

1. 引入Yotech.Sdk.Meta 、Yotech.WorkflowEngine.Web 、Yotech.Sdk.Data、Yotech.Sdk.NoSQL
2. 写个方法注入 private readonly IInitMataHandler _initMata;
3. 初始化调用请看目录【初始化数据库和工作流】
    a. 作用：会初始化所有打 [GenerateTable]的实体进MetaEntity 和 MetaAttribute
    b. 参数：填的是初始化的dll命名空间
4. program.cs 【DbContext,ApiHelper,Meta和Redis是必须注入的】

```
builder.Services.AddApiHelper(builder.Configuration.GetSection("ApiHelperConfig"));
builder.Services.AddScoped<IInitMataHandler, InitMata>();
builder.Services.AddScoped<ICacheHandler, RedisHandler>();
...
var app = builder.Build();
ServiceProviderHelper.Instance = app.Services;
...
```

5. appsetting.cs或者nacos配置

```
"ApiHelperConfig": {
    "SsoUrl": "https://dev-sso.hnyotech.com",
    "ClientId": "opqrst-hijklmn-abcdefg",
    "BaseApi": "https://dev-api.hnyotech.com/baseapi",
    "LoginName": "admin",
    "LoginPwd": "123456"
},
"RedisConfig": {
    "Connection": "172.20.20.15:36308,allowadmin=true,password=1qaz2wsx~"
}

```

6. 使用【方法和原来的一样，唯一区别的是EngineProxy原来传的是DBHelper,现在传IUnitOfWork】

```
//启动工作流
using (var engine = new EngineProxy(Db))
{
    engine.StartProcessByName("公告审核", "hnprec", "k22b1j3313p2014184acc7t", "hnprec");
}

//通过
 using (var engine = new EngineProxy(Db))
{
    engine.CompleteApproveWorkItem("k2358f0629lf008a7414c6m",  ApproveResultEnum.Pass, "通过原因");
}

//驳回
using (var engine = new EngineProxy(Db))
{
    engine.CompleteApproveWorkItem("k2358g0815cl008a77c306q", ApproveResultEnum.Reject,"驳回原因");
}

//是否最后一步
InstanceHelper.IsEnd(Db, "k235ab5353jf008a721aczu");
```

工作流其他管理端调用接口不变

7. 因为工作流执行是按脚本执行，所以工作流审核通过或驳回后使用EFCore查询数据时，会发现数据的状态没有变化。这时可以使用 isEnd方法，这个方法用来判断当前审核是否走到最后一步，是最后一步+ 传参审核结果是通过 = 数据审核通过。

```
var isEnd = InstanceHelper.IsEnd(Db, workitemid);
if(isEnd && param.ApproveResult) //然后进行通过或驳回的操作
{
    ...
}
```

签核活动

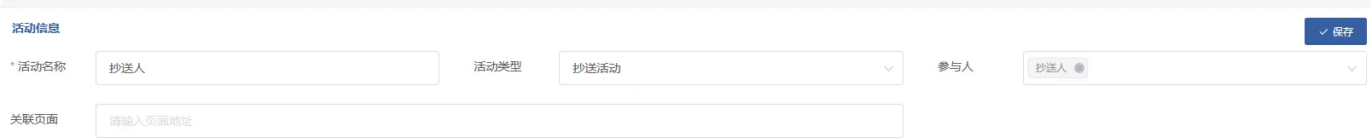


会签

配置：



抄送活动



配置的人会进行抄送，抄送活动在审核流的T_WorkItemBase表里存在，State为1（已完成），WorkItemType为1（抄送活动），OwnerId为抄送人

无人审核自动通过

配了查不到对应的人也可以自动通过。

抄送活动默认无人审核自动通过为true。

人脸识别

1. 引入 Yotech.Sdk.Common
2. 注入

```
builder.Services.Configure<FaceVerifyConfig>(builder.Configuration.GetSection("FaceVerifyConfig"));
builder.Services.AddScoped<IFaceVerify, BaiduFaceVerifyUtil>();
```

3. 配置

```
"FaceVerifyConfig": {
    "ClientId": "K3hXantgQ6GwDAqkLHE5Cxo8",
    "ClientSecret": "LGecOGiQFOEc0Vng4AGo7QdreXLYmpuo",
    "AuthAPI": "https://aip.baidubce.com",
    "BrainAPI": "https://brain.baidu.com",
    "ProjectPlanId": "12956",
    "ScoreThreshold": "80"
},
```

4. 使用注入IFaceVerify即可，方法和原来一样。具体可参照老Astec.Assets.Api项目HnPrecController的GetFaceVerifyH5Url和GetFaceVerifyResult方法。

```
/// <summary>
/// 生成AccessToken
/// </summary>
/// <returns></returns>
public AccessToken GetAccessToken();

/// <summary>
/// 生成VerifyToken
/// </summary>
/// <returns></returns>
public VerifyToken GetVerifyToken();

/// <summary>
/// 上传用户信息
/// </summary>
/// <returns></returns>
public SubmitUserInfoModel SubmitUserInfo(SubmitUserInfoParam param);

/// <summary>
/// 获取h5连接
/// </summary>
/// <returns></returns>
public string GetFaceVerifyH5Url(string verifyToken, string successUrl, string failUrl);

/// <summary>
/// 获取h5结果
/// </summary>
/// <returns></returns>
public bool GetFaceVerifyResult(string verifyToken);
```

OCR

1. 引入 Yotech.Sdk.Common
2. 注入

```
builder.Services.Configure<OcrConfig>(builder.Configuration.GetSection("OcrConfig"));
builder.Services.AddScoped<IOcr, OcrHelper>();
```

3. 配置

```
"OcrConfig" {
    "ApiKey": "lIyUp0IAPBnlzeGHrxzSyWdp",
    "SecretKey": "BIiujB4eCMg8V75OfUo1knRoW0soosIQ"
}
```

4. 使用注入IOcr即可，方法和原来一样。具体可参照老Astec.Base.Api项目的MemberApiController的IdCardDiscern方法。

```
/// <summary>
/// 创建ocr对象
/// </summary>
/// <returns></returns>
public Ocr GetOcr();

/// <summary>
/// 从文件中读取原始数据  要解密
/// </summary>
/// <param name="attach"></param>
public void SetFileData(v_attach attach);

/// <summary>
/// 身份证件的识别
/// </summary>
/// <param name="attach">附件（正面或背面的图片附件）</param>
/// <param name="cardSide"></param>
public OcrIdCardModel IdCard(v_attach attach, IdCardSide_Enum cardSide = IdCardSide_Enum.Front);

/// <summary>
/// 身份证件的识别
/// </summary>
/// <param name="attachs">附件，包含正面和国徽面两个附件</param>
public OcrIdCardModel IdCard(List<v_attach> attachs);

/// <summary>
/// 营业执照的识别
/// </summary>
/// <param name="attach">附件（正面或背面的图片附件）</param>
public OrcBusinessLicenseModel BusinessLicense(v_attach attach);
```

转PDF

1. 引入 Yotech.Sdk.Common

2. 注入

```
builder.Services.Configure<AttachOptions>(builder.Configuration.GetSection("AttachOptions"));
builder.Services.AddScoped<IPdfHandler, AsPdf2MQHandler>();
```

3. 配置

```
"AttachOptions": {
  "ToPdfQueueName": "Test.Assets.AsPdf"
},
```

4. 使用注入IPdfHandler即可，方法和原来一样

初始化数据库和工作流

1. 引用Yotech.Sdk.Init
2. program.cs加

JSON

```
builder.Services.AddScoped<IInitMataHandler, InitMata>();
```

3. 第一种用法：直接用postman调用
    a. 初始化工作流： /api/InitDb/InitMetadata?subsystemId=子系统Id 【⚡注意：子系统Id需要与基础数据库中配置的管理端子系统Id一致，权限也会用到这个】
    b. 初始化数据库：/api/InitDb/InitDb 【⚡注意：字段类型，如果相比原来的长度变长会改，变短不会改】
4. 第二种用法：注入 （可以看基础的feature/权限管理分支的BaseController，有使用）
    a. 初始化工作流：
        i. 注入 IInitMataHandler
        ii. 使用接口中的InitMetaEntityAsync("子系统Id")方法
    b. 初始化数据库：
        i. 注入 IInitEntityService
        ii. 使用接口中的InitDb()方法
5. 注意：如果项目API引用工作流的包，那么初始化会带上工作流所有实体表

网关

项目：002-yotech-gateway

这个只要区分普通转发和使用注册发现的两种配置就可以

用法1单纯转发

program.cs C#

```
...
//0.配置中心
builder.Configuration.AddConifgCenter(builder.Configuration.GetSection("ConfigCenter"));
//1.添加网关
builder.Services.AddOcelot();
//2.身份验证
builder.Services.AddAuth(opt =>
     {
          opt.Authority = builder.Configuration.GetSection("SSO").Value;
          opt.RequireHttpsMetadata = false;
          opt.TokenValidationParameters = new TokenValidationParameters
          {
              ValidateAudience = false
           };
      });
//3.配置跨域
builder.Services.AddCors(options =>
    {
          options.AddPolicy("CorsPolicy",
         builder => builder.AllowAnyOrigin()
         .AllowAnyMethod()
         .AllowAnyHeader());
    });
//4.网关注册到服务发现中心
//builder.Services.AddServiceRegistry(builder.Configuration.GetSection("ServiceRegistry"));
IdentityModelEventSource.ShowPII = true;
var app = builder.Build();

//配置跨域
app.UseCors("CorsPolicy");
app.UseSwagger();
app.UseSwaggerUI();
app.UseAuthorization();
app.MapControllers();
app.UseOcelot();
app.Run();

```

appsettings.cs JSON

```json
{
  "SSO": "https://dev-sso.hnyotech.com/adfs",
  "Routes": [
    {
      "DownstreamPathTemplate": "/todos/{id}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "jsonplaceholder.typicode.com",
          "Port": 443
        }
      ],
      "UpstreamPathTemplate": "/posts/{id}",
      "UpstreamHttpMethod": [
        "Get"
      ]
    },
    {
      "DownstreamPathTemplate": "/{businessparentid}api/api/{url}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "dev-api.hnyotech.com",
          "Port": 443
        }
      ],
      "UpstreamPathTemplate": "/{businessparentid}/api/{url}",
      "UpstreamHttpMethod": [ "Post", "Get" ],
      "Priority": 0
    },
    {
      "DownstreamPathTemplate": "/{url}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "node-exam-dev.hnyotech.com",
          "Port": 443
        }
      ],
      "UpstreamPathTemplate": "/Node/{url}",
      "UpstreamHttpMethod": [ "Post" ],
      "Priority": 0,
      "AuthenticationOptions": { //配置这个的就会验证
        "AuthenticationProviderKey": "Bearer",
        "AllowedScopes": []
```

```
        }
      }

    ],
    "GlobalConfiguration": {

    }
  }

```

用法2使用注册发现中心

program.cs C#

```
using Microsoft.IdentityModel.Logging;
using Microsoft.IdentityModel.Tokens;
using Microsoft.OpenApi.Models;
using Ocelot.DependencyInjection;
using Ocelot.Middleware;
using Ocelot.Provider.Nacos;
using Ocelot.Provider.Polly;
using System.Reflection;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
//0.配置中心
builder.Configuration.AddConifgCenter(builder.Configuration.GetSection("ConfigCenter"));

//1.配合服务发现中心做轮询
builder.Services.AddOcelot().AddNacosDiscovery("ServiceRegistry");
//2.身份验证
builder.Services.AddAuth(opt =>
     {
         opt.Authority = builder.Configuration.GetSection("SSO").Value;
         opt.RequireHttpsMetadata = false;
         opt.TokenValidationParameters = new TokenValidationParameters
         {
             ValidateAudience = false
             };
     });
//3.配置跨域
builder.Services.AddCors(options =>
    {
        options.AddPolicy("CorsPolicy",
        builder => builder.AllowAnyOrigin()
        .AllowAnyMethod()
        .AllowAnyHeader());
    });
//4.网关注册到服务发现中心
builder.Services.AddServiceRegistry(builder.Configuration.GetSection("ServiceRegistry"));
IdentityModelEventSource.ShowPII = true;
var app = builder.Build();

//配置跨域
app.UseCors("CorsPolicy");
```

```
app.UseSwagger();
app.UseSwaggerUI();
app.UseAuthorization();
app.MapControllers();
app.UseOcelot();
app.Run();
```

appsettings.cs JSON

```
{
  "SSO": "http://test3-sso.hnyotech.com.cn/adfs",
  "ServiceRegistry": {
    "ServerAddresses": [ "http://172.20.20.29:8848/" ],//218.28.18.12 //nacos地址 本地调试用218.28.18.12 发devdev服务器要用172.20.10.204
    "DefaultTimeOut": 15000,
    "Namespace": "test3",
    "ListenInterval": 1000,
    "ServiceName": "Yotech.Gateway.Web", //服务名
    "GroupName": "DEFAULT_GROUP",
    "ClusterName": "DEFAULT",
    "Weight": 100,
    "RegisterEnabled": true,
    "InstanceEnabled": true,
    "Ephemeral": true,
    "Secure": false,
    "UserName": "nacos",
    "Password": "nacos",
    "ConfigUseRpc": true,
    "NamingUseRpc": true,
    "LBStrategy": "WeightRandom", //WeightRandom WeightRoundRobin
    "Port": 50090
  },
  "Routes": [
    {
      "ServiceName": "Yotech.Base.API",  //注册发现的名字
      "DownstreamPathTemplate": "/{url}",
      "DownstreamScheme": "http",
      "UpstreamPathTemplate": "/baseapi/{url}",
      "UpstreamHttpMethod": [ "Post", "Get", "Options" ],
      "LoadBalancerOptions": {
        "Type": "RoundRobin"
      },
      // 是否使用服务发现
      "UseServiceDiscovery": true
    },
    {
      "ServiceName": "Yotech.Yolc.API",
      "DownstreamPathTemplate": "/{url}",
      "DownstreamScheme": "http",
      "UpstreamPathTemplate": "/yolcapi/{url}",
      "UpstreamHttpMethod": [ "Post", "Get" ],
      "LoadBalancerOptions": {
        "Type": "RoundRobin"
      },
```

```
        // 使用服务发现
        "UseServiceDiscovery": true
      },
      {
        "ServiceName": "Yotech.Indicator.API",
        "DownstreamPathTemplate": "/{url}",
        "DownstreamScheme": "http",
        "UpstreamPathTemplate": "/indicatorapi/{url}",
        "UpstreamHttpMethod": [ "Post", "Get" ],
        "LoadBalancerOptions": {
          "Type": "RoundRobin"
        },
        // 使用服务发现
        "UseServiceDiscovery": true
      }
    ],
    "GlobalConfiguration": {
      "ServiceDiscoveryProvider": {
        "Type": "Nacos"
      }
    }
  }
```

网关转发失败原因自查

如果遇到转发失败500 502 等等：

1. 检查网关是否启动，挂了没
2. 检查网关配置里是否有这个api的配置
3. 检查对应api是否启动，直接请求ip试试
4. 检查nacos管理页面 -> 服务管理-> 服务列表 是否有这个api
    a. 如果启动了但是服务发现没有就检查api注册发现的ServiceName配置

自动发布

1. 登录harbor http://172.20.20.36 admin 123456
    a. 新建项目 对应所在的发布环境
2. 登录jenkins http://172.20.20.36:8080/ admin 1qaz2wsx~
    a. 新建对应的视图 对应所在的发布环境
    b. 系统管理->系统配置->SSH Servers 添加新API服务器的地址，需要Test Configuration测试

c. 新建任务-> 构建一个自由风格的软件项目->复制 -> 输入 dev3-base-api

i. 需要改的地方（下面为页面配置标题）

1. GitLab Repository Name
2. 参数化构建过程 ->默认值
3. 源码管理 ->地址
4. Build Steps -> 执行 shell ->修改
5. Send files or execute commands over SSH ->改需要发布的ssh服务器，执行命令（Exec command）要改harbor环境和api名字

Exec command：以dev的baseapi举例 Shell

```
docker rm -f baseapi
docker rmi  172.20.20.36:80/dev3/baseapi
echo ---------------Launching Container...------------------
docker compose -f /home/yotech/dockercompose/baseapi.yaml up -d
```

3. 登录对应的要发布的新api服务器

a. /etc/docker/daemon.json 文件

```
{
    "insecure-registries":["172.20.20.36:80"]
}
```

b. 命令行输入

i. systemctl daemon-reload

ii. systemctl restart docker //重启docker

iii. gpasswd -a $USER docker //给用户docker权限

iv. newgrp docker //更新docker组

c. 在 /home/yotech/dockercompose/目录下建baseapi.yaml文件【注意按环境改挂载文件地址，按api改容器名字】

以base的yaml为例 YAML

```yaml
version: "3"
services:
  baseapi:
    image: 172.20.20.36:80/dev3/baseapi
    container_name: baseapi
    volumes:
      - /opt/nfs-file/Dev3:/opt/Dev3
    restart: always
    network_mode: host

```

4. 点击发布
5. 注意：docker用nuget官方包会因为网路波动出现拉取失败的情况，需要在dockerfile加下面几句话

```
RUN dotnet nuget remove source nuget.org
RUN dotnet nuget add source http://172.20.10.148:8081/repository/Yotech/
RUN dotnet nuget add source https://repo.huaweicloud.com/repository/nuget/v3/index.json
RUN dotnet nuget list source
```

模型校验

为了在进行校验的时候减少重复代码，在底层封装了公共的方法。

请求传参校验

这个会校验控制器接口的传参：

1. 在program.cs加配置

```
builder.Services.AddControllers(options =>
{
    options.Filters.Add<ValidateModelFilter>();

}).ConfigureApiBehaviorOptions(options =>
{
    options.SuppressModelStateInvalidFilter = true;
})
```

2. 在实体类中加验证和报错信息，以下只是常用具体，System.ComponentModel.DataAnnotations的相关验证都可以用

```
    /// <summary>
    /// 名称
    /// </summary>
    [MaxLength(200, ErrorMessage = "名称不可超过200字")]
    public string? Name { get; set; }

    /// <summary>
    /// 接口类型
    /// </summary>
    [Required(ErrorMessage = "接口类型不可为空")]
    public InterfaceType_Enum InterfaceType { get; set; }
```

结果长这样：

```
{"ErrorCode":1,"Message":"Name:名称不可超过200字","Data":null}
```

其他情况

比如有我们自己做导入导出时需要校验，但是模型是在请求发生后建的，可以调用

`ValidateModelHelper.Validate(校验的实体);`

这个也会根据实体是否符合验证进行报错。

NLP

1. 引入 Yotech.Sdk.Common
2. 注入

```
builder.Services.Configure<NlpConfig>(builder.Configuration.GetSection("NlpConfig"));
builder.Services.AddSingleton<INaturalLanguageProcess, BaiduNlpUtil>();
Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);
```

3. 配置

```
"NlpConfig": { //nlp
    "AuthApi": "https://aip.baidubce.com/oauth/2.0/token",
    "ApiUrl": "https://aip.baidubce.com/rpc/2.0/nlp/v2/simnet",
    "ApiKey": "33mGnHUnxJiNmiQ5PBtLYgUi",
    "SecretKey": "htoIw5IEClzqVaKI1a6G0AJI4kPgczlQ"
}
```

4. 使用注入INaturalLanguageProcess即可。

```
/// <summary>
/// nlp接口
/// </summary>
public interface INaturalLanguageProcess
{
    /// <summary>
    /// 获取短文本相似度（最大512字节）
    /// </summary>
    /// <param name="sourceText">源文本</param>
    /// <param name="targetText">目标文本</param>
    /// <returns>相似度0~1</returns>
    Task<decimal> GetTextSimnetScore(string sourceText, string targetText
);

}
```

```
//举例
public async Task<decimal> NLP()
{
    var score = await _nlpservice.GetTextSimnetScore("爸爸的爸爸是爷爷", "妈妈
的妈妈是姥姥");
    return score;
}
```