

仓储大叔讲系统架构

专注于 dotNet 架构与设计

主讲：仓储大叔，洋名：Lind

博客地址：<http://www.cnblogs.com/lori>

第一讲	说说架构师.....	3
第二讲	基础篇.....	4
第三讲	仓储模式.....	7
第四讲	缓存篇.....	8
第五讲	日志与异常处理及 Dispose 模式.....	11
第六讲	分布式的 Pub/Sub 和消息队列.....	17
第七讲	领域对象，事件总线 and 领域事件.....	19
第八讲	Redis 集群与客户端使用	19
第九讲	MongoDB 集群与客户端使用.....	21
第十讲	Cat 集群与使用.....	25
第十一讲	分布式文件存储和文件上传的设计.....	26
第十二讲	消息组件和第三方支付的统一.....	27
第十三讲	数据包和网络通讯.....	29
第十四讲	ORM 和分布式事务.....	33
第十五讲	MVVM 和 KnockoutJS 的介绍.....	33
第十六讲	任务调度 Quartz.....	34
第十七讲	多并程与并行.....	36
第十八讲	： IOC 原理 和统一的 IOC 容器.....	38
第十九讲	： API 安全与校验.....	44
第二十讲	： 领域驱动的设计模式.....	45

第二十一讲：Bootstrap 和后台管理系统.....	51
第二十二讲：使用 xamarin 进行移动开发.....	53
第二十三讲：Node.js 与 Sails 框架.....	59
第二十四讲：跨语言开发 Thrift 框架.....	63
第二十五讲：什么样的代码需要重构.....	66
第二十六讲：基础篇续~接口与抽象类，集合与数组，树与链表.....	67
第二十七讲：设计模式在项目中很自然的出现了.....	68
第二十八讲：单点统一登陆 SSO 的设计.....	68
第二十九讲：Session 共享与 WEB 集群.....	68
第三十讲：Lind.DDD 设计初衷与它的未来（代码需要人性化）	68

第一讲 说说架构师

架构师要干什么

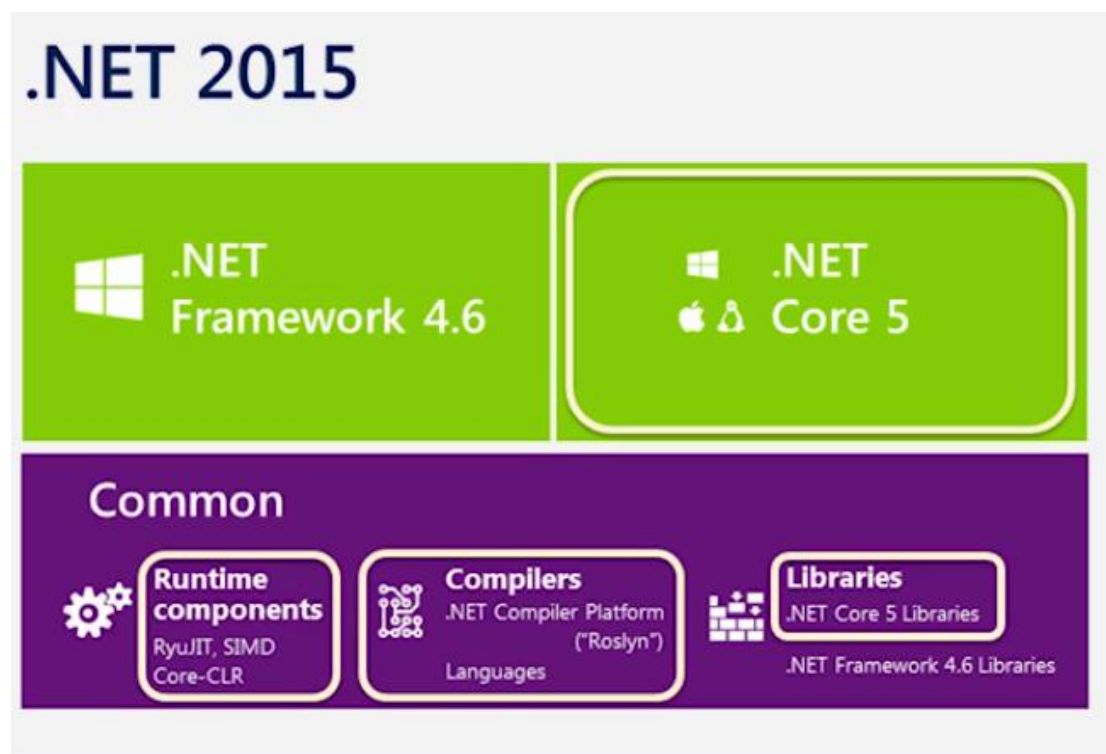
- 核心代码
- 新技术培训
- 框架设计
- 数据库设计
- 模型设计
- 核心疑难问题解决

如何成为一名合格的架构师

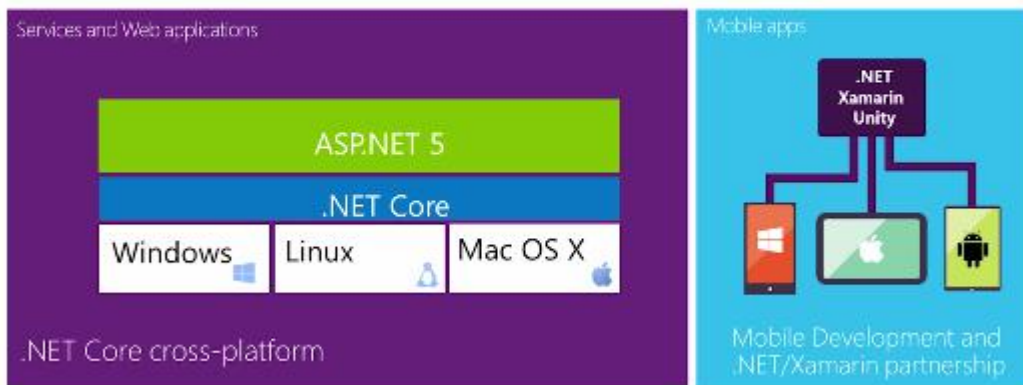
- 勤奋
- 善于总结
- 写代码比说话多

当前 dotNet 架构师应该掌握的

- 一套常用核心组件,自己维护,升级
- 大数据
- 关系型数据库与 NoSql
- 高并发
- 分布式
- 缓存技术
- 仓储设计
- 移动开发免费了



.NET Cross Platform



第二讲 基础篇

接口

一种操作规范，实现多态的基础，我们应该面向接口编程

```
#region 级别日志
/// <summary>
/// 将message记录到日志文件
/// </summary>
/// <param name="message"></param>
void Logger_Info(string message);
/// <summary>
/// 异常发生的日志
/// </summary>
/// <param name="message"></param>
void Logger_Error(Exception ex);
/// <summary>
/// 调试期间的日志
/// </summary>
/// <param name="message"></param>
void Logger_Debug(string message);
/// <summary>
/// 引起程序终止的日志
/// </summary>
/// <param name="message"></param>
void Logger_Fatal(string message);
/// </summary>
```

抽象类

一个种族的概念，一个拥有最基本属性和方法的类，它为派生类提供了这些公用的属性和行为

```

/// <summary>
/// 日志核心基类
/// 模版方法模式，对InputLogger开放，对其它日志逻辑隐藏，InputLogger可以有多种实现
/// </summary>
internal abstract class LoggerBase : ILogger
{
    private string _defaultLoggerName = DateTime.Now.ToString("yyyyMMddhh") + ".log";

    /// <summary>
    /// 日志文件地址
    /// 优化级为mvc方案地址，网站方案地址，console程序地址
    /// </summary>
    protected string FileUrl[...]

    /// <summary>
    /// 日志持久化的方法，派生类必须要实现自己的方式
    /// </summary>
    /// <param name="message"></param>
    protected abstract void InputLogger(string message);
}

```

枚举

定义一些枚举的值，它默认集成自 int，如果你的元素少，完整可以集成 short

Flags 特性：表示枚举支持位运算

```

/// <summary>
/// Http请求方式，支持位运算
/// </summary>
[Flags]
public enum HttpMethodFlags
{
    /// <summary>
    /// Get请求
    /// </summary>
    GET = 1,
    /// <summary>
    /// Post请求
    /// </summary>
    POST = 2,
    /// <summary>
    /// Put请求
    /// </summary>
    PUT = 4,
    /// <summary>
    /// Delete请求
    /// </summary>
    DELETE = 8,
}

```

位运算：加，减，包含

1 与运算可以用来判断某个数是位在另外一个数中存在：

```
10 & 2 //结果为2, 大于0表示操作数2在操作数1中存在 10 = 8+2
```

2 或运算可以用来将两个数相加在一起

```
8 | 2 //结果为10, 10 | 2 结果还是10, 你可以把它转换为二进制看一下, (1000 | 10 为1010,
```

3 非运算和与运算在一起使用, 可以将一个元素从一个集合里去除

```
4 & (~2) //结果为4, 4里不包含2, 所以直接返回原值4  
3 & (~2) //结果为1, 3由1和2组成, 所以去掉2后原值变为1
```

4 左移运算, 每移1位相当于乘2

```
4 << 2 //结果为16
```

5 右移运算, 每移1位相当于除2

```
4 >> 2 //结果为1
```

事件和委托

提供一种方法的原型, 实现方法的回调, 这个概念在 JS 中经常可以看到, 一个被封装的框架, 如果希望让外界调用它的已有方法, 需要使用方法回调的方式, 由外界提供方法体, 架构内容实现对它的调用。

```
CatClient.DoTransactionAction("Do", "Add", () =>  
{  
    CatClient.LogRemoteCallServer(CatClient.GetCatContextFromServer());  
    CatClient.LogEvent("Do", "Add", "0", "hello distribute api 4 5 6");  
    CatClient.LogError(new Exception());  
});
```

```

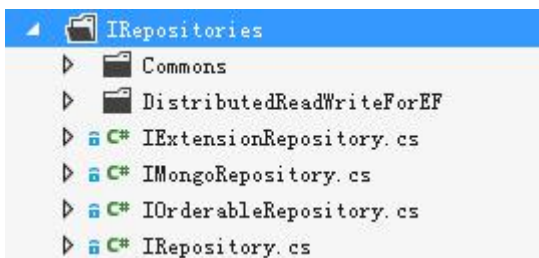
/// <summary>
/// 发布分布式事务，不返回上下文
/// </summary>
/// <param name="type"></param>
/// <param name="name"></param>
/// <param name="action"></param>
public static void DoTransactionAction(string type, string name, Action action)
{
    var tran = NewTransaction(type, name);
    try
    {
        action();
    }
    catch (Exception ex)
    {
        LogError(ex);
        tran.SetStatus(ex);
        throw;
    }
    finally
    {
        tran.Complete();
    }
}

```

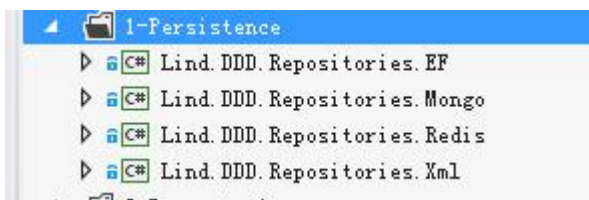
第三讲 仓储模式

架构中的面向接口是核心，IOC 开始登场，大叔仓储模式的最佳实战(EF,Mongo,Redis,XML)

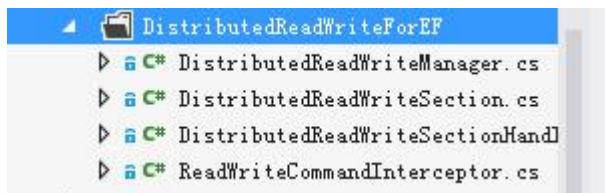
仓储接口规范



对仓储的实现，数据的持久化方式是多种多样的



E F 读写分离



实质是对 SQL 的一种拦截

```
/// <summary>
/// Linq to Entity生成的update,delete
/// </summary>
/// <param name="command"></param>
/// <param name="interceptionContext"></param>
public override void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<int> interceptionContext)
{
    base.NonQueryExecuting(command, interceptionContext); //update,delete等写操作直接走主库
}

/// <summary>
/// Linq to Entity生成的update,delete
/// </summary>
/// <param name="command"></param>
/// <param name="interceptionContext"></param>
public override void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<int> interceptionContext)
{
    base.NonQueryExecuting(command, interceptionContext); //update,delete等写操作直接走主库
}
```

第四讲 缓存篇

文章索引：<http://www.cnblogs.com/lori/p/3990744.html>

缓存术语

GET：得到缓存，有直接返回，没有查询实际数据并进行返回，同时进行Put操作

Put:插入一个缓存，当GET不到缓存时，直接Put一个缓存进去

Delete:删除缓存，当数据有更新时，Delete这个缓存

静态页缓存

<http://www.cnblogs.com/lori/p/4012823.html>

通过 HttpModule 对 HTTP 请求进行拦截，并以 URL 地址为键，将 HTTP 相应结果存储到磁盘，当下次请求到来时，首先检测缓存里是否有，如果有，就直接返回 HTML 文件，否则就生成新的 HTML 文件缓存，避免了服务端和数据库的开销。

```
public void Init(HttpApplication context)
{
    context.BeginRequest += new EventHandler(context_BeginRequest);
    context.ReleaseRequestState += new EventHandler(context_ReleaseRequestState);
}
```

页面是否需要被缓存，主要通过扩展名 (html,shtml)来确定的，所以使用这种静态页的方式进行缓存，首先需要我们做 URL 重写。扩展名我们可以进行配置。


```

if (IsNeedCache(application)) //检测当前请求是否需缓存
{
    string key = string.Empty; ;
    string extend = VirtualPathUtility.GetExtension(application.Context.Request.FilePath).ToLower();
    if (IsClearCache(application, out key))
    {
        if (CacheManage.Container(key, extend))//缓存中存在，则清除缓存，结束请求
        {
            application.Context.Response.Write(CacheManage.Remove(key, extend));
        }
        application.CompleteRequest();
    }
}

```

数据缓存

<http://www.cnblogs.com/lori/p/4010862.html>

将数据存储到一个公用的存储区域，它可以有自己的过期时间，一般不能主动失效

```

/// <summary>
/// 导航生产类
/// </summary>
public static class BannerBreadFactory
{
    /// <summary>
    /// BannerBread单例实例对象
    /// </summary>
    static readonly List<BannerBread> Instance = new List<BannerBread>();

    static BannerBreadFactory()
    {
        Instance.Add(new BannerBread { ID = 1, DisplayTitle = "首页", Parent = null, ParentID = 0, Url = "/" });
        Instance.Add(new BannerBread { ID = 2, DisplayTitle = "关于我们", Parent = null, ParentID = 1, Url = "/Home/About" });
        Instance.Add(new BannerBread { ID = 3, DisplayTitle = "联系我们", Parent = null, ParentID = 1, Url = "/Home/Contact" });
    }
}

```

分布式数据集缓存

<http://www.cnblogs.com/lori/p/4063807.html>

在方法签名上添加特性，它的失效依赖于其它的方法，如 GET 方法，它的结果集被缓存，它的失效可能依靠于 update, create 和 delete 方法。

```

/// <summary>
/// 表示由此特性所描述的方法，能够获得来自Microsoft.Practices.EnterpriseLibrary.Caching基础结构层所提供的缓存机制。
/// </summary>
[AttributeUsage(AttributeTargets.Method, AllowMultiple=false, Inherited=false)]
public class CachingAttribute : Attribute
{
    #region Ctor
    /// <summary>
    /// 初始化一个新的<c>CachingAttribute</c>类型。
    /// </summary>
    /// <param name="method">缓存方式。</param>
    public CachingAttribute(CachingMethod method)
    {
        this.Method = method;
    }
    #endregion
}

```

统一的缓存操作接口

```

/// <summary>
/// 表示实现该接口的类型是能够为应用程序提供缓存机制的类型。
/// </summary>
public interface ICacheProvider
{
    #region Methods
    /// <summary>
    /// 向缓存中添加一个对象。
    /// </summary>
    /// <param name="key">缓存的键值，该值通常是使用缓存机制的。</param>
    /// <param name="valKey">缓存值的键值，该值通常是由使用缓存机制的。</param>
    /// <param name="value">需要缓存的对象。</param>
    void Add(string key, string valKey, object value);
    /// <summary>
    /// 向缓存中更新一个对象。
    /// </summary>
    /// <param name="key">缓存的键值，该值通常是使用缓存机制的。</param>
    /// <param name="valKey">缓存值的键值，该值通常是由使用缓存机制的。</param>
    /// <param name="value">需要缓存的对象。</param>
    void Put(string key, string valKey, object value);
    /// <summary>
    /// 从缓存中读取对象。
    /// </summary>
    object Get(string key, string valKey);
    #endregion
}

```

具体缓存的实现

```

/// 向缓存中添加一个对象。
/// </summary>
/// <param name="key">缓存的键值，该值通常是使用缓存机制的方法的名称。</param>
/// <param name="valKey">缓存值的键值，该值通常是由使用缓存机制的方法的参数值所产生。</param>
/// <param name="value">需要缓存的对象。</param>
public void Add(string key, string valKey, object value)
{
    Dictionary<string, object> dict = null;
    if (_cacheManager.Contains(key))
    {
        dict = (Dictionary<string, object>)_cacheManager[key];
        dict[valKey] = value;
    }
    else
    {
        dict = new Dictionary<string, object>();
        dict.Add(valKey, value);
    }
    _cacheManager.Add(key, dict);
}

```

缓存在程序中的使用

```

public interface IUserService
{
    [Caching(CachingMethod.Get)]
    PagedList<WebManageUsers> GetWebManageUsers(PageParameters pp);
    [Caching(CachingMethod.Get)]
    PagedList<WebManageUsers> GetWebManageUsers(Expression<Func<WebManageUsers, bool>> predicate, PageParameters pp);
    [Caching(CachingMethod.Remove, "GetWebManageUsers")]
    void InsertManageUsers(NLayer_IoC_Demo.Entity.WebManageUsers entity);
}

```

第五讲 日志与异常处理及 Dispose 模式

项目中日志组件是必须的，第三方的日志组件很多，log4Net,commonLog,大叔在学习了这些第三方组件之后，自己也整理了一套日志组件，即 Lind.DDD.Logger，它有多种日志持久化的方式，同时也集成了 log4net 这种方法。

特点：

- 一、日志记录代码执行时间
- 二、日志记录代码中出现的异常
- 三、日志分为多种级别（ debug,warn,info,error,fatal ），在调试和生产环境灵活的切换
- 四、日志标准采用 IOC 进行生产，当不配置 IOC，则使用文件进行记录
- 五、日志组件让我们更好的学习设计模式，工厂方法，单例和策略模式

日志接口：

```

#region 功能日志
/// <summary>
/// 记录代码运行时间，日志文件名以codeTime开头的时间戳
/// </summary>
/// <param name="message">消息</param>
/// <param name="action">所测试的代码块</param>
void Logger_Timer(string message, Action action);
/// <summary>
/// 记录代码运行异常，日志文件名以Exception开头的时间戳
/// </summary>
/// <param name="message">消息</param>
/// <param name="action">要添加try...catch的代码块</param>
void Logger_Exception(string message, Action action);
#endregion

```

```

#region 级别日志
/// <summary>
/// 将message记录到日志文件
/// </summary>
/// <param name="message"></param>
void Logger_Info(string message);
/// <summary>
/// 异常发生的日志
/// </summary>
/// <param name="message"></param>
void Logger_Error(Exception ex);
/// <summary>
/// 调试期间的日志
/// </summary>
/// <param name="message"></param>
void Logger_Debug(string message);
/// <summary>
/// 引起程序终止的日志
/// </summary>
/// <param name="message"></param>
void Logger_Fatal(string message);
/// <summary>
/// 引起警告的日志
/// </summary>
/// <param name="message"></param>
void Logger_Warn(string message);
#endregion

```

日志级别：

```

/// <summary>
/// 日志级别：DEBUG|INFO|WARN|ERROR|FATAL|OFF
/// </summary>
internal enum Level
{
    /// <summary>
    /// 记录DEBUG|INFO|WARN|ERROR|FATAL级别的日志
    /// </summary>
    DEBUG,
    /// <summary>
    /// 记录INFO|WARN|ERROR|FATAL级别的日志
    /// </summary>
    INFO,
    /// <summary>
    /// 记录WARN|ERROR|FATAL级别的日志
    /// </summary>
    WARN,
    /// <summary>
    /// 记录ERROR|FATAL级别的日志
    /// </summary>
    ERROR,
    /// <summary>
    /// 记录FATAL级别的日志
    /// </summary>
    FATAL,
    /// <summary>
    /// 关闭日志功能
    /// </summary>
    OFF,

```

日志组件抽象的基类：

基类 `LoggerBase` 包括了所有子类（实现日志持久化功能子类）公有属性和方法，其中 `InputLogger` 方法主要用来完成日志的持久化工作，而每个子类的持久化方式都是不同的，所以在基类中这个方法被声明为 `abstract`，即抽象方法，这个方法要求子类必须重写它，即以自己的方式去重写它！

日志中的工厂方法模式：


```

/// <summary>
/// 对外不能创建类的实例
/// </summary>
private LoggerFactory()
{
    string type = "file";
    if (ConfigManager.Config != null)
        type = ConfigManager.Config.Logger_Type.ToLower();

    switch (type)
    {
        case "file":
            ILogger = new NormalLogger();
            break;
        case "log4net":
            ILogger = new Log4Logger();
            break;
        case "mongodb":
            ILogger = new MongoLogger();
            break;
        case "catlogger":
            ILogger = new CatLogger();
            break;
        default:
            throw new ArgumentException("请正确配置AppSetting的LoggerType节点 ( file , log4net , mongodb ) ");
    }
}
}

```

日志中的单例模式

```

/// <summary>
/// 单例模式的日志工厂对象
/// </summary>
public static LoggerFactory Instance
{
    get
    {
        if (instance == null)
        {
            lock (lockObj)
            {
                if (instance == null)
                {
                    instance = new LoggerFactory();
                }
            }
        }
        return instance;
    }
}
}

```

日志中的策略模式：

ILogger 对象为日志的持久化提供了多种策略


```

/// <summary> ...
public sealed class LoggerFactory : ILogger
{
    Logger有多种实现时,需要使用Singleton模式

    #region ILogger 成员
    /// <summary> ...
    public void Logger_Timer(string message, Action action)
    {
        iLogger.Logger_Timer(message, action);
    }
    /// <summary> ...
    public void Logger_Exception(string message, Action action) ...
    /// <summary> ...
    public void Logger_Debug(string message) ...
    /// <summary> ...
    public void Logger_Info(string message) ...
    /// <summary> ...
    public void Logger_Warn(string message) ...
    /// <summary> ...
    public void Logger_Error(Exception ex) ...
    /// <summary> ...
    public void Logger_Fatal(string message) ...
    #endregion
}

```

异常捕捉

自己的异常类型:

Exception 在每个项目中都有自己的实现, 你可以定义自己的异常类, 如为 Lind.DDD 系统定义一个 LindException, 它总是集成基类 Exception, 自己可以重写自己的业务逻辑和框架逻辑。

框架刻意去抛异常:

这个框架级设计上是经常看到的, 当用户调用的参数不合法时, 你完成可以向程序 throw 一个参数异常的 Exception, 即 `ArgumentException`, 事实上, 在大叔日志组件中也有对应的实现

```

switch (type)
{
    case "file":
        iLogger = new NormalLogger();
        break;
    case "log4net":
        iLogger = new Log4Logger();
        break;
    case "mongodb":
        iLogger = new MongoLogger();
        break;
    case "catlogger":
        iLogger = new CatLogger();
        break;
    default:
        throw new ArgumentException("请正确配置AppSetting的LoggerType节点 ( file , log4net , mongodb ) ");
}

```

全局的异常捕获:

对于一个系统, 我们并不建议到处都写 try...catch, 而应该将所有异常都抛出来, 然后在一个地方进行捕捉并记录, 这才是我们希望看到时的, 这种实现方式在 MVC 里有现成的工具, 即 Filter 过滤器, 它可以对 Action 进行拦截, 这类似于 AOP 面向方面的编程, 其时这就是 AOP 在 MVC 架构中的一个实现。

```

public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        //错误日志
        filters.Add(new Lind.DDD.Filters.MvcExceptionHandlerLoggerAttribute());
        //用户授权
        filters.Add(new Lind.DDD.Authorization.Mvc.AuthorizationLoginFilter("AdminCommon", "LogOn"));
        //Cat拦截
        filters.Add(new Lind.DDD.CatClientPur.CatFilter());
    }
}

```

自己设计一个异常拦截对象

```

public class MvcExceptionHandlerLoggerAttribute : HandleErrorAttribute
{
    #region IExceptionHandler 成员

    public override void OnException(ExceptionContext filterContext)
    {
        Lind.DDD.Logger.LoggerFactory.Instance.Logger_Error(filterContext.Exception);

        Lind.DDD.Logger.LoggerFactory.Instance.Logger_Info("\r\n当前请求出现异常,时间:"
            + DateTime.Now
            + "\r\n异常URL:"
            + filterContext.RequestContext.HttpContext.Request.Url.AbsoluteUri
            + "\r\n异常信息:" + filterContext.Exception.Message
            + "\r\n堆栈信息:" + filterContext.Exception.StackTrace);
        filterContext.RequestContext.HttpContext.Response.Clear();

        filterContext.RequestContext.HttpContext.Response.Write(
            "<form id='errForm' method='post' action='/AdminCommon/Error'><input type='hidden' name='msgErr' value='"
        );
        filterContext.RequestContext.HttpContext.Response.End();

        base.OnException(filterContext);
    }

    #endregion
}

```

Dispose 模式

```

/// <summary>
/// 实现 IDisposable, 对非托管系统进行资源回收
/// 作者: 仓储大叔
/// </summary>
public abstract class DisposableBase : IDisposable
{
    /// <summary>
    /// 标准 Dispose, 外界可以直接调用它
    /// </summary>
    public void Dispose()
    {
        Logger.LoggerFactory.Instance.Logger_Debug("Dispose");

        this.Dispose(true);///释放托管资源
        GC.SuppressFinalize(this);///请求系统不要调用指定对象的终结器. //该方法在对象头中设置一个位, 系统在调用终结器
        时将检查这个位
    }

    private void Dispose(bool disposing)
    {
        if (!_isDisposed)//_isDisposed 为 false 表示没有进行手动 dispose
        {
            //清理托管资源和清理非托管资源
            Finalize(disposing);
        }
    }
}

```

```

    }
    Logger.LoggerFactory.Instance.Logger_Debug("Dispose  complete!");
    _isDisposed = true;
}

/// <summary>
/// 由子类自己去实现自己的 Dispose 逻辑 ( 清理托管和非托管资源 )
/// </summary>
/// <param name="disposing"></param>
protected abstract void Finalize(bool disposing);

private bool _isDisposed;

/// <summary>
/// 是否完成了资源的释放
/// </summary>
public bool IsDisposed
{
    get { return this._isDisposed; }
}

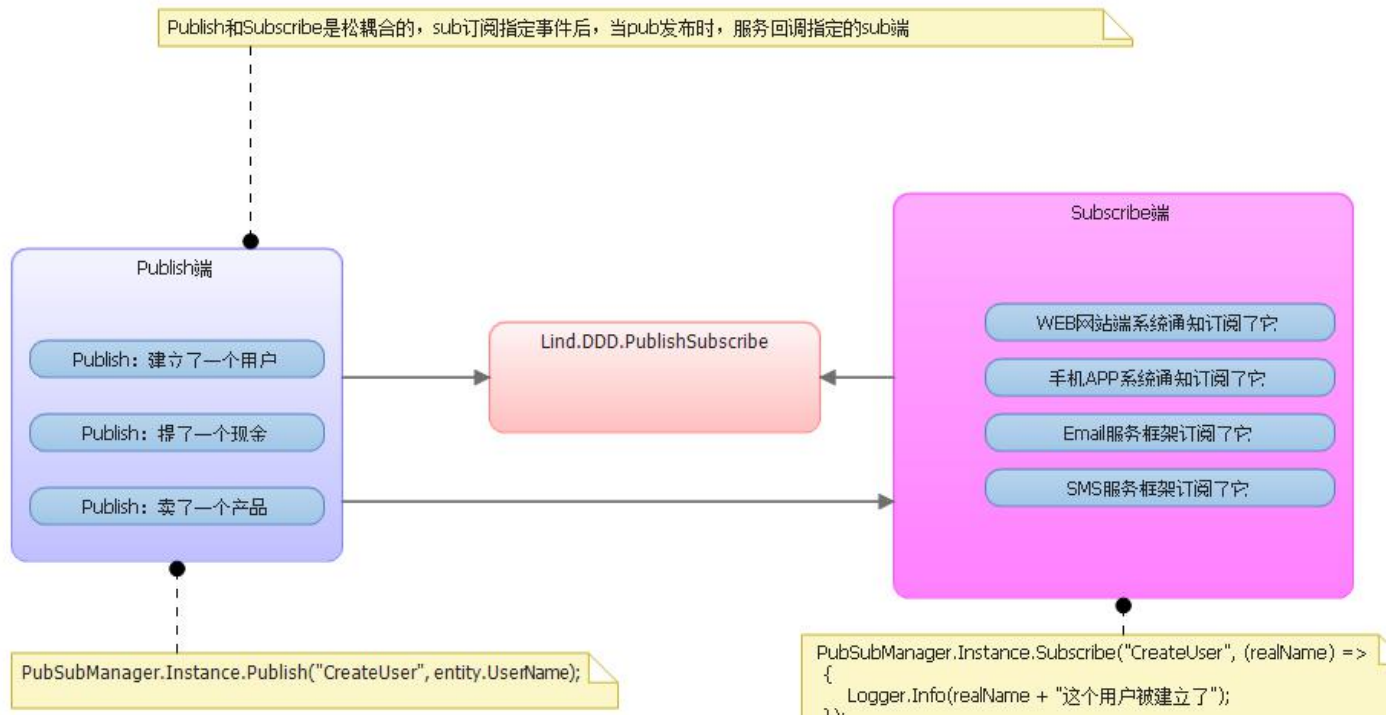
/// <summary>
/// 析构方法 - 在类被释放前被执行
/// </summary>
~DisposableBase()
{
    Logger.LoggerFactory.Instance.Logger_Debug("析构方法");

    //执行到这里，托管资源已经被释放
    this.Dispose(false); //释放非托管资源，托管资源由终极器自己完成了
}
}

```

第六讲 分布式的 Pub/Sub 和消息队列

Pub/Sub 即发布与订阅模式，首先有一些订阅者去制订一些业务功能，然后有一些发布者在某种情况下触发某些功能，这时，**被订阅的方法将会被执行！**



它通常发生在一个系统或者多个系统中，如一个邮件发送功能，你可以当成是一个订阅者，它可以在 A 系统初始化时被订阅，名称可以是 SendNotify；当业务系统真正要发邮件时，只需要发布这个 SendNotify 即可，**这时 A 系统订阅的发送 Email 代码段就会被执行**；向 SMS 也是一样，与 Email 一起去订阅，名称也是 SendNotify；这时就有两个订阅者，当业务系统发布 SendNotify 时，A 系统在发 Email 的同时还会发 SMS，这种松耦合的设计也经常被用户各个系统之间，A 系统订阅一些服务，在业务代码中去触发，这也是未来的一种趋势。

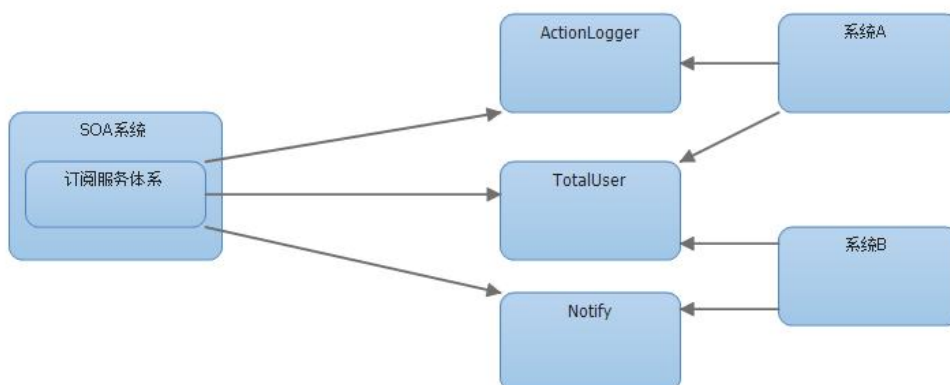
```
PubSubManager.Instance.Subscribe("SendNotify", (msg) =>
{
    //订阅者A, 进行Email的发送
    Console.WriteLine("Email发送消息{0}", msg);
});
PubSubManager.Instance.Subscribe("SendNotify", (msg) =>
{
    //订阅者B, 进行SMS的发送
    Console.WriteLine("SMS发送消息{0}", msg);
});
```

//用户登陆模块

```
PubSubManager.Instance.Publish("SendNotify", "占岭这个用户登录了");
```

分布式 Pub/Sub 实现立即队列

在 SOA 服务体系中去订阅某个核心功能 ActionLogger，然后在各个业务系统的各种业务中发布这个 ActionLogger，这时，ActionLogger 这个对象在发布后会 **立即执行 SOA 订阅的方法代码段**。



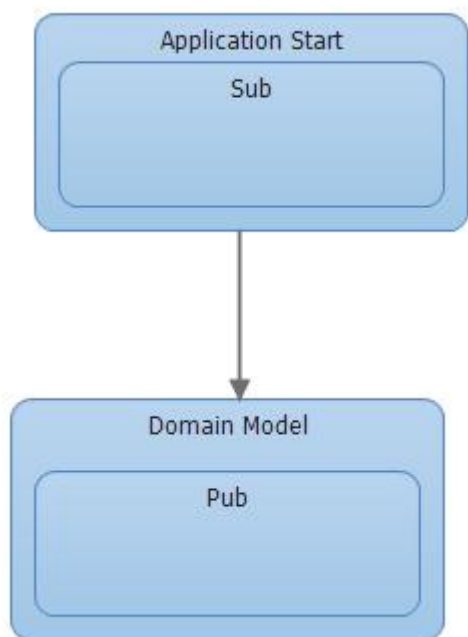
消息队列 (Queue)

特点：FIFO，先进先出

第七讲 领域对象，事件总线 and 领域事件

一个事件处理系统，让订阅和发布更加灵活介绍在项目中如何实现多种消息发送功能的并存，包括了 Email, SMS, RTX 等常用的通讯机制；领域事件主要封装了，现时可以完全与领域实体有效的结合在一起。

我们一步一步设计一个领域事件和事件总线



第八讲 Redis 集群与客户端使用

Redis 的介绍及它的使用场合，5 个数据结构的介绍，服务端集群的配置，客户端集群的配置，读写分离的配置，客户端的使用及如何实现消息队列。

发并写性：10W/S，每秒 10W 个请求

众多 NoSQL 百花齐放，如何选择

最近几年，业界不断涌现出很多各种各样的 NoSQL 产品，那么如何才能正确地使用好这些产品，最大化地发挥其长处，是我们需要深入研究和思考的问题，实际归根结底最重要的是了解这些产品的定位，并且了解到每款产品的 tradeoffs，在实际应用中做到扬长避短，总体上这些 NoSQL 主要用于解决以下几种问题

少量数据存储，高速读写访问。此类产品通过数据全部 in-memory 的方式来保证高速访问，同时提供数据落地的功能，实际这正是 Redis 最主要的适用场景。

海量数据存储，分布式系统支持，数据一致性保证，方便的集群节点添加/删除。

这方面最具代表性的是 dynamo 和 bigtable 2 篇论文所阐述的思路。前者是一个完全无中心的设计，节点之间通过 gossip 方式传递集群信息，数据保证最终一致性，后者是一个中心化的方案设计，通过类似一个分布式锁服务来保证强一致性，数据写入先写内存和 redo log，然后定期 compact 归并到磁盘上，将随机写优化为顺序写，提高写入性能。

Schema free，auto-sharding 等。比如目前常见的一些文档数据库都是支持 schema-free 的，直接存储 json 格式数据，并且支持 auto-sharding 等功能，比如 **mongodb**。

面对这些不同类型的 NoSQL 产品，我们需要根据我们的业务场景选择最合适的产品。

Redis 要了解它的本质

Redis 除了作为存储之外还提供了一些其它方面的功能，比如聚合计算、pubsub、scripting 等，对于此类功能需要了解其实现原理，清楚地了解到它的局限性后，才能正确的使用，比如 pubsub 功能，这个实际是没有任何持久化支持的，消费方连接闪断或重连之间过来的消息是会全部丢失的，又比如聚合计算和 scripting 等功能受 Redis 单线程模型所限，是不可能达到很高的吞吐量的，需要谨慎使用。

总的来说 Redis 作者是一位非常勤奋的开发者，可以经常看到作者在尝试着各种不同的新鲜想法和思路，针对这些方面的功能就要求我们需要深入了解后再使用。

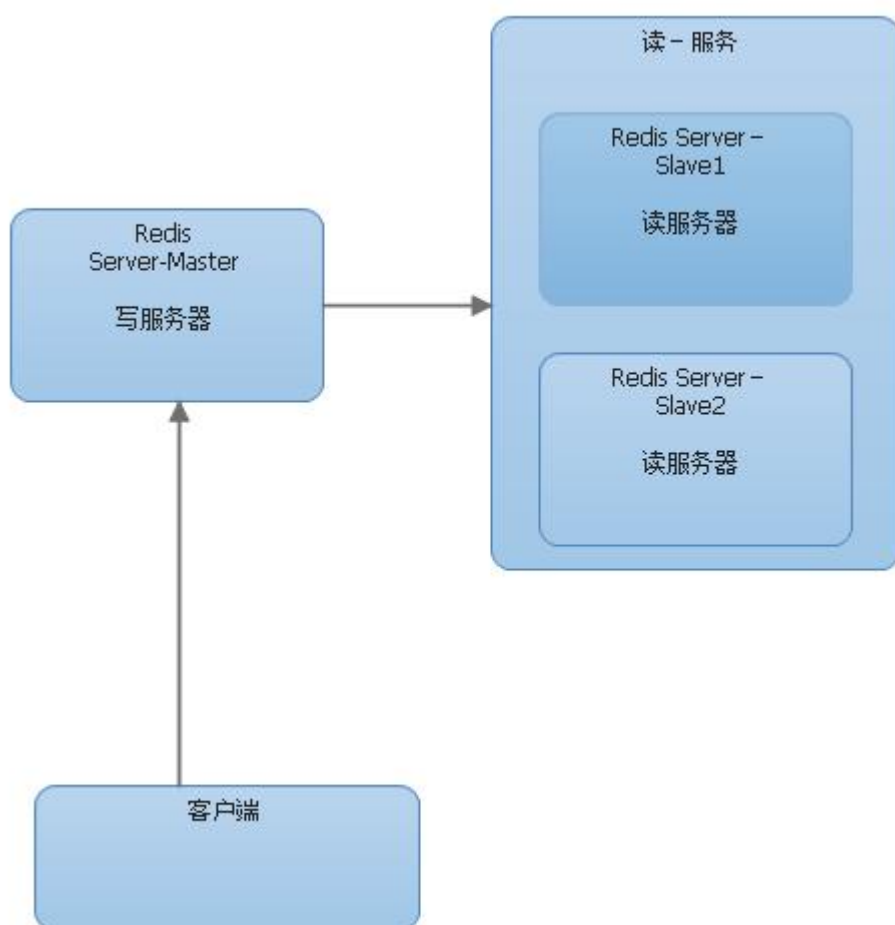
总结：

- Redis 使用最佳方式是全部数据 in-memory。
- Redis 更多场景是作为 Memcached 的替代者来使用。
- 当需要除 key/value 之外的更多数据类型支持时，使用 Redis 更合适。
- 当存储的数据不能被剔除时，使用 Redis 更合适。

五大数据结构

```
//string, 字符串  
//hash, 对象  
//list, 双向链表  
//set, 去重集合  
//sortset, 根据权重值排序
```

Redis 集群



打开从服务器的 redis.config，输入它认为的主 redis 的地址和端口，如下
slaveof 192.168.77.211 12002

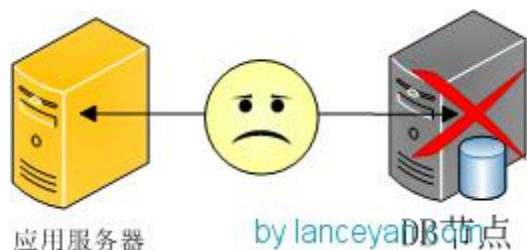
第九讲 MongoDB 集群与客户端使用

MongoDB 的介绍及它的使用场合，文档型数据库的优势，以 JSON 和 BSON 为基础对数据进行存储，对读写性能都很不错，向日志型数据，行为型数据都可以存储到 MongoDB 里，现时还会介绍复制集，切片的知识，对原生客户端的使用。

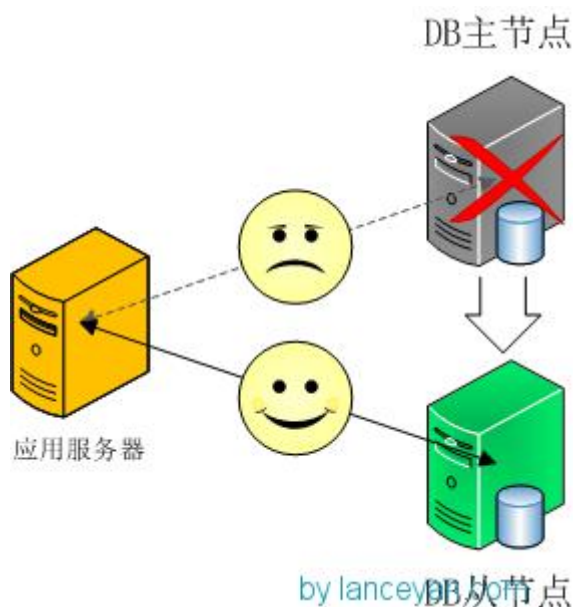
搭建高可用 mongodb 集群

在大数据的时代，传统的关系型数据库要能更高的服务必须要解决高并发读写、海量数据高效存储、高可扩展性和高可用性这些难题。不过就是因为这些问题 Nosql 诞生了。

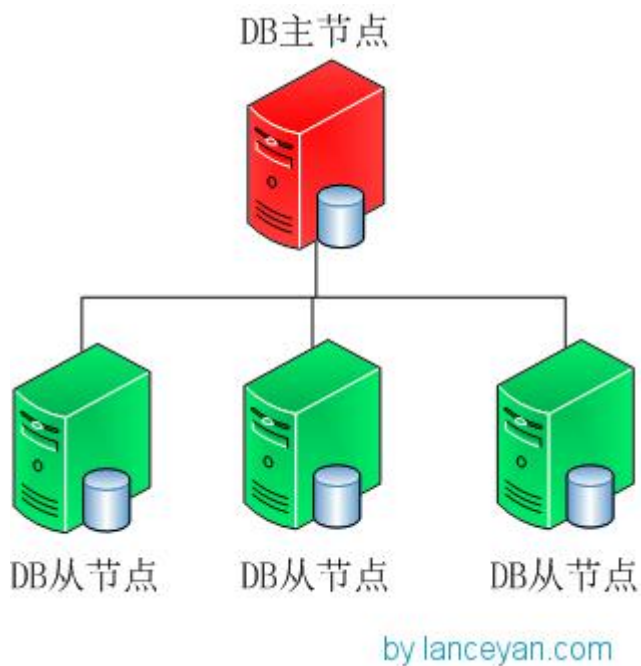
一、mongodb 单实例。这种配置只适合简易开发时使用，生产使用不行，因为单节点挂掉整个数据业务全挂，如下图。



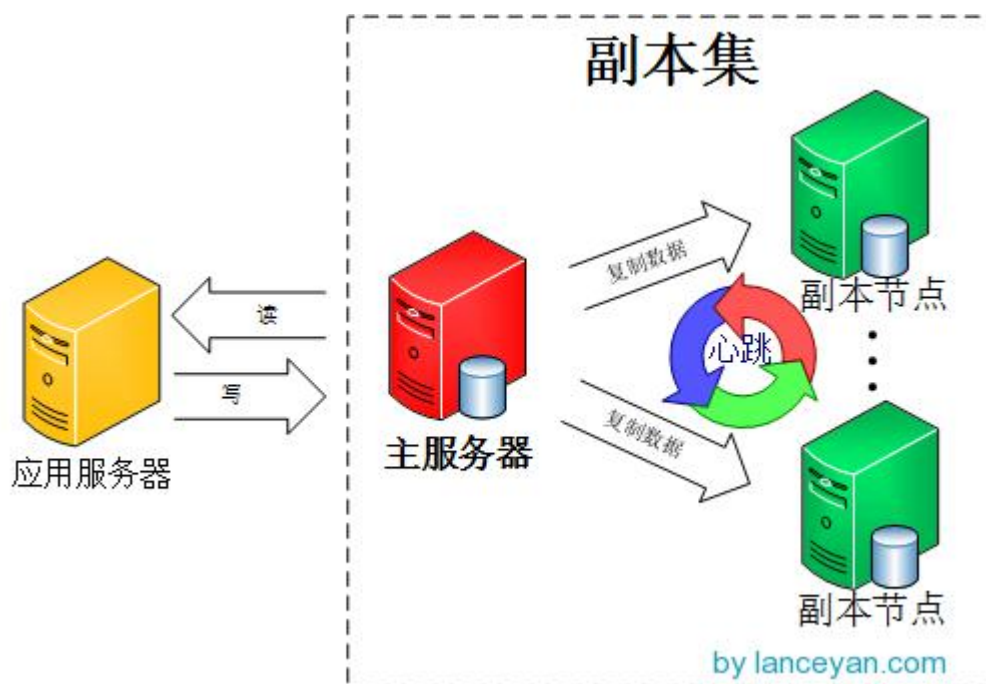
二、主从模式。使用 mysql 数据库时大家广泛用到，采用双机备份后主节点挂掉了后从节点可以接替主机继续服务。所以这种模式比单节点的高可用性要好很多。



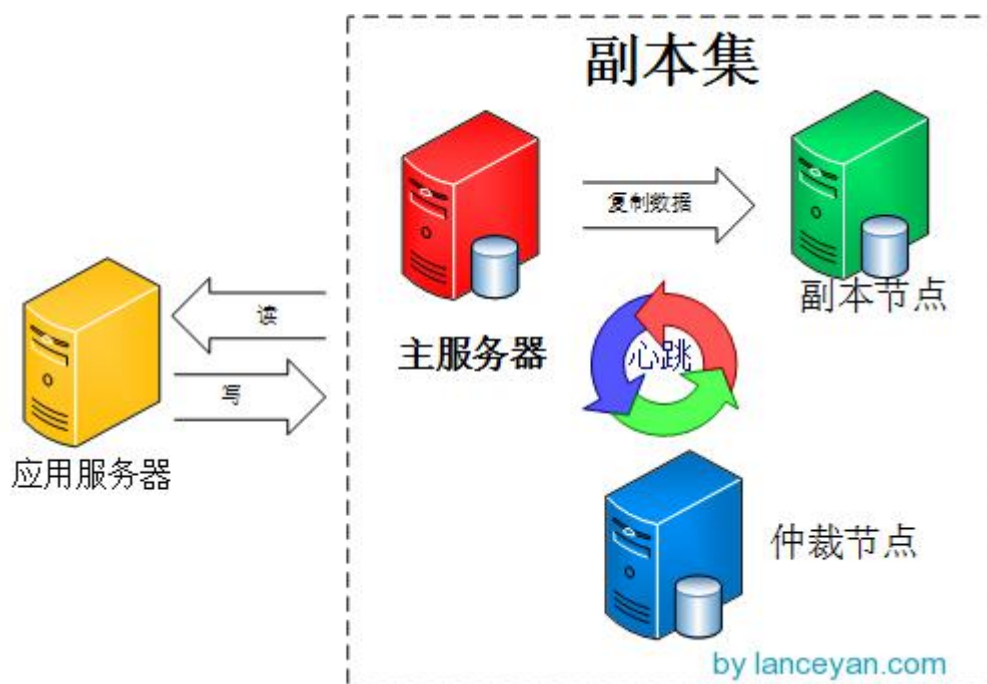
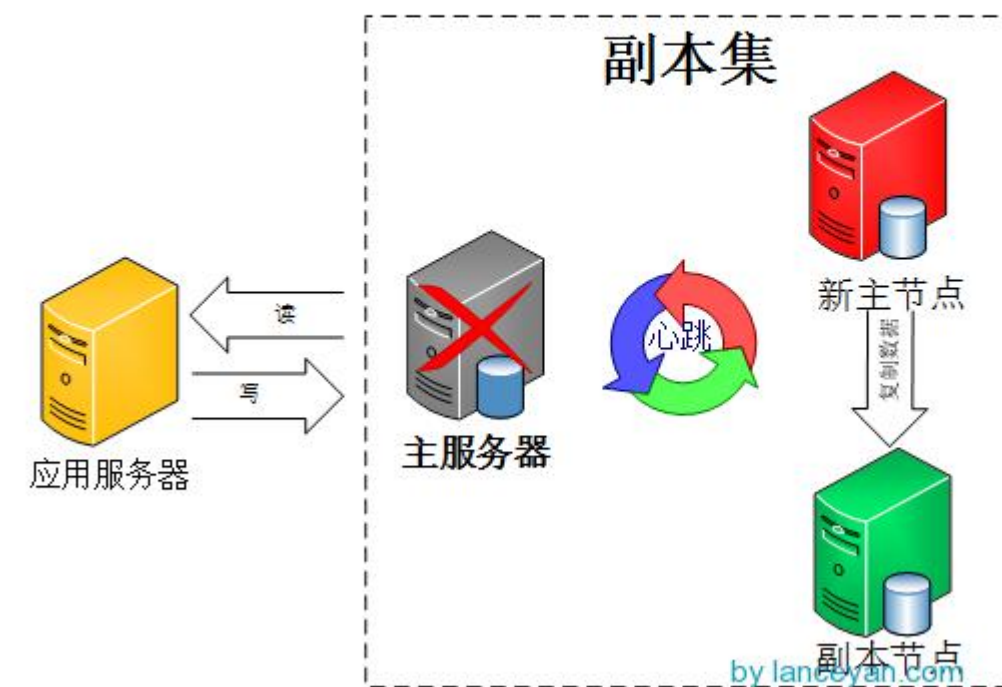
多个从节点。现在只是一个数据库服务器又提供写又提供读，机器承载会出现瓶颈。大家还记得 mysql 里的读写分离吗？把 20%的写放到主节点，80%的读放到从节点分摊了减少了服务器的负载。但是大部分应用都是读操作带来的压力，一个从节点压力负载不了，可以把一个从节点变成多个节点。那 mongodb 的一主多从可以支持吗？答案是肯定的。



三 副本集呢__mongoDB 主从模式其实就是一个单副本的应用，没有很好的扩展性和容错性。而副本集具有多个副本保证了容错性，就算一个副本挂掉了还有很多副本存在，并且解决了上面第一个问题“主节点挂掉了，整个集群内会自动切换”。难怪 mongoDB 官方推荐使用这种模式。我们来看看 mongoDB 副本集的架构图：

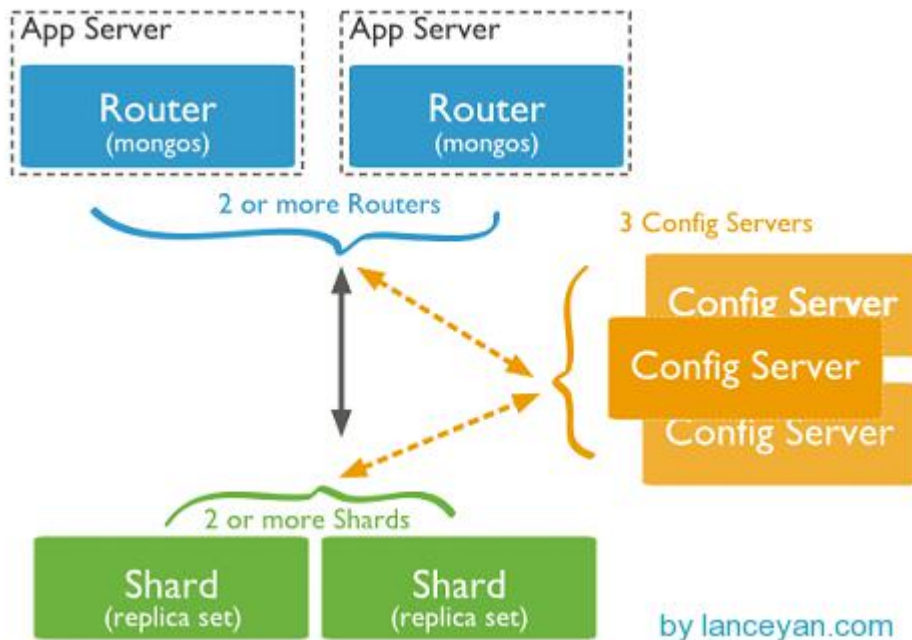


主服务器负责整个副本集的读写，副本集定期同步数据备份，
一旦主节点挂掉，副本节点就会选举一个新的主服务器，
这一切对于应用服务器不需要关心。我们看一下主服务器挂掉后的架构：



其中的仲裁节点不存储数据，只是负责故障转移的群体投票，这样就少了数据复制的压力。

分片：

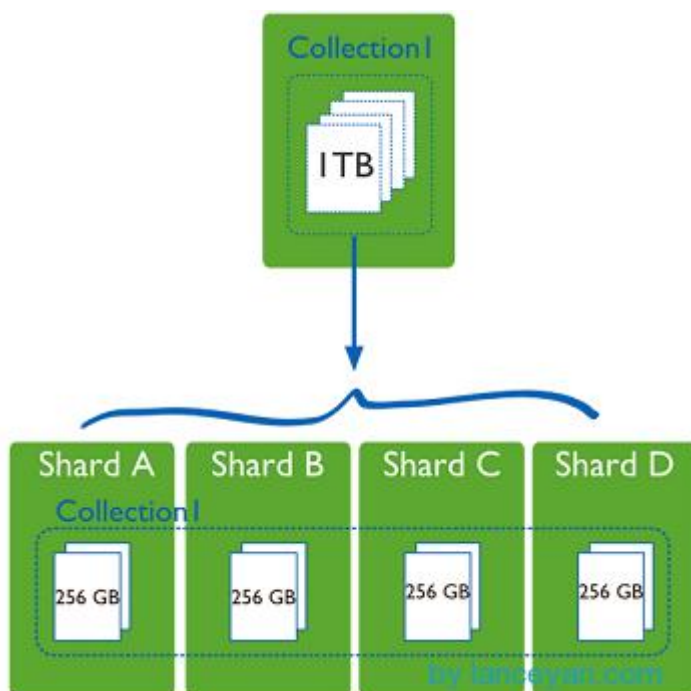


从图中可以看到有四个组件：mongos、config server、shard、replica set。

mongos，数据库集群请求的入口，所有的请求都通过 mongos 进行协调，不需要在应用程序添加一个路由选择器，mongos 自己就是一个请求分发中心，它负责把对应的数据请求请求转发到对应的 shard 服务器上。在生产环境通常有多 mongos 作为请求的入口，防止其中一个挂掉所有的 mongod 请求都没有办法操作。

config server，顾名思义为配置服务器，存储所有数据库元信息（路由、分片）的配置。mongos 本身没有物理存储分片服务器和数据路由信息，只是缓存在内存里，配置服务器则实际存储这些数据。mongos 第一次启动或者关掉重启就会从 config server 加载配置信息，以后如果配置服务器信息变化会通知到所有的 mongos 更新自己的状态，这样 mongos 就能继续准确路由。在生产环境通常有多个 config server 配置服务器，因为它存储了分片路由的元数据，这个可不能丢失！就算挂掉其中一台，只要还有存货，mongodb 集群就不会挂掉。

shard，这就是传说中的分片了。上面提到一个机器就算能力再大也有天花板，就像军队打仗一样，一个人再厉害喝血瓶也拼不过对方的一个师。俗话说三个臭皮匠顶个诸葛亮，这个时候团队的力量就凸显出来了。在互联网也是这样，一台普通的机器做不了的多台机器来做，如下图：

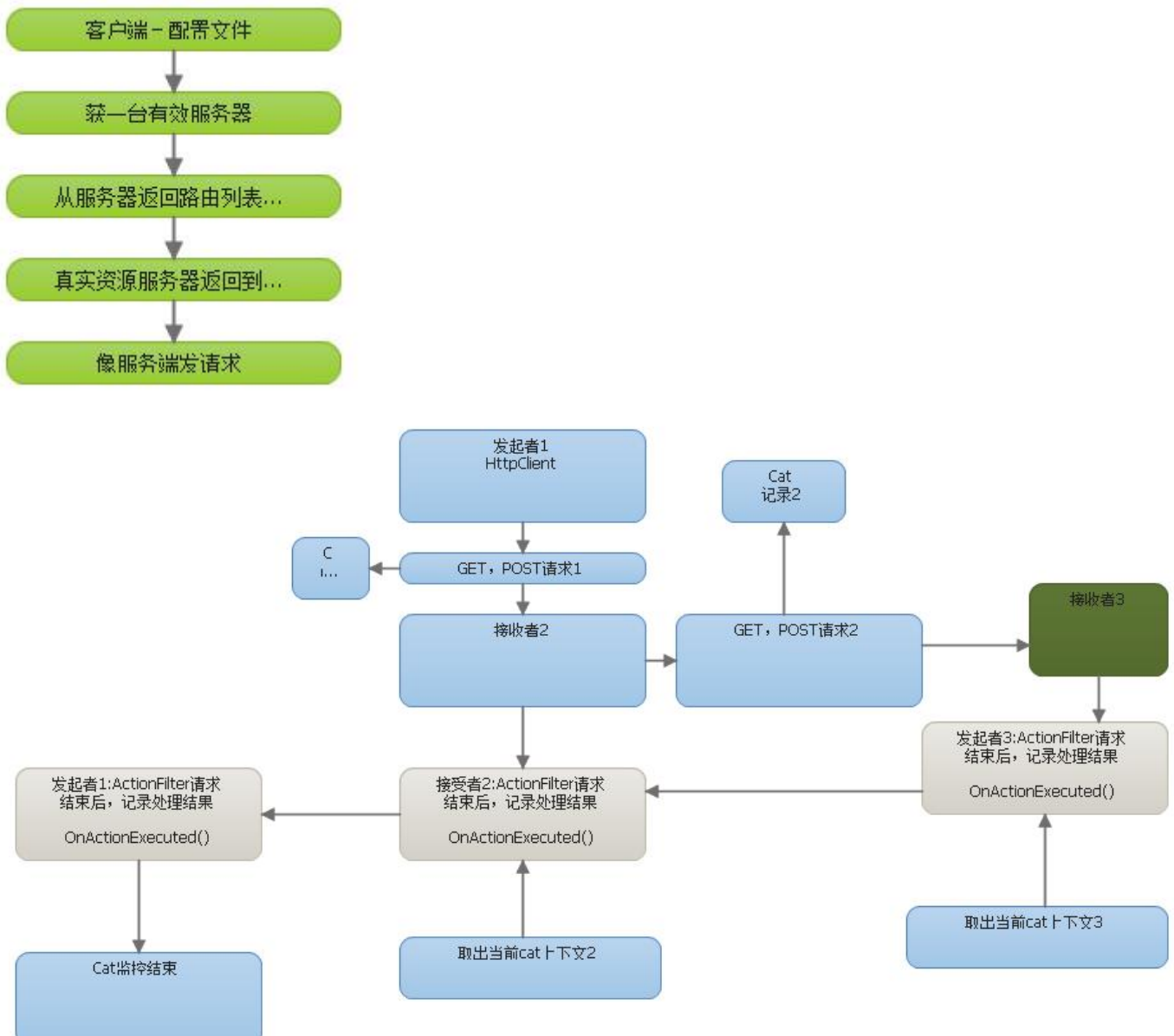


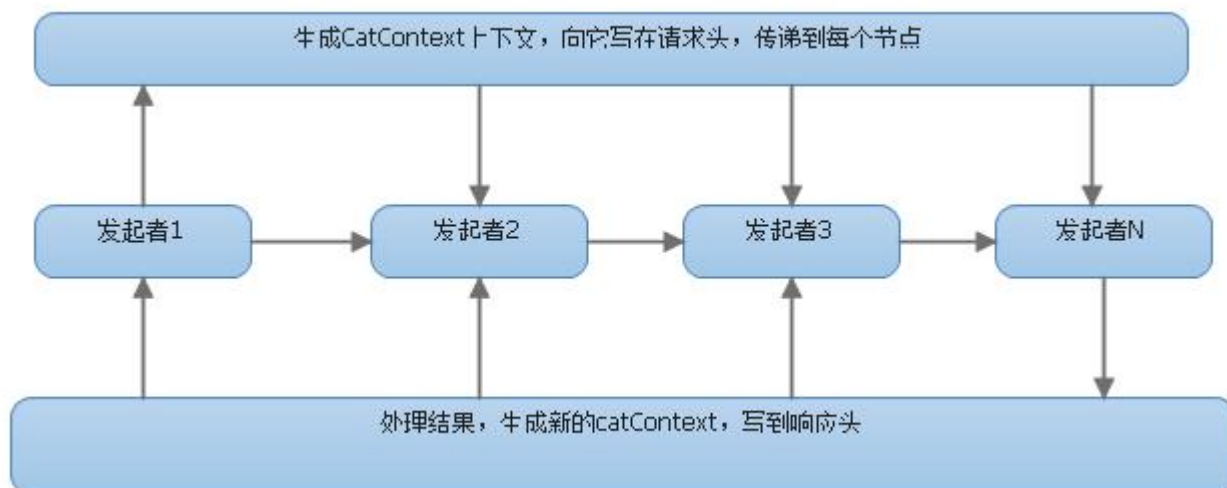
一台机器的一个数据表 Collection1 存储了 1T 数据，压力太大了！在分给 4 个机器后，每个机器都是 256G，则分摊了集中在一台机器的压力

第十讲 Cat 集群与使用

Cat 的分布式消息树

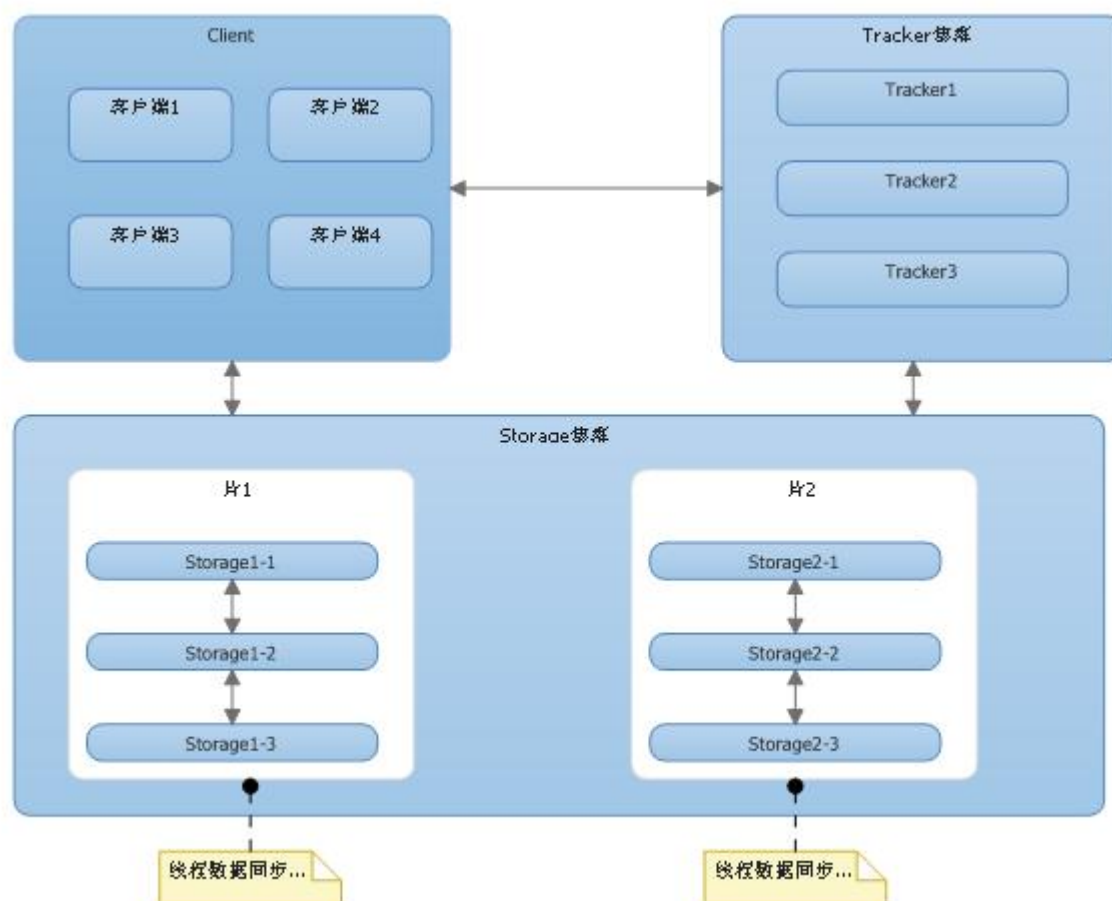
Cat 上下文，它与其它数据上下文，Http 上下文，文件上下文的意思是一样的，都是指一种对象的封装，在 cat 里它的上下文由三个 ID 组成，ROOT, Parent 和 Child，他们类似于数据库里的联合主键，在让多个消息进行关联时，需要通过这些键值，我们在跨网络记录日志时，也需要把这三个对象传过去，在目标服务器上进行解析，然后这两个消息就组成了一个消息树了。



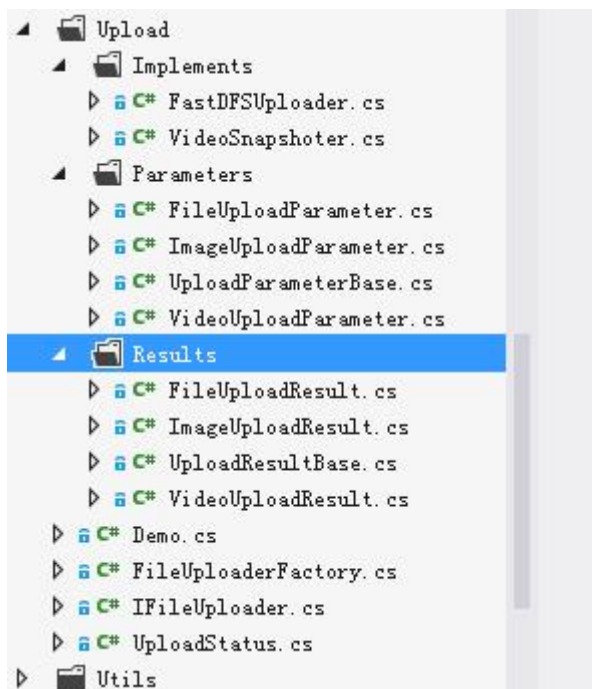


第十一讲 分布式文件存储和文件上传的设计

FastDFS 是一个分布式存储的工具，同时也是一个集群的环境，由 tracker 和 storage 组成。

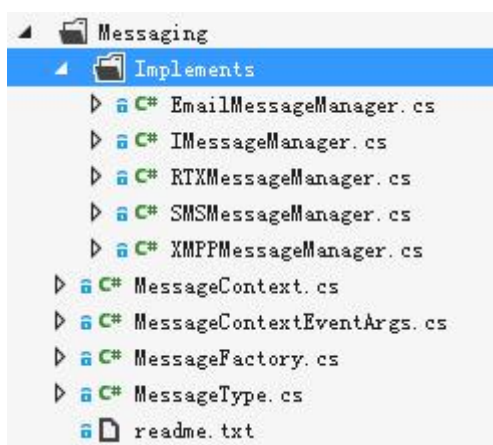


文件上传，对文件，图片，视频的封装



第十二讲 消息组件和第三方支付的统一

几种消息通知机制以策略的方式进行封装，以工厂方法模式进行实例的生产，以单例模式对程序性能进行优化。

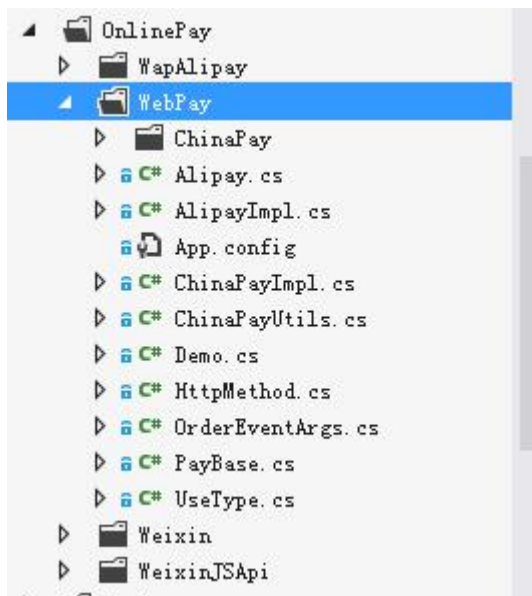


```

/// <summary>
/// 消息生产者
/// 具体消息生产者是单例，如Email,SMS,Rtx等
/// </summary>
public sealed class MessageFactory
{
    /// <summary>
    /// 消息工厂
    /// </summary>
    public static IMessageManager GetService(MessageType messageType)
    {
        switch (messageType)
        {
            case MessageType.Email:
                return EmailMessageManager.Instance;
            case MessageType.SMS:
                return SMSMessageManager.Instance;
            case MessageType.RTX:
                return RTXMessageManager.Instance;
            case MessageType.XMPP:
                return XMPPMessageManager.Instance;
            default:
                throw new NotImplementedException("消息生产者未被识别...");
        }
    }
}

```

在线支付的整理与封装，集成了支付宝，微信，银联等



```

#region Server Send Methods
protected override string GenerateUrl(UseType useType,
    string body,
    decimal vMoney,
    int userID,
    ref string exchangeID,
    string returnUrl,
    string notifyUrl,
    string name,
    string orderid = "")[...]

    [...] <summary> ...
private string CreatUrl(string[] para, string _input_charset, string key)[...]

#endregion

#region Server Return Methods

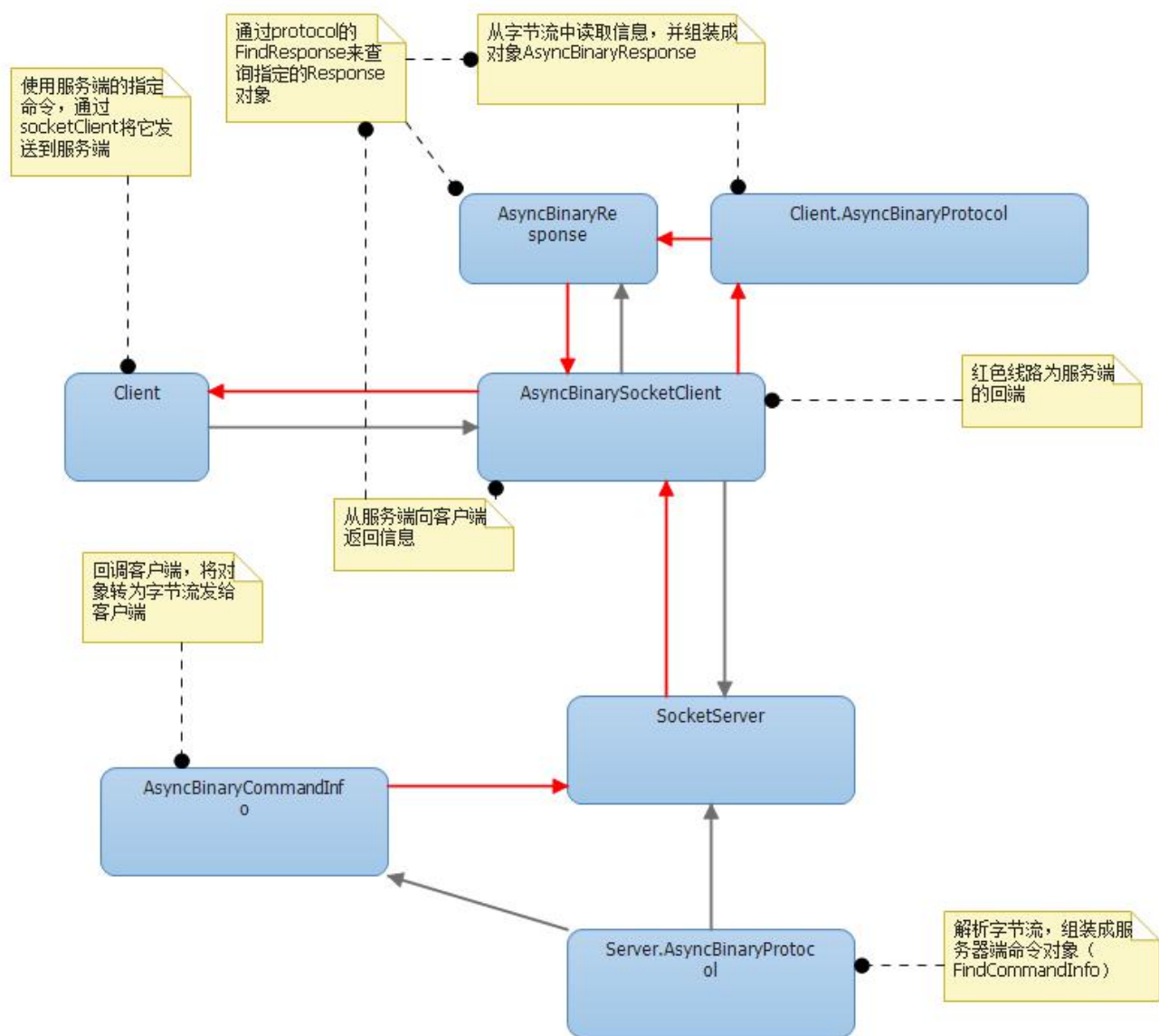
    [...] <summary> ...
public override void Notify_Return(HttpMethod httpMethod)[...]

    [...] <summary> ...
bool AlipayReturnParam(
    int userID,
    string orderID,
    string exchangeID,
    string notify_id,
    string sign,
    NameValueCollection coll,
    System.Web.HttpContext http)[...]
#endregion

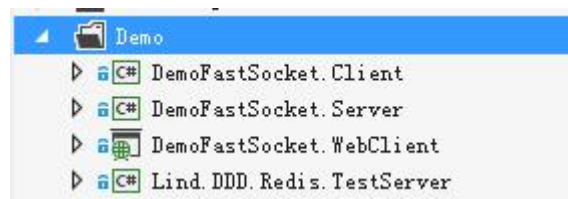
```

第十三讲 数据包和网络通讯

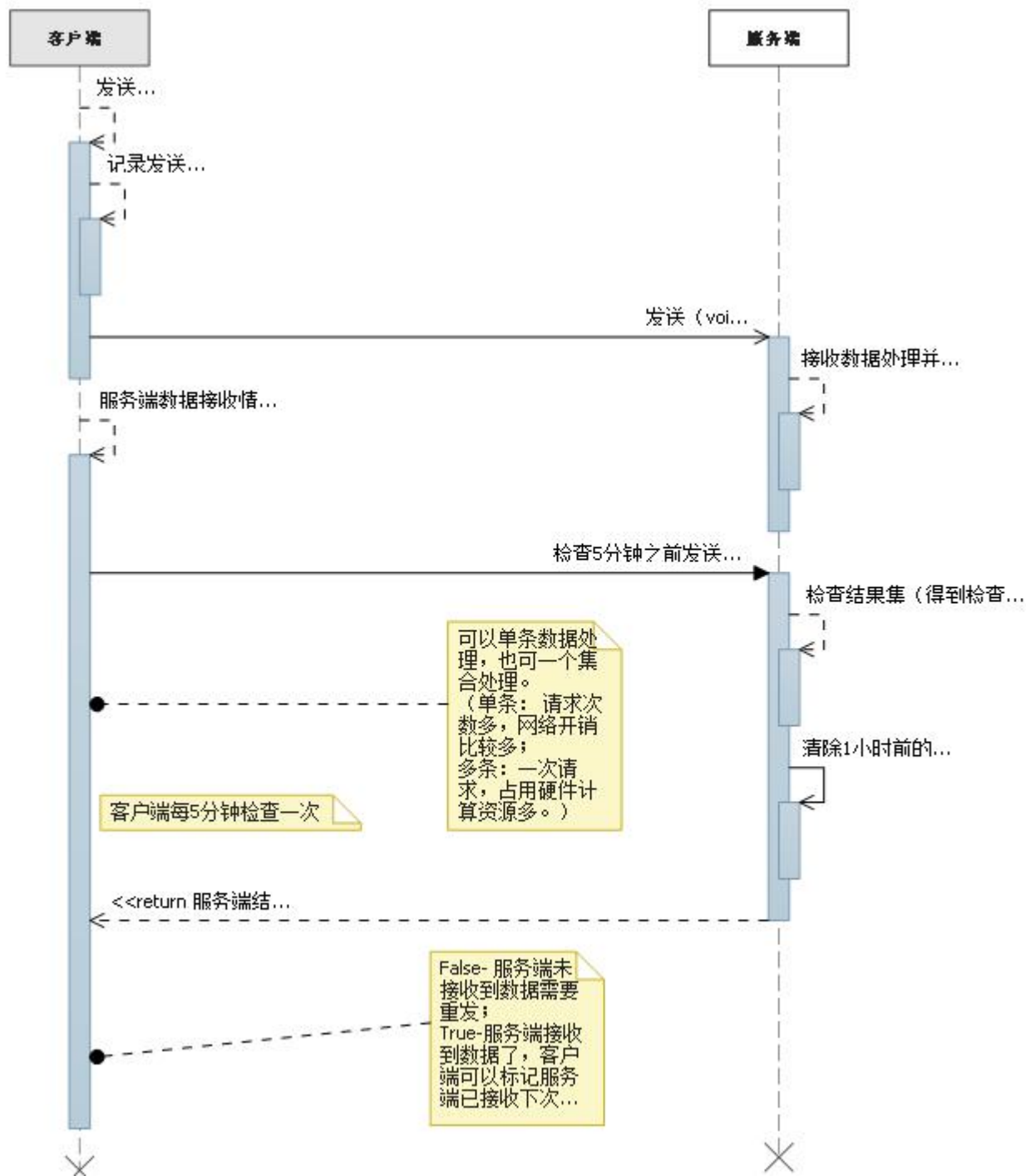
[FastSocket 框架结构](#)



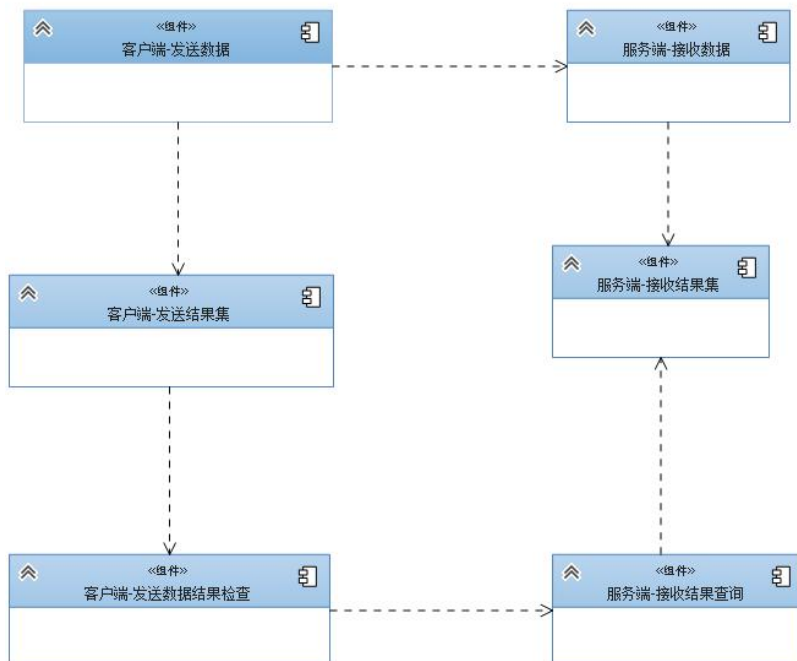
网络通讯 FastSocket 的项目结构



客户端与服务端通讯的时序图



数据有效性的检查



客户端这样为服务端发消息

```

var client = new DSSBinarySocketClient(8192, 8192, 3000, 3000);

#region DataInsert
//注册服务器节点, 这里可注册多个(name不能重复)
client.RegisterServerNode("127.0.0.1:8403", new System.Net.IPEndPoint(System.Net.IPAddress.Parse("127.0.0.1"), 8403));
client.Send("UserInsert", 1, "zzl", 1, "test"
    , SerializeMemoryHelper.SerializeToBinary("hello world!"), res => res.Buffer)
    .ContinueWith(c =>
    {
        if (c.IsFaulted)
        {
            throw c.Exception;
        }
        Console.WriteLine(Encoding.UTF8.GetString(c.Result));
    }).Wait();

#endregion
  
```

服务端这样处理消息 (可以动态配置服务端)

```

public void ExecuteCommand(IConnection connection, DSSBinaryCommandInfo commandInfo)
{
    if (commandInfo.Buffer == null || commandInfo.Buffer.Length == 0)
    {
        Console.WriteLine("UserInsert参数为空");
        connection.BeginDisconnect();
        return;
    }

    var entity = SerializeMemoryHelper.DeserializeFromBinary(commandInfo.Buffer);
    string str = "result:1,versionNumber:" + commandInfo.VersionNumber
        + ",extProperty:" + commandInfo.ExtProperty
        + ",projectName:" + commandInfo.ProjectName
        + ",message:" + entity.ToString();
    Console.WriteLine(str);
    commandInfo.Reply(connection, Encoding.UTF8.GetBytes(str)); //返回到客户端...
  }
  
```


第十四讲 ORM 和分布式事务

ORM:对象关系映射（英语：**Object Relational Mapping**，简称 **ORM**，或 **O/RM**，或 **O/R mapping**），是一种程序技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的“虚拟对象数据库”。

产品：

Linq2Sql

EntityFrameworks

分布式事务(MSDTC):分布式事务是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的[分布式系统](#)的不同节点之上。

```
/// <summary>
/// 事务提交
/// </summary>
public void Commit()
{
    try
    {
        using (TransactionScope transactionScope = new TransactionScope())
        {
            提交到工作单元

            //提交事务，程序中如果出错，这行无法执行，即事务不会被提交，这就类似于rollback机制
            transactionScope.Complete();

            清空当前上下文字典
        }
    }
    catch (Exception ex)
    {
        Logger.LoggerFactory.Instance.Logger_Error(ex);
        throw ex;
    }
}
```

第十五讲 MVVM 和 KnockoutJS 的介绍

- M V VM:模型 视图 视图模型
- 一般 MVVM 的架构

- Silverlight
- Angularjs
- KnockoutJS
- **KO 对象**
- 理解 KO：一切皆为对象
- 数据绑定实现
- Select 级联实现
- 表单验证实实现
- 前台 CURD 的实现
- \$.when.do.then 的使用

第十六讲 任务调度 Quartz

新增任务			
运行中的任务...			
任务名	任务组	任务开始时间	任务结束时间
JobStudentTotal	JobStudentTotalGroup	1982-06-28T18:15:00.0Z	2020-05-04T18:13:51.0Z

	时间间隔 (ms)	重复次数 (-1表示永久)	状态	操作
	2000	-1	正常	编辑 删除 恢复 停止

一个 cron 表达式有至少 6 个 (也可能 7 个) 有空格分隔的时间元素。

按顺序依次为

秒 (0~59)

分钟 (0~59)

小时 (0~23)

天 (月) (0~31 , 但是你需要考虑你月的天数)

月 (0~11)

天 (星期) (1~7 1=SUN 或 SUN , MON , TUE , WED , THU , FRI , SAT)

年份 (1970 - 2099)

其中每个元素可以是一个值(如 6),一个连续区间(9-12),一个间隔时间(8-18/4)(/表示每隔 4 小时),一个列表(1,3,5),通配符。由于"月份中的日期"和"星期中的日期"这两个元素互斥的,必须要对其中一个设置?

0 0 10,14,16 * * ? 每天上午 10 点 , 下午 2 点 , 4 点

0 0/30 9-17 * * ? 朝九晚五工作时间内每半小时

0 0 12 ? * WED 表示每个星期三中午 12 点

"0 0 12 * * ?" 每天中午 12 点触发

"0 15 10 ? * *" 每天上午 10:15 触发

"0 15 10 * * ?" 每天上午 10:15 触发

"0 15 10 * * ? *" 每天上午 10:15 触发

"0 15 10 * * ? 2005" 2005 年的每天上午 10:15 触发

"0 * 14 * * ?" 在每天下午 2 点到下午 2:59 期间的每 1 分钟触发

"0 0/5 14 * * ?" 在每天下午 2 点到下午 2:55 期间的每 5 分钟触发

"0 0/5 14,18 * * ?" 在每天下午 2 点到 2:55 期间和下午 6 点到 6:55 期间的每 5 分钟触发

"0 0-5 14 * * ?" 在每天下午 2 点到下午 2:05 期间的每 1 分钟触发

"0 10,44 14 ? 3 WED" 每年三月的星期三的下午 2:10 和 2:44 触发

"0 15 10 ? * MON-FRI" 周一至周五的上午 10:15 触发

"0 15 10 15 * ?" 每月 15 日上午 10:15 触发

"0 15 10 L * ?" 每月最后一日的上午 10:15 触发

"0 15 10 ? * 6L" 每月的最后一个星期五上午 10:15 触发

"0 15 10 ? * 6L 2002-2005" 2002 年至 2005 年的每月的最后一个星期五上午 10:15 触发

"0 15 10 ? * 6#3" 每月的第三个星期五上午 10:15 触发

第十七讲 多并程与并行

非阻塞

异步操作无须额外的线程负担，并且使用回调的方式进行处理，在设计良好的情况下，处理函数可以不必使用共享变量（即使无法完全不用，最起码可以减少 共享变量的数量），减少了死锁的可能。当然异步操作也并非完美无暇。编写异步操作的复杂程度较高，程序主要使用回调方式进行处理，与普通人的思维方式有些出入，而且难以调试。当需要执行 I/O 操作时，使用异步操作比使用线程+同步 I/O 操作更合适。异步和多线程两者都可以达到避免调用线程阻塞的目的，从而提高软件的可响应性。

异步本质为方法的回调

异步操作无须额外的线程负担，并且使用回调的方式进行处理，在设计良好的情况下，处理函数可以不必使用共享变量（即使无法完全不用，最起码可以减少 共享变量的数量），减少了死锁的可能。当然异步操作也并非完美无暇。编写异步操作的复杂程度较高，程序主要使用回调方式进行处理，与普通人的思维方式有些出入，而且难以调试。当需要执行 I/O 操作时，使用异步操作比使用线程+同步 I/O 操作更合适。

线程争夺与死锁

多线程中的处理程序依然是顺序执行，符合普通人的思维习惯，所以编程简单。但是多线程的缺点也同样明显，线程的使用（滥用）会给系统带来上下文切换 的额外 负担。并且线程间的共享变量可能造成死锁的出现。多线程的适用范围则是那种需要长时间 CPU 运算的场合，例如耗时较长的图形处理和算法执行。

技术点说明

1 Task.Run(=>{}); 将一个任务添加到线程池里，排队执行

2 async 标识一个方法为异步方法，可以与主线程并行执行，发挥 CPU 的多核优势

3 await 在调用一个 async 方法前可以添加这个修饰符，它意思是等待当前异步方法执行完后，再执行下面的代码

4 ConfigureAwait(true)，代码由同步执行进入异步执行时，当前线程上下文信息就会被捕获并保存至 SynchronizationContext 中，供异步执行中使用，并且供异步执行完成之后的同步执行中使用

5 ConfigureAwait(false)，不进行线程上下文信息的捕获，async 方法中与 await 之后的代码执行时就无法获取 await 之前的线程 的上下文信息，在 ASP.NET 中最直接的影响就是 HttpContext.Current 的值为 null，但不会出现非空引用的错误

Async 引起的死锁，w3wp.exe 挂的原因

对于将异步方法偷懒的人，即使用 Wait()和 Result 的人，将会为些付出代价，因为它会引起线程的死锁，最终导致 w3wp 挂掉，注意在控制器 console 程序中，这件事不会发生。

```

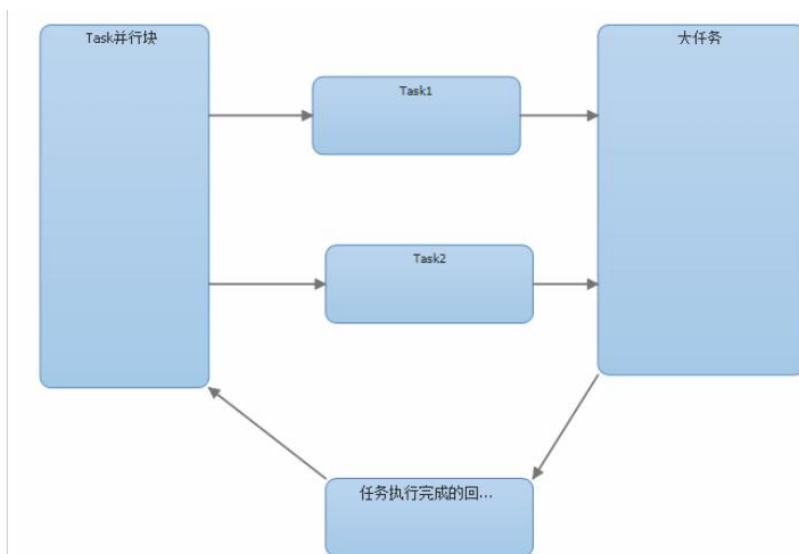
public class tools
{
    public static async Task TestAsync()
    {
        await Task.Delay(1000);
    }
}

public class HomeController : Controller
{
    public ActionResult Index()
    {
        tools.TestAsync().Wait(); //产生死锁, w3wp.exe挂掉
        ViewBag.Message = "test";
        return View();
    }
}

```

并行这个概念出自.net4.5，它被封装在 System.Threading.Tasks 命名空间里，主要提供一些线程，异步的方法，或者说它是 对之前 Thread 进行的二次封装，为的是让开发人员更方便的调用它，对于异步与多线程我们在之前的几讲里已经介绍过了，今天主要说说并行，并行也可以叫 并行计算，即对于一个大任务，使用多个线程去计算它，这可以充分发挥多核 C P U 的优势，可以说是大势所趋！

先看一下并行编程（并行计算）的图像



对于两个任务，任务 1 执行需要 1 秒，任务 2 执行需要 3 秒，那么，如果顺序执行，它需要的时间为 3 秒（ 1 + 2 ），而如果是并行编程，那就运行时间为 3 秒，即（ 1 和 3 一起运行，取最长的时间 ），这就是并行计算的魅力！

下面看两种并行的实现方式

一 Task 实现的并行



#region 并行 Task

```
Console.WriteLine(DateTime.Now);
```

```
var task = Task.WhenAll(Task.Run(() => { Thread.Sleep(1000); }), Task.Run(() => { Thread.Sleep(2000); })); //多个
```

task 并行执行，不阻塞

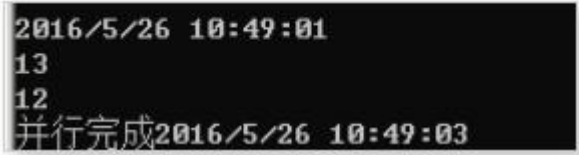
```
task.ContinueWith((ctw) => //当 task 完成后，执行这个回调
```

```
{
```

```
    Console.WriteLine("并行完成" + DateTime.Now);
```



```
});  
Console.WriteLine(DateTime.Now);  
#endregion
```

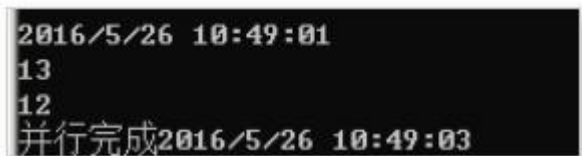


```
2016/5/26 10:49:01  
13  
12  
并行完成2016/5/26 10:49:03
```

二 Parallel 实现的并行



```
#region 并行 Parallel  
Console.WriteLine(DateTime.Now);  
Parallel.Invoke() =>  
{  
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId);  
    Thread.Sleep(1000);  
}, () =>  
{  
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId);  
    Thread.Sleep(2000);  
});  
Console.WriteLine(DateTime.Now);  
Console.ReadKey();  
  
#endregion
```



```
2016/5/26 10:49:01  
13  
12  
并行完成2016/5/26 10:49:03
```

通过上面的图我们可以看到，在进行并行测试时，运行时间为两秒！

第十八讲：IOC 原理 和统一的 IOC 容器

几个名词

依赖倒置原则 (DIP)：一种软件架构设计的原则（抽象概念）。

依赖倒置原则，它转换了依赖，高层模块不依赖于低层模块的实现，而低层模块依赖于高层模块定义的接口。通俗的讲，就是高层模块定义接口，低层模块负责实现。

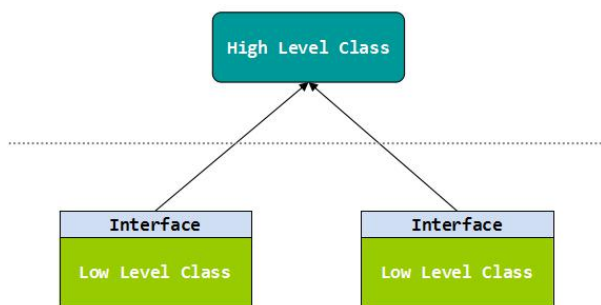
Bob Martins 对 DIP 的定义：

高层模块不应依赖于低层模块，两者应该依赖于抽象。

抽象不不应该依赖于实现，实现应该依赖于抽象。

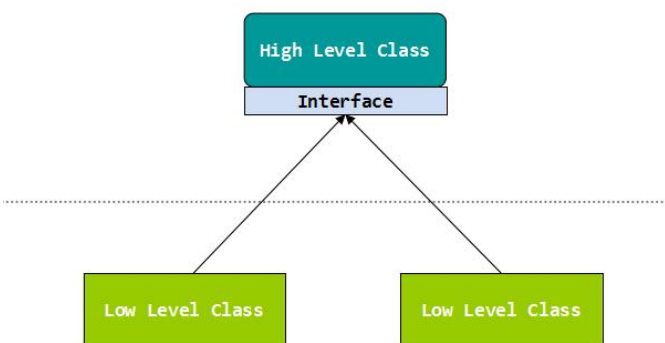
如果生活中的实例不足以说明依赖倒置原则的重要性，那下面我们将通过软件开发的场景来理解为什么要使用依赖倒置原则。

场景一 依赖无倒置（低层模块定义接口，高层模块负责实现）



从上图中，我们发现高层模块的类依赖于低层模块的接口。因此，低层模块需要考虑到所有的接口。如果有新的低层模块类出现时，高层模块需要修改代码，来实现新的低层模块的接口。这样，就破坏了开放封闭原则。

场景二 依赖倒置（高层模块定义接口，低层模块负责实现）



在这个图中，我们发现高层模块定义了接口，将不再直接依赖于低层模块，低层模块负责实现高层模块定义的接口。这样，当有新的低层模块实现时，不需要修改高层模块的代码。

由此，我们可以总结出使用 DIP 的优点：

系统更柔韧：可以修改一部分代码而不影响其他模块。

系统更健壮：可以修改一部分代码而不会让系统崩溃。

系统更高效：组件松耦合，且可复用，提高开发效率。

控制反转 (IoC)：一种反转流、依赖和接口的方式 (DIP 的具体实现方式)。

DIP 是一种 **软件设计原则**，它仅仅告诉你两个模块之间应该如何依赖，但是它并没有告诉如何做。IoC 则是一种 **软件设计模式**，它告诉你应该如何做，来解除相互依赖模块的耦合。**控制反转 (IoC)**，它为相互依赖的组件提供抽象，将依赖 (低层模块) 对象的获得交给第三方 (系统) 来控制，即依赖对象不在被依赖模块的类中直接通过 **new** 来获取。在图 1 的例子我们可以看到，ATM 它自身并没有插入具体的银行卡 (工行卡、农行卡等等)，而是将插卡工作交给人来控制，即我们来决定将插入什么样的银行卡来取钱。同样我们也通过软件开发过程中场景来加深理解。

软件设计原则：原则为我们提供指南，它告诉我们什么是对的，什么是错的。它不会告诉我们如何解决问题。它仅给出一些准则，以便我们可以设计好的软件，避免不良的设计。一些常见的原则，比如 DRY、OCF、DIP 等。

软件设计模式：模式是在软件开发过程中总结得出的一些可重用的解决方案，它能解决一些实际的问题。一些常见的模式，比如工厂模式、单例模式，策略，装饰等等。

```
#region IoC ( 解决问题的方法 )
interface IATM
{
    /// <summary>
    /// 取钱
    /// </summary>
    /// <returns></returns>
    decimal Pull(decimal money);
}

class BankChina : IATM
{
    public decimal Pull(decimal money)
    {
        Console.WriteLine("中国银行取钱:" + money);
        return money;
    }
}

class BankXingYe : IATM
{
    public decimal Pull(decimal money)
    {
        Console.WriteLine("兴业银行取钱:" + money);
        return money;
    }
}

class User
{
    //没有告诉我们如何去实现这种免依赖
    IATM atm = new BankChina();
    public void PullMoney()
    {
        atm.Pull(1000);
    }
}
```

依赖注入 (DI)：IoC 的一种实现方式，用来反转依赖 (IoC 的具体实现方式)。

```

#region 构造方法注入 ( autofac )
IATM _atm;
public UserDI(IATM atm)
{
    _atm = atm;
}
public UserDI()
    : this(new BankChina())
{
}
#endregion

#region 属性注入
private IATM _pAtm; //定义一个私有变量保存抽象

//属性, 接受依赖
public IATM PAtm
{
    set { _pAtm = value; }
    get { return _pAtm; }
}
#endregion

#region 接口注入
IATM _iAtm;
public void SetATM(IATM iAtm)
{
    _iAtm = iAtm;
}
#endregion

public void PullMoney()
{
    _atm.Pull(1000);
}

```

IoC 容器：依赖注入的**框架**，用来映射依赖，管理对象创建和生存周期（DI 框架）。

IoC 容器的功能：

动态创建、注入依赖对象。

管理对象生命周期。

映射依赖关系。

几大框架：

- **.Ninject**: <http://www.ninject.org/>
- **.Castle Windsor**: <http://www.castleproject.org/container/index.html>

- **. Autofac**: <http://code.google.com/p/autofac/>
- **. StructureMap** : <http://docs.structuremap.net/>
- **. Unity** : <http://unity.codeplex.com/>
- **. Spring.NET** : <http://www.springframework.net/>

Autofac 的例子

```
#region IoC容器，依赖注入的框架，用来映射依赖，管理对象创建和生存周期（DI框架）
var builder = new ContainerBuilder();
builder.RegisterType(typeof(BankChina)).As(typeof(IATM));
var container = builder.Build();

var di = container.Resolve<IATM>();
di.Pull(100);
#endregion
```

统一的I O C容器

为了实现I O C容器的松耦合，对它们进行了抽象，在高层提出了 IContain 的接口，主要提供了注册和反转功能


```

/// <summary>
/// IoC容器规范
/// 作者：仓储大叔
/// </summary>
public interface IContainer
{
    /// <summary>
    /// 反射成对象
    /// </summary>
    /// <typeparam name="TService">接口类型</typeparam>
    /// <returns>具体类型</returns>
    TService Resolve<TService>();
    /// <summary>
    /// 反射成对象
    /// </summary>
    /// <typeparam name="TService">接口类型</typeparam>
    /// <returns>具体类型</returns>
    object Resolve(Type type);
    /// <summary>
    /// 反射成对象
    /// </summary>
    /// <typeparam name="TService">接口类型</typeparam>
    /// <param name="overriddenArguments">参数</param>
    /// <returns>具体类型</returns>
    TService Resolve<TService>(object overriddenArguments);
    /// <summary>
    /// 反射成对象
    /// </summary>
    /// <typeparam name="TService">接口类型</typeparam>
    /// <param name="overriddenArguments">参数</param>
    /// <returns>具体类型</returns>
    object Resolve(Type serviceType, object overriddenArguments);
    /// <summary>
    /// 注册抽象类型与具体实现的类型
    /// </summary>
    /// <param name="from">接口类型</param>
    /// <param name="to">具体类型</param>
    void RegisterType(Type from, Type to);
}

```

Lind.DDD 架构中主要对 autofac 和 unity 进行了实现



在 IOC 生产者中去构建它们的实现

```
private IoCFactory()
{
    switch (ConfigConstants.ConfigManager.Config.IocContaion.IoCType)
    {
        case 0:
            _CurrentContainer = new UnityAdapterContainer();
            break;
        case 1:
            _CurrentContainer = new AutofacAdapterContainer();
            break;
        default:
            throw new ArgumentException("不支持此IoC类型");
    }
}
```

在程序调用时，不用关注具体的实现，而为使用抽象的接口进行编程即可

```
#region IoC容器，依赖注入的框架，用来映射依赖，管理对象创建和生存周期 (
IoCFactory.Instance.CurrentContainer.RegisterType(
    typeof(IoCTest),
    typeof(IoCTestChina));
var helloIoC = IoCFactory.Instance.CurrentContainer.Resolve<IoCTest>();
helloIoC.Hello();
#endregion
```

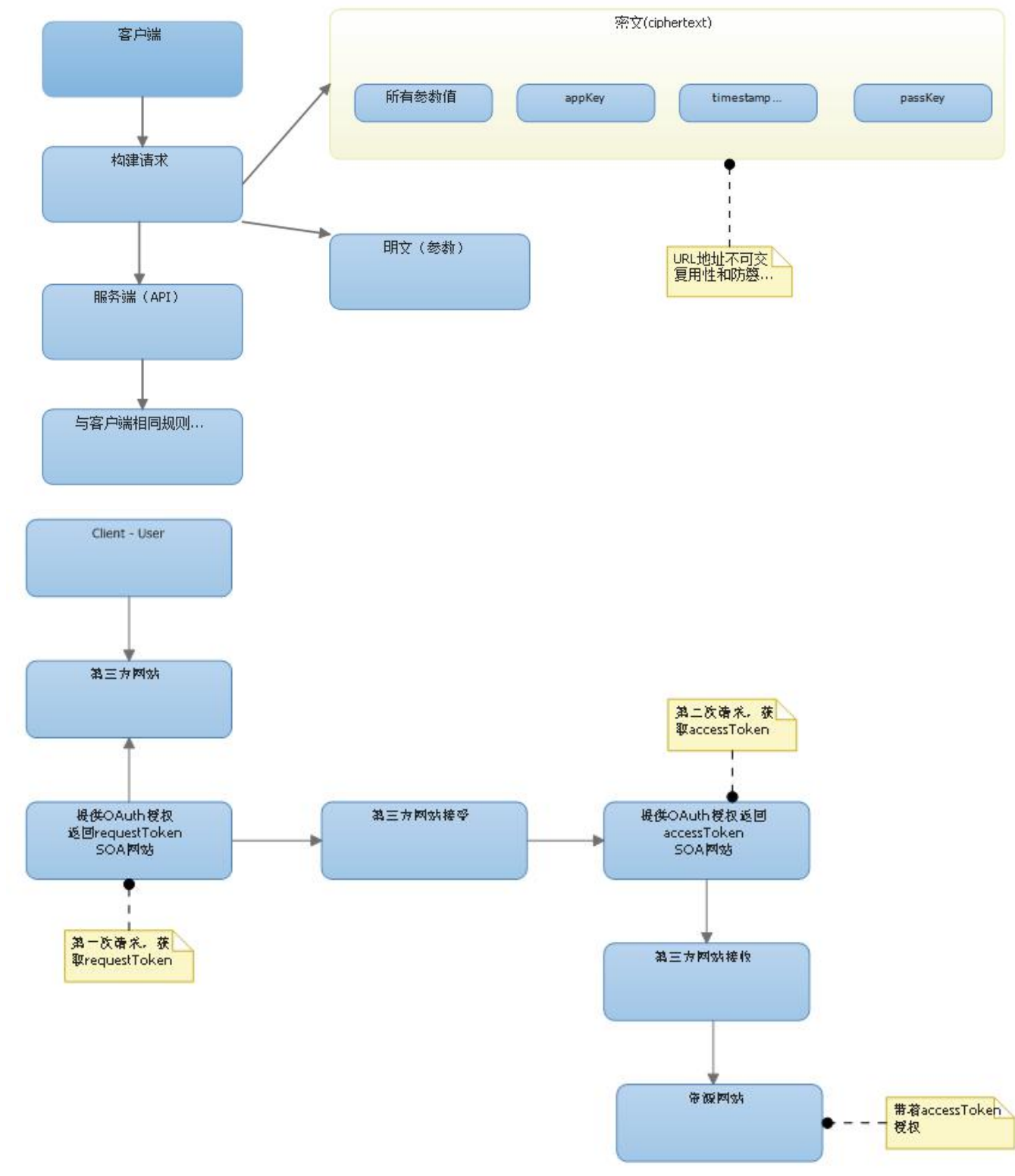
第十九讲：API 安全与校验

REST (英文：**Representational State Transfer**，简称 **REST**) 描述了一个架构样式的网 络系统，比如 web 应用程序。它首次出现在 2000 年 Roy Fielding 的博士论文中，他是 HTTP 规范的主要编写者之一。在目前主流的三种 Web 服务交互方案中，REST 相比于 SOAP (Simple Object Access protocol，简单对象访问协议) 以及 XML-RPC 更加简单明了，无论是对 URL 的处理还是对 Payload 的编码，REST 都倾向于用更加简单轻 量的方法设计和实现。值得注意的是 REST 并没有一个明确的标准，而更像是一种设计的风格。

REST 指的是一组架构[约束条件](#)和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。

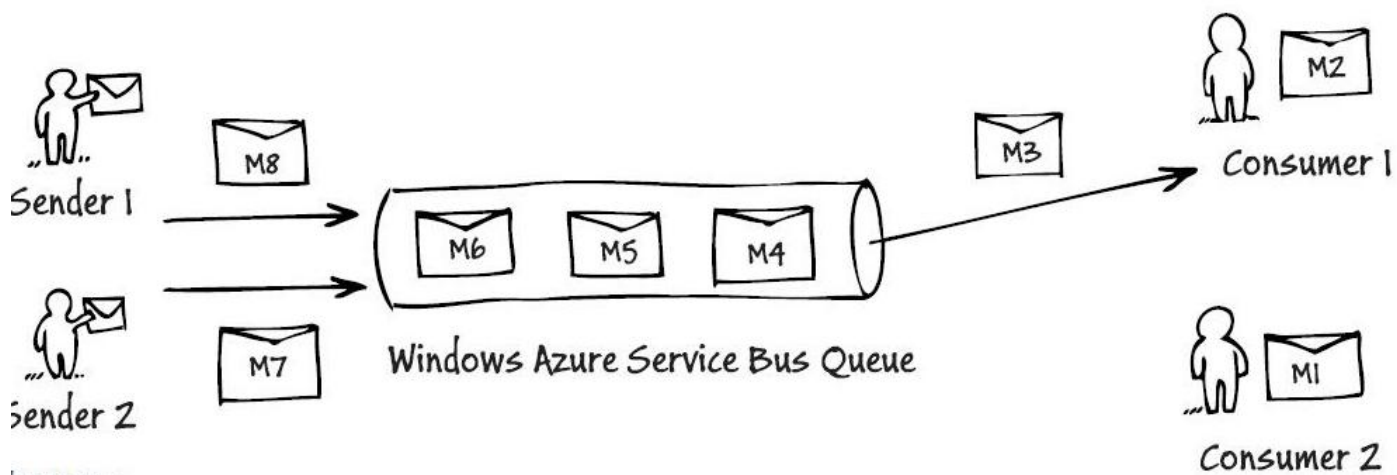
Web 应用程序最重要的 REST 原则是，客户端和服务端之间的交互在请求之间是无状态的。从客户端到服务器的每个请求都必须包含理解请求所必需的信息。如果服务器在请求之间的任何时间点 重启，客户端不会得到通知。此外，无状态请求可以由任何可用服务器回答，这十分适合[云计算](#)之类的环境。客户端可以缓存数据以改进性能。

在服务器端，应用程序状态和功能可以分为各种资源。资源是一个有趣的概念 实体，它向客户端公开。资源的例子有：应用程序对象、数据库记录、算法等等。每个资源都使用 URI (Universal Resource Identifier) 得到一个唯一的地址。所有资源都共享统一的接口，以便在客户端和服务端之间传输状态。使用的是标准的 HTTP 方法，比如 GET、PUT、[POST](#) 和 [DELETE](#)。[Hypermedia](#) 是应用程序状态的[引擎](#)，资源表示通过[超链接](#)互联。

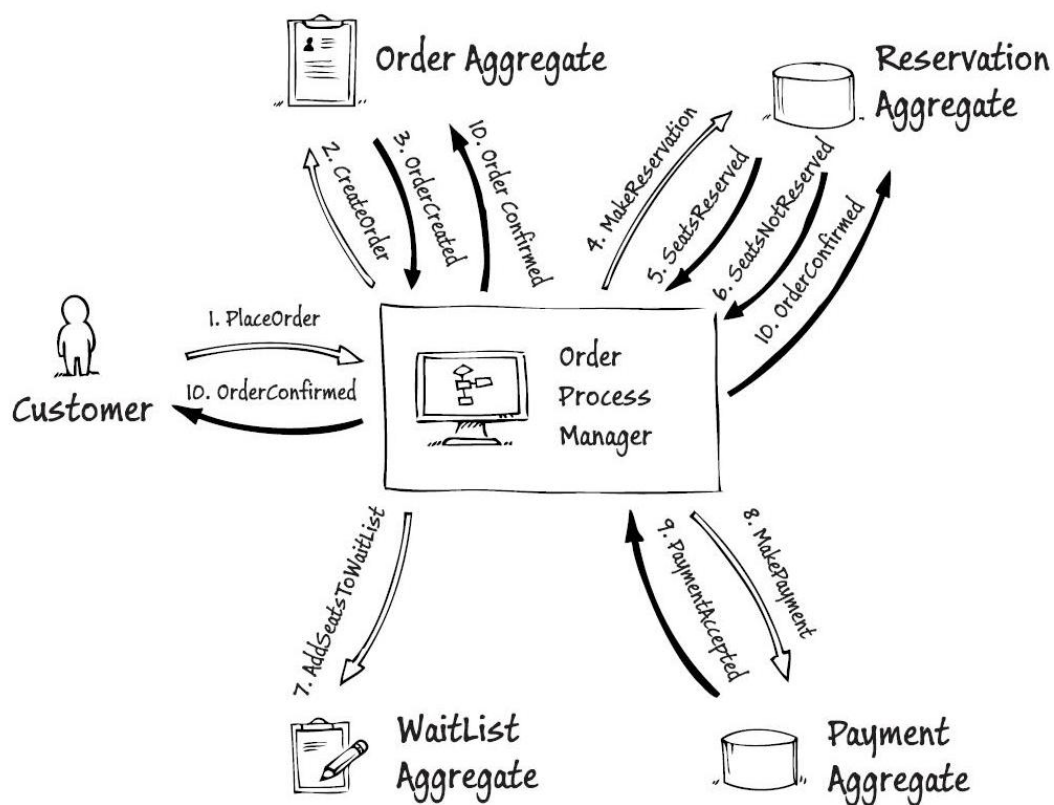


第二十讲：领域驱动的设计模式

领域事件

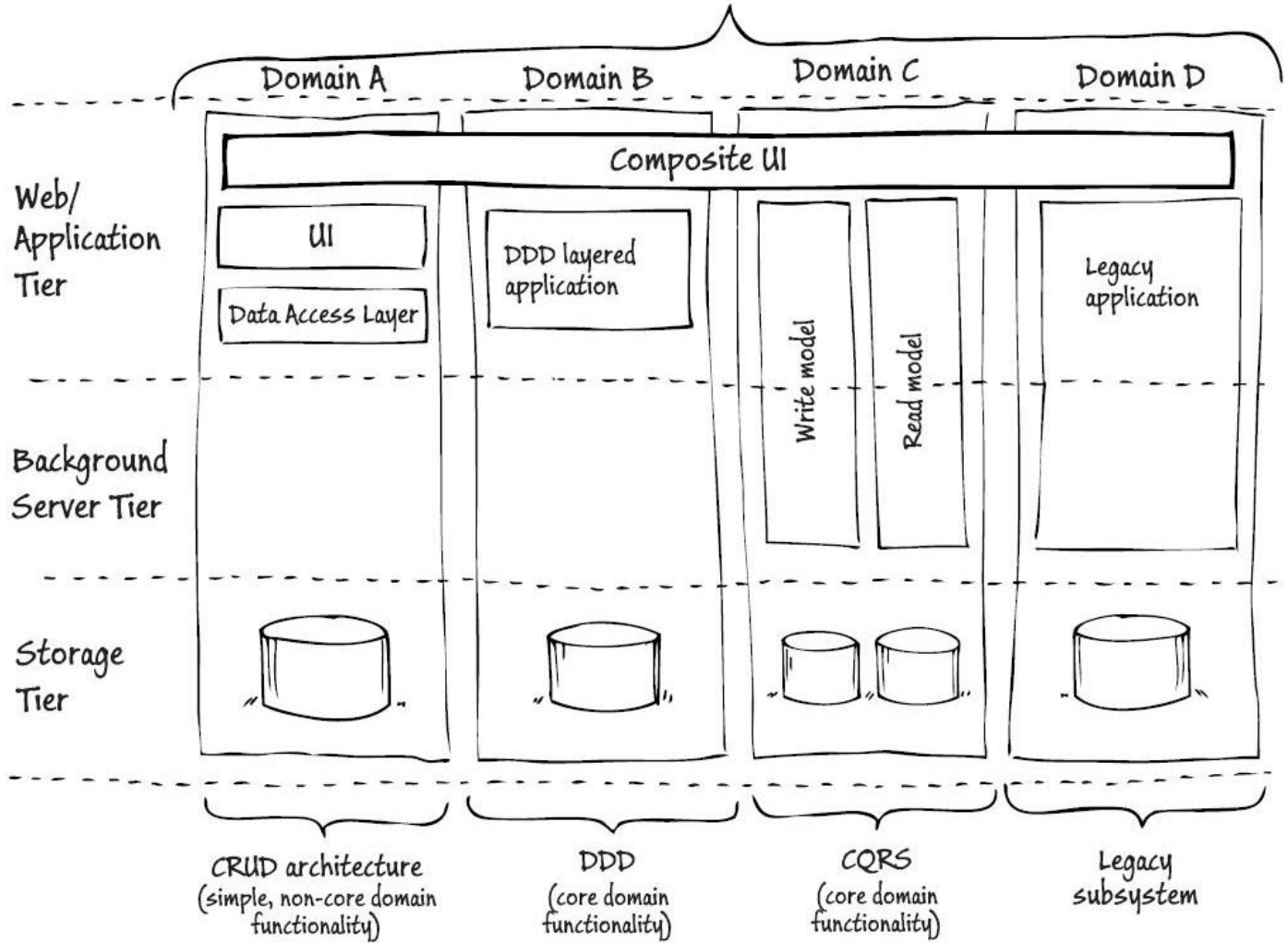


聚合根

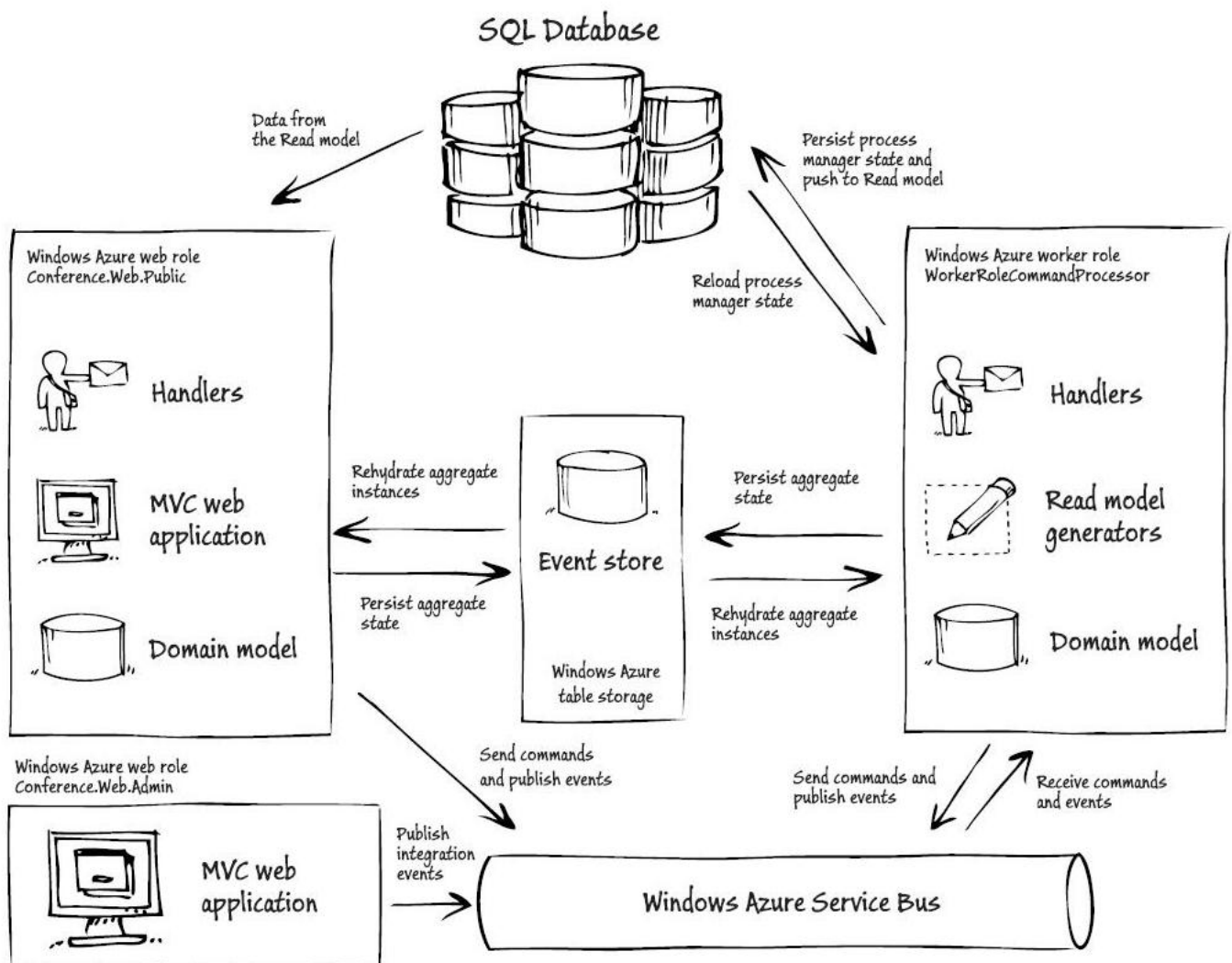


职责与分层

Scope of a Complex Domain



框架与细节



用户界面/展现层

负责向用户展现信息以及解释用户命令。更细的方面来讲就是：

请求应用层以获取用户所需要展现的数据；（显示数据）

发送命令给应用层要求其执行某个用户命令；（添加 / 删除 / 更新数据）

应用层

定义功能点。对外为展现层提供各种应用功能（包括查询或命令），对内调用领域层（领域对象或领域服务）完成各种业务逻辑，应用层不包含业务逻辑。

领域层

负责表达业务概念，业务状态信息以及业务规则，领域模型处于这一层，是业务软件的核心。

基础设施层

本层为其他层提供通用的技术能力；提供了层间的通信；为领域层实现持久化机制

实体（Entity）

实体就是领域中需要唯一标识的领域概念。因为我们有时需要区分是哪个实体。有两个实体，如果唯一标识不一样，那么即便实体的其他所有属性都一样，我们也认为他们两个不同的实体；因为实体有生命周期，实体从被创建后可能会被持久化到数据库，然后某个时候又会被取出来。所以，如果我们不为实体定义一种可以唯一区分的标识，那我们就无法区分到底是这个实体还是哪个实体。另外，不应该给实体定义太多的属性或行为，而应该寻找关联，发现其他一些实体或值对象，将属性或行为转移到其他关联的实体或值对象上。比如 Customer 实体，他有一些地址信息，由于地址信息是一个完整的有业务含义的概念，所以，我们可以定义一个 Address 对象，然后把 Customer 的地址相关的信息转移到 Address 对象上。如果没有 Address 对象，而把这些地址信息直接放在 Customer 对象上，并且如果对于一些其他的类似 Address 的信息也都直接放在 Customer 上，会导致 Customer 对象很混乱，结构不清晰，最终导致它难以维护和理解；

值对象（Value Object）

在领域中，并不是没一个事物都必须有一个唯一标识，也就是说我们不关心对象是哪个，而只关心对象是什么。就上面的地址对象

Address 为例，如 果有两个 Customer 的地址信息是一样的，我们就会认为这两个 Customer 的地址是同一个。也就是说只要地址信息一样，我们就认为是同一个地址。用程序的方式来表达就是，如果两个对象的所有的属性的值都相同我们会认为它们是同一个对象的话，那么我们就可以把这种对象设计为值对象。因此，值对象没有 唯一标识，这是它和实体的最大不同。另外值对象在判断是否是同一个对象时是通过它们的所有属性是否相同，如果相同则认为是同一个值对象；而我们在区分是否 是同一个实体时，只看实体的唯一标识是否相同，而不管实体的属性是否相同；值对象另外一个明显的特征是不可变，即所有属性都是只读的。因为属性是只读的，所以可以被安全的共享；当共享值对象时，一般有复制和共享两种做法，具体采用哪种做法还要根据实际情况而定；另外，我们应该给值对象设计的尽量简单，不要 让它引用很多其他的对象，因为他只是一个值，就像 `int a = 3`；那么“3” 就是一个我们传统意义上所说的值，而值对象其实也可以和这里的“3” 一样理解，也是一个值，只不过是对象来表示。所以，当我们在 C#语言 中比较两个值对象是否相等时，会重写 `GetHashCode` 和 `Equals` 这两个方法，目的就是为了解决对象的值；值对象虽然是只读的，但是可以被整个替 换掉。就像你把 a 的值修改为“4” (`a = 4`)一样，直接把“3” 这个值替换为“4” 了。值对象也是一样，当你要修改 Customer 的 Address 对象引用时，不是通过 `Customer.Address.Street` 这样的方式来实现，因为值对象是只读的，它是一个完整的不可分割的整体。我们可以这样 做：`Customer.Address = new Address(...)`;

领域服务 (Domain Service)

领域中的一些概念不太适合建模为对象，即归类到实体对象或值对象，因为它们本质上就是一些操作，一些动作，而不是事物。这些操作或动作往往会涉及到 多个领域对象，并且需要协调这些领域对象共同完成这个操作或动作。如果强行将这些操作职责分配给任何一个对象，则被分配的对象就是承担一些不该承担的职 责，从而会导致对象的职责不明确很混乱。但是基于类的面向对象语言规定任何属性或行为都必须放在对象里面。所以我们需要寻找一种新的模式来表示这种跨多个 对象的操作，DDD 认为服务是一个很自然的范式用来对应这种跨多个对象的操作，所以就有了领域服务这个模式。和领域对象不同，领域服务是以动词开头来命名 的，比如资金转帐服务可以命名为 `MoneyTransferService`。当然，你也可以把服务理解为一个对象，但这和一般意义上的对象有些区别。因为 一般的领域对象都是有状态和行为的，而领域服务没有状态只有行为。需要强调的是领域服务是无状态的，它存在的意义就是协调领域对象共完成某个操作，所有的 状态还是都保存在相应的领域对象中。我觉得模型（实体）与服务（场景）是对领域的一种划分，模型关注领域的个体行为，场景关注领域的群体行为，模型关注领 域的静态结构，场景关注领域的动态功能。这也符合了现实中出现的各种现象，有动有静，有独立有协作。

领域服务还有一个很重要的功能就是可以避免领域逻辑泄露到应用层。因为如果没有领域服务，那么应用层会直接调用领域对象完成本该是属于领域服务该做 的操作，这样一来，领域层可能会把一部分领域知识泄露到应用层。因为应用层需要了解每个领域对象的业务功能，具有哪些信息，以及它可能会与哪些其他领域对 象交互，怎么交互等一系列领域知识。因此，引入领域服务可以有效的防治领域层的逻辑泄露到应用层。对于应用层来说，从可理解的角度来讲，通过调用领域服务 提供的简单易懂但意义明确的接口肯定也要比直接操纵领域对象容易的多。这里似乎也看到了领域服务具有 `Façade` 的功能，呵呵。

说到领域服务，还需要提一下软件中一般有三种服务：应用层服务、领域服务、基础服务。

应用层服务

获取输入（如一个 XML 请求）；

发送消息给领域层服务，要求其实现转帐的业务逻辑；

领域层服务处理成功，则调用基础层服务发送 Email 通知；

领域层服务

获取源帐号和目标帐号，分别通知源帐号和目标帐号进行扣除金额和增加金额的操作；

提供返回结果给应用层；

基础层服务

按照应用层的请求，发送 Email 通知；

所以，从上面的例子中可以清晰的看出，每种服务的职责；

聚合及聚合根 (Aggregate , Aggregate Root)

聚合，它通过定义对象之间清晰的所属关系和边界来实现领域模型的内聚，并避免了错综复杂的难以维护的对象关系网的形成。聚合定义了一组具有内聚关系的相关对象的集合，我们把聚合看作是一个修改数据的单元。

聚合有以下一些特点：

每个聚合有一个根和一个边界，边界定义了一个聚合内部有哪些实体或值对象，根是聚合内的某个实体；

聚合内部的对象之间可以相互引用，但是聚合外部如果要访问聚合内部的对象时，必须通过聚合根开始导航，绝对不能绕过聚合根直接访问聚合内的对象，也就是说聚合根是外部可以保持 对它的引用的唯一元素；

聚合内除根以外的其他实体的唯一标识都是本地标识，也就是只要在聚合内部保持唯一即可，因为它们总是从属于这个聚合的；

聚合根负责与外部其他对象打交道并维护自己内部的业务规则；

基于聚合的以上概念，我们可以推论出从数据库查询时的单元也是以聚合为一个单元，也就是说我们不能直接查询聚合内部的某个非

根的对象；

聚合内部的对象可以保持对其他聚合根的引用；

删除一个聚合根时必须同时删除该聚合内的所有相关对象，因为他们都同属于一个聚合，是一个完整的概念；

关于如何识别聚合以及聚合根的问题：

我觉得我们可以先从业务的角度深入思考，然后慢慢分析出有哪些对象是：

有独立存在的意义，即它是不依赖于其他对象的存在它才有意义的；

可以被独立访问的，还是必须通过某个其他对象导航得到的；

如何识别聚合？

我觉得这个需要从业务的角度深入分析哪些对象它们的关系是内聚的，即我们会把他们看成是一个整体来考虑的；然后这些对象我们就可以把它们放在一个聚合内。所谓关系是内聚的，是指这些对象之间必须保持一个固定规则，固定规则是指在数据变化时必须保持不变的一致性规则。当我们在修改一个聚合时，我们必须 在事务级别确保整个聚合内的所有对象满足这个固定规则。作为一条建议，聚合尽量不要太大，否则即便能够做到在事务级别保持聚合的业务规则完整性，也可能会 带来一定的性能问题。有分析报告显示，通常在大部分领域模型中，有 70% 的聚合通常只有一个实体，即聚合根，该实体内部没有包含其他实体，只包含一些值对象；另外 30% 的聚合中，基本上也只包含两到三个实体。这意味着大部分的聚合都只是一个实体，该实体同时也是聚合根。

如何识别聚合根？

如果一个聚合只有一个实体，那么这个实体就是聚合根；如果有多个实体，那么我们可以思考聚合内哪个对象有独立存在的意义并且可以和外部直接进行交互。

工厂（Factory）

DDD 中的工厂也是一种体现封装思想的模式。DDD 中引入工厂模式的原因是：有时创建一个领域对象是一件比较复杂的事情，不仅仅是简单的 new 操作。正如对象封装了内部实现一样（我们无需知道对象的内部实现就可以使用对象的行为），工厂则是用来封装创建一个复杂对象尤其是聚合时所需的知识，工厂的作用是将创建对象的细节隐藏起来。客户传递给工厂一些简单的参数，然后工厂可以在内部创建出一个复杂的领域对象然后返回给客户。领域模型中其他元素都不适合做这个事情，所以需要引入这个新的模式，工厂。工厂在创建一个复杂的领域对象时，通常会知道该满足什么业务规则（它知道先怎样实例化一个对象，然后对这个对象做哪些初始化操作，这些知识就是创建对象的细节），如果传递进来的参数符合创建对象的业务规则，则可以顺利创建相应的对象；但是如果由于参数无效等原因不能创建出期望的对象时，应该抛出一个异常，以确保不会创建出一个错误的对象。当然我们也并不总是需要通过工厂来创建对象，事实上大部分情况下领域对象的创建都不会太复杂，所以我们只需要简单的使用构造函数创建对象就可以了。隐藏创建对象的好处是显而易见的，这样可以不会让领域层的业务逻辑泄露到应用层，同时也减轻了应用层的负担，它只需要简单的调用领域工厂创建出期望的对象即可。

仓储（Repository）

仓储被设计出来的目的是基于这个原因：领域模型中的对象自从被创建出来后不会一直留在内存中活动的，当它不活动时会被持久化到数据库中，然后当需要的时候我们会重建该对象；重建对象就是根据数据库中已存储的对象的状态重新创建对象的过程；所以，可见重建对象是一个和数据库打交道的过程。从更广义的角度来理解，我们经常会像集合一样从某个类似集合的地方根据某个条件获取一个或一些对象，往集合中添加对象或移除对象。也就是说，我们需要提供一种机制，可以提供类似集合的接口来帮助我们管理对象。仓储就是基于这样的思想被设计出来的；

仓储里面存放的对象一定是聚合，原因是之前提到的领域模型中是以聚合的概念去划分边界的；聚合是我们更新对象的一个边界，事实上我们把整个聚合看成是一个整体概念，要么一起被取出来，要么一起被删除。我们永远不会单独对某个聚合内的子对象进行单独查询或做更新操作。因此，我们只对聚合设计仓储。

仓储还有一个重要的特征就是分为仓储定义部分和仓储实现部分，在领域模型中我们定义仓储的接口，而在基础设施层实现具体的仓储。这样做的原因是：由于仓储背后的实现都是在和数据库打交道，但是我们又不希望客户（如应用层）把重点放在如何从数据库获取数据的问题上，因为这样做会导致客户（应用层）代码很混乱，很可能会因此而忽略了领域模型的存在。所以我们需要提供一个简单明了的接口，供客户使用，确保客户能以最简单的方式获取领域对象，从而可以让它专心的不会被什么数据访问代码打扰的情况下协调领域对象完成业务逻辑。这种通过接口来隔离封装变化的做法其实很常见。由于客户面对的是抽象的接口并不是具体的实现，所以我们可以随时替换仓储的真实实现，这很有助于我们做单元测试。

尽管仓储可以像集合一样在内存中管理对象，但是仓储一般不负责事务处理。一般事务处理会交给一个叫“工作单元（Unit Of Work）”的东西。关于工作单元的详细信息我在下面的讨论中会讲到。

另外，仓储在设计查询接口时，可能还会用到规格模式（Specification Pattern），我见过的最厉害的规格模式应该就是 LINQ 以及 DLINQ 查询了。一般我们会根据项目中查询的灵活度要求来选择适合的仓储查询接口设计。通常情况下只需要定义简单明了的具有固定查询参数的查询接口就可以了。只有是在查询条件是动态指定的情况下才可能需要用到 Specification 等模式。

CQRS 架构

核心思想是将应用程序的查询部分和命令部分完全分离，这两部分可以用完全不同的模型和技术去实现。比如命令部分可以通过领域驱动设计来实现；查询部分可以直接用最快的非面向对象的方式去实现，比如用 SQL。这样的思想有很多好处：

实现命令部分的领域模型不用经常为了领域对象可能会被如何查询而做一些折中处理；

由于命令和查询是完全分离的，所以这两部分可以用不同的技术架构实现，包括数据库设计都可以分开设计，每一部分可以充分发挥其长处；

高性能，命令端因为没有返回值，可以像消息队列一样接受命令，放在队列中，慢慢处理；处理完后，可以通过异步的方式通知查询端，这样查询端可以做数据同步的处理；

Event Sourcing（事件溯源）

对于 DDD 中的聚合，不保存聚合的当前状态，而是保存对象上所发生的每个事件。当要重建一个聚合对象时，可以通过回溯这些事件（即让这些事件重新发生）来让对象恢复到某个特定的状态；因为有时一个聚合可能会发生很多事件，所以如果每次要在重建对象时都从头回溯事件，会导致性能低下，所以我们会在一定时候为聚合创建一个快照。这样，我们就可以基于某个快照开始创建聚合对象了。

DCI 架构

DCI 架构强调，软件应该真实的模拟现实生活中对象的交互方式，代码应该准确朴实的反映用户的心智模型。在 DCI 中有：数据模型、角色模型、以及上下文这三个概念。数据模型表示程序的结构，目前我们所理解的 DDD 中的领域模型可以很好的表示数据模型；角色模型表示数据如何交互，一个角色定义了某个“身份”所具有的交互行为；上下文对应业务场景，用于实现业务用例，注意是业务用例而不是系统用例，业务用例只与业务相关；软件运行时，根据用户的操作，系统创建相应的场景，并把相关的数据对象作为场景参与者传递给场景，然后场景知道该为每个对象赋予什么角色，当对象被赋予某个角色后就真正成为有交互能力的对象，然后与其他对象进行交互；这个过程与现实生活中我们所理解的对象是一致的；

DCI 的这种思想与 DDD 中的领域服务所做的事情是一样的，但实现的角度有些不同。DDD 中的领域服务被创建的出发点是当一些职责不太适合放在任何一个领域对象上时，这个职责往往对应领域中的某个活动或转换过程，此时我们应该考虑将其放在一个服务中。比如资金转帐的例子，我们应该提供一个资金转帐的服务，用来对应领域中的资金转帐这个领域概念。但是领域服务内部做的事情是协调多个领域对象完成一件事情。因此，在 DDD 中的领域服务在协调领域对象做事时，领域对象往往是处于一个被动的地位，领域服务通知每个对象要求其做自己能做的事情，这样就行了。这个过程中我们似乎看不到对象之间交互的意思，因为整个过程都是由领域服务以面向过程的思维去实现了。而 DCI 则通用引入角色，赋予角色以交互能力，然后让角色之间进行交互，从而可以让我们看到对象与对象之间交互的过程。但前提是，对象之间确实是在交互。因为现实生活中并不是所有的对象在做交互，比如有 A、B、C 三个对象，A 通知 B 做事情，A 通知 C 做事情，此时可以认为 A 和 B，A 和 C 之间是在交互，但是 B 和 C 之间没有交互。所以我们需要分清这种情况。资金转帐的例子，A 相当于转帐服务，B 相当于帐号 1，C 相当于帐号 2。因此，资金转帐这个业务场景，用领域服务比较自然。有人认为 DCI 可以替换 DDD 中的领域服务，我持怀疑态度。

第二十一讲：Bootstrap 和后台管理系统

Bootstrap 作为支持响应式布局的一个前端插件，确实发挥着重要的作用，无论你是在手机，平板还是 PC 上浏览网页，都能达到不错的效果，这一切一切，都是 bootstrap 带给我们的！

今天主要说下页面的布局，这是最基础的东西了，当我们设计一个站点时，应该为它设计一个全局性的统一的规范页面，这种页面我们叫它布局页，而在页面上体现出来的东西，就是布局的元素，在 bootstrap 里当然也是不可缺少的东西。

Bootstrap 的布局是一种栅格系统，即它由行和列组成，在使用时需要为页面内容和栅格系统包裹一个 .container 容器。

— .container 类用于固定宽度并支持响应式布局的容器。

```
<div class="container">
...</div>
```

.container-fluid 类用于 100% 宽度，占据全部视口（viewport）的容器。

```
<div class="container-fluid">
...</div>
```

二 栅格系统的行和列，在 bootstrap 里，行和列使用 row 和 col 表示，而一行中最多有 12 个列单元组成，col-md-1 表示占用 1 个单元的宽度，而 col-md-7 表示占用 7 个单元的宽度，它们加在一起最多为 12 个单元



```
<div class="row">
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div></div><div class="row">
  <div class="col-md-8">.col-md-8</div>
  <div class="col-md-4">.col-md-4</div></div><div class="row">
  <div class="col-md-4">.col-md-4</div>
  <div class="col-md-4">.col-md-4</div>
  <div class="col-md-4">.col-md-4</div></div>
```



第二和第三行显示出来的效果类似这样

.col-xs-12 .col-md-8								.col-xs-6 .col-md-4			
.col-xs-6 .col-md-4				.col-xs-6 .col-md-4				.col-xs-6 .col-md-4			

三 嵌套列，列中还可以有列，这种嵌套我们需要把 md 改为 sm



```
<div class="row">
  <div class="col-sm-9">
    Level 1: .col-sm-9
    <div class="row">
      <div class="col-xs-8 col-sm-6">
        Level 2: .col-xs-8 .col-sm-6
      </div>
      <div class="col-xs-4 col-sm-6">
        Level 2: .col-xs-4 .col-sm-6
      </div>
    </div>
  </div>
</div></div>
```



效果类似于这样

Level 1: .col-sm-9	
Level 2: .col-xs-8 .col-sm-6	Level 2: .col-xs-4 .col-sm-6

[Bootstrap~Panel 和 Table](#)

```
<div class="panel panel-default">
  <div class="panel-heading">Panel heading without title</div>
  <div class="panel-body">
    Panel content
  </div>
</div>
```

Panel heading without title
Panel content

```
<table class="table table-hover">
  ...
</table>
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

第二十二讲：使用 xamarin 进行移动开发

[环境部署](#)

现在移动开发很 HOT,以至于很多人都转向了它,大叔也不例外,这次有机制接触一下 xamarin 这个东西,其实之前也用于 xamarin,只是用来写网页程序,没有接触到移动开发,对于 xamarin 的移动开发分为三个分支,android,ios 和 winform,以后可能还会支持其它的开发,xamarin 这个东西是跨平台的,本身也有 mac>window 版本,根据你的需要和习惯而定.

Android 需要涉及的几个东西

java jdk:(Java Development Kit)大名鼎鼎的 java 运行环境,JDK 是整个 Java 的核心,包括了 Java 运行环境、Java 工具和 Java 基础类库

Android sdk(android software development kit)软件开发工程师用于为特定的软件包,软件框架,硬件平台、操作系统等建立应用程序的开发工具的集合。我们平时说的 sdk4.2,sdk5.1,sdk6.0 说的就是这个东西

Android ndk:Android NDK 是在 SDK 前面又加上了“原生”二字,即 Native Development Kit,因此又被 Google 称为“NDK”。

众所周知,Android 程序运行在 Dalvik 虚拟机中,NDK 允许用户使用类似 C / C++之类的原生代码语言执行部分程序。

Android AVD:(android virtual device),安卓的虚拟机,本机提供的速度超慢,还不如自己下载第三方的

sdk 相关介绍

NDK 包括了:

从 C / C++生成原生代码库所需要的工具和 build files。

将一致的原生库嵌入可以在 Android 设备上部署的应用程序包文件 (application packages files , 即.apk 文件) 中。

支持所有未来 Android 平台的一些列原生系统头文件和库

为何要用到 NDK?

概括来说主要分为以下几种情况:

1. 代码的保护,由于 apk 的 java 层代码很容易被反编译,而 C/C++库反编译难度较大。
2. 在 NDK 中调用第三方 C/C++库,因为大部分的开源库都是用 C/C++代码编写的。
3. 便于移植,用 C/C++写的库可以方便在其他的嵌入式平台上再次使用。

环境搭建

1 下载 java jdk 并安装

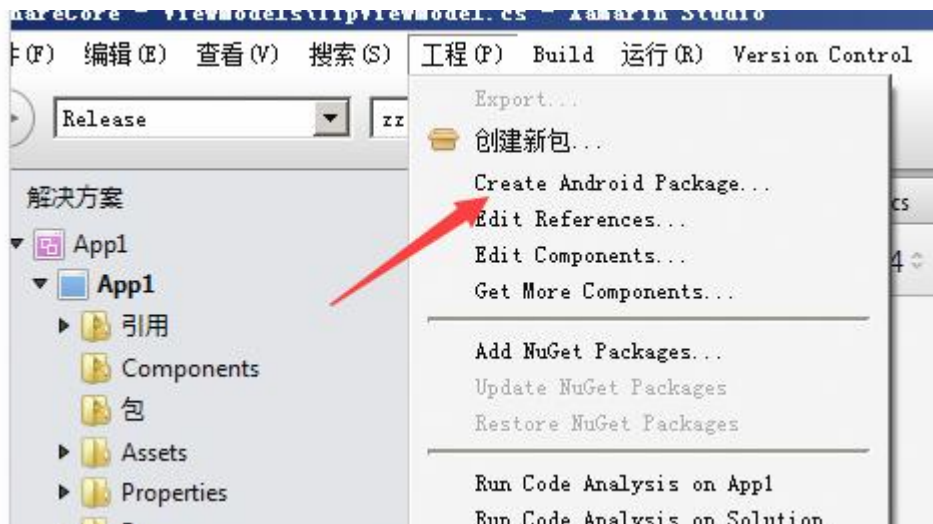
2 下载 android sdk 并解压

3 下载 xamarin,版本为 3.11.666,目前这个版本破解后比较稳定,没有大小的限制

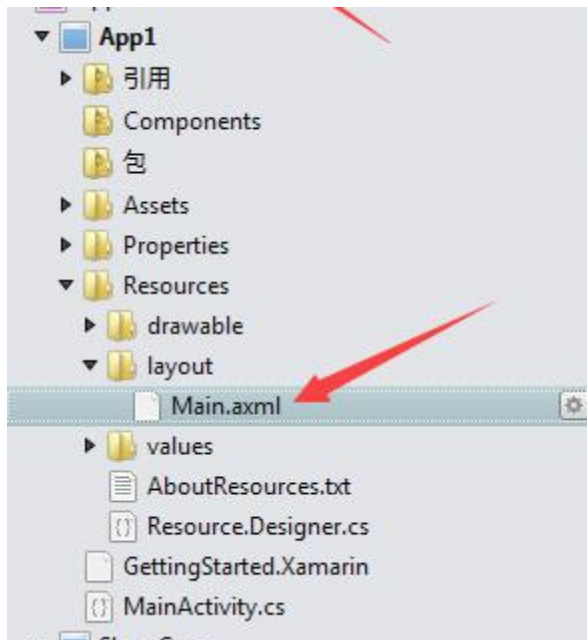
4 下载 xamarin studio 开发环境

建立 android,并生成 apk

注意生成时需要使用 release 模式



可以通过从工具箱拖动工具来实现简单的布局



生成 apk 文件,使用指定模拟器进行打开,或者真机测试



现在我们的第一个 android 程序就完成了!

[基础语法](#)

程序的入口

```
protected override void OnCreate (Bundle savedInstanceState)
{
    base.OnCreate (savedInstanceState);
```

页面的渲染

```
// Set our view from the "main" layout resource
SetContentView (Resource.Layout.Main);
```

按钮的调用，采用了匿名委托的方式

```
// Get our button from the layout resource,
// and attach an event to it
Button button = FindViewById<Button> (Resource.Id.myButton);

button.Click += delegate {
    button.Text = string.Format ("{0} clicks!", count++);
};
```

手机上日志的输出

```
Android.Util.Log.Info ("normal", "日志zz1");
```

页面上元素的获取，泛型方法的类型为要获取对象的类型，如 button, textview 等

```
var loginBtn = FindViewById<Button> (Resource.Id.loginBtn);
var username = FindViewById<TextView> (Resource.Id.username);
var password = FindViewById<TextView> (Resource.Id.password);
var result = FindViewById<TextView> (Resource.Id.result);
```

开启一个新的窗口

```
Intent intent = new Intent(this, typeof(ViewPageActivity));
StartActivity(intent);
```

使用 WebView 打开 H 5 页面，这个在开发时经常用到

```
var webView = FindViewById<WebView> (Resource.Id.webView);
//启用Javascript Enable
webView.Settings.JavaScriptEnabled = true;
//载入网址
webView.LoadUrl ("http://www.sina.com");
//直接在当前webView上打开
webView.SetWebViewClient (new CustWebViewClient ());
```

自定义的 WebViewClient 让页面在当前窗口打开（避免唤起手机浏览器）

```
public class CustWebViewClient : WebViewClient
{
    public override bool ShouldOverrideUrlLoading (WebView view, string url)
    {
        view.LoadUrl (url);
        return true;
    }
}
```

多个窗口 (Active)之前的数据传递

原窗口

```
Intent intent = new Intent(this, typeof(UserInfoLayoutActivity));
/* 通过Bundle对象存储需要传递的数据 */
Bundle bundle = new Bundle();
/*字符、字符串、布尔、字节数组、浮点数等等，都可以传*/
intent.PutExtra("Title", datas[e.Position].Title);
intent.PutExtra("Desc", datas[e.Position].Desc);
intent.PutExtra("AssistsCount", datas[e.Position].AssistsCount);
intent.PutExtra("Fails", datas[e.Position].Fails);
intent.PutExtra("Score", datas[e.Position].Score);
intent.PutExtra("Level", datas[e.Position].Level);
intent.PutExtra("Image", datas[e.Position].Image);
/*把bundle对象assign给Intent*/

intent.PutExtras(bundle);
StartActivity(intent);
```

目标窗口

```

Title = Intent.GetStringExtra("Title"),
Desc = Intent.GetStringExtra("Desc"),
AssistsCount = Intent.GetIntExtra("AssistsCount", 0),
Level = Intent.GetIntExtra("Level", 0),
Fails = Intent.GetIntExtra("Fails", 0),
Image = Intent.GetIntExtra("Image", 0),
Score = Intent.GetIntExtra("Score", 0),

```

手机上返回按钮，双击退出程序



实现的原理就是在 activity 里设置一个时间，单击后把这个时间赋值并与当前时间进行比较，如果在 2 秒内，就认为是双击操作，当然这个时间间隔你可以自己设置，在认为是双击操作后，执行 Finish() 方法即可以返回到主窗口，当然，单击操作你也可以设置自己的行为，本例中是在 webView 中打开新的页面。

```

DateTime? lastBackKeyDownTime;
public override bool OnKeyDown(Keycode keyCode, KeyEvent e)
{
    if (keyCode == Keycode.Back && e.Action == KeyEventActions.Down)
    {
        if (!lastBackKeyDownTime.HasValue || DateTime.Now - lastBackKeyDownTime.Value > new TimeSpan(0, 0, 2))
        {
            Toast.MakeText(this.ApplicationContext, "再按一次退出程序", ToastLength.Short).Show();
            lastBackKeyDownTime = DateTime.Now;
            loadUrl("/Task/CurrentTaskList");
        }
        else
        {
            Finish();
        }
        return true;
    }
    return base.OnKeyDown(keyCode, e);
}

```

[Api 与安卓 Session 的同步机制](#)

- > 客户端
- > (Request)访问服务端页面

- > 服务端产生 SessionId
- > 存储到服务端
- > **(Response)**同时向客户端相应
- > 客户端存储把 SessionID 到 Cookies 里 (.net 平台 cookies 里键名为 ASP.NET_SessionId)
- > 下次请求，客户端将在 Request 头信息中把当前 SessionID 发到服务端
- > 服务端的 SessionID 通过过期时间维护它的有效性

实现安卓端向 A P I 请求头写 session

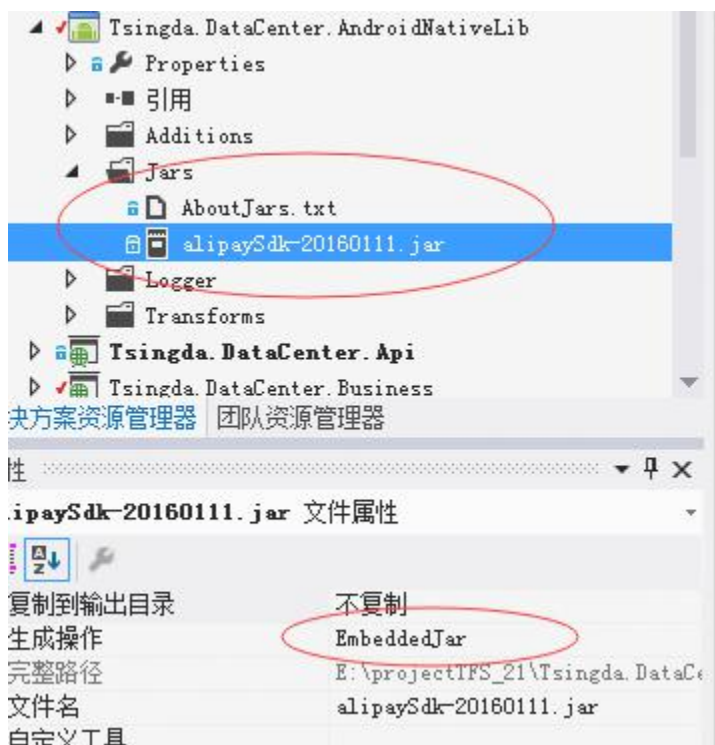
```
Uri uri = new Uri(GetString(Resource.String.apiHost));
HttpClientHandler handler = new HttpClientHandler();
handler.CookieContainer = new CookieContainer();
handler.CookieContainer.Add(uri, new Cookie("ASP.NET_SessionId", InMemory.SessionID)); // Adding a Cookie

using (var http = new HttpClient(handler))
{
    var content = new FormUrlEncodedContent(new Dictionary<string, string>() { });
    var response = http.PostAsync(GetString(Resource.String.apiHost) + "/Test/CurrentTaskListApi", content);

    var obj = JsonConvert.DeserializeObject<List<Task_Info>>(response.Result.Content.ReadAsStringAsync().Result);
    listView.Adapter = new Task_InfoListAdapter(this, obj);
}
```

[安卓原生库的使用 jar，支付宝原生 A P P](#)

我们新建一个“android 绑定库项目”，然后把 jar 文件放到 jars 文件夹里，把它的生成方式改成“EmbeddedJar”，然后在目标的 android 应用程序里引用它即可



当用户手机或者设置上没有安装支付宝时，我们应该让它跳到支付宝的 H 5 页面进行支付，这个配置我们可以在 AndroidManifest.xml 里进行设置，将下面代码添加到 application 节点下


```

<application android:label="Tsingda.DataCenter.AndroidNative" android:icon="@drawable/Icon">
    <activity android:name="com.alipay.sdk.app.H5PayActivity" android:configChanges="orientation|keyboardHidden|navigation"
android:exported="false" android:screenOrientation="behind"></activity>
    <activity android:name="com.alipay.sdk.auth.AuthActivity" android:configChanges="orientation|keyboardHidden|navigation"
android:exported="false" android:screenOrientation="behind"></activity>
</application>

```

在 A P P 里进行调用，主要对支付参数的组装和对支付宝原生的调用

```

private void Pay()
{
    try
    {
        var con = getOrderInfo("test", "testbody");
        var sign = SignatureUtils.Sign(con, RSA_PRIVATE);
        sign = URLEncoder.Encode(sign, "utf-8");
        con += "&sign=\"" + sign + "\"&" + MySignType;
        Com.Alipay.Sdk.App.PayTask pa = new Com.Alipay.Sdk.App.PayTask(this);
        var result = pa.Pay(con, false);
        Logger_Info("支付宝result:" + result);
    }
    catch (Exception ex)
    {
        Logger_Info("2" + ex.Message + ex.StackTrace);
    }
}

```

Xamarin 这讲今天就到这里吧！

第二十三讲：Node.js 与 Sails 框架

安装：<https://nodejs.org/en/>

Node.js:后端开发语言，类似 asp,.net,php,jsp，它寄宿在 node.exe 这个进程。

特点：单线程，非阻塞的语言

阻塞：对于 C #程序来说，它有阻塞，多个线程访问同一个资源（ public static 属性），如果不加锁，这时会出现并发现象（结果就是它的不确定性），解决并发，我们需要对资源加锁，这时一个线程访问它时，其它线程处理等待，这种等待我们叫它“阻塞”。

Node.js 如何实现非阻塞：通过方法的回调实现的（ callback ）。

Node 框架：express 框架，sails.js 框架（ mvc ）

开发工具：webstorm10

包管理工具：NPM 实现对包的管理，添加，删除，包都会放在 modules 目录下

包文件：package.json 是项目移植的基础，它存储了项目中用到的所有的包

Sails.js：一个MVC框架，可以通过NPM去安装，<http://sailsjs.org/>

Sails 安装

```
npm -g install sails
```

Sails 项目的建立

```
sails new testProject
```

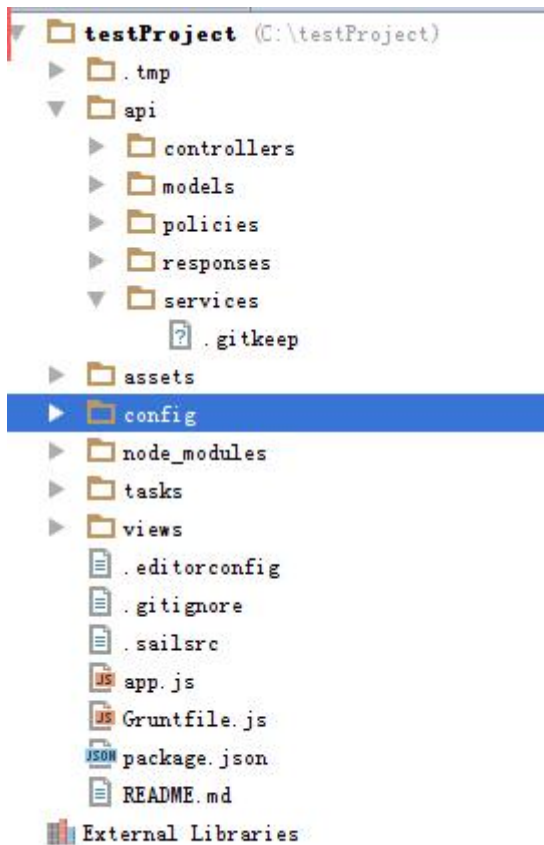
运行 sails 程序，这个过程是监听某个端口的过程

```
cd testProject  
sails lift
```

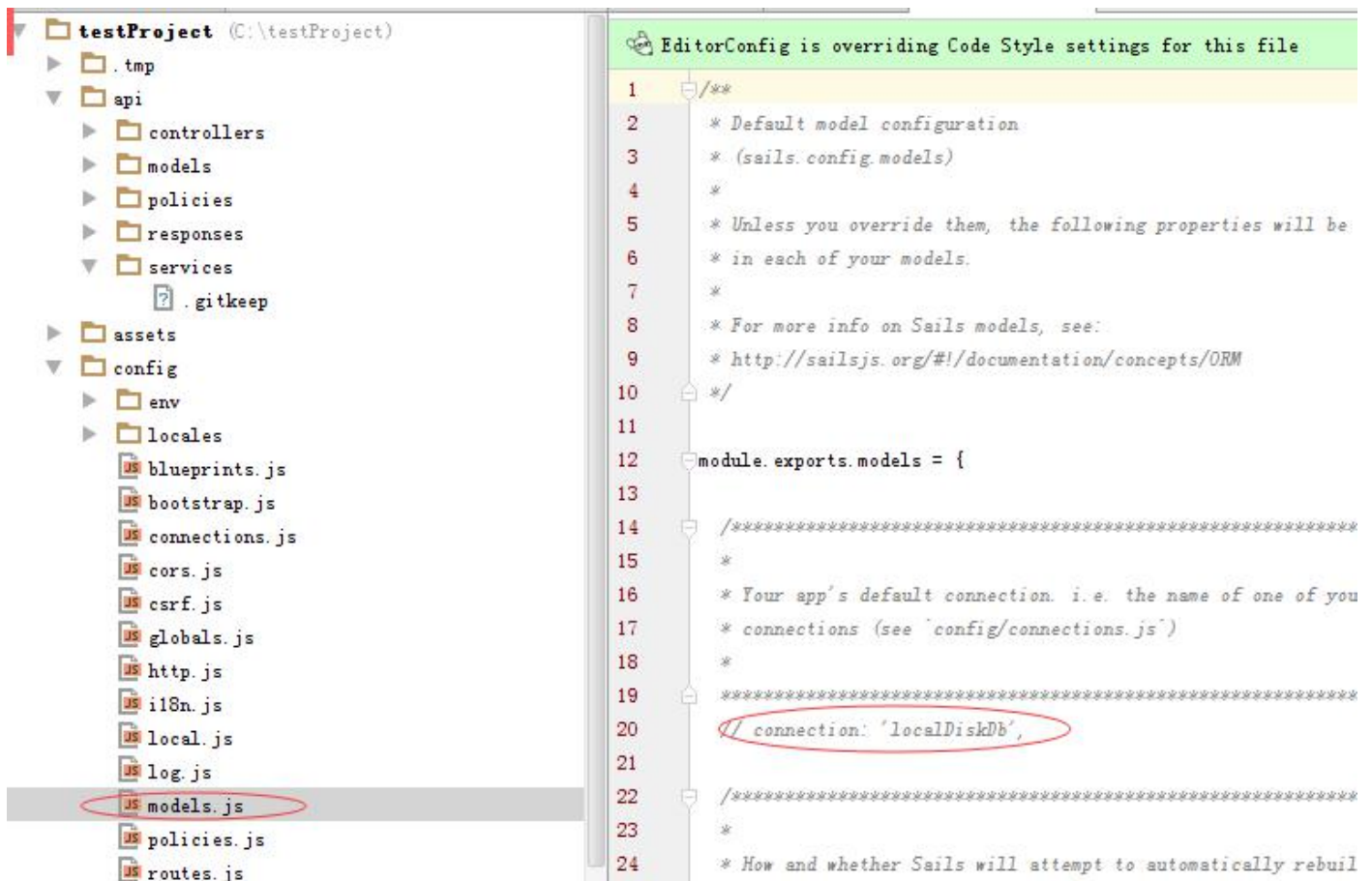
EJS:文件的布局，让HTML代码与node动态代码有效的结合

App.js:sails 应用程序的入口

Sails 结构



Sails 里设置ORM：设置数据持久化的方式



Sails 里的页面是通过 api/controller 去渲染的

api

controllers

.gitkeep

userController.js

models

policies

responses

services

assets

config

node_modules (Library home)

projectFilesBackup

tasks

views

.editorconfig

.gitignore

.sailsrc

app.js

Gruntfile.js

package.json

README.md

16

return res.redirect("http://www.cnblogs.com");

17

}

18

index: function (req, res) {

19

var params={};

20

if(req.param("name")!=undefined)

21

params["name"]=req.param("name");

22

UserService.getUser(params,function (err, data) {

23

if (err) {

24

console.log(err);

25

}

26

return res.view({user: data});return res.view("vie

27

28

});

29

},

30

add: function (req, res) {

31

return res.view();

32

},

33

save: function (req, res) {

34

var tmp = ['name', 'sex', 'updateCount', 'passw

35

var id = req.param("id");

36

var opt = {};

Sails 里的模型：模型是被数据持久化的对象

module.exports = {

autoPK: true, //这是默认值，可以省略

tableName: 'UserInfo',

attributes: {

name: {

type: 'string',

size: 255

},

password: {

type: 'string',

size: '30'

},

sex: {

type: 'integer',

defaultsTo: 0

},

updateCount: {

type: 'integer',

defaultsTo: 0

}

}

},

behaviors: { //扩展方法

md5Password: function (password) {

return md5(password, "");

}

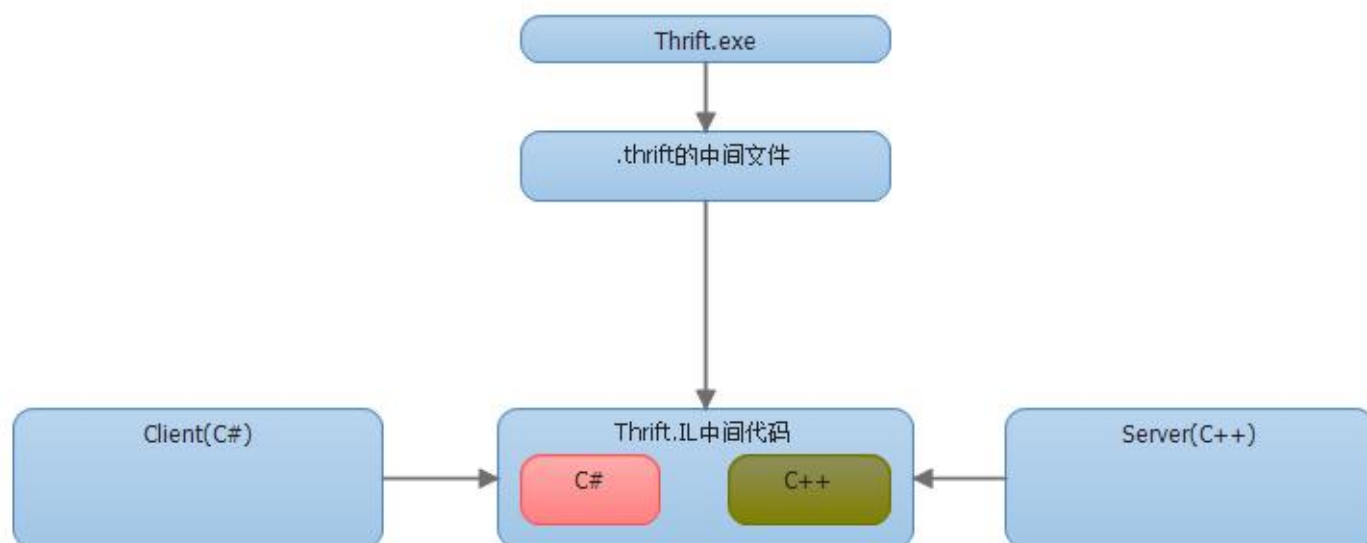
}

ORM为我们提供了标准的CURD操作

```
addUser: function (param, cb) {  
    var opt = param || { name: 'zxl' };  
  
    Person.create(opt).exec(function (err, record) {  
  
        console.log("添加")  
        if (err) {  
            cb('ERROR_DATABASE_EXCEPTION');//输出错误  
        } else {  
  
            cb(null, record);//正确返回  
        }  
    });  
};
```

第二十四讲：跨语言开发 Thrift 框架

Thrift 是一个跨语言的服务部署框架，最初由 Facebook 于 2007 年开发，2008 年进入 Apache 开源项目。Thrift 通过一个中间语言 (IDL, 接口定义语言) 来定义 RPC 的接口和数据类型，然后通过一个编译器生成不同语言的代码（目前支持 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk 和 OCaml），并由生成的代码负责 RPC 协议层和传输层的实现。



Thrift 实际上是实现了 C/S 模式，通过代码生成工具将接口定义文件生成服务器端和客户端代码（可以为不同语言），从而实现服务端和客户端跨语言的支持。用户在 Thrift 描述文件中声明自己的服务，这些服务经过编译后会生成相应语言的代码文件，然后用户实现服务（客户端调用服务，服务器端提供服务）便可以了。其中 protocol（协议层，定义数据传输格式，可以为二进制或者 XML 等）和 transport（传输层，定义数据传输方式，可以为 TCP/IP 传输，内存共享或者文件共享等）被用作运行时库。上图的详细解释参考引用【1】。

支持的数据传输格式、数据传输方式和服务模型

(1) 支持的传输格式

TBinaryProtocol – 二进制格式.

TCompactProtocol – 压缩格式

TJSONProtocol – JSON 格式

TSimpleJSONProtocol –提供 JSON 只写协议, 生成的文件很容易通过脚本语言解析。

TDebugProtocol – 使用易懂的可读的文本格式, 以便于 debug

(2) 支持的数据传输方式

TSocket -阻塞式 socket

TFramedTransport – 以 frame 为单位进行传输, 非阻塞式服务中使用。

TFileTransport – 以文件形式进行传输。

(3) 支持的服务模型

TSimpleServer – 简单的单线程服务模型, 常用于测试

TThreadPoolServer – 多线程服务模型, 使用标准的阻塞式 IO。

TNonblockingServer – 多线程服务模型, 使用非阻塞式 IO (需使用 TFramedTransport 数据传输方式)

程序流程图

- 服务端设计数据模块,.thrift
- ->
- Thrift.exe 生成中间代码,C#和 C++版
- ->
- 开发服务端程序,并引用 C++的模型
- ->
- 开发客户端端,并引用 C#的模型
- ->
- 调试代码完成

实例代码

- Thrift 数据模型


```
namespace csharp HelloThriftspace

exception Xception {
    1: i32 errorCode,
    2: string message
}

service HelloThrift{
    void HelloWorld() throws (1:Xception ex)
}
```

- 生成指定语言的数据模型

```
#if SILVERLIGHT
public IAsyncResult send_HelloWorld(AsyncCallback callback, object state)
#else
public void send_HelloWorld()
#endif
{
    oprot_.WriteMessageBegin(new TMessage("HelloWorld", TMessageType.Call, seqid_));
    HelloWorld_args args = new HelloWorld_args();
    args.Write(oprot_);
    oprot_.WriteMessageEnd();
    #if SILVERLIGHT
    return oprot_.Transport.BeginFlush(callback, state);
    #else
    oprot_.Transport.Flush();
    #endif
}

public void recv_HelloWorld()
{
    TMessage msg = iprot_.ReadMessageBegin();
    if (msg.Type == TMessageType.Exception) {
        TApplicationException x = TApplicationException.Read(iprot_);
        iprot_.ReadMessageEnd();
        throw x;
    }
    HelloWorld_result result = new HelloWorld_result();
    result.Read(iprot_);
    iprot_.ReadMessageEnd();
    if (result.__isset.ex) {
        throw result.Ex;
    }
    return;
}
```

- 服务端代码

```

public class HelloThriftHandler :
    HelloThriftspace.HelloThrift.Iface
{
    public void HelloWorld()
    {
        Console.WriteLine("hello world!");
    }
}

//开始Thrift rpc服务
new Thread(() =>
{
    var processor1 = new HelloThrift.Processor(new HelloThriftHandler());
    TMultiplexedProcessor processor = new TMultiplexedProcessor();
    processor.RegisterProcessor("HelloThriftHandler", processor1);
    var serverTransport = new TServerSocket(9090);
    var server1 = new TThreadedServer(processor, serverTransport);
    Console.WriteLine("向客户端输出服务开启");
    server1.Serve();
}).Start();

```

- 客户端代码

```

static void Main(string[] args)
{
    //调用服务端的HelloThrift的HelloWorld方法
    TTransport transport = new TSocket("localhost", 9090);
    class Thrift.Transport.TTransportFactoryProtocol(transport);
    TMultiplexedProtocol mp1 = new TMultiplexedProtocol(protocol, "HelloThriftHandler");
    HelloThrift.Client client = new HelloThrift.Client(mp1);
    transport.Open();
    client.HelloWorld();
    transport.Close();
}

```

- 总结

使用 Thrift 进行数据通讯,让我们可以在多个语言中自由的实现交互,它不向过去的 RPC,还需要封装一个个的数据协议包,使用 Thrift 真正做到了面向接口的编程,公开的是更友好的方法约定,而不是抽象的数据包.

第二十五讲：什么样的代码需要重构

参考文章：大叔代码重构 <http://www.cnblogs.com/lori/archive/2012/07/23/2604729.html>

由来

在我们的项目开发完成后，需要对原来代码进行 review，对一些代码结构混乱，逻辑不清晰，违背 DRY 和 OCP 的一些代码进行重构的工作，这是必要的，也是改善一个软件系统的必经之路！

啥时应该进行重构

对于大部分开发者来说，重构的含义可能是系统开发完成后的检查工作，但我认为，重构应该是遍及软件开发的各个阶段的，从开发到调试再到最后的代码审查及改版，整个阶段都会出现代码重构的身影，这是可以理解的，任何开发都不可能写出一个自己不去修改的永恒程序，反之，好的开发者应该是会经常发现自己代码的缺陷，从而去改善它，使它们的程序变得更大强大，扩展性，维护性，稳定性都更强！

本讲将会涉及的点

[1 封装成员变量 \(Encapsulate Field \)](#)

[2 提取方法 \(Extract Method \)](#)

[3 提取到类 \(Extract Class \)](#)

[4 方法归父 \(方法上移 \)](#)

[5 方法归子 \(方法下移 \)](#)

[6 方法更名](#)

[7 代码注释](#)

第二十六讲：基础篇续～接口与抽象类，集合与数组，树与链表

接口：行为接口，标识接口

第二十七讲：设计模式在项目中很自然的出现了

第二十八讲：单点统一登陆 SSO 的设计

第二十九讲：Session 共享与 WEB 集群

第三十讲：Lind.DDD 设计初衷与它的未来（代码需要人性化）