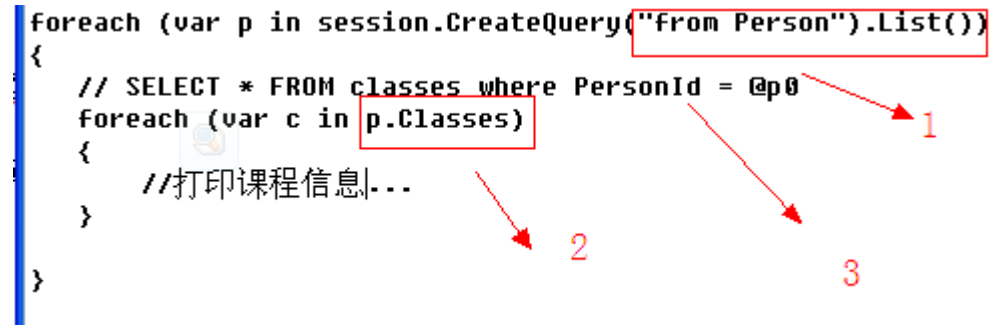


Nhibernate 性能问题性能问题 03—臭名昭著的 N+1 问题

图 1

关于 N+1 问题，网上有很多资料。不讲概念。先看代码

```
foreach (var p in session.CreateQuery("from Person").List())
{
    // SELECT * FROM classes where PersonId = @p0
    foreach (var c in p.Classes)
    {
        //打印课程信息|...
    }
}
```



代码很简单。也不用 ide 了，是不是很多人有这种感觉。没问题啊。这代码没有问题，我就是这么写的。也没见有啥问题。

说下场景，一个班，有学生，有课程，学生有选课，三张表，学生表，课程表和选课表。学生和课 N-N 的关系，不多讲。我们知道的是一个学生可以选多门课。

1 处获取所有学生集，好理解

2 处遍历学生关联的所有课程。然后打印。

1 处一次性加载了所有的学生实体。在学生和课程建立关系以后，1 处加载学生实体的时候，每一个学生关联的课程也一起加载，注意，这个加载仅仅加载了该学生关联所有课程的主键 id 集合。这就是 NH 设计中的 Lazy Load 思想。延迟加载的意义是减小内存消耗，按需加载。这一特性，完全是代理模式的完美应用，Perfect

看 2 处，拿到每个学生的选课 id 集合售后，我们遍历他，取出每个选课的课程信息。这个时候，NH 做了什么。其实就是 2 处循环，拿每一个课程 id 去 db 中查询。也能正确的获得信息。

问题来了，上面这段代码进行了多少次数据库访问呢？

1 处获取所有学生一次，2 处，有多课程就有多少次访问，N 次。这就是 N+1 查询。

N+1 查询并不会会有灾难性或者错误的数据返回。但是，这确是非常低效的，跟 db 通信耗时。加剧了数据库的访问次数。增加了 DB server 的压力。

我们完全可以一次数据库访问获得我们需要的所有数据，在这种场景下，应该避免 Lazy load，相反，应该一次性加载。(NH 默认启用了延迟加载)虽然可以通过配置修改 NH 默认不采用延迟加载，但是，需要 Lazy Load 的场景远远比需要一次性 Load 的场景多，。嗯，在查询的时候，可以一次性“抓取”关联集合的所有信息。

就像下面这样

```

var persons = session
    .CreateQuery("from Person p left join fetch p.classes")
    .List();

var persons=session.CreateCriteria(typeof(Person))
    .SetFetchMode("Classes", FetchMode.Eager)
    .List();

var persons = (from p in session.Query<Person>().FetchMany(x =>
x.Classes) select p).ToList();

```

图 2

- 1 处使用 Hql, 注意 fetch 关键字的使用。
- 2 处, 使用 Criteria 接口, 注意 FetchMode 的使用
- 3 处, 使用 Linq 的查寻方式。FetchMany 扩展方法的使用、

注意:这个问题是一个非常普遍非常常见的问题。之前已经说过。数据不会有问题, 问题是带来数据库的低性能。

上面的这段代码。只是一种场景而已。那么, 也有其他的场景带来了 N+1 问题。比如在循环中使用数据库访问。或者遍历一些更复杂的对象(对象中有关联的集合作为属性, 而这个集合中每一个对象又还有别的集合作为这个对象的属性。比如顾客-订单-车票, 一个顾客有多张订单与之关联, 一个订单有多张车票与之关联), 在遍历诸如这种对象的时候, 都是会带来 N+1 问题的、

所以, 对于 N+1, 需要格外的注意。找到是哪一种场景, 然后采用一些处理消除。同时对循环中的数据库访问是要格外保持敏感性的。

来看下华通内的一小段代码。

```

var temp = adminFacade.GetMemberSer().GetVipUsers(order, page, rows, sort, vipUserBo);
if (temp != null && temp.ListT != null)
{
    var formatted = temp.ListT.Select(p =>
    {
        float[] leftMoney = new float[6];
        if (p.VipId != null)
        {
            leftMoney = adminFacade.GetMemberSer().GetReliableTotalRMBAndHKD((int)p.VipId);
        }
        return new
        {
            p.VipId,
            Station = p.Station == null ? "" : p.Station.Name ?? "",
            p.LoginName,
            p.UserName,
            CardNo = p.CardNo ?? "",
            CardNum = p.CardNum ?? ""
        };
    });
}

```

图 3

这段代码想返回一个会员分页列表。并且同时要返回会员的账户余额
会员的余额是必须从消费和充值记录表中做统计运算才能算出来的。

- 1 处获取一个无余额信息的会员列表
- 2 处 Linq 循环，对每一个会员去求余额

循环中进行数据库访问，这是典型的 **N+1** 问题。破解之法，用视图或者存储过程，把每一个账户都需要统计计算的任务交到数据库端，关系运算和统计那是它的强项。在数据库端将需要进行余额计算的会员计算好了，一起返回。

可能有人会说，可以通过多表查询来避免 **N+1** 问题。是的，有的情景下，简单多表联合查询确实可以规避 **N+1** 问题。但是，**NH** 对表关系的处理是自动关联的。比如顾客关联到所有的订单，每个订单又关联到所有的车票，每张车票又关联到与车票关联的所有信息。这样一来，使用 **Nh** 来进行多表联合查询，执行查询的时候，**nh** 会生成大量的 **sql** 语句去交给 **db**，而这些 **sql** 语句，有很多很多 **sql** 语句查询的结果是我们并不需要的。这又会带来另一个性能问题—大量 **sql** 查询，以后再讲。

NH 的弱项似乎就是多表联合查询，尤其是无关系表的联合查询。至于其他一些复杂场景，比如我上面那段华通的代码，每一行记录中有字段是需要多条统计求和才能得到记录结果值的。应对这种场景，**NH** 绵软无力。

那段代码出自我之手，将问题呈现出来，使用 **NH** 要注意这个问题。

上面图 1 和图 2 是传统的 **N+1** 问题，而事实上，我们在实际应用中，场景会比他们复杂些，比如图 3 中的场景。好，既然贴出了图 3 的代码，那就再深入一点。

图 3 的代码中存在的并不是传统的 **N+1** 问题，而是另外一种 **N+1**—同一个请求下 **N+1** 查询问题。这种 **N+1** 查询的根源是单请求多 **Session**(One more Session Per Request)和传统 **N+1** 问题的复合。

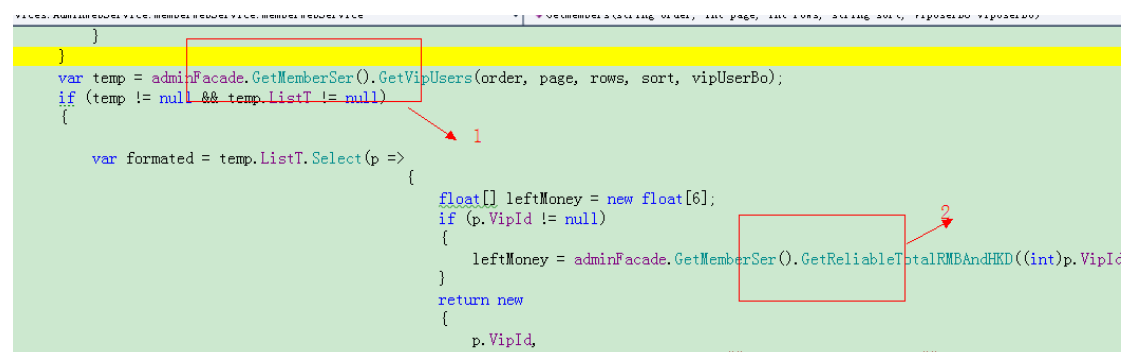


图 4

还是原来那段代码，1 处和 2 处都是 **face** 封装的接口方法。是独立的。看下具体的定义
1 处方法

```

/// <summary>
/// 返回分页数据
/// add by ys on 2013-6-4
/// </summary>
/// <param name="order">asc/desc</param>
/// <param name="page">页码</param>
/// <param name="rows">每页的记录行数</param>
/// <param name="sort">排序关键字</param>
/// <param name="vipUserBo">搜索用的bo</param>
/// <returns>返回一页的数据集合</returns>
public Page<VipUserBO> GetVipUsers(string order, int page, int rows, string sort, VipUserBO vipUserBo)
{
    #region

    try
    {
        //调用扩展的分页方法，默认降序
        using (var context = new MiniSysDataContext())
        {
            //获取分页数据
            Page<Vipuser> pagedata = context.Vipuser.GetPagedListQuery(page, rows, wheres, orderBy,
                selector, isSortOrderAsc);

            Mapper.CreateMap<Page<Vipuser>, Page<VipUserBO>>();
            Mapper.CreateMap<Vipuser, VipUserBO>();
            Mapper.CreateMap<Station, StationBO>();
            Mapper.CreateMap<Payrecord, PayRecordBO>();
        }
    }
}

```

图 5

很明显的可以看到有一个 context 的创建过程，实际上本质还是一个 Isession 创建的过程，并且使用 using 自动释放托管的 ISession。

再看 2 处的代码，跟 1 处类似

```

/// <summary>
/// 统计充值实冲余额，赠送余额，账户可用余额
/// 包括人民币和港币的，返回一个float?[]数组
/// 数据存放顺序按照索引位置分别为
/// 0处人民币实冲余额，1处港币实冲余额
/// 2处人民币赠送余额，3处港币赠送余额
/// 4处人民币账户可用余额，5处港币账户可用余额
/// add by ys on 2013-8-1
/// </summary>
/// <returns>返回浮点数组，包含6个统计数据</returns>
public float[] GetReliableTotalRMBAndHKD(int vipId)
{
    var datas = CountReliableTotalRMBAndHKD(vipId);
    return datas;
}

```

图 6

```

protected static float[] CountReliableTotalRMBAndHKD(int vipId)
{
    float[] temp = new float[6];
    float consumerRecordTotalHKD = default(float);
    float consumerRecordTotalRMB = default(float);
    float payRecordTotalHKD = default(float);
    float payRecordTotalRMB = default(float);
    float givenNumTotalHKD = default(float);
    float givenNumTotalRMB = default(float);
    using (var context = new MiniSysDataContext())
    {
        //统计该会员的港币消费总额
        consumerRecordTotalHKD = GetMemberTotalConsumerRecordHKDCountByVipUserId(context, vipId);
        //统计该会员的人民币消费总额
        consumerRecordTotalRMB = GetMemberTotalConsumerRecordRMBCountByVipUserId(context, vipId);
        //统计该会员的港币充值总额
        payRecordTotalHKD = GetMemberTotalPayNumHKDCountByVipUserId(context, vipId);
        //统计该会员的人民币充值总额
        payRecordTotalRMB = GetMemberTotalPayNumRMBCountByVipUserId(context, vipId);
        //统计该会员的港币赠送总额
        givenNumTotalHKD = GetMemberTotalGivenHKDCountByVipUserId(context, vipId);
        //统计该会员的人民币赠送总额
        givenNumTotalRMB = GetMemberTotalGivenRMBCountByVipUserId(context, vipId);
    }
}

```

图 7

有啥问题呢。

图 5 中的代码。得到最后带有余额的会员列表，这个过程中，有过几次数据库访问实例的产生和销毁？

假设一页会员数据有 10 条，获取不带余额的会员，有一次 `ISession` 的创建和销毁。紧接着使用 `Linq` 一次循环对会员进行求余额、操作，这里调用的又是独立的求余额方法，毫无疑问，因为封装方法的独立性，每一次调用都是一次 `ISession` 的创建和销毁过程。

好，问题来了，我仅仅只需要获取一页带有额度数据的会员信息，那么在这个请求中有 11 次 `ISession` 实例的产生和销毁过程。即使 DB 可能有一些连接池，缓存了刚刚数据连接(不敢确定，需要实验)下次连接起来可以非常快。

这仍然会带来严重的性能问题。首先是 `WebServer` 端，11 次的 `ISession` 的创建和销毁是不敢想象的，`Session` 的创建过程要创建很多相关实例，用于跟踪与 db 通信过程中的数据状态，相当于创建一个活动现场把。一个 `request`，我们只需要一次 `Session` 的创建和销毁，就已经能够达到我们的目的。其次才是 DB 端，11 次的 `Session` 创建后，都有 `sql` 产生给 db 执行。其实，这些 `sql` 可以在一个 `Session` 中产生，一次性交给 db 批处理执行掉(这是理想中的状态，严格的来说，NH 做不到这点，所以取代的办法是存储过程或者视图)。

注意看图 7 的 1 处，可以看到有一个 `context` 参数，为什么，余额统计方法又是独立的，最后的余额值是需要用多个统计值计算得出的。这些统计方法内部如果又有 `Context` 的实例创建的话，那么无疑会有更多 `ISession` 的产生和销毁操作。所以这里，将 `context` 作为参数传入。四个统计方法公用一个 `context`，减少数据库访问实例的产生和销毁次数，

如果不那么做，会有多少次的 `ISession` 的产生和销毁呢， $1+10*6=61$ ，一共会有 61 次的 `ISession` 的创建和销毁。不敢想象把，一页 10 数据可能没什么问题，但是这种情况下，一页如果有 500 行数据， $1+500*6=3001$ 次，一次请求会有 3001 次的 `ISession` 产生和销毁过程。GC 的

压力是不能忽视的。呵呵。

有时候，对于一些方法，`context` 作为方法参数，外层公用同一个 `DateContex` 对象，也是减少 `Session` 产生个数，提高性能的一种处理手段。

好了。华通中这个问题，叫作，单次请求中产生多个 `Session`(One more Session Per Request)，属于更加复杂的一种 **N+1** 问题。有没有办法解决呢。自然有，针对整套框架采用 **One Session Per Request** 模式。保证在一次请求中，只产生一个 `ISession`，所有的数据库访问公用一个 `ISession`。但是，现在来说，这种模式能不能稳定的适用在华通项目中是一个问题，另外一个，改动带来的影响太大了。所以只能暂时这样了。**Mark** 一下，嗯，有这问题存在。后期有足够时间再实验这种模式在大型项目中的稳定应用。

这次分享，主要是介绍两种 **N+1** 问题，第一种是传统的 **N+1** 查询问题，循环中进行 `db` 访问，第二种则是一次请求多个 `Session` 所带来的 **N+1** 问题。

两者带来的性能问题都是不乐观的。

2013-10-24