

DotNet架构设计和性能优化

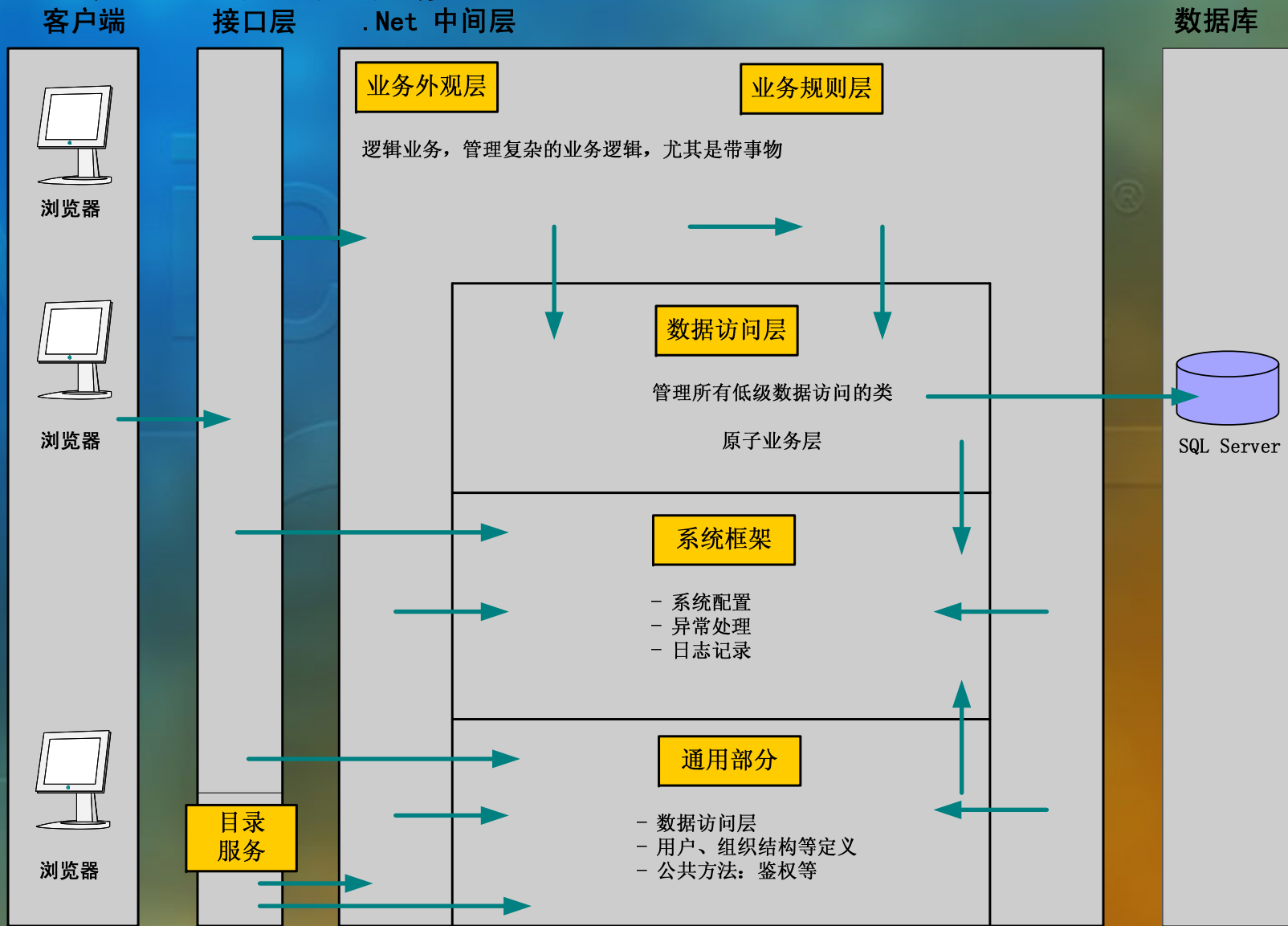
.Net 分层结构

- ◆ **Microsoft** 最近几年为降低企业范围内应用程序的复杂性而开发了这个结构。
- ◆ 这一模型有时被称为“多层结构”(Multitier)，原因是它可以非常容易地按照逻辑分组来加以理解。

企业级模板和多层分布式应用程序

层	功能
用户服务	Windows 用户界面和/或 Web 用户界面、Web 服务
业务服务	业务外观、业务规则
数据访问和持久性	数据访问
其他	系统框架

分层结构模型



备注：箭头表示系统层次间的调用关系

用户界面层

- ◆ 用户服务是指在应用程序中实现的客户端，包括 **Web** 客户端或 **Windows** 客户端。
 - ❖ 管理 **Web** 页或 **Windows** 界面的呈现和行为
 - ❖ 显示数据
 - ❖ 捕获数据
 - ❖ 数据验证检查
 - ❖ 为用户提供任务指南
 - ❖ 向“业务外观”发送用户输入
 - ❖ 从“业务外观”接收结果
 - ❖ 向用户显示错误

业务外观层

- ◆ 向基础业务对象提供一致的接口，并将客户端同基础业务逻辑的更改隔离开。该块存在时，它活跃于 **WebUI** 客户端（或 **WinUI** 客户端）与业务逻辑层之间。
 - ❖ 从“用户服务”层（**Windows** 用户界面或 **Web** 用户界面客户端应用程序）接收用户输入。
 - ❖ 如果请求需要对数据进行只读访问，则可能使用“数据访问”层。
 - ❖ 将请求传递到“业务规则”层。
 - ❖ 将响应从“业务规则”层返回到“用户服务”层。
 - ❖ 在对“业务规则”层的调用之间维护临时状态。

业务规则

- ◆ 包含业务对象本身以及应用于它们的规则。这也是主要业务对象所在的位置。它们实现业务实体或系统对象。系统的业务规则将在这些对象中编码，尽管部分业务规则可能实际上已在数据库的存储过程和触发器中进行了编码。
 - ❖ 从“业务外观”层接受请求。
 - ❖ 根据编码的业务规则处理请求。
 - ❖ 使用“数据访问”层从“数据服务”层获取数据或将数据发送到“数据服务”层。
 - ❖ 将处理结果传递回“业务外观”层。

数据访问层

- ◆ 执行从数据库（或其他数据服务）获取数据或向数据库发送数据的功能。
 - ❖ 从“业务规则”层接收请求，从“数据服务”获取数据或向其发送数据。
 - ❖ 使用存储过程获取数据，并可选用 **ADO.NET** 向数据库发送数据。
 - ❖ 将数据库查询结果返回到“业务规则”层，作为强类型的 **ADO.NET** 数据集。

系统框架层

- ◆ 提供系统服务、系统基础结构功能或其他共享功能的应用程序的组件。这一功能可能不是特定于任何给定的应用程序的。
- ◆ 可能包括 **HTML** 页缓存服务或对象查询缓存（名称服务/负载平衡服务）。
- ◆ “系统框架项目”层可由结构中的任何层使用。此外，强类型的数据集驻留于系统类型层。当某些共享功能只在一个应用程序中使用，而其他共享功能在若干个应用程序之间共享时，可以考虑将它们分成不同的系统类型项目。

一个具体实现的例子

Microsoft®

Internet

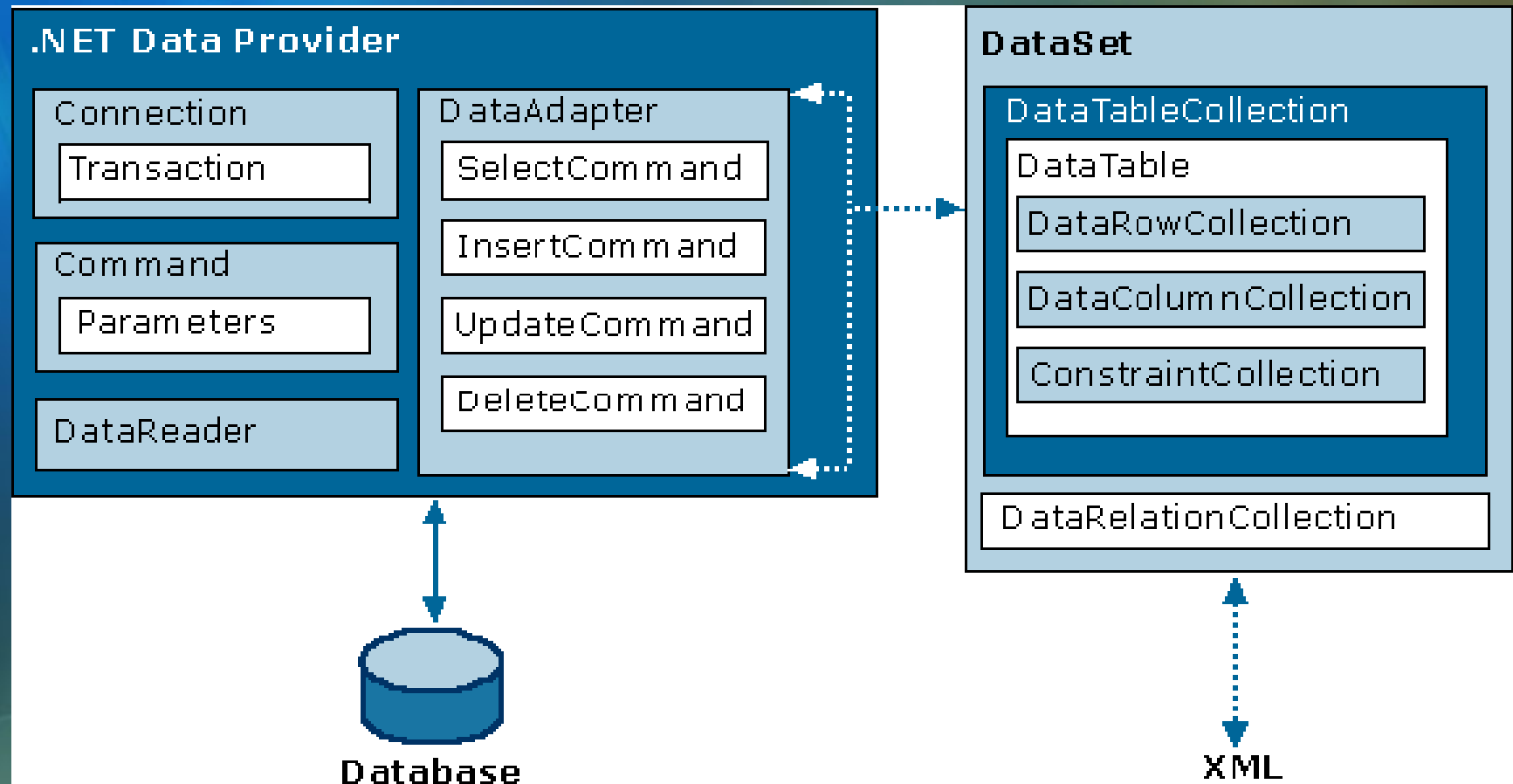
系统框架层

- ◆ 定义BusinessException
- ◆ 定义公用常量、属性和方法
- ◆ 定义通用调试环境变量

数据访问层

- ◆ 底层数据访问控件
 - ❖ 采用微软的**DataAccess**组件
 - ❖ 尽晚打开，尽早关闭连接
 - ❖ 连接池由**.Net**进行维护
- ◆ 自定义的数据对象
- ◆ 原子业务逻辑

ADO.NET 构成



ADO.NET对象

- ◆ **Connection** : 描述与特定数据源的连接
- ◆ **Command** : 操作数据库的数据, 可执行sql命令, 也可调用存储过程
- ◆ **DataReader** : 浏览数据, 从数据源中获得只读、只能向前访问的数据流
- ◆ **DataAdapter** : 生成 **DataSet** , 更新数据源
- ◆ **DataSet**: 多个表、约束、关系的集合, 可以认为是内存中的数据库
- ◆ **DataView**: 对**DataSet**进行过滤、排序

ADO的运作方式

- ◆ DataSet的运作方式
- ◆ DataReader的运作方式

连接漏洞

- ◆ 在应用程序压力很小的时候,数据库连接数量持续增加,直至达到连接池大小的极限
- ◆ 负载稳定但比较大的情况下,应用程序开始建立越来越多的连接很快达到最大连接数
- ◆ 应用程序对**Open**方法, 引发超时的异常
- ◆ 应用程序单个运行很好,但随着用户量的上升,很快速度变慢,渐渐崩溃
- ◆ 应用程序在压力运行下反应缓慢,并显示出数据库机器低处理器的使用
- ◆ **IIS**或**Application Server CPU**空闲, 数据库服务器**CPU**也空闲, 但是数据库不断发生某个库或表的堵塞或锁

连接字符串的秘密

- ◆ ConnString="Server=local;database=northwind;UserID=aaa;Password=bbb;connection reset=false; connection lifetime=5;enlist=true;min pool size=1;max pool size=50;
- ◆ Config文件
<appSettings>
 <add key="NorthWind" Value="data source=localhost....userid
 ...pass"
</appSettings>
ConnectionString = ConfigurationSettings.AppSettings("NorthWind")
- ◆ 直接使用
- ◆ 保存在一个单独的DLL组件中
- ◆ COM+ 中连接字符串
- ◆ 加密装置
- ◆ 连接字符串也会影响到ADO.NET的数据库连接池机制

ADO 连接池的算法

- ◆ **Connection.Open** 连接池机制被启动
- ◆ 首先检查这个连接字符串下面是否有一个连接池.
- ◆ 如果没有则建立一个新的连接,并且建立一个连接池
- ◆ 如果已经有,那么看池中是否有连接,如果有则成功
- ◆ 如果没有,那么看是否达到最大连接数量,如果没有则建立一个新的连接,池数量加1
- ◆ 如果已经到达池最大连接数,那么等待**30秒**,**30秒**后看是否有可用连接.
- ◆ 如果这时有那么使用这个连接,如果没有则报错

连接池的秘密

- ◆ 每个连接池可以打开的连接数量是有限的,这个可以在连接字符串中进行修改.每个数据库**Provider**提供一个缺省的值,并且设置方法不同
- ◆ 一个连接池为一个应用程序域服务,一个应用程序域可能有一个以上的连接池。每个**ASP.NET**应用程序被加载到工作进程(**aspnet_wp.exe**)中的一个应用程序域中.应用程序域消失时池也销毁。
- ◆ 连接池是基于连接字符串的,不同的连接字符串会生成不同的连接池(字母大小写)
- ◆ 连接池也需要匹配身份,当使用信任连接时,会鉴别不同的身份,身份不同连接池也不同。
- ◆ 调用**Close**或**Dispose**方法会将打开的连接返回到连接池.不过不调用,那么最终会被垃圾回收器销毁,而且不会回到连接池中。

使用的原则

- ◆ 总是相同的连接字符串和同一数据库通信,这将确保连接会回到同一个连接池中.
- ◆ 总是调用**Close**或**Dispose**.这将确保使用过的连接再次回到池中.
- ◆ 尽可能晚的打开连接,而且尽可能早的关闭.提高使用效率,避免等待引起的超时.
- ◆ 保证每个连接的单调性.通过提交或回滚来清算所有在这个连接上的事务,这样你不会有数据库被锁而连接限制在池中的情况.消除连接上的所有临时对象,特别是临时表.
- ◆ 连接池是客户端技术,数据库并不知道和了解它的机制.

建议

- ◆ 尽量多的使用存储过程.
- ◆ 索引
- ◆ 加密**SQL**语句或用户名密码
- ◆ 密码强制超过**8**位,并且数字、字母和符号组合
- ◆ 大块调用, 实用为主
- ◆ 异步通信/使用**Windows Services**
- ◆ 归档和搬移
- ◆ 事务最小
- ◆ **COMInterop**的大块调用
- ◆ 减少往返, 将多个结果一次返回
- ◆ 使用**StringBuilder**
- ◆ **ExecuteScalar, ExecuteNoQuery**

原子业务

- ◆ 定义对单表的操作
 - ❖ 例如：生成自增长序号
 - ❖ **INSERT INTO tbSeqBusiSerialNo (cnvcFill) VALUES ('0');**
SELECT scope_identity() AS id

业务逻辑层

- ◆ 定义两个统一的数据处理流程模板
 - ❖ 带事务处理和不带事务处理
- ◆ 跨数据库事务
 - ❖ 存储过程

```

//取数据库连接串
string strDbBoss = "";
//初始化输出包
DataSet dsOut = new DataSet();
//连接数据库
using (SqlConnection conn = new SqlConnection(strDbBoss))
{
    //打开数据库连接
    conn.Open();

    //事务类
    using (SqlTransaction trans = conn.BeginTransaction(IsolationLevel.ReadCommitted, "SampleTransaction"))
    {
        DASample DASampleInst = new DASample();

        //查询类
        DASampleInst = new DASample();
        try
        {
            //原子事务1取操作员信息
            DataTable dtOper = DASampleInst.GetOperData(conn, trans, "strOperID", "strOperPwd");

            if(dtOper == null || dtOper.Rows.Count == 0)
                throw new BusinessException("BFSample:", "E1002");
            dtOper.TableName = "tbOper";
            dsOut.Tables.Add(dtOper);
            //原子事务2...
            //原子事务3...
        }
        catch (BusinessException bx)
        {
            //业务异常
            {
                //处理错误返回包
                //dsOut = ReturnMsg.ErrorDataSet(bx);
            }
            catch (SQLException sqllex) //数据库异常
            {
                //处理错误返回包
                //dsOut = ReturnMsg.ErrorDataSet(sqllex);
            }
            catch (Exception ex) //其他异常
            {
                //处理错误返回包
                //dsOut = ReturnMsg.ErrorDataSet(ex);
            }
        }
        finally
        {
            //释放连接
            conn.Close();
        }
    }
    //返回数据
    return dsOut;
}

```



```

//取数据库连接串
string strDbBoss = "";
//初始化输出包
DataSet dsOut = new DataSet();
//连接数据库
using (SqlConnection conn = new SqlConnection(strDbBoss))
{
    //打开数据库连接
    conn.Open();
    //查询类
    SqlTransaction trans = null;
    DASample DASampleInst = new DASample();
    try
    {
        //原子事务1取操作员信息
        DataTable dtOper = DASampleInst.GetOperData(conn, trans, "strOperID", "strOperPwd");

        if(dtOper == null || dtOper.Rows.Count == 0)
            throw new BusinessException("BFSample:", "E1002");
        dtOper.TableName = "tbOper";
        dsOut.Tables.Add(dtOper);
        //原子事务2...
        //原子事务3...
    }
    catch (BusinessException bx) //业务异常
    {
        //处理错误返回包
        dsOut = ReturnMsg.ErrorDataSet(bx);
    }
    catch (SqlException sqllex) //数据库异常
    {
        //处理错误返回包
        dsOut = ReturnMsg.ErrorDataSet(sqllex);
    }
    catch (Exception ex) //其他异常
    {
        //处理错误返回包
        dsOut = ReturnMsg.ErrorDataSet(ex);
    }
    finally
    {
        //释放连接
    }
}
//返回数据
return dsOut;
}

```

接口层

- ◆ Remoting or Webservice

Web表现层

- ◆ 根据数据库中用户权限表动态生成 OutlookBar 菜单
 - ❖ user、right、menuitem
- ◆ 代码重用：抽象web基类页面和 UserControl 等（含权限控制,Session 控制、显示）等
- ◆ 采用 Application、Session、Cache、Hashtable 等进行数据共享和性能优化

性能优化-页面缓存

◆ 页面缓存

```
<%@ OutputCache Duration= "20"  
    VaryByParam="none"%>
```


性能优化-变量缓存

- ◆ Using ASP.NET Cache
- ◆ Using web.config Variables
- ◆ Using **Session** and **Application** Variables
- ◆ Saving **Session** and **Application** Variables in a Database

ASP.NET Cache

◆ASP.NET Cache:

- ❖ 在应用程序中存储对象和变量以便重复使用

◆Placing Objects in ASP.NET Cache(frm8)

```
Cache.Insert("mykey", myValue, _  
            Nothing, DateTime.Now.AddHours(1), _  
            TimeSpan.Zero)
```

◆Retrieving Objects From ASP.NET Cache

```
myValue = Cache("mykey")  
If myValue <> Nothing Then  
    DisplayData(myValue)  
End If
```

使用web.config变量(frm11)

◆在 web.config存储变量

```
<configuration>  
  <appsettings>  
    <add key="pubs" value=  
      "server=localhost;uid=sa;pwd=;database=pubs"/>  
  </appsettings>  
</configuration>
```

◆在.aspx或Global.asax文件中调用

```
Dim appSetting As NameValueCollection  
Dim strConn As String  
appSetting = CType(Context.GetConfig _  
  ("appsettings"), NameValueCollection )  
strConn = appSetting("pubs").ToString()
```

使用Session和Application变量

- ◆ **Session** 为每个不同用户存储变量

```
Sub Session_Start(s As Object, e As EventArgs)  
    Session("BackColor") = "beige"  
    Session("ForeColor") = "black"  
End Sub
```

- ◆ **Application** 存储的变量是所有用户共用的

```
Sub Application_Start(s As Object, e As EventArgs)  
    Application("NumberofVisitors") = 0  
End Sub
```


性能优化-内存中的检索

- ◆ 内存检索：
 - ❖ **DataSet**: 检索用**find**,不能用**fielter**或**Select**
 - ❖ **Arraylist**
 - ❖ 两级 **HashTable**
 - ❖ 存储用户资料、参数、套餐等
- ◆ 操作间数据交换：文件
 - ❖ 瓶颈在数据库存储
 - ❖ 保留文件，便于回退和错误处理
 - ❖ 避免不稳定

性能优化-SessionState

- ◆ 当不使用会话状态时禁用它
 - ❖ `<%@ Page EnableSessionState="false"%>`。
- ◆ 需要访问会话变量，但不打算创建或修改它们，则将 `@ Page` 指令中的 `EnableSessionState` 属性设置为 `ReadOnly`

性能优化-ServerControl

- ◆ 在适当的环境中使用 **ASP.NET** 服务器控件
- ◆ 只在必要时保存服务器控件**ViewState**
 - ❖ **ViewState**服务器控件的功能，该功能使服务器控件可以在往返过程上重新填充它们的属性值（您不需要编写任何代码）。
 - ❖ 因为**ViewState**在隐藏的窗体字段中往返于服务器，所以该功能确实会对性能产生影响。您应该知道在哪些情况下视图状态会有所帮助，在哪些情况下它影响页的性能。

性能优化-不要依赖代码中异常

- ◆ 异常大大地降低性能，所以您不应该将它们用作控制正常程序流程的方式。不要在处理该状态之前捕获异常本身。

```
try { result = 100 / num; }  
catch (Exception e)  
    { result = 0; }
```

```
if (num != 0)  
    result = 100 / num;  
else  
    result = 0;
```


性能优化-数据库SQL

- ◆ 使用存储过程
- ◆ **SqlDataReader** 类用于快速只进数据游标
- ◆ 日志、索引和数据文件分布到不同通道的硬盘上
- ◆ 大批量数据入库: **bcp**
 - ❖ 带五个索引, 1万条约1秒

性能优化-一定要禁用调试模式

- ◆ 如果启用了调试模式，应用程序的性能可能受到非常大的影响

WebService调试

- ◆ 当成本地类引用便于调试
- ◆ 部署时再远程引用
- ◆ 注意对象的序列化

各服务器内存利用率走势图	<i>Memory\ Available Bytes, Memory\ Committed Bytes</i>
各服务器磁盘利用率的走势图	<i>PhysicalDisk\%Disk Time</i>
各服务器网络带宽利用率的走势图	<i>Bytes Received/sec, Byte Send/sec, Bytes Total/sec</i>
SQLServer:General Statistics每秒钟登陆到SQL Server的数量	<i>Logins/sec</i>
SQLServer:Cache Manager显示在cache中找到数据的比率。这个数据持续小于85%表明有内存问题。	<i>Cache Hit Ratio (all instances)</i>
SQLServer:General Statistics SQL 活动用户的数量。参考ASP.NET Application中的Request/sec，可以知道测试用例是否对SQL Server作了足够的压力测试。	User Connections
SQL Server:Databases事务数	<i>Transactions/sec</i>
SQLServer:Locks每秒锁住的请求数。如果这个值持续大于0说明事务有问题。	<i>Lock Waits/sec</i>
RPS（每秒请求）和 并发用户数的走势图	<i>requests/sec(avg) vs Browser connections</i>
PPS（每秒处理页面）和并发用户数的走势图	<i>PPS = RPS / 每页面requests</i>
请求处理错误率	<i>error/all%</i>
响应时间（TTLB/TTFB）和并发用户数的走势图	<i>Time to Last Byte (avg) vs Bowser connectionsr</i>
每秒执行的请求数	Requests/sec
失败的请求数Requests failed	
超时的请求数Requests timed out	
Requests Queued	

Microsoft[®]

Thanks