

# Plinq NH 后台框架中常见问题和解决办法 V1.2

## 目录

Plinq NH 后台框架中常见问题和解决办法 V1.2.....	1
1. 简单 N+1 问题 .....	1
2. 复杂 N+1 问题 .....	3
3. 合理选用事务.....	5
4. 大粒度事务.....	6
5. 事务中的异常处理 .....	7
6. 数据库同步的滥用 .....	8
7. 冗余的数据库访问 .....	9
8. 使用不当的 Linq 排序.....	10
9. 查询优化 .....	11
10. 检索条件转移 .....	13
11. Linq 的过度依赖.....	14
12. 不安全代码.....	15
13. 错误的数据过滤点.....	17
14. 错误的 NULL 返回.....	20
15. 错误的数据格式化形式 .....	20
16. 合理的定义 Bo 中属性类型和属性名称.....	23
17. 序列化数据冗余 .....	24
18. AutoMapper 的正确使用.....	27
19. Json.Net 的正确使用.....	30
20. Nhibernate profile 的使用.....	31
21. 低性能的全表连接.....	31

## 1. 简单 N+1 问题

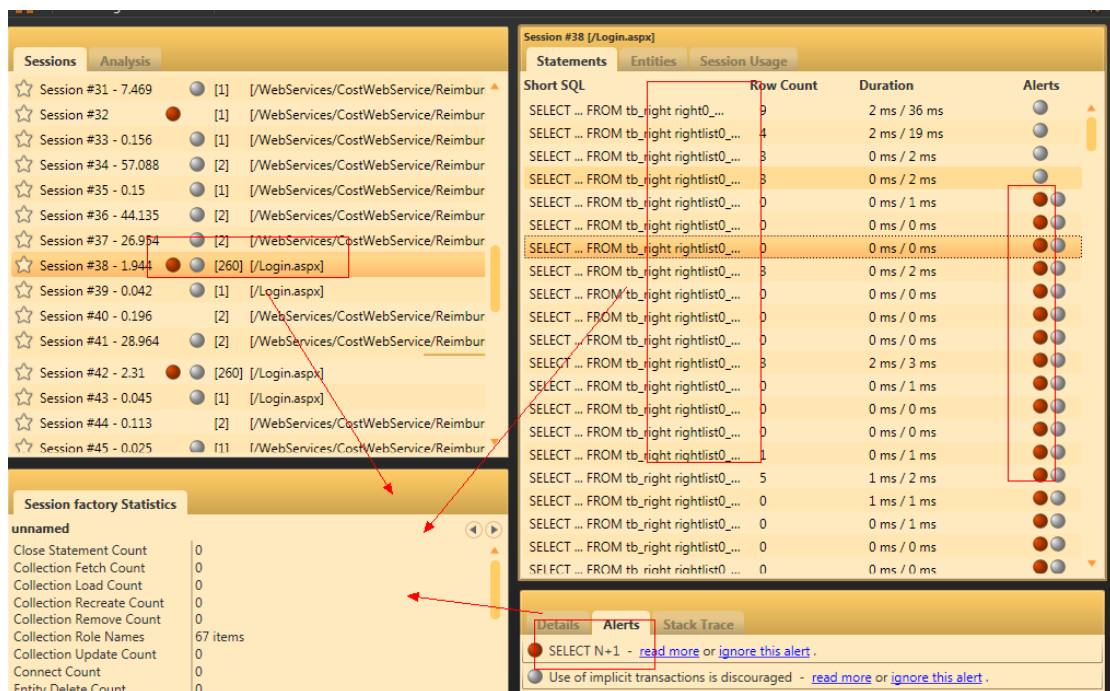
这个问题一般是过量查询，属于一个 Session 中产生太多查询语句

```

/// <param name= rgids />param/
/// </returns></returns>
private List<RightBO> GetRightBoList(IEnumerable<Right> pList, List<long> rgIds)
{
    var mList = new List<RightBO>();
    if (pList != null)
    {
        foreach (Right bo in pList.Where(p => rgIds.Contains(p.RgId)))
        {
            mList.Add(new RightBO
            {
                Name = bo.Name,
                RgId = bo.RgId,
                Code = bo.Code,
                Url = bo.Url,
                RightBoList = GetRightBoList(bo.RightList, rgIds)
            });
        }
    }
    return mList;
}
/// </summary>

```

递归查询，NH 内部用了延迟加载，递归的话，一定会产生大量的 sql  
Profile 查看



一共产生了 260 条语句，提示有两类问题，未显示声明事务，N+1 查询，N+1 说的够多了，不多说。

这个权限加载的方式，目前认定存在较大问题。  
业务需求是，根据权限，加载该角色的对应的权限树。

在业务层通过方法去递归访问数据库加载权限树，会产生大量的 sql，每一行 sql 执行都会采用隐式事务处理。严重的低性能。

怎么解决呢？

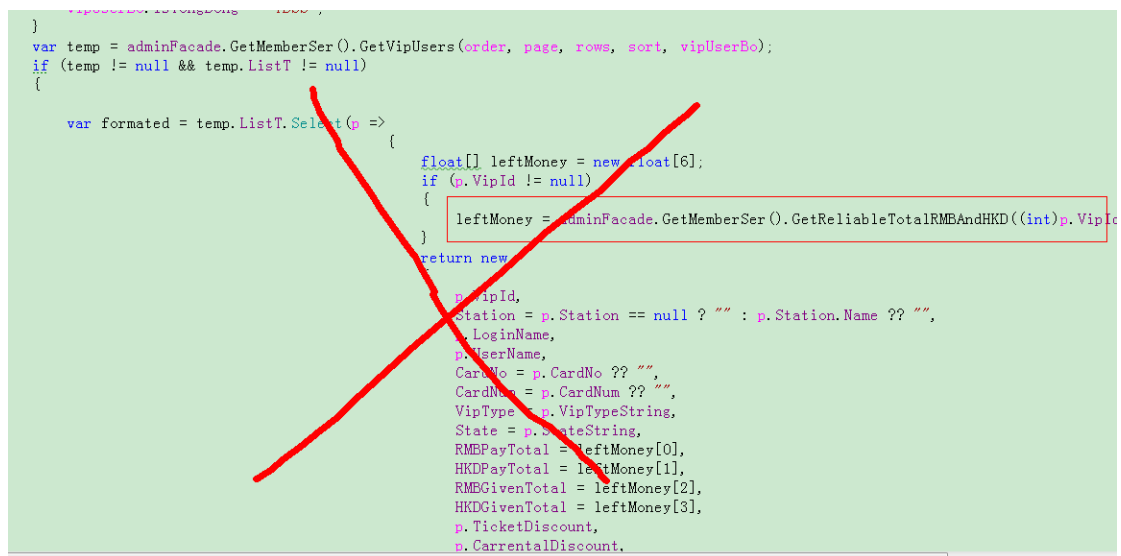
1. 在业务层来达到这个目的代价有点大，递归不可避免，不排除有巧妙的办法。探索出较优方案的请找我向我反馈。

2. 用存储过程, 在 db 一次性处理好了返回。具体是, Sp 暴露一个角色 id 或者角色 name,。从角色权限表筛选角色对应的权限数据, 然后再与权限表关联, 用自然连接, 权限 id 作为连接条件, 取出数据以后, 按照 parentId(模块编号)进行升序排序操作, 将树结构的权限树, 水平拉伸成线性结构, 然后按照 parentId 依次加载。以上所有过程只需要一次数据库访问。不会产生递归。而且理论上应该可行

Tips, 慎用递归调用,可能造成资源过度消耗, 线程同步和死锁问题

## 2. 复杂 N+1 问题

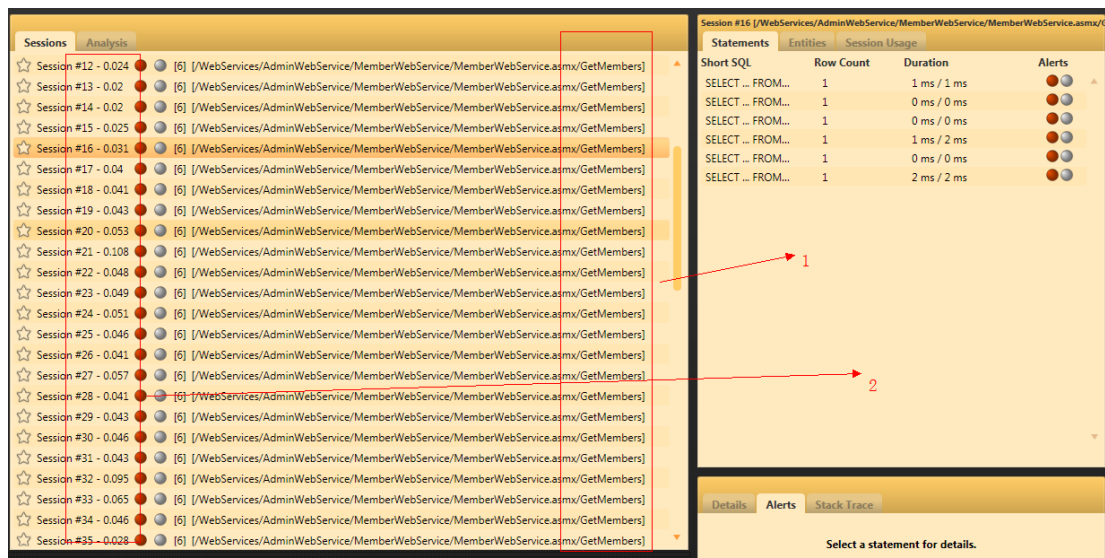
这个问题是非常严重的一类问题。也不多说了。简单说下



循环中进行数据库的访问, 并且访问的是独立的方法。

Ok, 一页数据, 有多少条数据就产生多少 Session,

用 Nh profile 查看, 不堪入目。



一处可以看到这些 session 来自同一个方法，2 处看到产生了整整一页数据长度的 Session

太多的 Session，太多的 Sql 语句。

解决办法，如何考虑在业务层实现，通常可以这么做

```
/// <returns>返回售出订单实体</returns>
public SoldOrder GetAllOrderForUser(string username, DateTime startDate, DateTime endDate)
{
    SoldOrder soldOrder = new SoldOrder();
    using (var context = new MiniSysDataContext())
    {
        List<TicketLineOrderBo> ticketlineorders = GetTicketLineOrdersForUser(context, username, startDate, endDate);
        List<CarrentalOrderBo> carrentalorders = GetCarrentalOrdersForUser(context, username, startDate, endDate);
        List<BagTicketBo> bagTicketBos = GetBagTicketOrdersForUser(context, username, startDate, endDate);
        List<FreeorderBo> freeorderBos = GetFreeOrdersForUser(context, username, startDate, endDate);
        soldOrder.TicketLineOrderBos = ticketlineorders;
        soldOrder.CarrentalOrderBos = carrentalorders;
        soldOrder.BagTicketBos = bagTicketBos;
        soldOrder.FreeorderBos = freeorderBos;
    }
    return soldOrder;
}
```

提取 context 数据库上下文实例作为参数，子方法设置一个 context 参数。多个独立子方法公用 context 进行数据获取，可削减 Session 的产生个数。父方法外层使用 using 创建 context，内层做到 context 的公用。并在父方法上做异常处理。

但是，带来的弊端 是，方法带来了耦合性升高，重用性降低

考虑到性能又怎么解决呢？考虑到有带参搜索，用存储过程。

经过，两类 N+1 问题表明，处理复杂数据查询，如果在业务层实现代价过大。请直接考虑用存储过程实现。可兼顾成本和性能。

### 3. 合理选用事务

```
using (var context = new MiniSysDataContext())
{
    try
    {
        if (ticketLineOrder.PaymentType == 3)
        {
            int hangingUser = context.Vipuser.ByLoginName(ticketLineOrder.CreatedBy).ByIsAccount(1).Count();
            if (hangingUser > 0)
            {
                ticketLineOrder.IsHanging = 1;
            }
        }
        //订单新增业务处理开始
        context.BeginTransaction();
        context.Ticketlineorder.InsertOnSubmit(ticketLineOrder); //订单写入
        //提交下, 写入订单
        context.SubmitChanges();
        //提交下, 写入车票, 注意订单是车票的外键, 应该先写入 (切记不要删除, 下面的车票写入方案有问题, 后期会重构)
        // context.Orderticket.InsertAllOnSubmit(orderTickets);
        //context.SubmitChanges();
        //消费记录写入(modified by ys on 2013-9-28, 按折扣价格扣款)
        var recordResult = service.AddConsumerRecordForVipAccountPay(context, ticketLineOrderBo.CreatedBy,
                                                                    ticketLineOrderBo.RealOrderPrice,
                                                                    ticketLineOrderBo.CurrencyType, 1, ticketLineOrderBo.OrderNum);

        //更改订单支付状态
        var stateChangeResult = ChangeOrderPayState(context, ticketLineOrderBo.OrderNum, 1);
        if (recordResult && stateChangeResult)
        {
            context.CommitTransaction();
        }
    }
}
```

这里有一个 using, 开启了一次 db 访问

完成的业务处理是, 生成订单并写入, 写入消费数据, 修改订单状态

还有一个流程没有走, 那就是写入订单关联的车票数据, 写入车票关联的座位

```
131
132         return false;
133     }
134     #region 车票写入, 调整车票写入进事务, 暂不删除
135
136     using (var context = new MiniSysDataContext())
137     {
138         try
139         {
140             //准备车票基础数据
141             var banciSeats = new List<Linebanciseat>();
142
143             //生成去成票
144             //生成返程票
145
146             context.BeginTransaction();
147             context.Orderticket.InsertAllOnSubmit(orderTickets);
148             context.Linebanciseat.InsertAllOnSubmit(banciSeats);
149             context.CommitTransaction();
150
151             catch (Exception ex)
152             {
153                 context.RollbackTransaction();
154                 LogHelper.WriteLog(string.Format("TicketLineOrderService.AddTicketLineOrder({0})生成车票异常",
155                 return false;
156             }
157         }
158     }
159 }
```

戏剧性的是, 车票和座位号的写入是单独又开了一次数据库连接, 按事务写入

下单这个业务, 要写入的数据多, 订单数据, 消费数据, 车票数据, 座位数据, 并且要更改订单的支付状态。这一连串的操作, 应该放在一个数据库访问里, 放在同一个事务里操作。这里不仅仅分开成两次数据库访问, 还分成了两个事务, 会有问题么?

应用在复杂的生产环境中, 一定是会出问题的。

需要元子级的事务操作的业务, 慎用分开操作。因为无法保证业务出异常以后, 整个数据库操作的回滚。

## 4. 大粒度事务

```
ticketLineOrder = context.Ticketlineorder.GetByKey(ticketLineOrder.OrderId);
context.BeginTransaction();
var orderTickets = new List<Orderticket>();
Orderticket orderTicketTemp = new Orderticket();
orderTicketTemp.CurrencyType = ticketLineOrder.CurrencyType;
//orderTicketTemp.IsAppend = 1; //非补客
orderTicketTemp.IsRefund = 1; //未退票
orderTicketTemp.IsOut = 1; //未出票
orderTicketTemp.IsDelete = 0; //未删除
orderTicketTemp.AuditState = 1; //未核票
orderTicketTemp.CreatedOn = DateTime.Now;
orderTicketTemp.CreatedBy = ticketLineOrder.CreatedBy;
orderTicketTemp.ModifiedOn = orderTicketTemp.CreatedOn;
orderTicketTemp.ModifiedBy = orderTicketTemp.CreatedBy;
生成去成票
生成返程票
context.Orderticket.InsertAllOnSubmit(orderTickets);
context.CommitTransaction();
}
catch (Exception ex)
{
    context.RollbackTransaction();
    LogHelper.WriteLog(string.Format("TicketLineOrderService.AddTicketLineOrder({0})生成车票异常", ticketLineOrderBo), ex);
    return false;
}
return true;
```

本处有超过50行的代码

本处事务开始

真正的批量操作在这里

事务粒度过大，事务开启，会有表锁，上面的代码，表锁一定时间较长，这种锁持续时间应该尽可能的短。

你应该尽可能的缩小事务的粒度(事务块应该尽可能的小)，避免将非数据库批量操作，比如业务层的创建对象，赋值，循环等放到事务块里。

你应该

```
#endregion

//订单业务处理开始
context.BeginTransaction();
context.Ticketlineorder.InsertOnSubmit(ticketLineOrder); //订单写入
//提交下，写入车票，注意订单是车票的外键，应该先写入
context.Orderticket.InsertAllOnSubmit(orderTickets);
context.Linebanciseat.InsertAllOnSubmit(banciSeats);
//context.SubmitChanges();
context.CommitTransaction();
}
```

1 开启事务，

2 写入父亲，写入订单数据

3 写入儿子，写入车票

4 写入儿子，写入座位数据

- 1 开启事务，
  - 2 写入父亲，写入订单数据
  - 3 写入儿子，写入车票
  - 4 写入儿子，写入座位数据
- 再提交，所有处理业务放到事务外，合理的降低事务粒度

因框架所限，有些大粒度事务是必须的，而且也是合理的，如

```

using (var context = new MiniSysDataContext())
{
    try
    {
        context.BeginTransaction();
        context.Applyrecord.InsertOnSubmit(entity);
        context.SubmitChanges(); //先Submit下产生报销单
        foreach (var applydetailBo in applyRecordBo.ApplydetailBoList)
        {
            Applydetail singleDetail = new Applydetail();
            singleDetail.Applyitem =
                context.Applyitem.FirstOrDefault(p => p.ItemId == applydetailBo.ItemId.GetValueOrDefault());
            singleDetail.Applyrecord = entity;
            singleDetail.CreatedOn = entity.CreatedOn;
            singleDetail.CreatedBy = entity.CreatedBy;
            singleDetail.Count = applydetailBo.Count;
            singleDetail.Price = applydetailBo.Price;
            details.Add(singleDetail);
            recordTotalPrice = recordTotalPrice +
                (singleDetail.Count.GetValueOrDefault()) * (singleDetail.Price.GetValueOrDefault());
        }
        entity.ApplydetailList = details;
        entity.Price = recordTotalPrice; //提交事务产生报销详情记录
        context.CommitTransaction();
        result = true;
    }
    catch (Exception e)
    {
        context.RollbackTransaction();
        result = false;
        LogHelper.WriteLog(string.Format("ApplyRecordService.AddApplyRecord({0}), 事务错误提交过程中产生错误", applyRec

```

1 处开启事务

2 处提交父数据

3 那个循环体是准备子数据用的，这里准备子数据要进行 db 查询。

在有一片分享中我提到过循环中的数据库访问造成 N+1 问题，也许你认为这是 N+1，但是请注意这个循环在同一个 context 下，所以这个不是 N+1 问题，更不是复杂的 N+1，相反，这么做是我目前能想到的是消耗最小的办法

4 处，关联已经是持久态的父亲对象，  
接下来提交事务。父子数据可顺利写入。

## 5. 事务中的异常处理

```

using (var context = new MiniSysDataContext())
{
    try
    {
        context.BeginTransaction();
        context.Applyrecord.InsertOnSubmit(entity);
        context.SubmitChanges(); //先Submit下产生报销单
        foreach (var applydetailBo in applyRecordBo.ApplydetailBoList)
        {
            Applydetail singleDetail = new Applydetail();
            singleDetail.Applyitem =
                context.Applyitem.FirstOrDefault(p => p.ItemId == applydetailBo.ItemId.GetValueOrDefault());
            singleDetail.Applyrecord = entity;
            singleDetail.CreatedOn = entity.CreatedOn;
            singleDetail.CreatedBy = entity.CreatedBy;
            singleDetail.Count = applydetailBo.Count;
            singleDetail.Price = applydetailBo.Price;
            details.Add(singleDetail);
            recordTotalPrice = recordTotalPrice +
                (singleDetail.Count.GetValueOrDefault()) * (singleDetail.Price.GetValueOrDefault());
        }
        entity.ApplydetailList = details;
        entity.Price = recordTotalPrice; //提交事务产生报销详情记录
        context.CommitTransaction();
        result = true;
    }
    catch (Exception e)
    {
        context.RollbackTransaction();
        result = false;
        LogHelper.WriteLog(string.Format("ApplyRecordService.AddApplyRecord({0}), 事务错误提交过程中产生错误", applyRec

```

所有存在较大异常可能的代码，都应该用 try 块进行异常处理。

Using 开启数据库访问块内部，进行数据库事务操作，必须进行异常处理，保证 commit 和 rollback 的配对使用。保证出现异常的时候，能回滚操作。

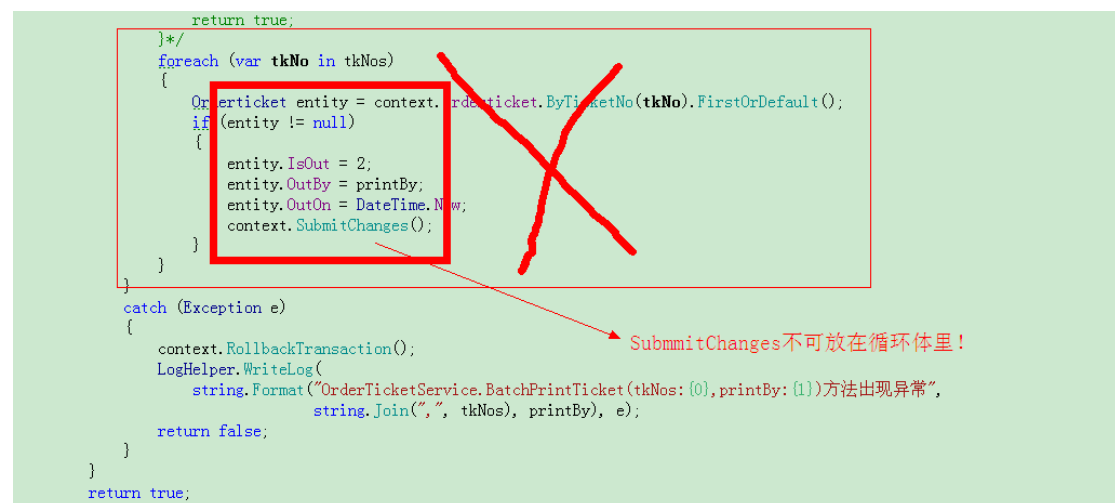
Tips:对于多表操作和批量操作，Commit 和 rollback 必须配对使用，

在 using 外层还用不用 try 块呢。我个人强烈建议使用，外层有 AutoMapper 的映射操作。尤其还有 using 这个创建非托管资源的操作。

都是有非常大的可能会抛异常的。

有人说 try 耗性能，没错。但是，牺牲一点性能来确保业务处理的可靠性，我觉得这是值得的。

## 6. 数据库同步的滥用



```
return true;
}*/
foreach (var tkNo in tkNos)
{
    OrderTicket entity = context.OrderTicket.ByTicketNo(tkNo).FirstOrDefault();
    if (entity != null)
    {
        entity.IsOut = 2;
        entity.OutBy = printBy;
        entity.OutOn = DateTime.Now;
        context.SubmitChanges();
    }
}
}
catch (Exception e)
{
    context.RollbackTransaction();
    LogHelper.WriteLog(
        string.Format("OrderTicketService.BatchPrintTicket(tkNos: {0}, printBy: {1})方法出现异常",
            string.Join(", ", tkNos), printBy), e);
    return false;
}
}
return true;
```

SubmitChanges不可放在循环体里!

慎用在循环中进行数据库同步。原因不多说



## 7. 冗余的数据库访问

```
/// <param name="carNo">车牌号</param>
/// <param name="driverNo">司机工号</param>
/// <returns></returns>
[ScriptMethod]
[WebMethod]
[SoapHeader("htSoapHeader", Direction = SoapHeaderDirection.InOut | SoapHeaderDirection.Fault)]
public string CheckCarNoDriverNo(string carNo, string driverNo)
{
    if (htSoapHeader.ValidateUser(htSoapHeader.UserName, htSoapHeader.PassWord))
    {
        bool t1 = adminFacade.GetService().IsCarExist(carNo);
        if (!t1)
        {
            //如果该车牌号不存在
            return new JavaScriptSerializer().Serialize(new { result = 1 });
        }
        bool t2 = adminFacade.GetService().IsEmployeeExist(driverNo);
        if (!t2)
        {
            //如果该司机工号不存在
            return new JavaScriptSerializer().Serialize(new { result = 2 });
        }
        return new JavaScriptSerializer().Serialize(new { result = 3 }); //验证通过
    }
    return null;
}
```

分别想判断车牌存在和司机存在, 这里最佳做法是压缩数据库访问请求, 提取到一个方法(置入两个参数即可)里面, 减少远程的数据库访问, 原因不多说。

AdminFacade.GetService().IsExist(string carNo, string driverNo)

返回布尔数组, 取出判断即可、

```
//订单业务处理开始
context.BeginTransaction();
context.TicketLineorder.InsertOnSubmit(ticketLineOrder); //订单写入
//提交下, 写入车票, 注意订单是车票的外键, 应该先写入
context.OrderTicket.InsertAllOnSubmit(orderTickets);
context.LineBanciseat.InsertAllOnSubmit(banciSeats);
context.SubmitChanges();
context.CommitTransaction();
```

这种事务提交前的 SubmitChanges 是不必要的, 因为事务的 commit 内部会先 Submit, 前面那个 summitchange 造成了一次不必要的数据库同步

而有些事务中的 SubmitChanges 又是必须的, 如

```

#endregion
using (var context = new MiniSysDataContext())
{
    try
    {
        context.BeginTransaction();
        context.Applyrecord.InsertOnSubmit(entity);
        context.SubmitChanges(); //先Submit下产生报销单
        foreach (var applydetailBo in applyRecordBo.ApplydetailBoList)
        {
            Applydetail singleDetail = new Applydetail();
            singleDetail.Applyitem =
                context.Applyitem.FirstOrDefault(p => p.ItemId == applydetailBo.ItemId.GetValueOrDefault());
            singleDetail.Applyrecord = entity;
            singleDetail.CreatedOn = entity.CreatedOn;
            singleDetail.CreatedBy = entity.CreatedBy;
            singleDetail.Count = applydetailBo.Count;
            singleDetail.Price = applydetailBo.Price;
            details.Add(singleDetail);
            recordTotalPrice = recordTotalPrice +
                (singleDetail.Count.GetValueOrDefault()) * (singleDetail.Price.GetValueOrDefault());
        }
        entity.ApplydetailList = details;
        entity.Price = recordTotalPrice; //提交事务产生报销详情记录
        context.CommitTransaction();
        result = true;
    }
    catch (Exception e)
    {
        context.RollbackTransaction();
        result = false;
        LogHelper.WriteLog(string.Format("ApplyRecordService.AddApplyRecord({0}), 事务错误提交过程中产生错误", applyRecordBo),
    }
}

```

这个是父子表单类型的处理，在子数据写入前，需要保证父数据已经写入，但是，这些写入操作又必须纳入一个事务中，那么，就需要在父数据写入以后，SubmitChanges 来确保父数据写入。

## 8. 使用不当的 Linq 排序

Net 平台的 Linq 技术以其优雅和简洁的使用特性，颇受人的欢迎，而且 Linq 提供了不少统一查询接口，比如排序，查找，删除，添加等操作。但是排序往往会被误用和错用

使用 Linq 排序很方便，但是大数据集合应该慎用。看代码

```

using (var context=new MiniSysDataContext())
{
    carstartlists=context.Carstartlist.Where(p=>p.LineName.Contains(lineName)).Where(p=>p.StartDate==time.Date)
        .ToList();
    carstartlists = carstartlists.Where(p =>
    {
        DateTime banciTime = DateTime.Parse(p.BanCi);
        bool isManzu = (time.AddHours(-ago) <= banciTime) &&
            (time <= banciTime.AddHours(later));
        if (isManzu)
        {
            return true;
        }
        else
        {
            return false;
        }
    }).ToList();
    returnBos=carstartlists.Select(p => new CheckTicketByCarStartlistBo()
    {
        LineName = p.LineName,
        CslId = p.CslId,
        BanciDate = p.StartDate.GetValueOrDefault().ToString("yyyy-MM-dd"),
        BanciTime = p.BanCi,
        CarNo = p.CarNo,
        DriverNo = p.DriverNo,
        CheckCounts = p.Passenger.GetValueOrDefault()
    }).OrderByDescending(p=>p.BanciTime).ToList();
}

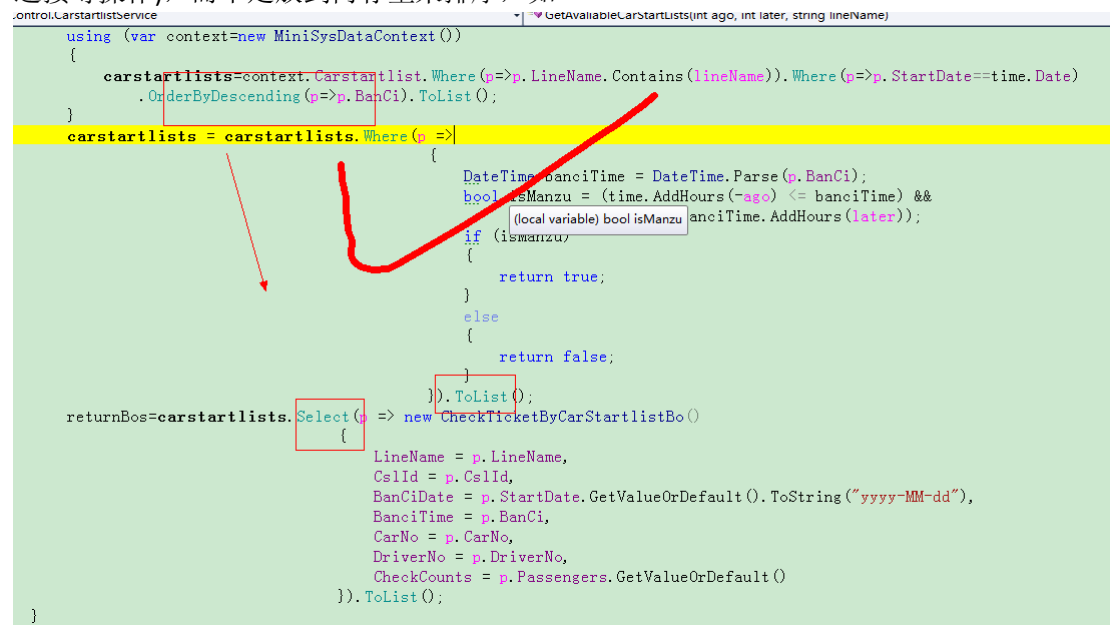
```

1 处, Linq 表达式树形式的查询, 动态的创建表达式树, 然后转换为 sql, 是跟数据库打交到的 linq 操作。实际上底层是用的 Nhibernate, Nhibernate 现在已经提供了对 linq 的语法支持。所以 Plino NH 这东西既然是对 NH 的封装。那么也自然会 linq 那简洁的语法了。

2, 3, 4, 集合数据都已经加载到内存中, 相关的筛选和排序操作都在内存里。

我想说明的是 4 处的代码。这是不合理的 Linq 排序操作。在内存中进行排序操作, 不仅吃内存, 并且耗时。

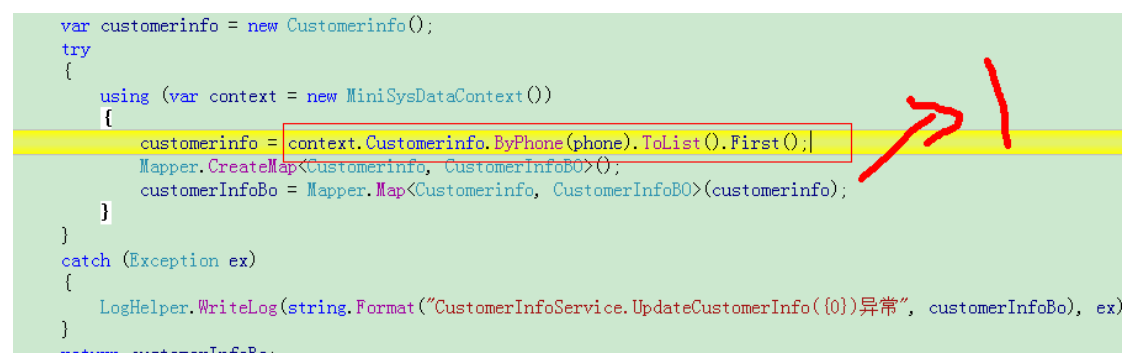
最佳的办法是, 将排序操作, 交给数据库去做, 关系型数据库它们擅长干这类事情(排序, 连接等操作), 而不是放到内存里来排序, 如



```
using (var context=new MiniSysDataContext())
{
    carstartlists=context.Carstartlist.Where(p=>p.LineName.Contains(lineName)).Where(p=>p.StartDate==time.Date)
    .OrderByDescending(p=>p.BanCi).ToList();
}
carstartlists = carstartlists.Where(p =>
{
    DateTime banCiTime = DateTime.Parse(p.BanCi);
    bool isManzu = (time.AddHours(-ago) <= banCiTime) &&
    (local variable) bool isManzu banCiTime.AddHours(later));
    if (ismanzu)
    {
        return true;
    }
    else
    {
        return false;
    }
}).ToList();
returnBos=carstartlists.Select(p => new CheckTicketByCarStartlistBo()
{
    LineName = p.LineName,
    CslId = p.CslId,
    BanCiDate = p.StartDate.GetValueOrDefault().ToString("yyyy-MM-dd"),
    BanCiTime = p.BanCi,
    CarNo = p.CarNo,
    DriverNo = p.DriverNo,
    CheckCounts = p.Passengers.GetValueOrDefault()
}).ToList();
}
```

数据集让数据库排好序返回。

## 9. 查询优化



```
var customerinfo = new Customerinfo();
try
{
    using (var context = new MiniSysDataContext())
    {
        customerinfo = context.Customerinfo.ByPhone(phone).ToList().First();
        Mapper.CreateMap<Customerinfo, CustomerInfoBo>();
        customerInfoBo = Mapper.Map<Customerinfo, CustomerInfoBo>(customerinfo);
    }
}
catch (Exception ex)
{
    LogHelper.WriteLog(string.Format("CustomerInfoService.UpdateCustomerInfo({0})异常", customerInfoBo), ex)
}
return customerInfoBo;
```

```

var customerInfoBo = new CustomerInfoBO();
var customerinfo = new Customerinfo();
try
{
    using (var context = new MiniSysDataContext())
    {
        //customerinfo = context.Customerinfo.ByPhone(phone).ToList().First();
        customerinfo = context.Customerinfo.FirstOrDefault(p=>p.Phone.Equals(phone));
        Mapper.CreateMap<Customerinfo, CustomerInfoBO>();
        customerInfoBo = Mapper.Map<Customerinfo, CustomerInfoBO>(customerinfo);
    }
}
catch (Exception ex)
{
    LogHelper.WriteLog(string.Format("CustomerInfoService.UpdateCustomerInfo({0})异常", customerInfoBo), ex);
}
return customerInfoBo;

```

1,2 查询结果最后一致，但是有很大的差别。

- 1, load 满足条件的数据到内存，然后 linq to object 检索 first
- 2, 生成的 sql 语句会直接使用 top 关键字，返回一条数据

那种写法更合理呢？

再放一张搞瞎眼睛的代码写法

```

public object GetAllCarrentalOrdersBySome()
{
    var mList = OrderFacade.GetCarrentalOrderSer().GetAllCarrentalOrders();
    var tt = new[]
    {
        (from m in mList select m).Count(),
        (from m in mList where m.PaymentState==0 select m).Count(),
        (from m in mList where m.PaymentState==1 select m).Count(),
        (from m in mList where m.TicketState==0 select m).Count(),
        (from m in mList where m.TicketState==1 select m).Count(),
        (from m in mList where m.OrderState==1 select m).Count()
    };
    return tt;
}

```

- 1 处获取所有包车订单，load 到内存。这个表的数据不算大，但是是在不断增长的。
- 2 处 linq to object 统计数据，分类统计，在虚拟机去统计计算。我非常不懂，先不说 load 所有表数据极有可能撑爆内存的。要是前面下面车票表数百万的数据，按照这种写法统计，客户不找你那才真是奇迹。

这种 sum 操作不是应该交给数据库去统计更合适么。为什么要拿到所有数据到内存交给虚拟机去做？效率很低，很慢的:.....。

屁话不说，放码过来。代码初步重构如下

条件动态注入，最后执行 count。

当然这只是利用 linq 优雅语法糖的一种更高效率的写法，还有其他写法，用 stringBuilder 可直接上 hql 或者 sql，条件参数也动态注入。

```

public CarrentalOrderCountSumBo GetAllCarrentalOrdersBySome(CarrentalOrderBO carOrderBo)
{
    CarrentalOrderCountSumBo returnBo = new CarrentalOrderCountSumBo();
    using (var context = new MiniSysDataContext())
    {
        try
        {
            var totalQueryable = context.Carrentalorder.Where(p => 1 != 0); //全部
            var unpaidQueryable = context.Carrentalorder.Where(p => p.PaymentState == 0); //未付款
            var paidQueryable = context.Carrentalorder.Where(p => p.PaymentState == 1); //已付款
            var uncheckQueryable = context.Carrentalorder.Where(p => p.TicketState == 0); //未核票
            var checkQueryable = context.Carrentalorder.Where(p => p.TicketState == 1); //已核票
            var refundQueryable = context.Carrentalorder.Where(p => p.OrderState == 1); //已取消

            if (!string.IsNullOrEmpty(carOrderBo.UserName))
            {
                var tempUserName = carOrderBo.UserName.Trim();
                //注入UserName条件
                totalQueryable = totalQueryable.Where(p => p.UserName == tempUserName); //全部
                unpaidQueryable = unpaidQueryable.Where(p => p.UserName == tempUserName); //未付款
                paidQueryable = paidQueryable.Where(p => p.UserName == tempUserName); //已付款
                uncheckQueryable = uncheckQueryable.Where(p => p.UserName == tempUserName); //未核票
                checkQueryable = checkQueryable.Where(p => p.UserName == tempUserName); //已核票
                refundQueryable = refundQueryable.Where(p => p.UserName == tempUserName); //已取消
            }

            //注入phone条件

            //可继续注入其他检索条件
            returnBo.Total = totalQueryable.Count();
            returnBo.Unpaid = unpaidQueryable.Count();
            returnBo.Paid = paidQueryable.Count();
            returnBo.Uncheck = uncheckQueryable.Count();
            returnBo.Check = checkQueryable.Count();
            returnBo.Refund = refundQueryable.Count();
        }
    }
}

```

此种写法无论是时间复杂度还是空间复杂度，都远比上面第一种低。不服来辩。

同样的需求，别人用 8 核阿里云，你也用 8 核，有这种代码存在，有多少差距大家都懂。

Linq 对 object, xml , sql 等提供统一的操作接口，好用的确不假，要用的合理才好。

## 10. 检索条件转移

针对百万级大表数据。检索和高级检索，速度会成为很大的问题，必须进行检索优化

W23005	2014-04-24 14:04:28
W23005	2014-04-24 14:04:28
W23005	2014-04-24 14:04:28
W23005	2014-04-24 14:04:28
W23005	2014-04-24 14:04:28
W23005	2014-04-24 14:04:28
W23005	2014-04-24 14:04:28
W23005	2014-04-24 14:04:28

显示1到100,共1099839记录

目前车票表接近 **110w** 数据量，并且还在不断增长。按订单号检索车票，经常找不到车票，其实是查询超时的问题。

```

if (!string.IsNullOrEmpty(orderTicketBo.OrderNum))
{
    //Expression


```

1 处的代码是之前条件查询写入的条件。

2 处，是调整代码之后的。

2 处做的处理是，首先去掉模糊搜索，然后将订单号查询条件转换成订单 id，因为 id 上是主键，而主键上建有索引。这样，进行表连接后再按照主键字段查询，速度会提高几个数量级。

所以，当大表条件检索时，尤其是唯一性条件，合理将其转换成主键条件去查询，提高检索速度。

如果无法转换成主键条件的，应该合理大表查询字段建立索引。

优化后，速度刷刷的，你懂的，客户再也不说这里找不到数据了:)

## 11. Linq 的过度依赖

Linq 好用不多说，但是，用的过度，就有点不合适。

看代码

```

[HttpPost]
[Method(EnableSession = true)]
public object GetCarTypes(int page, int rows, string order, string sort, CarTypeBo carTypeBo)
{
    var pages = adminFacade.GetCarTypeSer().GetCarTypes(page, rows, order, sort, carTypeBo);
    return new { total = pages.ListT.Count, rows = pages.ListT.Select(q => new
    {
        q.CartId,
        q.Type,
        q.CreatedBy,
        TypeSizeCode = q.TypeSizeCode == 0 ? "小车" : q.TypeSizeCode == 1 ? "中车",
        CreatedOn = string.Format("{0:yyyy-MM-dd}", q.CreatedOn)
    }) };
}

```

这里存在几种问题，

第一种，pages 为 null 和 list 为 null 的潜在为空判断，可能会导致 ws 抛异常，并且这种异常不会被捕获。

第二种，linq 依赖过度。给调试会带来一些不便。尤其是 linq 遍历中，取得子项关联对象属性的时候，当关联子项为 null，则会报错，如下

```

u.Select(q => new Linebanci
{
    BeId = q.BeId,
    LineName = q.Ticketline.LineName,
    StrateDate = q.StrateDate,
    BanCiTime = q.BanCiTime,
    PriceAdjustRMB = q.PriceAdjustRMB,
    PriceAdjustHKD = q.PriceAdjustHKD,
    BackPriceAdjustRMB = q.BackPriceAdjustRMB ?? 0,
    BackPriceAdjustHKD = q.BackPriceAdjustHKD ?? 0,
    GuestPriceRmb = q.Ticketline.GuestPriceRMB,
    GuestPriceHkd = q.Ticketline.GuestPriceHKD,
    GuestBackForthPriceRmb = q.Ticketline.GuestBackForthPriceRMB,
    GuestBackForthPriceHkd = q.Ticketline.GuestBackForthPriceHKD,
    AgentPriceRmb = q.Ticketline.AgentPriceRMB,
    AgentPriceHkd = q.Ticketline.AgentPriceHKD,
    VipPriceRmb = q.Ticketline.VipPriceRMB,
    VipPriceHkd = q.Ticketline.VipPriceHKD,
    BanCiNum = q.BanCiNum,
    SeatNum = q.SeatNum,
    CanBeSold = q.CanBeSold,
    VehicleModelName = q.Cartype.Type,
    ModifidBy = q.ModifidBy,
    ModifidOn = q.ModifidOn
}

```

Ticketline, cartype 都是可 null 类型, 那么当出现异常的时候, 我如何得知是哪里的异常呢? 要知道 linq query 是不会进入到 lamda 表达式里的。给调试带来很大的不便  
第三种, 数据格式化形式不对, 后面讲。

## 12. 不安全代码

```

public object GetUsers(int page, int rows, string order, string sort, UserBO userBo)
{
    var list = adminFacade.GetUserSer().GetUsers(page, rows, order, sort, userBo);
    return new
    {
        total = list.TotalCount - 1,
        rows = list.ListT.Where(q => q.LoginName != ConfigurationManager.AppSettings["administrator"])
            .Select(q =>
            {
                q.CreatedOnStr = string.Format("{0:yyyy-MM-dd}", q.CreatedOn);
                return q;
            })
    };
}

```

1 处内部异常导致 list 为 null, 会发生什么呢?

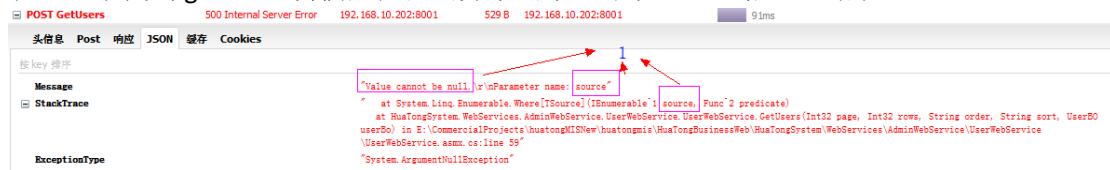
2 处, linq 循环筛选, 会进行多少次 xmlDocument 对象节点的读取?(不会是文件 io, webconfig 只会加载一次, 除非中途被修改), 不变的条件要拿出来处理, 而不是每次循环都去把相同的事情做一遍。

```

/// <returns></returns>
[ScriptMethod]
[WebMethod(EnableSession = true)]
public object GetUsers(int page, int rows, string order, string sort, UserBO userBo)
{
    string adminString = ConfigurationManager.AppSettings["administrator"];
    var list = adminFacade.GetUserSer().GetUsers(page, rows, order, sort, userBo);
    if(list==null||list.ListT==null)
    {
        return new {total = 0, rows = 0};
    }
    return new
    {
        total = list.TotalCount - 1,
        rows = list.ListT.Where(q => q.LoginName !=adminString)
            .Select(q =>
            {

```

从日志中找到 getUsers 会偶然性抛出异常出来，导致 Server 报 500 错误。



原因是 list 是 null，导致 webService 报异常  
不安全代码，不多说

做好相应的检查，非空，越界等，保证健壮性

```

var temp = adminFacade.GetConsumerSer().GetConsumerRecords(order, page, rows, sort, consumerRecordBo);
if (temp != null&&temp.ListT!=null)
{
    var formatted = temp.ListT.Select(p => new
    {
        p.CRId,
        p.Vipuser.CardNo,
        VipNameUserName=p.Vipuser.UserName,
        VipNameUserName1 = p.Vipuser.UserName,
        p.Vipuser.LoginName,
        p.Vipuser.CardNum,
        p.ConsumeSum,
        p.CurrencyTypeString,
        p.ConsumerDateString,
        p.OrderNo,
        p.OrderTypeString,
        p.RemainingSum
    }).ToList();
    return new { total = temp.TotalCount, rows = formatted };
}
else
{
    return new { total = 0, rows = new List<string>() };
}

```

注意 2 处，所有集合类型数据的返回，不可返回 null 或者其它的数据，  
请一定要返回对应类型的空集合，避免造成前端解析异常，否则在父方法内层需要添加 null 判断，来保证 linq 使用不抛异常

在 ht 前端框架中，要求是特定的数据返回格式，如果是空数据，返回的 json，rows 应该是空[],类似于 total=0 rows=[]这样的形式，解析不会报错。

具体返回的数据格式视要求而定

Nhibernate 带来的好处的，能自动关联到关系对象，并且给这些对象提供延迟加载，但是，



这一特性也会带来潜在的不安全代码，如下

```
//筛选器pageIndex-1是由于数据库的索引是从0开始的！
Func<IQueryable<Linebanci>, List<Linebanci>> selector =
    u => u.Select(q => new Linebanci
    {
        BeId = q.BeId,
        LineName = q.Ticketline.LineName,
        StrateDate = q.StrateDate,
        BanCiTime = q.BanCiTime,
        PriceAdjustRMB = q.PriceAdjustRMB,
        PriceAdjustHKD = q.PriceAdjustHKD,
        BackPriceAdjustRMB = q.BackPriceAdjustRMB ?? 0,
        BackPriceAdjustHKD = q.BackPriceAdjustHKD ?? 0,
        GuestPriceRmb = q.Ticketline.GuestPriceRMB,
        GuestPriceHkd = q.Ticketline.GuestPriceHKD,
        GuestBackForthPriceRmb = q.Ticketline.GuestBackForthPriceRMB,
        GuestBackForthPriceHkd = q.Ticketline.GuestBackForthPriceHKD,
        AgentPriceRmb = q.Ticketline.AgentPriceRMB,
        AgentPriceHkd = q.Ticketline.AgentPriceHKD,
        VipPriceRmb = q.Ticketline.VipPriceRMB,
        VipPriceHkd = q.Ticketline.VipPriceHKD,
        BanCiNum = q.BanCiNum,
        SeatNum = q.SeatNum,
        CanBeSold = q.CanBeSold,
        VehicleModelName = q.Cartype.Type,
        ModifidBy = q.ModifidBy,
        ModifidOn = q.ModifidOn
    }).Skip((pageIndex - 1) * pageSize).Take(pageSize).ToList();
```

本处代码中，如果关联的 Ticketline 和 carttype 对 null，代码必然报错无疑。

正确的做法是,坐好安全检查，如下

```
PriceAdjustRMB = q.PriceAdjustRMB,
PriceAdjustHKD = q.PriceAdjustHKD,
BackPriceAdjustRMB = q.BackPriceAdjustRMB ?? 0,
BackPriceAdjustHKD = q.BackPriceAdjustHKD ?? 0,
GuestPriceRmb = q.Ticketline == null ? default(float) : q.Ticketline.GuestPriceRMB ?? 0.0f,
GuestPriceHkd = q.Ticketline == null ? default(float) : q.Ticketline.GuestPriceHKD ?? 0.0f,
GuestBackForthPriceRmb = q.Ticketline.GuestBackForthPriceRMB,
GuestBackForthPriceHkd = q.Ticketline.GuestBackForthPriceHKD,
    AgentPriceRmb = q.Ticketline.AgentPriceRMB,
    AgentPriceHkd = q.Ticketline.AgentPriceHKD,
    VipPriceRmb = q.Ticketline.VipPriceRMB,
    VipPriceHkd = q.Ticketline.VipPriceHKD,
BanCiNum = q.BanCiNum,
SeatNum = q.SeatNum,
CanBeSold = q.CanBeSold,
VehicleModelName = q.Cartype == null ? string.Empty : q.Cartype.Type,
ModifidBy = q.ModifidBy,
ModifidOn = q.ModifidOn
}).Skip((pageIndex - 1) * pageSize).Take(pageSize).ToList();
//ToLower() [float? PriceAdjustRMB], [float?
//ToLower() [float? PriceAdjustRMB], [float?
```

## 13. 错误的数据过滤点


Service 方法

```

public List<ApplydetailBO> GetApplyDetailsByRecordId(int id)
{
    Applyrecord entity = null;
    List<Applydetail> detailEntities = null;
    List<ApplydetailBO> detailBos = null;
    try
    {
        using (var context=new MiniSysDataContext())
        {
            entity=context.Applyrecord.FirstOrDefault(p => p.ApplyId == id);
            if (entity != null)
            {
                detailEntities = entity.ApplydetailList.ToList();
            }
            Mapper.Reset();
            Mapper.CreateMap<Applydetail, ApplydetailBO>();
            Mapper.CreateMap<Applyitem, List<ApplyitemBO>>();
            Mapper.CreateMap<Applyitem, ApplyitemBO>();
            Mapper.CreateMap<Applyrecord, ApplyRecordBO>();
            detailBos=Mapper.Map<List<Applydetail>, List<ApplydetailBO>>(detailEntities);
        }
        return detailBos.Select( =>new ApplydetailBO()
        {
            DetailId = p.DetailId,
            ItemId=p.Applyrecord.ApplyId??0, //报销单编号
            ItemName =p.Applyitem.ItemName, //报销项目名称
            Count = p.Count, //数量
            Price = p.Price, //报销子项单价
            TotalPrice = p.Count*p.Price??0.0d, //总价
            CreatedOnDateLongStr = p.CreatedOnDateLongStr //填写时间
        }).ToList();
    }
}

```

在Service进行数据过滤是不正确的  
正确的做法是，在最顶层的数据获取  
方法中进行过滤



## webService 方法

```

/// <param name="id"></param>
/// <returns></returns>
[WebMethod(EnableSession = true)]
[ScriptMethod]
public List<ApplydetailBO> GetApplyDetailsByRecordId(Int32? id)
{
    if (id == 0 || id == null)
    {
        return null;
    }
    return costFacade.GetApplyRecordService().GetApplyDetailsByRecordId(id.GetValueOrDefault());
}
}

```



正确做法=>

Service 方法

```

/// <returns></returns>
public List<ApplydetailBO> GetApplyDetailsByRecordId(int id)
{
    Applyrecord entity = null;
    List<Applydetail> detailEntities = null;
    List<ApplydetailBO> detailBos = null;
    try
    {
        using (var context = new MiniSysDataContext())
        {
            entity = context.Applyrecord.FirstOrDefault(p => p.ApplyId == id);
            if (entity != null)
            {
                detailEntities = entity.ApplydetailList.ToList();
            }
            Mapper.Reset();
            Mapper.CreateMap<Applydetail, ApplydetailBO>();
            Mapper.CreateMap<Applyitem, List<ApplyitemBO>>();
            Mapper.CreateMap<Applyitem, ApplyitemBO>();
            Mapper.CreateMap<Applyrecord, ApplyRecordBO>();
            detailBos = Mapper.Map<List<Applydetail>, List<ApplydetailBO>>(detailEntities);
        }
        return detailBos;
    }
    catch (Exception e)
    {
        LogHelper.WriteLog(string.Format("ApplyRecordService.GetApplyDetailsByRecordId({0})", id), e);
        return null;
    }
}

```

## webService 方法

```

/// <returns></returns>
[WebMethod(EnableSession = true)]
[ScriptMethod]
public object GetApplyDetailsByRecordId(Int32? id)
{
    if (id == 0 || id == null)
    {
        return null;
    }
    var tempdatas = costFacade.GetApplyRecordService().GetApplyDetailsByRecordId(id.GetValueOrDefault());
    return tempdatas.Select(p => new
    {
        p.DetailId,
        ItemId = p.Applyrecord.ApplyId ?? 0, //报销单编号
        p.Applyitem.ItemName, //报销项目名称
        p.Count, //数量
        p.Price, //报销子项单价
        TotalPrice = p.Count * p.Price ?? 0.0d, //总价
        p.CreatedOnDateLongStr //填写时间
    }).ToList();
}

```

选择正确的数据过滤点，减少返回的数据量。最大限度的保证易编辑，易修改，易扩展。按照第二种办法，不仅仅数据量少了，而且，如果有其它的数据返回需求，只需要在 webService 扩展接口，重用 service 的返回全部数据的那个方法，过滤一次返回即可。而不是从新在 service 又写方法。

这么做的原因，也是跟 JavaScript MVC 思想保持一致。保持前端业务实体的无污染

如果是 mvc 架构，请在 Controlner 内完成数据过滤

## 14. 错误的 NULL 返回

```
public List<ApplydetailBO> GetApplyDetailsByRecordId(int id)
{
    Applyrecord entity = null;
    List<Applydetail> detailEntities = null;
    List<ApplydetailBO> detailBos = null;
    try
    {
        using (var context=new MiniSysDataContext())
        {
            entity=context.Applyrecord.FirstOrDefault(p => p.ApplyId == id);
            if (entity != null)
            {
                detailEntities = entity.ApplydetailList.ToList();
            }
            Mapper.Reset();
            Mapper.CreateMap<Applydetail, ApplydetailBO>();
            Mapper.CreateMap<Applyitem, ApplyitemBO>();
            Mapper.CreateMap<Applyrecord, ApplyRecordBO>();
            detailBos=Mapper.Map<List<Applydetail>, List<ApplydetailBO>>(detailEntities);
        }
        return detailBos;
    }
    catch (Exception e)
    {
        LogHelper.WriteLog(string.Format("ApplyRecordService.GetApplyDetailsByRecordId({0})", id), e);
        return null;
    }
}
```

```
public List<ApplydetailBO> GetApplyDetailsByRecordId(int id)
{
    Applyrecord entity = null;
    List<Applydetail> detailEntities = null;
    List<ApplydetailBO> detailBos = new List<ApplydetailBO>();
    try
    {
        using (var context=new MiniSysDataContext())
        {
            entity=context.Applyrecord.FirstOrDefault(p => p.ApplyId == id);
            if (entity != null)
            {
                detailEntities = entity.ApplydetailList.ToList();
            }
            Mapper.Reset();
            Mapper.CreateMap<Applydetail, ApplydetailBO>();
            Mapper.CreateMap<Applyitem, ApplyitemBO>();
            Mapper.CreateMap<Applyrecord, ApplyRecordBO>();
            detailBos=Mapper.Map<List<Applydetail>, List<ApplydetailBO>>(detailEntities);
        }
        return detailBos;
    }
    catch (Exception e)
    {
        LogHelper.WriteLog(string.Format("ApplyRecordService.GetApplyDetailsByRecordId({0})", id), e);
        return detailBos;
    }
}
```

集合类型数据，慎用 null 返回，null 返回会增加父层的判断，并且，异常后返回到前端的是 null，会造成 js 错误，为什么呢，null 和空有区别的。

Null 是没有容器，空是容器中没有元素，前者作为 json 解析报错，后者不会。

约定集合类型的返回，不允许返回 null，返回对应类型的空集合

## 15. 错误的数据格式化形式

在业务处理过程中，有一些数据是没有关系表的，比如少量的类型网络类型(Wifi,WLAN,3G)，

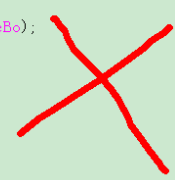
还有一些状态标志，比如 1234 分别代表新增，部门审核，财务审核，最终审核  
这些信息是以数字是存在数据库里的，取出来的时候，需要格式化文字。

还有时间等数据，为了便于前端的时候，返回前需要格式化。

做法是，统一在 bo 中格式化处理、

```
//获取分页数据
Page<RoleBO> pageRoleBos = adminFacade.GetRoleSer().GetRoles(order, page, rows, sort, roleBo);
//对分页数据日期格式化
var tempRowsFormatted = pageRoleBos.ListT.Select(bo => new
{
    bo.RoleId,
    bo.Name,
    bo.Description,
    bo.CreatedBy,
    bo.ModifiedBy,
    CreatedOn = string.Format("{0:yyyy-MM-dd}", bo.CreatedOn),
    ModifiedOn = string.Format("{0:yyyy-MM-dd}", bo.ModifiedOn),
});

//返回格式化日期后的列表数据和总的记录行
return new { total = pageRoleBos.TotalCount, rows = tempRowsFormatted };
```



使用 Linq 在循环中进行数据格式，有必要么？

要知道 AutoMapper 在进行 dto 到 bo 映射的时候，已经把所有数据遍历了一次，

```
try
{
    using (var context=new MiniSysDataContext())
    {
        entity=context.Applyrecord.FirstOrDefault(p => p.ApplyId == id);
        if (entity != null)
        {
            detailEntities = entity.ApplydetailList.ToList();
        }
        Mapper.Reset();
        Mapper.CreateMap<Applydetail, ApplydetailBO>();
        Mapper.CreateMap<Applyitem, ApplyitemBO>();
        Mapper.CreateMap<Applyrecord, ApplyRecordBO>();
        detailBos=Mapper.Map<List<Applydetail>, List<ApplydetailBO>>(detailEntities);
    }
    return detailBos;
}
```

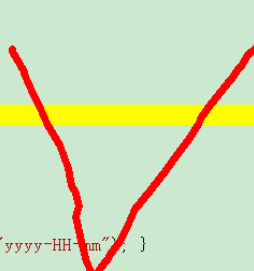


难道仅仅是为了格式化一个日期数据，在数据返回前就需要把全部数据遍历一遍，循环是耗时的，应该避免这种低效的无用功。

正确的在 bo 中处理数据格式化的问题。如

```
public string FinalAuditBy { get; set; }
[ScriptIgnore]
public DateTime? CreatedOn { get; set; }
public string CreatedBy { set; get; }
[ScriptIgnore]
public DateTime? ModifiedOn { set; get; }
public string ModifiedBy { get; set; }
[ScriptIgnore]
public List<ApplydetailBO> ApplydetailBoList { get; set; }

/*辅助字段*/
public string CreatedOnDateStr
{
    get { return CreatedOn == null ? "" : CreatedOn.GetValueOrDefault().ToString("yyyy-MM-dd HH:mm:ss"); }
}
public string ModifiedOnDateStr
{
    get { return ModifiedOn == null ? "" : ModifiedOn.GetValueOrDefault().ToString("yyyy-MM-dd HH:mm:ss"); }
}
public string CreatedOnDateLongStr
{
    get { return CreatedOn == null ? "" : CreatedOn.GetValueOrDefault().ToString("yyyy-MM-dd HH:mm:ss"); }
}
public string ModifiedOnDateLongStr
{
    get { return ModifiedOn == null ? "" : ModifiedOn.GetValueOrDefault().ToString("yyyy-MM-dd HH:mm:ss"); }
}
}
```



```

public string PaymentTypeStr
{
    get
    {
        switch (PaymentType)
        {
            case 1:
                return "支付宝";
            case 2:
                return "在线支付";
            case 3:
                return "IC卡支付";
            default:
                return "";
        }
    }
}

```

```

/// <summary>
/// 出票状态
/// </summary>
public string IsOutTicketStr
{
    get
    {
        switch (IsOutTicket)
        {
            case 0:
                return "未出票";
            case 1:
                return "已出票";
            case 2:
                return "部分出票";
            default:
                return "";
        }
    }
}

```

很简单不多说。

**Tips:** 不要封装独立的格式化方法，在 `linq` 循环中，调用格式化方法去格式化。可以做，但是是 `level` 很低的做法

## 16. 合理的定义 Bo 中属性类型和属性名称

确保 bo 是 poco 类，只含有属性，无继承，无接口实现，无方法等其它任何 oo 的特性  
另外确保 bo 中属性的类型是 CLR 定义的简单类型(Int,float,Int64,double,Datetime,string 等)  
List<>, 数组等简单数据结构

请不要搞一些复杂的数据结构在 bo 中，如字典或者 hashtable 等。

有一些属性类型经过了框架扩展成了自定义属性，如 ISet, IList 等，NH 扩展了一些集合类  
再次强调，请使用 C# 定义的集合类型。

```
<?xml-namespace="urn:hibernate-mapping-2.2" namespace="Data" assembly="Data">
  <class name="Applyrecord" table="tb_applyrecord" >
    <id name="ApplyId" column="ApplyId" type="Int32" >
      <generator class="native" />
    </id>
    <property name="OrId" column="Or_Id" type="Int32" not-null="false" />
    <property name="Price" column="Price" type="Single" not-null="false" />
    <property name="Currency" column="Currency" type="String" not-null="false" length="30" />
    <property name="ApplyBy" column="ApplyBy" type="String" not-null="false" length="600" />
    <property name="ApplyOn" column="ApplyOn" type="DateTime" not-null="false" />
    <property name="Description" column="Description" type="String" not-null="false" length="1500" />
    <property name="IsCar" column="IsCar" type="Int32" not-null="false" />
    <property name="CarNo" column="CarNo" type="String" not-null="false" length="60" />
    <property name="AuditStatus" column="AuditStatus" type="Int32" not-null="false" />
    <property name="DepAuditBy" column="DepAuditBy" type="String" not-null="false" length="600" />
    <property name="FinanceAuditBy" column="FinanceAuditBy" type="String" not-null="false" length="600" />
    <property name="FinalAuditBy" column="FinalAuditBy" type="String" not-null="false" length="600" />
    <property name="CreatedOn" column="CreatedOn" type="DateTime" not-null="false" />
    <property name="CreatedBy" column="CreatedBy" type="String" not-null="false" length="600" />
    <property name="ModifiedOn" column="ModifiedOn" type="DateTime" not-null="false" />
    <property name="ModifiedBy" column="ModifiedBy" type="String" not-null="false" length="600" />
    <property name="OrName" column="Or_Name" type="String" not-null="false" length="600" />
    <bag name="ApplydetailList" inverse="true" cascade="all-delete-orphan" lazy="true" >
      <key>
        <column name="ApplyId" />
      </key>
      <one-to-many class="Applydetail" />
    </bag>
  </class>
```

```
#region Associations mappings

[System.CodeDom.Compiler.GeneratedCode("CodeSmith", "6.0.0.0")]
private IList<Applydetail> _applydetailList;

[System.Runtime.Serialization.DataMember(Order = 19, EmitDefaultValue = false)]
[System.CodeDom.Compiler.GeneratedCode("CodeSmith", "6.0.0.0")]
public virtual IList<Applydetail> ApplydetailList
{
    get { return _applydetailList; }
    set
    {
        OnApplydetailListChanging(value, _applydetailList);
        SendPropertyChanging("ApplydetailList");
        _applydetailList = value;
        SendPropertyChanged("ApplydetailList");
        OnApplydetailListChanged(value);
    }
}
```

对应的 bo，定义如下

```

/// </summary>
public string FinalAuditBy { get; set; }
[ScriptIgnore]
public DateTime? CreatedOn { get; set; }
public string CreatedBy { set; get; }
[ScriptIgnore]
public DateTime? ModifiedOn { set; get; }
public string ModifiedBy { get; set; }
[ScriptIgnore]
public List<ApplydetailBO> ApplydetailBoList { get; set; }

/*辅助字段*/
public string CreatedOnDateStr
{
    get { return CreatedOn == null ? "" : CreatedOn.GetValueOrDefault().ToString(); }
}
public string ModifiedOnDateStr
{

```

针对非集合简单数据类型，string，datetime，int 等

Bo 属性命名请和 xml(entity)中命名保持一致，并且属性类型要保持一致，如果没有一致的数据类型，至少要保证是兼容的数据类型，比如 entity 中是 int，那么 bo 中用 long 是允许的，这样保证 AutoMapper 类的映射组件自动映射，并且不出错。

如果是 sql server，那么生成的 xml 中的数据类型，在 clr 是不一定有的。所以在设计的，请格外注意，设计 DB 和 CLR 兼容的数据类型(byte 类型会直接被视为 0，而 0 确是 int 类型的)

否则会 AutoMapper 映射引擎在执行映射的过程中会抛类型不一致的异常

真对集合类型，请不要命名成一样。Bo 中的集合映射是非常耗时的，所以，最佳实践是，entity 中的集合属性和 bo 中的集合属性区分命名，如上，entity 为 Applydetaillist，bo 中为 ApplyDetailBoList。

这样做的目的是，避免在不需要集合数据的时候，automapper 自动映射，这是不必要的。

当我需要的时候，手动进行映射指定。从而减少不必要的循环操作，提高性能。

## 17. 序列化数据冗余

垃圾数据的返回，这些 Datetime 格式的数据返回没有起到任何作用，还有一些 null 的属性，前端根本没有用，但是也一股脑的返回，会不会给 js 执行引擎造成不必要的压力和处理时间呢？



```
{
  "d": {
    "total": 238,
    "rows": [
      {
        "type": "BusinessObject.TicketBo.TicketlineBO",
        "TlId": 247,
        "CreatedOn": "/Date(1385975600000)/",
        "ModifiedOn": "/Date(1385981602000)/",
        "StrModifiedOn": "2013-12-02",
        "ModifiedBy": "admin",
        "CreatedBy": "admin",
        "BeginCity": "东莞",
        "EndCity": "广州",
        "LineState": 1,
        "BeginCitySanCode": "DNG",
        "EndCitySanCode": "CAN",
        "LineStateStr": "禁用",
        "BeginStationId": 2,
        "EndStationId": 6,
        "Station": 2,
        "StationStr": "常平站点",
        "VehicleModel": 2,
        "VehicleModelText": "商务小车",
        "LineType": 1,
        "LineTypeStr": "市区线",
        "IsHot": null,
        "IsShortLine": 1,
        "IsShortLineString": "否",
        "IsHotStr": "否",
        "MiddleStation": 1,
        "SecondMiddleStation": 0,
        "ThirdMiddleStation": 0,
        "LineSort": 100,
        "LineName": "常平站点-香港太子",
        "GuestPriceRMB": 0,
        "GuestPriceHKD": 0,
        "GuestBackForthPriceRMB": 0,
        "GuestBackForthPriceHKD": 0,
        "AgentPriceRMB": 0,
        "AgentPriceHKD": 0,
        "AgentBackForthPriceRMB": 0,
        "AgentBackForthPriceHKD": 0,
        "VipPriceRMB": 0,
        "VipPriceHKD": 0,
        "VipBackForthPriceRMB": 0,
        "VipBackForthPriceHKD": 0,
        "CheckNum": 1,
        "BigPersonsAllowanceRMB": 0,
        "BigPersonsAllowanceHKD": 0,
        "MiddlePersonsAllowanceRMB": 0,
        "MiddlePersonsAllowanceHKD": 0,
        "SmallPersonsAllowanceRMB": 0,
        "SmallPersonsAllowanceHKD": 0,
        "BigAllowanceRMB": 0,
        "BigAllowanceHKD": 0,
        "MiddleAllowanceRMB": 0,
        "MiddleAllowanceHKD": 0,
        "SmallAllowanceRMB": 0,
        "SmallAllowanceHKD": 0,
        "StrCurrencyType": null,
        "StrLinebanci": null,
        "StrLinebanciId": null,
        "StrTicketonpoint": null,
        "StrTicketoffpoint": null,
        "Num": null,
        "StartDate": null,
        "EndDate": null,
        "TimeDifference": 0,
        "Center": 0,
        "BackLineName": null,
        "BackLineId": null,
        "TotalTrip": null,
        "IsBack": 0,
        "BackGuestBackForthPriceRMB": null,
        "BackGuestBackForthPriceHKD": null,
        "StrLinebanciBack": null,
        "StrLinebanciBackId": null,
        "StrSeat": null,
        "StrSeatBack": null,
        {
          "type": "BusinessObject.TicketBo.TicketlineBO",
          "TlId": 246,
          "CreatedOn": "/Date(1385434820000)/",
          "ModifiedOn": "/Date(1385433312000)/",
          "StrModifiedOn": "2013-11-26",
          "ModifiedBy": "admin",
          "CreatedBy": "admin",
          "BeginCity": "香港",
          "EndCity": "深圳",
          "LineState": 0,
          "BeginCitySanCode": "HKG",
          "EndCitySanCode": "SZX",
          "LineStateStr": "启用",
          "BeginStationId": 7,
          "EndStationId": 1,
          "Station": 7,
          "StationStr": "香港荃湾",
          "VehicleModel": 2,
          "VehicleModelText": "商务小车",
          "LineType": 1,
          "LineTypeStr": "市区线",
          "IsHot": null,
          "IsShortLine": 1,
          "IsShortLineString": "否",
          "IsHotStr": "否",
          "MiddleStation": 0,
          "SecondMiddleStation": 0,
          "ThirdMiddleStation": 0,
          "LineSort": 100,
          "LineName": "香港荃湾-福田站点",
          "GuestPriceRMB": 2.1,
          "GuestPriceHKD": 2.2,
          "GuestBackForthPriceRMB": 0,
          "GuestBackForthPriceHKD": 0,
          "AgentPriceRMB": 0,
          "AgentPriceHKD": 0,
          "AgentBackForthPriceRMB": 0,
          "AgentBackForthPriceHKD": 0,
          "VipPriceRMB": 0,
          "VipPriceHKD": 0,
          "VipBackForthPriceRMB": 0,
          "VipBackForthPriceHKD": 0,
          "CheckNum": 1,
          "BigPersonsAllowanceRMB": 0,
          "BigPersonsAllowanceHKD": 0,
          "MiddlePersonsAllowanceRMB": 0,
          "MiddlePersonsAllowanceHKD": 0,
          "SmallPersonsAllowanceRMB": 0,
          "SmallPersonsAllowanceHKD": 0,
          "BigAllowanceRMB": 0,
          "BigAllowanceHKD": 0,
          "MiddleAllowanceRMB": 0,
          "MiddleAllowanceHKD": 0,
          "SmallAllowanceRMB": 0,
          "SmallAllowanceHKD": 0,
          "StrCurrencyType": null,
          "StrLinebanci": null,
          "StrLinebanciId": null,
          "StrTicketonpoint": null,
          "StrTicketoffpoint": null,
          "Num": null,
          "StartDate": null,
          "EndDate": null,
          "TimeDifference": 0,
          "Center": 0,
          "BackLineName": null,
          "BackLineId": null,
          "TotalTrip": null,
          "IsBack": 0,
          "BackGuestBackForthPriceRMB": null,
          "BackGuestBackForthPriceHKD": null,
          "StrLinebanciBack": null,
          "StrLinebanciBackId": null,
          "StrSeat": null,
          "StrSeatBack": null
        }
      ]
    }
  }
}
```

paste copy | format strip to y | remove new lines (y)

```
"Station": {
  "Identification": 3,
  "Name": "长安站点",
  "ComplexName": "长安站点",
  "EnglishName": "长安站",
  "Type": 0,
  "StrType": "公司站点",
  "Phone": "0",
  "Address": "",
  "LinkMan": "0",
  "City": "东莞",
  "CitySZM": "DNG",
  "CreatedOn": "/Date(1373940400000)/",
  "CreatedBy": "admin",
  "ModifiedOn": "/Date(1377074961000)/",
  "StrCreatedOn": "2013-07-16",
  "ModifiedBy": "v22007"
},
"StationName": "长安站点",
"Organization": {
  "OrgId": 9,
  "Name": "东莞长安站点",
  "CreatedOn": "/Date(1373953467000)/",
  "Description": "",
  "CreateBy": null,
  "ModifiedOn": null,
  "ModifiedBy": null
},
"OrgId": null,
"Job": {
  "Identification": 32,
  "Name": "司机",
  "CreatedOn": "/Date(1373944653000)/",
  "ModifiedOn": null,
  "CreatedBy": "zhd",
  "ModifiedBy": null
},
"JobId": null,
"WorkNo": "P16046",
"Name": "金永华",
"Password": "67B14728AD9902AECBA32E22FA4F6BD",
"Sex": 0,
"StrSex": "男",
"IdentityCar": "362302197205231019",
"Mobile": "13798404947",
"Email": "",
"WorkDate": "/Date(1378828800000)/",
"StrWorkDate": "2013-09-11",
"DepartureDate": null,
"StrDepartureDate": "",
"IsDeparture": null,
"IsDepartureStr": "",
"IsDepartureSearch": null,
"DepartureKesson": "",
"CreatedOn": "/Date(1378862358000)/",
"StrCreatedOn": "2013-09-11",
"CreatedBy": "zhd",
"ModifiedOn": "/Date(1378969449000)/",
"ModifiedBy": "zhd",
"DriverType": 1,
"WorkType": null,
"EmpIdentity": null,
"DriverLicense": 4,
```



五花八门的数据都返回了。。。。连密码的密文也返回了。这里面有大量的数据 client 根本不需要，返回做设什么呢

该怎么做？如下

```

    /// 最终审核人
    /// </summary>
    public string FinalAuditBy { get; set; }
    [ScriptIgnore]
    public DateTime? CreatedOn { get; set; }
    public string CreatedBy { set; get; }
    [ScriptIgnore]
    public DateTime? ModifiedOn { set; get; }
    public string ModifiedBy { get; set; }
    [ScriptIgnore]
    public List<ApplydetailBO> ApplydetailBoList { get; set; }

```

Ajax 返回的 json

```

{"d":[{"DetailId":4,"ItemId":53,"ItemName":"报销测试项","Count":1,"Price":1,"TotalPrice":1,"CreatedOnDateLongStr":
"2013-12-07 18:39:36"}, {"DetailId":5,"ItemId":53,"ItemName":"11132239235","Count":2,"Price":1,"TotalPrice"
:2,"CreatedOnDateLongStr":"2013-12-07 18:39:36"}, {"DetailId":6,"ItemId":53,"ItemName":"报销更新测试项","Count"
:2,"Price":45,"TotalPrice":90,"CreatedOnDateLongStr":"2013-12-07 18:39:36"}, {"DetailId":7,"ItemId":53
,"ItemName":"报销更新测试项","Count":1,"Price":1,"TotalPrice":1,"CreatedOnDateLongStr":"2013-12-07 18:54:47"
}, {"DetailId":8,"ItemId":53,"ItemName":"22","Count":11,"Price":11,"TotalPrice":121,"CreatedOnDateLongStr"
:"2013-12-07 18:54:47"}, {"DetailId":9,"ItemId":53,"ItemName":"11132239235","Count":11,"Price":11,"TotalPrice"
:121,"CreatedOnDateLongStr":"2013-12-07 18:54:47"}]}

```

保持前端数据干净，减少冗余数据返回

序列化的时候，所以前端不需要的数据，注意加上 ScriptIgnore 标签

尤其是属性为 DateTime 和属性是集合类型的返回

## 18. AutoMapper 的正确使用

简单说下常见的使用

```

else
{
    List<Applydetail> details=new List<Applydetail>();
    float recordTotalPrice = default(float); //报销单总价
    try
    {
        Mapper.Reset();
        Mapper.CreateMap<ApplydetailBO, Applydetail>();
        var entity = Mapper.Map<ApplyRecordBO, Applyrecord>(applyRecordBo);
        entity.CreatedBy = UserInfo.IsSysAccess ? UserInfo.GetUserName() : "非系统访问";
        entity.CreatedOn = DateTime.Now;
        entity.AuditStatus = 0; //新增报销项的状态为 '未审核'
        if(entity.OrId==0)
        {
            entity.OrId = null;
        }
        if(entity.IsCar!=null&&entity.IsCar.GetValueOrDefault()==1) //是车辆成本
        {
            entity.IsCar = 1;
        }
        else
        {
            entity.IsCar = 0;
        }
    }
}

```

1 重置 AutoMapper 的映射规则字典，映射规则是存在一个字典里的，映射的时候会进行查找，清除掉映射规则，避免不必要的映射规则查找

2 创建映射规则，存在映射规则字典

### 3 执行映射操作，得到实体

注意，ApplydetailBo 中有集合类型，但是 bo 和 entity 的集合类型属性命名不一样，这里使用 AutoMapper 映射的时候，直接忽略掉集合的映射，能大幅度的减少执行时间的。

简单的集合到集合的映射

如果需要执行集合映射怎么办呢？创建映射规则时候，跟简单 bo 类型一样，区别在执行映射

请使用

Var entities=Mapper.Map<List<T>,List<T>>(bos); 即可得到正确的实体集合

```
List<FreeorderBO> returnOrders = null;
try
{
    var orders = context.Freeorder.Where(
        p => p.CreatedBy == username && p.CreatedOn >= startDate && p.CreatedOn < endDate).OrderByDescending(p =>
        p.CreatedOn);
    Mapper.Reset();
    Mapper.CreateMap<Passport, PassInfoModel>();
    var map = Mapper.CreateMap<Freeorder, FreeorderBO>();
    map.ForMember(h => h.Title, opt => opt.MapFrom(s => s.Freepackage.Title));
    returnOrders = Mapper.Map<List<Freeorder>, List<FreeorderBO>>(orders);
}
catch (Exception e)
{
    LogHelper.WriteLog(
        string.Format("TicketInfoForUserService.GetFreeOrdersForUser({0},{1},{2})", context, startDate,
            endDate), e);
}
```

如果需要 bo 中的集合自动映射呢？请按照一下操作

```
Mapper.Reset();
var map=Mapper.CreateMap<ApplyRecordBO, Applyrecord>();
map.ForMember(d => d.ApplydetailList, opt => opt.MapFrom(n => n.ApplydetailBoList));
Mapper.CreateMap<ApplydetailBO, Applydetail>();
var entity = Mapper.Map<ApplyRecordBO, Applyrecord>(applyRecordBo);
entity.CreatedBy = UserInfo.IsSysAccess ? UserInfo.GetUserName() : "非系统访问";
entity.CreatedOn = DateTime.Now;
entity.AuditStatus = 0; //新增报销项的状态为 '未审核'
if(entity.OrId==0)
{
    entity.OrId = null;
}
```

1 处目的属性，2 处源属性

不同名的简单类型映射，也可按照以上操作。

```

Mapper.Reset();
var map = Mapper.CreateMap<Ticketlineorder, TicketLineOrderBo>();
Mapper.CreateMap<Orderticket, OrderTicketBO>();
Mapper.CreateMap<Ticketline, TicketlineBO>();
//手动制定映射规则，将与订单关联的车票集合进行映射上
map.ForMember(d => p.OrderticketListForWeb, opt => opt.MapFrom(d => d.OrderticketList));
map.ForMember(d => b.FirstLineName,
    opt =>
    opt.MapFrom(
        d =>
        {
            var firstOrDefault = d.OrderticketList.FirstOrDefault(q => q.IsRefund == 1);
            return firstOrDefault != null ? (!d.OrderticketList.Any(q => q.IsRefund == 1)
                ? ""
                : firstOrDefault.LineName) : null;
        }
    ));
map.ForMember(d => b.FirstLeaveDate,
    opt =>
    opt.MapFrom(
        d =>
        {
            var orDefault = d.OrderticketList.FirstOrDefault(q => q.IsRefund == 1);
            return orDefault != null ? (!d.OrderticketList.Any(q => q.IsRefund == 1)
                ? null
                : orDefault.LeaveDate) : null;
        }
    ));
map.ForMember(d => b.FirstLeaveTime,
    opt =>
    opt.MapFrom(
        d =>
        {
            var orDefault = d.OrderticketList.FirstOrDefault(q => q.IsRefund == 1);
            return orDefault != null ? (!d.OrderticketList.Any(q => q.IsRefund == 1)
                ? null
                : orDefault.LeaveTime) : null;
        }
    ));

```

上面这种映射其实是性能非常低的。每一个属性，内层还有嵌套一层循环

Bo 中嵌套其它类型的 bo，该如何映射呢？

在 entity 和 bo 中对应属性命名一致的情况下，请参考以下

```

//调用扩展的分页方法，默认降序
using (var context = new MiniSysDataContext())
{
    //获取分页数据
    pagedata = context.Consumerrecord.GetPagedListQuery(page, rows, wheres, orderBy,
        selector, isSortOrderAsc);
    Mapper.CreateMap<Page<Consumerrecord>, Page<ConsumerRecordBO>>();
    Mapper.CreateMap<Consumerrecord, ConsumerRecordBO>();
    Mapper.CreateMap<Vipuser, VipUserBO>();
    Mapper.CreateMap<Station, StationBO>();
    Mapper.CreateMap<Payrecord, PayRecordBO>();
    boPage = Mapper.Map<Page<Consumerrecord>, Page<ConsumerRecordBO>>(pagedata);
    return boPage;
}

```

Bo 中嵌套的是什么类型，那么在父类型映射的时候，应该加上子类型映射规则，父类型和子类型规则创建顺序应该是父类在前，子类在后，其实顺序对映射结果并无影响

因为规则是放在字典里的，区别只不过是，查找规则的那一点点微乎其微的时间。随意我个人的建议还是，父 bo 类型的规则创建在前。子 bo 类型规则的创建在后。一级级下来

AutoMapper 可实现，bo 到实体，实体到 bo 的双向映射。详细 google，或者 github 上的开源库文档

<https://github.com/AutoMapper/AutoMapper/wiki>

# General Features

- [Flattening](#)
- [Projection](#)
- [Configuration Validation](#)
- [Lists and Arrays](#)
- [Nested Mappings](#)
- [Custom Type Converters](#)
- [Custom Value Resolvers](#)
- [Null Substitution](#)
- [Containers](#)
- [Mapping Inheritance](#)
- [Queryable Extensions](#)
- [Configuration](#)
- [Conditional Mapping](#)

另外，tips：在循环体内慎用 AutoMapper

## 19. Json.Net 的正确使用

远程 remoting 到 json 数据以后，client 需要进一步处理，json 需要反序列为 bo，那么 json.net 是绝佳利器

入门和注意要点参考=>[Json.NET 序列化利器](#)

更详细和深入的东西，参考 github 上的开源库和 wiki，不多说。

另外多提一点，就是 DateTime 格式的数据序列化后，在反序列话后，时间相差 8 小时的问题。这个时间差值碰巧是格林威治时间和北京东八区时间的差值。

这个不是异常，是因为 json.net 内的序列化和反序列化的时候，选取的时区问题。需要改源码。

替代方案，序列化前，对 bo 中的 Datetime 属性使用 ScriptIgnore。序列化的时候忽略，使用 string 类型的字段格式化他，反序列话的时候，在还原成 dateTime

Bo 中的属性定义应该如此，就像这样

```

/// <summary>
/// 创建时间(长时间字符串格式)
/// </summary>
public string CreatedOnLongStr
{
    get { return CreatedOn.HasValue ? CreatedOn.Value.ToString("yyyy-MM-dd HH:mm:ss") : ""; }
    //添加setter访问器, 反序列化时间数据异常, add-by-ys-on-2013-9-26
    set { CreatedOn = string.IsNullOrEmpty(value) ? (DateTime?) null : Convert.ToDateTime(value); }
}

```

关于 json.net 更详细的使用

参考 <http://james.newtonking.com/json/help/index.html>

## 20. Nhibernate profile 的使用

这个工具，已经花了太多文字介绍了。不多说。

详情参考=>[Nhibernate profile](#)

## 21. 低性能的全表连接

自动完成: [Tab]-> 下一个标签, [Ctrl+Space]-> 列出所有标签, [Ctrl+Enter]-> 列出匹配标签

```

4  `st`.`StationName` AS `StationName`,
5  `tlorder`.`CreatedOn` AS `CreatedOn`,
6  `vip`.`UserName` AS `UserName`
7  FROM
8  (
9      {
10         `tb_ticketlineorder` `tlorder`
11         JOIN `tb_vipuser` `vip`
12         ON (
13             `vip`.`LoginName` = `tlorder`.`CreatedBy`
14             AND `tlorder`.`IsDelete` <> 1
15         )
16     }
17     JOIN `tb_station` `st`
18     ON (
19         (
20             `st`.`Station_Id` = `vip`.`Station_Id`
21         )
22     )
23 )
24 ORDER BY `tlorder`.`CreatedOn` DESC $$
25
26 DELIMITER ;

```

全表连接