

EXPERIENCE RECORD

DOLPHIN PROJECT

Computer Science

Author:

Xiaoqiang JIANG

Supervisor:

Donald KNUTH

July 13, 2016

经验归纳记录

蒋小强¹

2015.04

¹本书由 LATEX 工具生成，作者 mail:jiangtingqiang@gmail.com

表 1: 修改记录¹

序号	修改人	修改日期	备注
1	蒋小强	2015-04-22	创建基础版本
2	蒋小强	2015-06-10	添加《程序设计原则》节
3	蒋小强	2015-07-01	添加设计模式相关章节
4	蒋小强	2015-07-12	添加 Web 设计相关目录，添加 MVC 相关章节
5	蒋小强	2015-08-08	添加 WCF 配置相关章节
6	蒋小强	2015-09-12	添加 WCF REST 跨域调用
7	蒋小强	2015-09-12	添加 SQL Server 链接服务器内容
8	蒋小强	2015-09-22	添加 JavaScript、jQuery 相关内容
9	蒋小强	2015-09-29	添加 NodeJS 路由、NodeJS RESTful Service、NodeJS 数据库连接相关内容
10	蒋小强	2015-11-14	添加 Spring.NET 依赖注入相关内容
11	蒋小强	2015-11-26	添加 Chrome 调试网页相关内容
12	蒋小强	2015-12-25	添加浏览器渲染 (Render) 相关内容
13	蒋小强	2016-02-05	添加 RESTful API 授权流程
14	蒋小强	2016-06-11	添加 Wireshark 捕获包与 TCP 协议结合理解相关内容

¹表格 LATEX 代码生成可到此处<http://www.tablesgenerator.com/>

目录

I 整体设计	37
1 规范准则	41
1.1 程序设计原则	42
1.1.1 编程经验	42
1.1.2 程序语言喜好统计	42
1.1.3 Architectural Requirements	42
1.1.4 不要自我重复 (DRY - Don't Repeat Yourself)	44
1.1.5 单一职责原则 (Single Responsibility Principle,SRP)	44
1.1.6 开放/封闭原则 (Open Closed Principle)	44
1.1.7 保持简单 (Keep It Simple)	44
1.1.8 用最简单的方法让程序跑起来	45
1.1.9 避免过早优化	45
1.1.10 依赖倒置原则 (Dependence Inversion Principle)	45
1.1.11 接口隔离原则 (Interface Segregation Principle)	45
1.1.12 关注点分离 (Separation of Concerns)	46
1.1.13 封装变化点	46
1.1.14 不要开发你目前用不到的功能 (You Ain't Gonna Need It, YAGNI 原则)	46
1.1.15 合成/聚合复用原则 (Composite/Aggregate Reuse Principle, CARP)	46
1.1.16 依赖倒置 (Dependency Inversion Principle)	46
1.1.17 读写分离	47
1.1.18 动静分离	47
1.1.19 数据库表拆分 & 分片	47
1.1.20 主从同步	47

1.1.21 负载均衡	47
1.1.22 去中心化	47
1.1.23 主从分离	47
1.1.24 服务隔离	47
1.1.25 灰度部署	47
1.1.26 异步化	47
1.1.27 统一监控 & 统一日志	48
1.1.28 技术选型	48
1.1.29 一般网站架构	49
1.2 编码规范 (Programming Specification)/编程风格 (Programming Style)	49
1.2.1 良好编码习惯	49
1.2.2 变量 (Variable) 命名规范	50
1.2.3 命名空间 (Namespace) 命名规范	51
1.2.4 类 (Class) 命名规范	51
1.2.5 类 (Class) 注释规范	52
1.2.6 接口 (Interface) 命名规范	52
1.2.7 常量命名规范	53
1.2.8 方法 (Method) 命名规范	53
1.2.9 事件 (Event) 命名规范	53
1.2.10 数据库命名规范	54
1.2.11 创建类标准模板	54
1.2.12 文件夹命名	55
1.2.13 目录结构	55
1.2.14 代码规范检查	56
1.2.15 避免空引用异常 (Null Reference Exception)	56
1.3 运行效率 (Performance)	57
1.3.1 避免开辟内存	57
1.3.2 避免装箱	57
1.3.3 常见数据结构及复杂度	57
1.4 数据库	57

目录	7
1.4.1 字段	57
1.4.2 登录用户权限	58
1.5 环境	58
1.5.1 开发环境	58
1.5.2 Web Server	60
1.5.3 Apache	60
1.5.4 常见架构	61
1.5.5 主流平台	61
2 WCF	63
2.1 基础应用	63
2.1.1 WCF 测试客户端	66
2.1.2 WCF 和 Web Service 的异同	67
2.1.3 WCF 的配置	68
2.1.4 WCF 契约	69
2.1.5 WCF 的调用	69
2.1.6 Web Request 和 ajax 调用	71
2.1.7 WCF REST Ajax 跨域请求调用	75
2.1.8 WCF RESTful 接口传输文件	80
2.1.9 WebGet 和 WebInvoke	82
2.1.10 WCF 开启日志	82
2.1.11 WCF 引用安全	84
2.1.12 WCF RESTful Security	90
2.1.13 WCF 不使用 svc 文件	93
2.1.14 WCF 自定义路由映射	93
2.1.15 WCF 传递 Cookie	94
2.1.16 WCF REST 传递 Cookie	97
2.1.17 调取 HTTPS 类型的 Web Service	97
2.2 WCF 异常处理	98
2.2.1 常见问题	101

3 ASP.NET Web API(Apache License 2.0)	105
3.1 Web Api 帮助文档	107
3.1.1 Controller 分类	107
3.2 Web Api Area 支持	107
3.3 Web Api 路由	107
3.4 参数传递	107
3.4.1 Web Api Json 传递	107
3.4.2 错误处理 (Exception Handling in ASP.NET Web API)	107
3.4.3 消息处理 (HTTP Message Handlers in ASP.NET Web API)	108
3.4.4 Model 绑定 (Model Binding)	108
3.5 Automated Documentation for REST APIs	109
3.5.1 安装 Swagger-UI	109
3.5.2 Swagger	109
4 数据操作 (Data Operate)	111
4.1 生成二维码 (Generate QRCode)	113
4.1.1 获取程序运行路径	114
4.2 序列化 (Serialization) 与反序列化 (Deserialization)	114
4.2.1 序列化 (Serialization) 字典值	117
4.2.2 反序列化 XML	118
4.2.3 XML 增加内容	118
4.2.4 XML 删除内容	118
4.3 数据转换	118
4.3.1 将 DataTable 转为 List	118
4.3.2 Double 转换	120
4.3.3 List 与 string 的转换	120
4.3.4 获取集合的列	121
4.3.5 去除 List 中重复的对象	121
4.3.6 List 中的元素排序	121
4.3.7 Unix 时间戳格式转换	122
4.3.8 Linq 操作	122

4.3.9 分页查询语句	124
4.3.10 事务	125
4.3.11 随机查询	126
4.3.12 自定义异常 (Custom Exception)	126
4.3.13 根据经纬度计算距离	127
4.3.14 为数组元素添加单引号	128
4.3.15 获取 foreach 循环索引	128
4.3.16 byte 操作	129
4.3.17 获取字符串中的数字	131
4.4 Windows 服务 (Windows Service)	131
4.4.1 Demo	131
4.4.2 调试服务	132
4.4.3 修改配置文件	132
4.4.4 读取配置文件	133
4.5 字符串格式化 (Format)	133
4.6 参数验证	134
4.6.1 代码契约 (Code Contract)	134
4.6.2 验证小时分钟	134
4.6.3 生成 GUID	135
4.7 代码生成 (Code Generate)	135
4.7.1 CSharpCodeProvider	135
4.7.2 CodeDOMProvider	136
4.8 配置	136
4.8.1 ini	136
4.8.2 自动注册 COM 组件	136
5 Web	139
5.0.1 URL 规范	141
5.0.2 Uri	141
5.0.3 CommonJS	142
5.1 通信	142

5.1.1 端口	142
5.1.2 Internet Information Service	142
5.1.3 经典模式 (Classic Model) 和集成模式 (Integrated Model)	145
5.1.4 Configuring an ASP.NET 4 Application to Auto-Start	145
5.1.5 CGI(Common Gateway Interface)	146
5.1.6 MIME	147
5.1.7 虚拟目录 (Virtual Directories)	149
5.2 HTTP	150
5.2.1 浏览器的请求流程	150
5.2.2 完整的 HTTP 请求流程	150
5.2.3 Http 请求在 IIS 中的处理流程	151
5.2.4 HttpRuntime	152
5.2.5 HttpContext	153
5.2.6 IHttpModule	154
5.2.7 IHttpHandler	154
5.2.8 HTTP 结构	154
5.2.9 HTTP 协议-压缩 (HTTP Protocol-Compress)	157
5.2.10 HTTP ETag	158
5.2.11 HTTP Proxy	159
5.2.12 HTTP Tunnel	160
5.2.13 REST 方式 POST、GET	161
5.2.14 HTTP Request and HTTP Response	163
5.2.15 HTTP 状态码 (HTTP Status Code)	164
5.2.16 获取客户端 IP	166
5.2.17 placeholder 属性	167
5.3 调试 (Debug)	167
5.3.1 Chrome Javascript 条件断点 (Condition Breakpoint)	168
5.3.2 SlidesJS 改变滑动间隔	168
5.3.3 IIS 6.0 中 MVC 项目部署	170
5.3.4 HTML	172

5.3.5 Ajax 跨域请求	172
5.3.6 文件上传	174
5.4 Cookie	177
5.4.1 编辑 Cookie	179
5.5 Session	180
5.5.1 Session 的优点与限制	181
5.5.2 Session 的生命周期	181
5.5.3 网页嵌入视频	182
5.6 缓存 (Cache)	182
5.6.1 Request 缓存相关首部字段	183
5.6.2 Response 缓存相关首部字段	185
5.6.3 System.Web.Caching	185
5.7 浏览器渲染 (Browser Render)	186
5.7.1 重绘 (Redraw) 与重排 (Reflow)	187
5.8 网页跳转 (Page Redirect)	189
5.8.1 Redirect	189
5.8.2 Meta Refresh	190
5.8.3 Server.Transfer	190
5.8.4 URL 锚点 (Fragment URLs)	191
5.9 技巧 (Little Tricks)	191
5.9.1 查看 Cookie 登录密码	191
5.9.2 刷新当前页面 (Refresh Current Page)	192
5.9.3 URL 特殊参数传递	194
5.9.4 前端技巧	196
5.9.5 监视 log	196
5.9.6 网站统计	196
5.9.7 页面统计 (Pageview Tracking)	196
5.9.8 断点续传 (Resume Broken Transfer)	196
5.9.9 ASP.NET 应用程序与页面生命周期	197
5.10 页面适应多屏	197

5.10.1 适应手机浏览	197
5.11 性能 (Performance)	198
5.11.1 网页优化 (Website Optimized)	198
5.11.2 利用反向代理服务器加速和保护应用	199
5.11.3 增加一个负载均衡器	199
5.11.4 缓存静态和动态内容	200
5.12 Electron	200
5.12.1 安装	200
5.12.2 第一个程序	201
5.12.3 打包	201
5.13 常见问题	201
5.13.1 Bad Request (Invalid Hostname)	203
6 Windows Form	205
6.1 资源文件 (Resource)	205
6.1.1 简介	205
6.1.2 资源文件的分类	205
6.1.3 利用资源文件来做多国语言版本	206
6.1.4 MouseDown 事件后无法触发 MouseDoubleClick	206
6.1.5 指定 DllImport 文件目录	206
6.1.6 窗口句柄	206
6.1.7 Windows 弹出窗体	207
6.2 常见问题	207
6.2.1 关于嵌入互操作类型 (Embed Interop Types)	207
6.2.2 Class not registered	208
6.2.3 试图加载格式不正确的程序	208
6.2.4 Dictionary 索引超出数组界限	208
6.2.5 System.ComponentModel.Design.ExceptionCollection	209
7 ASP.NET MVC(Apache License 2.0)	211
7.1 MVC 权限	213

7.1.1	MVC 登录验证	213
7.2	MVC 生命周期	214
7.2.1	ASP.NET MVC 的处理流程	214
7.2.2	ASP.NET 请求生命周期 (ASP.NET MVC Request Life Cycle)	215
7.2.3	路由忽略	215
7.2.4	Controller 的激活过程 (Process of Controller Activation)	217
7.3	MVC 页面传值	218
7.3.1	(View → Controller)	218
7.3.2	PageRouteHandler vs MvcRouteHandler	221
7.4	MVC 异常处理 (MVC Error Handling)	223
7.4.1	Application_Error	223
7.4.2	OnException 继承 HandleErrorAttribute	224
7.5	脚本压缩和合并	224
7.6	参数的传递	225
7.6.1	ActionResult	226
7.6.2	MVC 中 Razor 语法	226
7.6.3	ViewData、ViewBag、PartialView、TempData、ViewModel、Tuple	227
7.6.4	JsonRequestBehavior	228
7.6.5	过滤器 (Filter) 记录访问日志	228
7.7	部分视图	229
7.8	常见错误	229
7.8.1	找不到编译动态表达式所需的一种或多种类型。是否缺少引用?	229
7.8.2	指定的转换无效	230
7.8.3	未能加载文件或程序集 “RR.Web.CCN.Tour” 或它的某一个依赖项。试图 加载格式不正确的程序。	230
8	JavaScript	231
8.0.1	规范 (Specification)	233
8.0.2	进入页面时执行 JavaScript	233
8.0.3	一道 Javascript 试题	234
8.0.4	Javascript 日期	235

8.0.5 jQuery Rotate 插件实现旋转	235
8.0.6 JavaScript 调试	235
8.0.7 匹配手机号码	236
8.0.8 JavaScript 清空数组	236
8.0.9 数组是否有包含关系	236
8.0.10 XMLHttpRequest	237
8.0.11 获取元素文本	237
8.0.12 序列化 (Serialize) 与反序列化 (Deserialize)	238
8.1 Ajax	239
8.1.1 Ajax 提交原生写法	239
8.1.2 Ajax 提交	240
8.1.3 Ajax 原理和 XMLHttpRequest 对象	240
8.2 jQuery	241
8.2.1 jQuery 事件绑定	242
8.2.2 \$(function () {}) 的作用	242
8.2.3 CheckBox	243
8.2.4 jQuery 中 prop 与 attr 区别	244
8.2.5 NETWORK_ERR: XMLHttpRequest Exception 101	244
8.2.6 Ajax 失效的	245
9 HTML	247
9.0.1 规范	247
9.0.2 HTML5	247
9.1 a 标签	248
9.1.1 href="#"	248
9.1.2 Meta	248
10 CSS	251
10.1 规范	251
10.1.1 选择器优先级	253
10.1.2 标签	253

目录	15
10.1.3 水平居中	254
10.1.4 宽高自适应	254
10.1.5 坚直居中	255
10.1.6 CSS3 Media Queries	256
10.1.7 属性	256
10.1.8 title	256
10.1.9 css !important	257
10.2 AngularJS	257
10.3 线程	257
10.3.1 托管线程	257
10.4 扩展方法	257
11 数据库	259
11.1 SQLite	259
11.1.1 SQLite 连接	259
11.2 数据恢复	259
11.3 数据库还原	260
11.3.1 切换到单用户还原	260
11.3.2 活动监视器	260
11.3.3 数据泵 (Data Pump)	261
11.3.4 常用 SQL	262
11.3.5 函数 (Function)	264
11.3.6 存储过程 (Procedure)	265
11.4 mysql	265
11.4.1 常用查询	265
11.5 定时任务	265
11.6 刷新页面	266
12 安全 (Security)	267
12.1 Encrypt Algorithms	269
12.1.1 网站登录安全 (Website Login Security)	270

12.2 RESTful API 认证	270
12.2.1 Basic Authentication	271
12.2.2 Api Key + Security Key + Sign	271
12.2.3 Amazon 的 API 鉴权流程	271
12.3 数字签名 (Digital Signature)	273
12.3.1 概念	273
12.3.2 工作原理	273
12.3.3 SHA1 算法	273
12.3.4 生成签名	275
12.3.5 验证签名	276
12.4 Cookie 窃取 (Cookie Hijacking)	276
12.4.1 XSS 漏洞	276
12.4.2 跨站请求伪造—CSRF	276
12.4.3 Google Chrome Cookie 读取	276
12.5 Session 劫持 (Session Hijacking)	277
12.6 HTTPS	277
12.6.1 TLS 历史	277
12.7 密码哈希加盐 (Salted Password Hashing)	278
12.7.1 Base64 编码	279
12.8 SQL 注入 (SQL Injection)	280
12.8.1 参数化 SQL 语句	280
12.8.2 Sqlmap	280
12.9 Nmap(GNU General Public License)	280
12.9.1 主机发现 (Host Discovery)	280
12.9.2 端口扫描 (Port Scanning)	281
12.9.3 版本侦测 (Version Detection)	283
12.10 AppScan	283
12.10.1 操作系统侦测 (Operating System Detection)	283
12.11 OpenVPN	283
12.11.1 OpenVPN 配置 (OpenVPN Configuration)	283

目录	17
12.11.2 添加新客户端 (Add New Client)	284
12.12 Windows 自带 VPN	285
12.12.1 VPN 类型	285
12.12.2 常见错误	288
12.13 DIRB	289
12.14 Network	289
12.14.1 Ubuntu 使用 aircrack 破解 WiFi	289
13 Node.js	291
13.0.1 Introduce	293
13.0.2 Install	293
13.0.3 基础的 HTTP 服务器	293
13.0.4 独立出服务端的模块	294
13.0.5 服务端路由	295
13.0.6 NodeJS RESTful Service	295
13.0.7 NodeJS connect SQL Server	296
13.0.8 配置文件管理	298
13.0.9 Debug	299
13.0.10 exports	299
13.0.11 处理 POST 请求	299
13.0.12 Express	299
13.0.13 MEAN	301
13.1 常用脚本	301
13.1.1 BAT	301
14 异常 (Exception)	303
14.1 Winows SEH	305
14.1.1 Don't ever swallow exceptions	306
14.1.2 Log Exception.ToString(); never log only Exception.Message	306
14.1.3 异常列表 (Exception List)	306
14.1.4 Windows Form 中统一异常处理	306

14.1.5 Windows Service Error Handling	307
14.1.6 DllNotFoundException	308
14.2 静态文件自动化管理	308
15 网络 (Network)	309
15.0.1 RAML	309
15.0.2 RestSharp	309
15.1 TCP	310
15.1.1 Ethernet v2	310
15.1.2 传输层 (Transport Layer) 的由来	310
15.1.3 什么是 Socket	311
15.1.4 三次握手	311
15.1.5 子网掩码	312
15.1.6 TCP 流量控制	312
15.1.7 通讯协议分析	312
15.1.8 查看网络状态	312
15.2 X11	313
15.3 即时通信	313
15.3.1 title	313
15.4 RPC	314
15.4.1 RPC(远程过程调用) 是什么	314
15.4.2 远程过程调用发展历程	314
16 .NET	317
16.1 委托 (Delegate)	319
16.2 泛型 (Generic) 与集合 (Collection)	319
16.2.1 List 与深拷贝	319
16.3 Microsoft Build Engine(MIT License)	320
16.4 Windows dll 装载过程	321
16.5 PE 文件	323
16.5.1 .reloc	323

16.5.2 Dos stub	325
16.5.3 NT Header	326
16.5.4 Section	328
16.6 CLR	328
16.6.1 PE Loader 载入 PE 文件	328
16.6.2 启动 CLR 服务	329
16.6.3 进入 Main 方法	332
16.7 Metadata	332
16.8 JIT(MIT Licence)	333
16.8.1 类型构造器 (Class Constructor)	334
16.9 FCL(MIT Licence)	334
16.10 GC(Garbage Collection)	334
16.10.1 为什么要 GC	334
16.10.2 对象分配过程	335
16.10.3 GC 回收原理	335
16.10.4 GC 触发时机	337
16.10.5 LOH 堆	337
16.11 IL(Intermediate Language)	337
16.11.1 简单的 IL 示例	339
16.11.2 属性 (Attribute)	340
16.11.3 编辑 DLL(Edit Dynamic Linkable Library)	340
16.12 调试 (Debug)	341
16.12.1 调试器工作原理	342
16.12.2 符号文件 (Symbol File)	342
16.12.3 符号文件服务器	342
16.12.4 常用调试技巧	343
16.12.5 不能设置断点的检查步骤	345
16.12.6 定位内存偏高、内存泄漏 (Memory Leak)	347
16.12.7 定位线程问题 (CPU 过高, 线程死锁)	347
16.12.8 PDB 文件	347

16.12.9 调试第三方 Dll	348
16.13 Visual Studio	348
16.13.1 Shared Source Common Language Infrastructure 2.0	348
16.13.2 Visual Studio 快捷键	348
16.13.3 Visual Studio 如何查找源文件	349
16.13.4 重置 Visual Studio	349
16.13.5 Visual Studio 显示行号	349
16.13.6 低版本打开高版本工程	349
16.13.7 使用后期生成事件	349
16.13.8 解决方案文件夹	350
16.13.9 插件	350
16.13.10 Visual Studio 加载外部工具	350
16.13.1 Configure Visual Studio 2013 for debugging .NET framework	351
16.13.12 Visual Studio 查看 Call Stack	352
16.14 发布 (Publish)	352
16.14.1 Visual Studio 中采用 FTP 发布	354
16.14.2 Browser Link	354
16.14.3 Shadow Copy	354
16.14.4 技巧 (Little Tricks)	355
16.14.5 Visual Studio 生成操作	356
16.14.6 字符串比较 (String Compare)	356
16.14.7 .NET 索引器	357
16.15 正则表达式 (Regular Expression)	357
16.15.1 元字符 (Metacharacter)	357
16.15.2 常用正则表达式	357
16.16 配置管理 (Configuration Management)	358
16.16.1 配置读取	358
16.16.2 针对不同发布环境调整配置 (Self-adaption Configuration)	358
16.17 Inversion of Control	360
16.17.1 Spring.NET	360

16.17.2 Spring.NET 配置独立的 Object.xml	362
16.17.3 Castle Windsor(Apache License Version 2.0)	363
16.17.4 Autofac	364
16.17.5 命令行开发.NET	364
16.18 AOP(Aspect-Oriented Programming)	365
16.18.1 AOP 验证登陆	365
16.18.2 AOP 验证权限	365
16.18.3 AOP 记录日志	365
16.18.4 AOP 事务处理	365
16.18.5 iMicros 辅助网页调试	365
16.19 异步 (Async)	366
16.19.1 使用 IAsyncResult 对象的异步操作	366
16.19.2 Task	366
16.19.3 async/await 特性	366
16.20 多线程	366
16.20.1 lock	366
16.20.2 线程简介	367
16.20.3 Timer	367
16.20.4 线程间操作无效: 从不是创建控件的线程访问它。	368
16.20.5 Lambda 表达式和委托 (delegate)	370
16.20.6 System.ObjectDisposedException	371
16.20.7 线程退出码 (Exit Code)	371
16.21 Windows 32 接口编程	372
16.21.1 自动注册 COM 控件	372
16.22 插件 (Plug-In)	373
16.22.1 定义	373
16.22.2 插件的优缺点	373
16.22.3 插件的实现	374
16.23 .NET Core	374
16.24 常见问题	374

17 基础引擎 (Fundation Engine)	375
17.1 HTTP 服务	375
17.2 解释器和编译器 (Compiler)	376
17.3 数据库 (Database)	376
17.4 操作系统 (Operation System)	376
18 Android	377
18.0.1 环境搭建	377
18.1 常见问题	377
18.1.1 The import android.support cannot be resolved	377
18.2 Android SDK	377
18.2.1 离线安装	377
18.3 PhoneGap	378
18.3.1 PhoneGap 安装	378
18.3.2 Cordova 安装	378
18.4 React Native for Android	379
18.4.1 环境搭建	379
19 Windows SDK	381
20 SOA	383
20.1 SOA 简介	383
20.2 微服务 (Microservices)	384
20.2.1 概念	384
20.2.2 用户体验适配层 (Backend For Frontend)	384
21 Java	385
21.1 JAX-WS(Java™ API for XML Web Services)	385
21.1.1 Google 调试 WebSocket	385
21.2 MyEclipse	385
21.2.1 常用操作	385
21.3 Maven	385

目录	23
21.4 Apache MINA(Multipurpose Infrastructure for Network Applications)	386
21.5 Apache Kafka	387
21.5.1 Apache Kafka 安装	387
21.5.2 安装 JDK	387
21.6 Apache Tomcat	388
21.6.1 常识 (Common Sense)	388
22 Python	389
22.1 框架	390
22.1.1 Django	390
22.1.2 paramiko	390
22.1.3 paramiko 安装	390
22.2 工具	391
22.2.1 编码风格	391
22.2.2 Python Tools for Visual Studio	391
22.2.3 PyCharm	391
22.3 初级任务	391
22.3.1 Python 进制转换	391
22.3.2 使用 Python 下载文件	392
22.3.3 Python 引入目录下文件	393
22.3.4 python xml 解析	393
22.3.5 Python 记录日志	393
22.3.6 试用配置文件定义日志	394
22.4 常见问题	394
22.4.1 TabError: inconsistent use of tabs and spaces in indentation	394
23 Linux	395
23.1 shell	395
23.1.1 风格	395
23.1.2 常用 Shell 脚本	395
23.1.3 shell 日志处理	397

23.2 基础	397
23.2.1 常用命令	397
23.2.2 日志查看	398
23.2.3 网络	398
23.2.4 tcpdump	400
23.2.5 vi	402
23.2.6 Vim	403
23.2.7 grep	403
23.2.8 anacron	404
23.2.9 cron 执行定时任务	404
23.2.10 CentOS 安装 PHP 5.4	405
23.3 Angile	405
23.3.1 Scrum	405
23.3.2 Redmine 慢	406
23.4 Code Review	406
23.4.1 Tools	406
23.4.2 Install Review Board	406

II 公共组件 409

24 日志消息组件 413	
24.1 Apache Kafka	414
24.2 log4net	414
24.2.1 创建日志表	414
24.2.2 log4net 配置 (RollingFileAppender)	414
24.2.3 log4net.config(ADONetAppender)	415
24.2.4 初始化配置文件	418
24.2.5 log 文件名累加	419
24.2.6 log4net 自定义字段	420
24.2.7 log4net 日志写入界面	421
24.2.8 Trace 和 Debug	423

目录	25
----	----

24.2.9 多个文件记录日志	423
24.2.10 统一记录日志	424
24.2.11 记录建议 (Logging Advisor)	424
24.2.12 远程 Tail 日志	424
24.2.13 常见问题	424
24.3 Opserver	424
24.3.1 title	425
24.4 ELMAH	425
24.5 NLog	425
24.6 log.io	425
24.7 ELK	425
24.7.1 logstash 简介	425
25 集成组件	429
25.1 SVN	430
25.1.1 SVN Revert	430
25.1.2 svn cleanup failed—previous operation has not finished; run cleanup if it was interrupted	430
25.1.3 SVN 文件重命名冲突	431
25.1.4 tags、trunk、branches	431
25.1.5 switch、relocate	431
25.1.6 SVN 锁	432
25.1.7 SVN 版本库镜像	432
25.1.8 备份库	432
25.1.9 SVN 自动更新	432
25.1.10 设置忽略列表 (Ignore List Setting)	434
25.1.11 SVN 提交规则 (SVN Commit Rule)	434
25.1.12 SVN 多个.svn 隐藏目录	435
25.1.13 diff 和 patch	435
25.2 Git	436
25.2.1 基本使用	436

25.2.2 推送到 GitHub	438
25.2.3 Git 配置	439
25.2.4 添加忽略列表 (Add Ignore List)	440
25.2.5 常见问题	440
25.3 StyleCop.exe	440
25.4 CruiseControl.Net	441
25.4.1 自动获取源代码	441
25.4.2 自动 Build	442
25.4.3 自动化回归测试	442
25.4.4 自动部署	443
25.4.5 邮件提醒	443
25.4.6 Dashboard Configuration	444
25.4.7 常见问题处理	444
25.5 Jenkins	445
25.5.1 插件安装 (Install Plug-in)	445
25.5.2 创建 Job	446
25.5.3 邮件通知 (E-Mail Notification)	449
25.5.4 自动打包	449
25.5.5 Jenkins 执行批处理命令	449
25.6 Travis CI	450
25.7 自动化脚本	450
25.7.1 生成更新文件	450
25.8 Gradle	450
25.8.1 安装	450
25.9 JIRA	450
25.9.1 JIRA 部署	451
25.10 Ansible	452
26 测试调试组件	453
26.1 WinDbg	454
26.1.1 基本使用	454

26.1.2 dump 文件	455
26.1.3 获取 dump 文件	455
26.1.4 调试 dump 文件	455
26.1.5 WinDbg 加载 SOS	455
26.1.6 symstore	456
26.2 Fiddler(Free,Not Open Source)	456
26.2.1 Fiddler Capture	457
26.2.2 Fiddler 捕获 HTTPS	458
26.2.3 Fiddler 应对 HSTS	458
26.2.4 Fiddler 移动端 Capture	460
26.2.5 Fiddler 测试 API	461
26.2.6 图标含义	461
26.2.7 配置过滤规则 (Configurate Filter Rule)	462
26.2.8 Composer	462
26.2.9 Statistics	463
26.2.10 Inspectors	463
26.2.11 断点调试 (Breakpoint Debugging)	464
26.3 NUnit (MIT License)	464
26.3.1 NUnit 在 .NET.Framework 4.0 调试	467
26.3.2 使用总结	468
26.4 xUnit	468
26.4.1 使用 xUnit	468
26.4.2 常用功能	469
26.5 MiniProfiler	469
26.5.1 安装	469
26.5.2 使用	470
26.6 dotTrace	471
26.6.1 dotTrace 分析 ASP.NET MVC 程序性能	471
26.7 CLR Profiler	471
26.8 JustDecompile	471

26.9 DebugView.exe	471
26.10 Selenium	471
26.11 JMeter	471
26.12 PerfView	471
26.13 SoapUI(Apahce Licence 2.0)	472
26.14 Wireshark(GNU General Public License)	472
26.14.1 常用过滤语句	472
26.14.2 各层报文理解	474
26.14.3 删除 Filter	475
26.14.4 抓取本机包	477
26.14.5 常见问题	477
27 辅助组件	479
27.1 Linux 有关工具	480
27.1.1 Putty	480
27.1.2 Vi	480
27.2 GMap.NET	480
27.3 DotNetBar(Commercial)	480
27.3.1 Introduce	480
27.3.2 加载 DotNetBar 控件到 Visual Studio 工具箱	480
27.4 ildasm.exe(MSIL Disassembler)	480
27.5 Source Insight(Commercial)	481
27.5.1 特性 (Feature)	481
27.6 Inno Setup	482
27.6.1 支持中文的安装界面	482
27.6.2 添加自定义界面	482
27.6.3 启用向导界面	483
27.6.4 自动打包.NET Framework 安装程序	483
27.7 Graphviz	483
27.7.1 中文乱码	484
27.7.2 箭头指向节点位置	485

目录	29
----	----

27.8 gVim	487
27.8.1 乱码	487
27.9 Atom	487
27.9.1 显示中文	487
27.10 Tex	487
27.10.1 Install	487
27.10.2 Tex 中注释代码	488
27.11 Rsync	489
27.12 Docbook	489
27.12.1 Docbook 安装	489
27.12.2 编写 Docbook 文档	491
27.12.3 将文档转换成 HTML 格式	491
27.13 PowerDesigner(Commercial)	492
27.13.1 显示注释	492
27.13.2 设置主键	492
27.13.3 设置默认值	492
27.14 WebStorm	494
27.14.1 配置 NodeJS	494
27.15 curl(CommandLine Uniform Resource Locator)	494
27.15.1 curl for HTTP	495
27.16 My Generation	495
27.17 Brackets(MIT License)	495
27.17.1 简介	495
27.17.2 常见问题	496
27.18 FileZilla	496
27.18.1 设置默认目录	496
27.19 Yeoman	496
27.19.1 简介	496
27.20 BrowserSync (MIT Licence)	497
27.20.1 简介	497

27.20.2 安装	497
27.20.3 使用	497
27.21 ReSharper(Commercial)	498
27.21.1 设置 (Settings)	499
27.21.2 快捷操作	499
27.21.3 Navigation and Search	501
27.21.4 Code analysis	501
27.22 Reflexil	502
27.23 Gulp	502
27.23.1 简介	502
27.23.2 安装	502
27.24 weinre	502
27.24.1 Install	502
27.24.2 启动	502
27.25 Shadowsocks	503
27.25.1 简介	503
27.25.2 使用	503
27.26 NuGet	503
27.26.1 打开命令行界面	503
27.27 Quartz.NET	504
27.27.1 定时任务配置	504
27.27.2 app.config	504
27.27.3 quartz.config	505
27.27.4 quartz_jobs.xml	506
27.27.5 实现 IJob 接口	507
27.27.6 启动作业	508
27.28 ThunderBird	508
27.29 PuTTY	508
27.29.1 PuTTY 保存密码	508
27.29.2 psftp	508

目录	31
27.29.3 putty 退出全屏	509
 28 科学上网	511
28.1 Lantern	511
28.1.1 How does Lantern work	511
28.2 Shadowsocks(影梭)	511
 III 设计	513
 29 字体	515
29.1 字体 (Font)	515
29.1.1 Museo Sans	515
29.1.2 Avenir Web	515
 IV 待移除模块	517
29.2 配置	520
29.2.1 app.config	520
29.2.2 定时任务	520
29.2.3 照片打印	531
29.2.4 获取机器 Mac 地址	534
 V 附录	537
 30 词汇	541
30.1 中英词汇对照	542
30.1.1 业务词汇对照	542
30.1.2 计算机词汇中英对照	545
30.2 常用术语	546
30.3 编程语言漫谈	548
 31 经验汇总	551
31.1 Windows 搜索	551

31.1.1 自动属性	551
31.1.2 批处理脚本启动服务	551
31.1.3 Windows 脚本 - % dp0 的含义	556
31.2 AppDomain	557
31.2.1 AppDomain VS 进程	557
31.2.2 AppDomain VS Assembly	558
31.2.3 AppDomain VS 对象	558
31.3 进程间通信	558
31.3.1 程序间参数传递	558
31.4 设计模式 (Design Pattern)	559
31.4.1 创建型模式	559
31.4.2 结构型模式	561
31.4.3 行为型模式	561
31.4.4 表驱动法 (Table-Driven Approach)	561
31.5 逆向工程 (Reverse Engineering)	561
31.5.1 radare2	561
31.5.2 反混淆 dll(Deobfuscation dll)	561
31.6 开发技巧	562
31.6.1 搜索技巧 (Search Tricks)	562
31.6.2 合理使用 Using 子句	565
31.6.3 使用 as 转换对象	565
31.6.4 枚举转换为字符串	565
31.6.5 字符串拼接	566
31.6.6 得到每天的开始时间结束时间	566
31.6.7 代码优化 (Code Optimized)	566
31.6.8 表驱动法	571
31.7 常用资源	571
31.7.1 常用网站搜藏	571
31.7.2 资源地址搜藏	572
31.7.3 LDAP(Lightweight Directory Access Protocol)	572

目录	33
----	----

32 Open Source	575
32.1 参与开源项目	575
32.1.1 Pull Request	575
33 FAQ	577
33.1 技术类	578
33.1.1 抽象类和接口的区别	578
33.1.2 POST 和 GET 的区别	578
33.1.3 == 和 equals() 的区别	579
33.1.4 Request.Params[]、Request[]、Request.Form[] 和 Request.QueryString[] 的区别	581
33.1.5 HTTP Cookie 和 Session 的区别	582
33.1.6 string.Empty、"" 和 null 的区别	582
33.1.7 SQL Server 中 exec 和 sp_executesql 的区别	584
33.1.8 Local system/Network service/Local Service 的区别	584
33.1.9 Margin 和 Padding 的区别	584
33.1.10 absolute 与 relative 区别	585
33.1.11 FTP 主动模式 (Active FTP) 和被动模式 (Passive FTP) 区别	585
33.1.12 const,readonly,static,readonly 的区别	586
33.1.13 string 是引用类型还是值类型	586
33.1.14 var,object,dynamic 的区别	587
33.1.15 typeof 和 GetType() 的区别	587
33.1.16 JavaScript 中 undefined 与 null 的区别	588
33.1.17 for/foreach/LINQ 的区别	588
33.1.18 装箱 (Box) 和拆箱 (Unbox) 的区别	589
33.1.19 程序集的加载机制	590
33.1.20 CSS 文字大小单位 px、em 的区别	590
33.1.21 为什么要使用 base64 编码	590
33.1.22 StringBuilder	591
33.1.23 C# 中 string 和 String 有什么区别	591
33.1.24 IIS 何时会重启	591

33.1.25 ASP.NET 网站第一次访问速度慢原因	592
33.1.26 ADO 和 ADO.NET 的区别	593
33.1.27 Javascript 如何判断是否是对象	594
33.1.28 什么是委托？什么是事件？为什么要用委托？	595
33.1.29 localhost、127.0.0.1、本机 IP 的区别	595
33.1.30 Struct 和 Class 的区别	596
33.1.31 Javascript 中 Window 和 window 的区别	596
33.1.32 BOM 和 DOM 的区别	596
33.1.33 internal 和 protect internal	597
33.1.34 IEnumerable 和 string[] 的区别	597
33.1.35 foreach 和 List<T>.Foreach 的区别	597
33.1.36 HashTable Hashmap 区别	598
33.1.37 3 种主流 Web 开发架构	598
33.1.38 ViewBag 和 ViewData 的区别	598
33.1.39 常用数据结构及复杂度	599
33.1.40 进程 (Process) 与线程 (Thread) 的区别	599
33.1.41 Form 表单中 onclick/submit/onsubmit 的关系	601
33.1.42 Model, ORM, DAO 和 Active Record 的区别	601
33.1.43 TCP 长连接与短连接的区别	602
33.1.44 HTTP 的长连接和短连接	602
33.1.45 为什么使用强名称 (强命名程序集)	603
33.1.46 Invoke 与 BeginInvoke 的区别	603
33.1.47 Activex、OLE、COM、OCX、DLL 之间的区别	604
33.1.48 Hadoop 和 Spark 区别	605
33.1.49 B 树、B-树、B+ 树、B* 树都是什么	607
33.1.50 ocx 是什么	609
33.1.51 什么是面向对象？为什么要面向对象？	609
33.1.52 Why is Dictionary preferred over HashTable	610
33.1.53 dll 和 COM 的区别	610
33.1.54 Func<T, TResult> 委托的由来和调用和好处	611

33.1.55 为什么要使用 LINQ 技术	611
33.1.56 c# 引用类型与值类型的区别	613
33.2 非技术类	613
33.2.1 Idea	614

Part I

整体设计

对计算机的访问（以及任何可能帮助你认识我们这个世界的事物）应该是不受限制的、完全的，任何人都有动手尝试的权利！

黑客伦理

技术只是解决问题的选择，而不是解决问题的根本。

Great engineers produce designs that are robust, intuitive, extensible, flexible, maintainable, operable, scalable, and efficient. In doing so, they strive to achieve a balance between quality and speed of execution.

计算机科学有两个艰难的事：1. 缓存失效；2. 命名。There are only two hard things in Computer Science: cache invalidation and naming things.— Phil Karlton

”我们的社会正越来越依赖计算机，我们使用的软件对保证未来社会的自由至关重要。自由软件使我们能够控制我们使用的技术，让技术造福个人和社会，而不是让技术被商业公司或政府控制，用来限制或监视我们。”

(As our society grows more dependent on computers, the software we run is of critical importance to securing the future of a free society. Free software is about having control over the technology we use in our homes, schools and businesses, where computers work for our individual and communal benefit, not for proprietary software companies or governments who might seek to restrict and monitor us.)

Chapter 1

规范准则

集群、负载均衡、排队、分库分表、锁、缓存。编程语言虽然该学，但是学习计算机算法和理论更重要，因为计算机算法和理论更重要，因为计算机语言和开发平台日新月异，但万变不离其宗的是那些算法和理论，例如数据结构、算法、编译原理、计算机体系结构、关系型数据库原理等等。

1.1 程序设计原则

1.1.1 编程经验

所有事情所花费的时间总是比你预期的要长 即使一切进展顺利，我们也很难对功能所需的时间做出正确的预算。并且，开发软件时碰到各种意想不到的问题是非常常见的。一个简单的合并操作会导致一系列小 bug，一次框架升级意味着一些函数必须改变或者一些 API 不按照你想象的那样工作。

Hofstadter Law (霍夫施塔特定律) 其实道出了真谛：做事所花费的时间总是比你预期的要长，即使你在预期中已经考虑了 Hofstadter Law (霍夫施塔特定律)。

从小事做起，然后再扩展 无论是创建一个新的系统，还是在现有的系统中添加新的功能，我总是从一个简单到几乎没有功能的版本开始，然后再一步一步地解决问题，直到满意为止。我从来没有妄想过能够一步登天。相反，我一边开发一边学习，同时新掌握的信息还可以用于解决方案中。我很喜欢 John Gall 的这句话：“复杂系统总是源于简单系统的演化。”

代码级别 可编译、可运行、可测试、可读、可维护、可重用。

1.1.2 程序语言喜好统计

图1.1是 Hacker News 对编程语言的喜爱做一个统计。

1.1.3 Architectural Requirements

- Easy to separate → Autonomy(自治，自治权)
- Easy to understand → Understandability
- Easy to extend → Extensibility
- Easy to change → Changeability
- Easy to replace → Replaceability
- Easy to deploy → Deployability
- Easy to scale → Scalability
- Easy to recover → Resilience(恢复力、弹力、顺应力)
- Easy to connect → Uniform Interface
- Easy to afford → Cost-efficiency(for develop&operation)

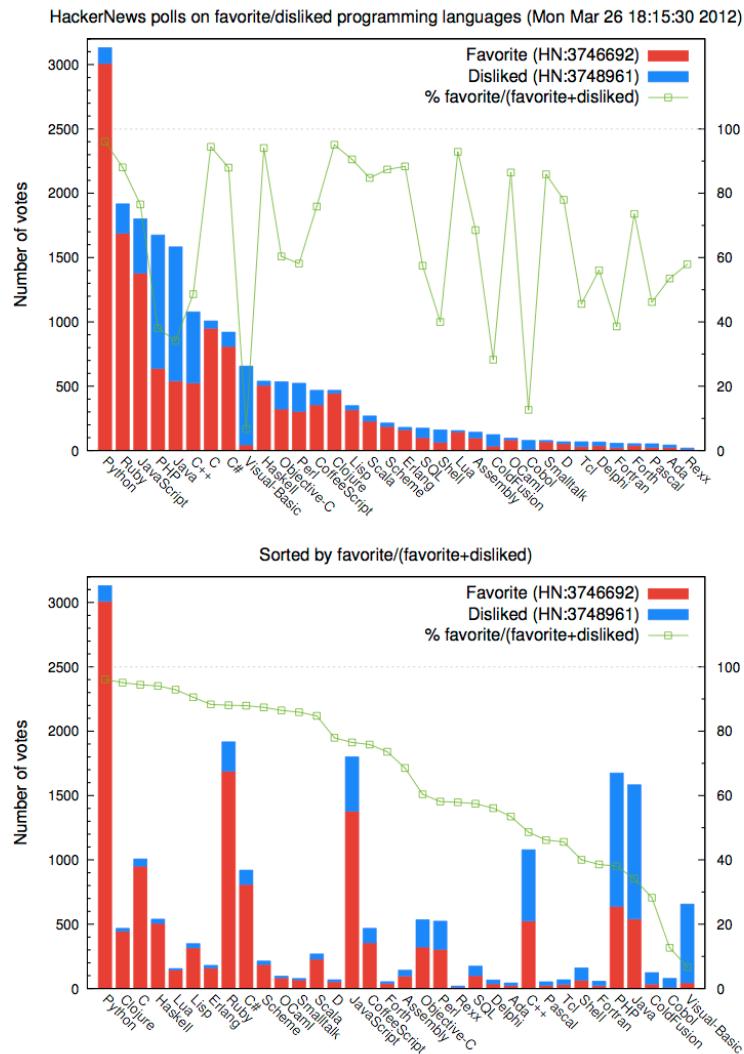


图 1.1: 对编程语言的喜好

Reliability/Fault tolerance/Reusability/Manageability/Monitorability/ Maintainability/Extensibility/Understandability/Auditability/ Performance/Availability/Scalability/Security

1.1.4 不要自我重复 (DRY - Don't Repeat Yourself)

这也许是在编程开发中最最基本的一个信条，就是要告诉你不要出现重复的代码。我们很多的编程结构之所以存在，就是为了帮助我们消除重复（例如，循环语句，函数，类，等等）。一旦程序里开始有重复现象的出现（例如很长的表达式、一大堆的语句，但都是为了表达相同的概念），你就需要对代码进行一次新的提炼，抽象。一次且仅一次（once and only once，简称 OAOO）又称为 Don't repeat yourself（不要重复你自己，简称 DRY）或一个规则，实现一次（one rule, one place）是面向对象编程中的基本原则，程序员的行事准则。旨在软件开发中，减少重复的信息。DRY 的原则是：系统中的每一部分，都必须有一个单一的、明确的、权威的代表：指的是（由人编写而非机器生成的）代码和测试所构成的系统，必须能够表达所应表达的内容，但是不能含有任何重复代码。当 DRY 原则被成功应用时，一个系统中任何单个元素的修改都不需要与其逻辑无关的其他元素发生改变。此外，与之逻辑上相关的其他元素的变化均为可预见的、均匀的，并如此保持同步。软件重复出现至少会导致以下问题：

- 其中的一个版本会过期
- 代码的责任会四处散开，导致代码难以理解
- 当你修改代码时，需要重复修改很多地方，一不小心就会遗漏
- 你不能很好地进行性能优化

1.1.5 单一职责原则 (Single Responsibility Principle,SRP)

单一职责原则¹是指一个代码组件（例如类或函数）应该只执行单一的预设的任务。

1.1.6 开放/封闭原则 (Open Closed Principle)

程序里的实体项（类，模块，函数等）应该对扩展行为开放，对修改行为关闭（Open for extension, closed for modification）。换句话说，不要写允许别人修改的类，应该写能让人们扩展的类。

1.1.7 保持简单 (Keep It Simple)

保持简单²简单化（避免复杂）永远都应该是你的头等目标。简单的程序让你写起来容易，产生的 bug 更少，更容易维护修改。

¹http://en.wikipedia.org/wiki/Single_responsibility_principle

²http://en.wikipedia.org/wiki/KISS_principle

1.1.8 用最简单的方法让程序跑起来

在开发时有个非常好的问题你需要问问自己，“怎样才能最简单的让程序跑起来？”这能帮助我们在设计时让程序保持简单。

1.1.9 避免过早优化

只有当你的程序没有其它问题，只是比你预期的要慢时，你才能去考虑优化工作。只有当其它工作都做完后，你才能考虑优化问题，而且你只应该依据经验做法来优化。

Premature optimization is the root of all evil!

“对于小幅度的性能改进都不该考虑，要优化就应该是 97% 的性能提升：过早优化是一切罪恶的根源” —Donald Knuth。编程的一个最核心的目的是解决问题，满足用户的需求。先保证解决方案的正确性（确实正确地解决了问题），再考虑其他一些特性，比如易操作性、运行速度、界面友好等等，经过适当的迭代一般在这些方面就能越来越好。但是，解决方案的正确性却不一定慢慢「越来越正确」的，比如用户需要监测一个金融市场指标是否发生了分布性质的突变，从而根据突变的情况做出不同的决策，结果一个研究小组开发出一套做监测的系统过早地做了优化，计算速度飞快、用户体验也很好，但是由于研究不到位，使用的基础统计检验整个就是错误的，由此经过大量性能优化的算法基本也都是白费了。

因此，为了减少无用功，提高一定时间内的有效生产力，先保证解决方案在原理上是正确的，可以解决问题，可以满足需求，再考虑更高层次的要求，例如性能等等。

1.1.10 依赖倒置原则 (Dependence Inversion Principle)

- A. 高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。
- B. 抽象不应该依赖于具体，具体应该依赖于抽象。

1.1.11 接口隔离原则 (Interface Segregation Principle)

使用多个专门的接口比使用单一的总接口要好。

一个类对另外一个类的依赖性应当是建立在最小的接口上的。

一个接口代表一个角色，不应当将不同的角色都交给一个接口。没有关系的接口合并在一起，形成一个臃肿的大接口，这是对角色和接口的污染。“不应该强迫客户依赖于它们不用的方法。接口属于客户，不属于它所在的类层次结构。”这个说得很明白了，再通俗点说，不要强迫客户使用它们不用的方法，如果强迫用户使用它们不使用的方法，那么这些客户就会面临由于这些不使用的方法的改变所带来的改变。

The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use. ISP splits interfaces which are very large

into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces. ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of Object-Oriented Design, similar to the High Cohesion Principle of GRASP.

好的接口定义应该是具有专一功能性的，而不是多功能的，否则造成接口污染。如果一个类只是实现了这个接口中的一个功能，而不得不去实现接口中的其他方法，就叫接口污染。

1.1.12 关注点分离 (Separation of Concerns)

好的架构设计必须把变化点错落有致地封装到软件系统的不同部分，为此，必须进行关注点分离。Ivar Jacobson 在《AOSD 中文版》中写道：

好的架构必须使每个关注点相互分离，也就是说系统中的一部分发生了改变，不会影响其他部分。即使需要改变，也能够清晰地识别出哪些部分需要改变。如果需要扩展架构，影响将会最小化。已经可以工作的每个部分都将继续工作。

1.1.13 封装变化点

1.1.14 不要开发你目前用不到的功能 (You Ain't Gonna Need It, YAGNI 原则)

YAGNI 原则指的是只需要将应用程序必需的功能包含进来，而不要试图添加任何其他你认为可能需要的功能。

1.1.15 合成/聚合复用原则(Composite/Aggregate Reuse Principle, CARP)

尽量使用合成/聚合，尽量不要使用继承。

1.1.16 依赖倒置 (Dependency Inversion Principle)

DIP 的两大原则：

1、高层模块不应该依赖于低层模块，二者都应该依赖于抽象。

2、抽象不应该依赖于细节，细节应该依赖于抽象。

具体来讲，依赖倒置的核心思想是针对接口而不是实现编程。应用 DIP 可以降低模块之间的耦合度，只要接口保持稳定，模块可以独立演化而互不影响。

1.1.17 读写分离

OneProxy 是平民软件完全自主开发的分布式数据访问层，帮助用户在 MySQL/PostgreSQL 集群上快速搭建支持分库分表的分布式数据库中间件，也是一款具有 SQL 白名单（防 SQL 注入）及 IP 白名单功能的 SQL 防火墙软件。采用与 MySQL Proxy 一致的反向协议输出模式，对应用非常简单和透明易用，让用户畏惧的分库分表（Horizontal Partitioning）工作变得极其简单可控！基于 Libevent 机制实现，单个实例可以实现 25 万的 SQL 转发能力，用一个 OneProxy 节点可以带动整个 MySQL 集群。

互联网企业在开源软件的基础之上打造了足以支撑每天十亿笔交易、支付处理能力的在线系统，其中分布式架构在其中发挥了关键的作用。在架构里面又分为应用层架构和数据层架构，应用层架构着重于解决应用之间的远程调用、服务的发现、消息的流转，数据层架构着重于解决底层数据库、缓存等的透明扩展。关系数据库的水平扩展，或者说 MySQL 数据库的水平扩展能力是架构中的关键所在，OneProxy 是一款构建在 MySQL 数据库之上的透明数据访问中间件，由在互联网行业（电商、互联网金融）工作多年的资深架构师精心打造，旨在降低对上层应用开发要求，实现对应用基本透明的底层数据库单点切换（Failover）、读写分离、水平拆分功能，使任何公司都可以轻松地拥有与大型互联网企业一样底层数据库扩展能力，也不用受制于开发语言的限制，以帮助业务实现再一次腾飞。

1.1.18 动静分离

1.1.19 数据库表拆分 & 分片

1.1.20 主从同步

1.1.21 负载均衡

1.1.22 去中心化

1.1.23 主从分离

1.1.24 服务隔离

1.1.25 灰度部署

避免上线导致大规模应用故障。

1.1.26 异步化

减少系统之间的强依赖，避免一个服务无法使用整个系统都无法使用的问题。

1.1.27 统一监控 & 统一日志

1.1.28 技术选型

淘宝的黄裳讲解淘宝网站架构发展的时候，说起 2004 年底淘宝为何从 PHP 向 Java 转移的事情。为何转换，他阐述了几个理由，其中一个是非常有趣的：当时的 PHP 缺少一个 IDE。而合适的 IDE 能够有效提升规模化软件开发的效率。

我们知道 eBay 在 2002 年的时候也在 Sun 技术团队的帮助下，将整个应用架构从 C++ 迁移到 J2EE。也就是 eBay 内部所说的 V3 版本。

选 Windows 和选 Linux 有没有哪个更好，我想说的是，都还不错，不要以为选 Windows 就不好（StackOverflow 就是.NET 平台）。

很多金融机构都是用 Windows 的平台（你可能会和我争吵国内的银行都不是 Windows 的平台，都是 Unix 的平台，是的，我也是在银行里做过的，中国的银行几乎都是 IBM/SUN/ORACLE 的领地，所以，那里都是 AIX、RISC600，Solaris，Java，C/C++ 的地方），但是国外很多金融机构却更多用的是 Windows。

以前淘宝架构师黄裳说过，或许是 PHP 缺少一个 IDE；) 当然实际上不是这样的，当时的 PHP 远没有现在成熟。比如，PHP 缺少中间件（或许现在也是），而 Java 当时起码有 Weblogic /Jboss 可用。至于解决方案，Java 相对更为成熟，从人的角度来说，Java 企业级应用背景的人才更多一些，再说可以得到 Sun 中国技术团队的协助。应该是 2004 年底的技术决定吧，同时间段迁移的还有从 MySQL 迁移到 Oracle，从 PC 服务器迁移到小型机。这是一个正确的决定。

其实当时最早更换的不是 PHP 语言，而是 MySQL，MySQL 当年太弱了，读写性能问题严重，容易死锁。而阿里当时在 Oracle 方面的积累非常强大，有冯春培、汪海、Fenng 这样的人物（fenng 比他们稍微晚点来，来的时候已经换了），于是数据库就迁移到了 Oracle。PHP 不具备连接池的功能，一开始弄了一个 sqlRelay 中间件来充当连接池，结果这个东东也经常死掉，于是开发语言也必须跟着换，至于为什么换 java，fenng 已经说的很清楚了。当然现在的 MySQL 已经不是当年的吴下阿蒙了，现在用 LAMP 架构的话应该能支持很高的流量。

轮回罢了。当业务快速发展的時候必须靠 DB 的能力顶上，所以第一批做 DB 的人冒出了头。接下来 DB 扛不住了，架构上场……出了一批架构师。架构搞好之后，DB 就不再那么重要了，于是开始可以降低 Oracle 规模了，Mysql 上场以及 KV 出场。oracle 收购了 mysql 要围剿的话，谁知道几年后是不是 postgresql 上场？

现在国外用 PHP 很多的，典型的是 Facebook。Java 运行时对内存的管理和监控非常方便，之前 twitter 使用 ruby 写了一个消息队列，因为无法管控内存，换成用 Scala 重写了一个新的消息队列

58 同城使用的是 Windows、iis、SQL-Sever、C# 这条路。现在很多创业公司可能就不会这么做。58 同城为什么当时选择了这条路？原因是公司招聘的第一个工程师和第二个工程师只会这个，所以只能走这条路。

网站在不同的阶段遇到的问题不一样，而解决这些问题使用的技术也不一样，流量小的时

候，我们主要目的是提高开发效率，在早期要引入 ORM、DAO 这些技术。随着流量变大，使用动静分离、读写分离、主从同步、垂直拆分、CDN、MVC 等方式不断提升网站的稳定性。面对更大的流量时，通过垂直拆分、服务化、反向代理、开发框架（站点/服务）等等，不断提升高可用。在面对上亿级的更大流量时，通过中心化、柔性服务、消息总线、自动化（回归，测试，运维，监控）来迎接新的挑战。未来的就是继续实现移动化，大数据实时计算，平台化

1.1.29 一般网站架构

网站架构一般分为网页缓存层、负载均衡层、Web 层、数据库层、文件服务器层。Nginx 已经具备 Squid 所拥有的 Web 缓存加速功能。此外，Nginx 对多核 CPU 的利用胜过 Squid，现在越来越多的架构师都喜欢将 Nginx 同时作为“负载均衡服务器”与“Web 缓存服务器”来使用

1.2 编码规范 (Programming Specification)/编程风格 (Programming Style)

编程规范就是为了便于自己和他人阅读理解源程序，而制定的一个规范。编程规范只是一个规范，也可以不遵守，但是要做一个具有良好编程风格的程序员，就一定要遵守编程规范，不仅方便自己以后的阅读，也方便与其他程序员的交流。代码一旦编写完毕，剩下的就是阅读和修改。所以，需要谨慎对待代码的撰写。编码约定的细节要达到这样的精确度：在编写完软件之后，几乎不可能改变（翻新）软件所遵循的编码约定。用规范和约定来使大脑从记忆不同代码段的随意性、偶然性差异中解脱出来³。不管有多少人共同参与同一项目，一定要确保每一行代码都像是同一个人编写的。

1.2.1 良好编码习惯

1. 确保没有任何警告（warnings）
2. 去掉所有没有用到的 usings，编码过程中尽量去掉多余代码
3. 尽早且经常的重构代码（Refactor often and sooner）
4. 请确保你了解 SOLID⁴原则

在程序设计领域，SOLID（单一功能、开闭原则、里氏替换、接口隔离以及依赖反转）是由罗伯特·C·马丁在 21 世纪早期引入的记忆术首字母缩略字，指代了面向对象编程和面向对象设计的五个基本原则。当这些原则被一起应用时，它们使得一个程序员开发一个容易进行软件维护和扩展的系统变得更加可能。SOLID 所包含的原则是通过引发编程者进行软件源代码的代码重构进行软件的代码异味清扫，从而使得软件清晰可读以及可扩展时可以应用的指南。SOLID 被典型的应用在测试驱动开发上，并且是敏捷开发以及自适应软件开发的基本原则的重要组成部分

³ 《代码大全》34.1 节

⁴http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29

5. 始终遵循命名规范⁵。一般而言变量参数使用驼峰命名法 (dataOper), 方法名和类名使用 Pascal 命名法 (DataOper), 本标准中摒弃使用匈牙利命名法 (data_Oper)
6. 如果需要多次串联, 请使用 StringBuilder 代替 string, 这可以节省堆内存
7. 为了避免在阅读代码时不得不滚动源代码编辑器, 每行代码或注释在 1024*800 的分辨率下不得超过一显示屏, 代码中方法的行数不超过 30 到 40 行
8. 避免 for/foreach 循环嵌套和 if 条件嵌套 (Restrict all code to very simple control flow constructs)
9. 在一个尽可能小的范围内赋值或初始化变量、结构等, 限制全局变量的使用 (Data objects must be declared at the smallest possible level of scope)。降低变量的作用域有如下好处: 其一: 变量被无意修改的可能性降低; 其二: 使代码更具有可读性; 其三: 当需要对大程序分拆成小的子程序时, 降低变量的作用域 (Comments on Minimizing Scope) 会使分拆起来更加简单⁶。全局变量也会大大增加需要兼顾的代码的比例⁷。
10. 重视日志的采集, 保证所有的系统异常和关键信息有记录
11. 使用标准库函数和公共函数
12. 足够的单元测试加上持续集成工具让你修改代码时更加有自信
13. 修改越是细微越好, 如果大量的修改后引入了错误, 会很难发现到底是哪一步修改所导致
14. 所有类型、方法、参数、变量的命名不推荐缩写, 包括大家熟知的缩写, 例如 msg, 使用缩写常见的 2 个问题: 一是代码的读者可能不理解这些缩写, 其他程序员可能会用多个缩写代表相同的词, 从而产生不必要的混乱. 如果没有项目级的标准缩写文档, 不推荐使用缩写.
15. 方法应避免过多参数, 合理的参数个数, 其上限大概是 7 个左右⁸
16. 为变量指定单一用途, 避免在 2 个位置把同一变量用于不同用途, 例如 temp 变量在程序里 2 处用到, 但代表不同的含义⁹
17. 记住典型的布尔变量命名:done,error,found,success 或 ok
18. 将复杂度降到最低是高质量代码的关键
19. 可考虑将大的工具类拆分成互不干扰的小的工具类, 如 NetworkUtil 可拆分成 HttpUtil,FTPUtil,TelnetUtil

1.2.2 变量 (Variable) 命名规范

- 1) 使用名词、名词性词组或者名词地缩写来命名。

⁵更多命名规范见《代码大全》第 11 章 变量名的力量

⁶《代码大全》10.4 节

⁷《代码大全》34.1 节

⁸《代码大全》7.1 节.

⁹《代码大全》10.8 节

1.2. 编码规范 (PROGRAMMING SPECIFICATION)/编程风格 (PROGRAMMING STYLE)51

- 2) 使用 Camel Case, 与.NET 的风格保持一致 (consistent with the Microsoft's .NET Framework and easy to read)。
- 3) 变量名中不使用下划线 (_)。
- 4) 成员变量不要使用匈牙利命名法。
- 5) 只要合适, 在变量名的末尾追加计算限定符 (Avg、Sum、Min、Max、Index)。
- 6) 在变量名中使用互补对, 如 min/max、begin/end 和 open/close。
- 7) 布尔变量包含 Is, 这意味着 Yes/No 或 True/False 值, 如 fileIsFound。
- 8) 状态变量命名时, 避免使用诸如 Flag 的术语。状态变量不同于布尔变量的地方是它可以具有两个以上的可能值。不是使用 documentFlag, 而是使用更具描述性的名称, 如 documentFormatType。
- 10) 数组后加 Array, 列表后加 List, 字典后加 Dictionary, DataSet 后加 Set。

1.2.3 命名空间 (Namespace) 命名规范

- 1) 使用公司名. 产品名这样的格式。
- 2) Namespace 中类的依赖关系应该体现在命名上, 比如 System.Web.UI.Design 中的类以 来于 System.Web.UI。
- 3) 使用 Pascal 大小写形式命名。
- 4) 当商标 (产品名) 的命名风格和 Pascal 风格不符时, 以商标 (产品名) 为准。
- 5) 在语意合适的情况下使用复数, 比如 System.Collections。例外是缩写和商标的情况。
- 6) Namespace 的名字不一定和 Assembly 一一对应。

1.2.4 类 (Class) 命名规范

- 1) 使用名词或者名词性词组命名 class。
- 2) 使用 Pascal case, 与.NET 的命名保持一致 (consistent with the Microsoft's .NET Framework and easy to read)。
- 3) 文件名应和类名相同。
- 4) 保守地使用缩写。
- 5) 不使用 type 前缀, 例如 C 来标识 Class。比如, 使用 FileStream 而不是 CFileStream。
- 6) 不使用下划线。
- 7) 偶尔的在 Class 名称组成中需要使用 I 开头的时候, 比如 IdentityStore, just use it。
- 8) 在合适的时候, 使用单词复合来标识从某个基类继承而来。比如 xxxException。

1.2.5 类 (Class) 注释规范

```
1  /*-----
2  <copyright file="$safeitemrootname$.cs" company="RRMall">
3  Copyright (c) RRMall.All Rights Reserved.
4  </copyright>
5  CLRVersion:$clrversion$
6  NameSpace:$rootnamespace$  

7  Author:$username$  

8  Email:jiangxiaoqiang@renrenmall.com  

9  CreateDate:$time$  

10 Stamp:$guid10$  

11 UserDomain:$userdomain$  

12
13 -----
14 Modifier:  

15 ModifyDate:  

16 ModifyDescription:  

17 -----*/  

18 namespace $rootnamespace$  

19 {  

20     using System;  

21
22     /// <summary>  

23     /// Write the class summary.  

24     /// </summary>  

25     public class $safeitemrootname$  

26     {  

27     }  

28 }
```

1.2.6 接口 (Interface) 命名规范

- 1) 使用名词或者名词性词组命名 Interface。
- 2) 使用 Pascal case。
- 3) 保守地使用缩写。
- 4) 在 interface 名称前加上字母 I 来表示 type 是 interface。
- 5) 在某个 class 是某个 interface 地标准实现地时候，用类似的名字来命名它们，仅仅在 interface 的名称前面多个 I。
- 6) 不要使用下划线。
- 7) 文件名应和类名相同。

1.2.7 常量命名规范

常量名也应当有一定的意义，格式为 NOUN 或 NOUN_VERB。常量名均为大写，字之间用下划线分隔。

例：

```
private const bool WEB_ENABLEPAGECACHE_DEFAULT = true;
private const int WEB_PAGECACHEEXPIRESINSECONDS_DEFAULT = 3600;
private const bool WEB_ENABLESSL_DEFAULT = false;
```

注：

变量名和常量名最多可以包含 255 个字符，但是，超过 25 到 30 个字符的名称比较笨拙。此外，要想取一个有实际意义的名称，清楚地表达变量或常量的用途，25 或 30 个字符应当足够了。

1.2.8 方法 (Method) 命名规范

- 1) 使用动词或者动词性词组命名。
- 2) 使用 Pascal case。
- 3) 如果方法返回的类型为 bool 类型，则其前缀为 Is、Can 或者 Try。

1.2.9 事件 (Event) 命名规范

- 1) 使用 Pascal case。
- 2) 不要使用匈牙利命名法。
- 3) 在 event handler 名字中使用 EventHandler 后缀。
- 4) 指定两个名字分别为 sender 和 e 的参数。sender 参数代表了发出事件的对象。sender 参数总是类型 object，即使可能使用一个更加精确的类型。和事件相关的状态封装在名字为 e 的 event class 的实体之中。给 e 指定恰当而且明确的 event class。
- 5) 使用 EventArgs 后缀命名事件参数 class。
- 6) 考虑使用动词命名事件。使用进行时态来标识事件正在进行之中，使用完成时态标识事件已经完成，不要使用 BeforeXxx/AfterXxx 命名法。
- 7) 不要在事件声明中使用前缀和后缀，比如，用 Close 而不是 OnClose。
- 8) 一般应同时提供一个名字为 OnXxx 的 protected method 供派生类来改写。
- 9) 事件以其对应的委托类型，去掉 EventHandler 后缀，并加上 On 前缀构成。

1.2.10 数据库命名规范

- 1) 总是使用单数名称，不管是表名称还是列名称¹⁰。例如 Product 而不是 Products。
- 2) 存储过程名称不要以 sp_ 开头，因为 SQL Server 系统存储过程都是以 sp_ 开头，所以 sp_Widget 会给人是否为标准存储过程的疑惑。

1.2.11 创建类标准模板

在新建类时，Visual Studio 会自动调取类模板，通过修改类模板，在新建类时自动添加类注释，避免手工注释的麻烦。模板的路径为：

```
1 D:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ItemTemplates\CSharp
  \Code\2052\Class
```

将类的注释规范（在节1.2.5中）粘贴到模板里面即可，如果想定义其他如新建接口的模板，新建 web 类模板，在临近的文件夹将之替换即可，添加类的注释规范之后可立即生效，不必重新启动 Visual Studio 2013。一个标准的用于新建模板类的模板如下代码片段所示（新建的类通过 StyleCop 检查）：

```
1 /*-----
2 <copyright file="$safeitemrootname$.cs" company="RRMall">
3 Copyright (c) RRMall.All Rights Reserved.
4 </copyright>
5 CLRVersion:$clrversion$
6 NameSpace:$rootnamespace$
7 Author:$username$
8 Email:jiangxiaoqiang@renrenmall.com
9 CreateDate:$time$
10 Stamp:$guid10$
11 UserDomain:$userdomain$

12 -----
13 -----
14 Modifier:
15 ModifyDate:
16 ModifyDescription:
17 -----
18 namespace $rootnamespace$ */
19 {
20     using System;
21
22     /// <summary>
23     /// Write the class summary.
24     /// </summary>
25     public class $safeitemrootname$ {
26     }
27 }
```

¹⁰ 《C# 高级编程（第 7 版）第 30.9.3 节》

28 }

1.2.12 文件夹命名

在 C# 开发中经常遇到添加新文件夹的情况，以下列出了一些常见的命名以供参考。

Client	客户端部署文件夹	Wrapper	包裹
Service	服务端部署文件夹	Factory	工厂
Control	控件存放文件夹	FactoryMethod	工厂方法
Controller	控制器	Facade	外观
ReportCenter	报表中心	Builder	建造者
View	界面	Common	公共
ViewModel	界面模型	Security	安全
Filter	过滤器	Config	配置
Config	配置文件	Adapter	适配器
Business	业务逻辑	Component	组件
Core	核心	Widget	物件
Utility	工具	Menu	菜单
Entity	实体	Static	静态
Pattern(s)	模式	Template	模板
Handler	处理器	Public ¹	公共
Action	动作	Route	路由
Base	基	Proxy	代理
Library	库	Tools	工具库
Platform	平台	NetworkProcess	网络进程
Storage	存储的共享代码	Loader	资源加载器
Preview ²	预演	SubView	子视图
Network	网络	Category	分类
Vendor ³	第三方	Expand	扩展

1.2.13 目录结构

```

1 root
2   |--Source
3   |   |--Plugin
4   |   |   |--Plugin 1
5   |   |   |--Plugin 2
6   |   |--Project A

```

¹Base 用于存放一些抽离提取或以共用的可被继承的内容

²Preview 用于存放一些练习的功能页面，集成一些第三插件实例或者实例代码都可以放在里面

³Vender(第三方) 存放一些可能被修改的第三方插件及一些自个封装插件

```
7 |   |--Project B
8 |   |--Test
9 |   |   |--Test Project A
10 |   |   |--Test Project B
11 |   |--Tool
12 |   |   |--Tool1
13 |   |   |--Tool2
```

1.2.14 代码规范检查

StyleCop analyzes C# source code to enforce a set of style and consistency rules. It can be run from inside of Visual Studio or integrated into an MSBuild project. StyleCop has also been integrated into many third-party development tools.

不是拿到需求就开始写代码，而是先考虑清楚。需求是否合理，是否能解决用户的问题，逻辑上是否有模糊或不完备的地方。然后考虑设计的问题，流程图是什么样的，类图是什么样的，接口是什么样的，对架构和模块的影响是什么样的，考虑清楚后才开始写代码。

1.2.15 避免空引用异常 (Null Reference Exception)

在使用对象之前做非空检查，保持代码的健壮性，获取的 Request 对象进行非空检查。

```
1 \\\ 推荐：对象为空时赋给进行默认值
2 var sellerId = (sellerModelList == null) ? 0 : sellerModelList[0].sellerId;
3
4 \\\ 不推荐写法
5 string requestUrl = Request.Url.OriginalString;
6 \\\ 推荐写法
7 var requestUrlObj = Request.Url;
8 if (requestUrlObj != null)
9 {
10     string requestUrl = requestUrlObj.OriginalString;
11 }
```

1.3 运行效率 (Performance)

1.3.1 避免开辟内存

1.3.2 避免装箱

1.3.3 常见数据结构及复杂度

Data Structure	Add	Find	Delete	GetByIndex
Array ($T[]$)	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Linked list (LinkedList< T >)	$O(1)$	$O(n)$	$O(n)$	$O(n)$

1.4 数据库

1.4.1 字段

表中应该避免可为空的列 虽然表中允许空列，但是，空字段是一种比较特殊的数据类型。数据库在处理的时候，需要进行特殊的处理。如此的话，就会增加数据库处理记录的复杂性。当表中有比较多的空字段时，在同等条件下，数据库处理的性能会降低许多。

要求二：表不应该有重复的值或者列。

要求三：表中记录应该有一个唯一的标识符。

要求四：数据库对象要有统一的前缀名。

要求五：尽量只存储单一实体类型的数据。

表的主键 (ID、PKID) 应当不具有任何业务含义 表通过主键来保证每条记录的唯一性，表的主键应当不具有任何业务含义，因为任何有业务含义的列都有改变的可能性。关系数据库学的最重要的一个理论就是：不要给关键字赋予任何业务意义。假如关键字具有了业务意义，当用户决定改变业务含义，也许他们想要为关键字增加几位数字或把数字改为字母，那么就必须修改相关的关键字。一个表中的主关键字有可能被其他表作为外键。就算是一个简单的改变，譬如在客户号码上增加一位数字，也可能会造成极大的维护上的开销。为了使表的主键不具有任何业务含义，一种解决方法是使用代理主键，例如为表定义一个不具有任何业务含义的 ID 字段（也可以叫其他的名字），专门作为表的主键¹¹。考虑每个表增加必备字段，用于记录该笔数据的创建时间，创建人，最后修改人，最后修改时间等等信息。

1 [ID] [int] NOT NULL,
 2 [Status] [int] NULL,
 3 [CreatedDate] [datetime] NULL,

¹¹孙卫琴《精通 Hibernate：Java 对象持久化技术详解》

```

4 [CreatedBy] [nvarchar] (10) COLLATE SQL_Latin1_General_CI_AS NULL,
5 [RevisedDate] [datetime] NULL,
6 [RevisedBy] [nvarchar] (10) COLLATE SQL_Latin1_General_CI_AS NULL,
7 [IsDel] [tinyint] NULL,
8 [Remark] [nvarchar] (2048) NULL,
9 [RowGuid] [nvarchar] (38) NULL,
10 [Sort] [int] NULL

```

GUID 是一个 128 位长的数字，一般用 16 进制表示。算法的核心思想是结合机器的网卡、当地时间、一个随即数来生成 GUID。从理论上讲，如果一台机器每秒产生 10000000 个 GUID，则可以保证（概率意义上）3240 年不重复。

1.4.2 登录用户权限

1.5 环境

1.5.1 开发环境

现在所使用的编程语言无非就两种，一种是解释型（Interpreted Language）的语言，另一种就是编译后才能够执行的语言。

An interpreted language is a programming language for which most of its implementations execute instructions directly, without previously compiling a program into machine-language instructions. The interpreter executes the program directly, translating each statement into a sequence of one or more subroutines already compiled into machine code.

解释型语言最大的弱点就是能够被反编译¹²。解释性语言先是编译为中间语言，然后才编译为计算机可以识别的机器语言。解释型语言每执行一次就需要编译一次，解释性语言开发的程序需要依赖于特定的编译框架。但是解释性语言有跨平台的优点。

A compiled language is a programming language whose implementations are typically compilers (translators that generate machine code from source code), and not interpreters (step-by-step executors of source code, where no pre-runtime translation takes place).

编译型语言（Compiled Language）的程序执行效率相对较高，而且只需要编译一次，运行时不需要编译。后者由于程序执行速度快，同等条件下对系统要求较低，因此像开发操作系统、大型应用程序、数据库系统等时都采用它，像 C/C++、Pascal/Object Pascal（Delphi）等都是编译语言，而一些网页脚本、服务器脚本及辅助开发接口这样的对速度要求不高、对不同系统平台间的兼容性有一定要求的程序则通常使用解释性语言，如 JavaScript、VBScript、Perl、

¹² 《加密与解密》（第 3 版）第四篇篇头语

Python、Ruby、MATLAB 等等。项目开发语言选择解释型的语言 C#，项目开发所用的工具以及环境如下表所示。

项	名称	版本
开发工具	Visual Studio	2013
日志组件	log4net	1.2.13.0
单元测试组件	NUnit	2.6.4
代码规范检查工具	StyleCorp	4.7
数据库	SQL Server	2008 R2
源码管理工具	Visual SVN	2.7.11
Json 处理与转换	Json.NET	4.5.11.15520
运行时环境	.NET Framework	4.0
BS 界面	easyUI	1.2.2
持续集成	CruiseControl.NET	1.8.5.0
接口通信	WCF (RESTful)	.NET 4.0
WCF REST 调试工具	HttpRequester	2.1.1
WCF REST 调试工具	RestClient	2.0.3
Json 浏览插件	JsonViewer	
REST 调试	Rest Client for Chrome	
WCF 认证	RSA 签名	
Web 调试	FireDebug	2.0.12
RESTful 接口	NodeJS	4.1.1
NodeJS 框架	Express	4.13.1
数据库管理工具	HeidiSQL	9.3.0.4984
接口压力测试	JMeter	2.13
WCF 测试	WCF Storm	
Google Cookie 管理	EditThisCookie	1.4.1
VS 代码缩进提示插件	Indent Guids	
代码性能分析	PerfView	1.8.0.0
HTTP 调试	Fiddler	4.6.1.4
网页性能优化	MiniProfiler	latest
后端性能优化	dotTrace	10.0.0.2(Commercial)
内存泄漏检测	dotMemory	10.0.0.2(Commercial)
Rest 客户端	RestSharp	latest

Visual Studio Code

在 Fedora 中安装完毕后的路径为：

1 /usr/share/code

运行文件夹下的 code 文件即可。

1.5.2 Web Server

OpenResty(BSD)

OpenResty（也称为 `ngx_openresty`）是一个全功能的 Web 应用服务器。它打包了标准的 Nginx 核心，很多的常用的第三方模块，以及它们的大多数依赖项。通过众多进行良好设计的 Nginx 模块，OpenResty 有效地把 Nginx 服务器转变为一个强大的 Web 应用服务器，基于它开发人员可以使用 Lua 编程语言对 Nginx 核心以及现有的各种 Nginx C 模块进行脚本编程，构建出可以处理一万以上并发请求的极端高性能的 Web 应用。OpenResty 致力于将你的服务器端应用完全运行于 Nginx 服务器中，充分利用 Nginx 的事件模型来进行非阻塞 I/O 通信。不仅仅是和 HTTP 客户端间的网络通信是非阻塞的，与 MySQL、PostgreSQL、Memcached、以及 Redis 等众多远方后端之间的网络通信也是非阻塞的。因为 OpenResty 软件包的维护者也是其中打包的许多 Nginx 模块的作者，所以 OpenResty 可以确保所包含的所有组件可以可靠地协同工作。

基于 OpenResty：优酷。

Tengine

Tengine 是由淘宝网发起的 Web 服务器项目。它在 Nginx 的基础上，针对大访问量网站的需求，添加了很多高级功能和特性。Tengine 的性能和稳定性已经在大型的网站如淘宝网，天猫商城等得到了很好的检验。它的最终目标是打造一个高效、稳定、安全、易用的 Web 平台。

基于 Tengine：凤凰网、淘宝、土豆。

Phusion Passenger

Phusion Passenger 是一个流行的 Web 应用服务器，它最初是针对 Ruby 的，现在也支持 Node.js 应用。在今年的早些时候该功能被引入了 Passenger 的企业版中，但是现在已经开源并随着最近的 4.0.21 免费版发布。Passenger 能与 Apache 或者 Nginx Web 服务器集成，旨在成为一个服务、监控和扩展 Web 应用程序的完整解决方案。Phusion 公司的总部位于荷兰，他们宣称在 Passenger 中运行 Node.js 应用的好处包括：多租户——通过最小的配置运行一些应用的能力监控——自动启动 Node.js 进程、如果进程崩溃了则重启它们扩展——根据要处理的请求的数量增加或者减少进程的数量统计——帮助显示运行中进程的状态的工具 Passenger 的作者还指出，与 Apache/Nginx 集成还带来了其他的好处，例如：加速了静态文件服务，阻止了很多常见的攻击和慢客户端。

1.5.3 Apache

配置文件位置（根据不同的操作系统而定）：

- 1 /etc/apache2/httpd.conf
- 2 /etc/apache2/apache2.conf
- 3 /etc/httpd/httpd.conf

```
4 | /etc/httpd/conf/httpd.conf
```

1.5.4 常见架构

架构名称	OS	数据库	服务端	客户端
MEAN	Linux/Windows	MongoDB	NodeJS+Express	AngularJS
LAMP	Linux	MySQL	Apache Tomcat	Perl/PHP/Python
WAMP	Windows	MySQL/MariaDB	Apache Tomcat	Perl/PHP/Python
LNMP	Linux	MySQL	Nginx	PHP
WSIC	Windows	Microsoft SQL Server	IIS	C#

1.5.5 主流平台

- Rails
- Java EE 平台
- LAMP 平台
- Microsoft .NET 平台
- Django

Chapter 2

WCF

Windows Communication Foundation(WCF) 是由微软发展的一组数据通信的应用程序开发接口，可以翻译为 Windows 通讯接口，它是.NET 框架的一部分。由.NET Framework 3.0 开始引入。WCF 的最终目标是通过进程或不同的系统、通过本地网络或是通过 Internet 收发客户和服务之间的消息。WCF 合并了 Web 服务、.net Remoting、消息队列和 Enterprise Services 的功能并集成在 Visual Studio 中。WCF 专门用于面向服务开发。WCF 与其他面向服务技术之间 (ASP.NET \J2EE \Web Service 技术等) 最大的区别在于传输可靠性 (Transport Reliability) 与消息可靠性 (Message Reliability)。传输可靠性 (例如通过 TCP 传输) 在网络数据包层提供了点对点保证传递 (Point-to-Point Guaranteed Delivery)，以确保数据包的顺序无误。传输可靠性不会受到网络连接的中断或其他通信问题的影响。.net Remoting 是在 DCOM 等基础上发展起来的一种技术，它的主要目的是实现跨平台、跨语言、穿透企业防火墙，这也是他的基本特点，与 WebService 有所不同的是，它支持 HTTP 以及 TCP 信道，而且它不仅能传输 XML 格式的 SOAP 包，也可以传输传统意义上的二进制流，这使得它变得效率更高也更加灵活。而且它不依赖于 IIS，用户可以自己开发 (Development) 并部署 (Dispose) 自己喜欢的宿主服务器，所以从这些方面上来讲 Web Service 其实上是.net Remoting 的一种特例。.net Remoting 主要用于 C/S 结构项目，将.net Remoting 采用 TCP 通讯，比 Web Service 效率高。

WCF 的配置编辑可用微软提供的工具 **SvcConfigEditor.exe**，该工具存放在 C:\Program Files (x86)\Microsoft SDKs\Windows\v8.1A\bin\NETFX 4.5.1 Tools 下。查看跟踪日志可用微软提供的工具 **SvcTraceViewer.exe**，在相同的目录下。两种常见的分布式应用架构风格包括：DO (分布式对象)、RPC (远程过程调用)。这两种架构风格在企业应用中得到了广泛的应用，然而，Web 架构的设计者们却有意避免采用这两种架构风格。主要的原因是运行 Web 应用的互联网环境，与运行企业应用的企业内网环境有很大的差别。

2.1 基础应用

REST 可以降低开发的复杂度，提高系统的可伸缩性，增强系统的可扩展性，简化应用系统之间的集成，一些新产品的开发甚至已经几乎完全抛弃了传统的类似 JSP 的技术，转而大量

使用 REST 风格的构架设计，即在服务器端所有商业逻辑都以 REST API 的方式暴露给客户端，所有浏览器用户界面使用 widget, Ajax, HTML5 等技术，用 HTTP 的方式与后台直接交互。WCF 主要是设计为 RESTful API 供客户端（手机、平板、桌面电脑、其他专用设备...）调用，如下是 RESTful API 设计建议的原则¹，供参考。

- API 与用户的通信协议，总是使用 HTTPS 协议
- 应该尽量将 API 部署在专用域名之下，如<https://api.example.com>。如果确定 API 很简单，不会有进一步扩展，可以考虑放在主域名下，如<https://example.org/api/>
- 在 RESTful 架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的“集合”（collection），所以 API 中的名词也应该使用复数
- API 的版本号不应加入 URI 中²，因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个 URI。版本号可以在 HTTP 请求头信息的 Accept 字段中进行区分
- 针对不同操作，服务器向用户返回的结果应该符合以下规范

1	GET /collection: 返回资源对象的列表（数组）
2	GET /collection/resource: 返回单个资源对象
3	POST /collection: 返回新生成的资源对象
4	PUT /collection/resource: 返回完整的资源对象
5	PATCH /collection/resource: 返回完整的资源对象
6	DELETE /collection/resource: 返回一个空文档

- 如果记录数量很多，服务器不可能都将它们返回给用户。API 应该提供参数，过滤返回结果

1	?limit=10: 指定返回记录的数量
2	?offset=10: 指定返回记录的开始位置。
3	?page=2&per_page=100: 指定第几页，以及每页的记录数。
4	?sortby=name&order=asc: 指定返回结果按照哪个属性排序，以及排序顺序。
5	?animal_type_id=1: 指定筛选条件

- API 的身份认证应该使用 OAuth 2.0 框架
- 服务器返回的数据格式，应该尽量使用 JSON，避免使用 XML
- 服务器向用户返回的状态码³（Status Code）和提示信息，常见的有以下一些（方括号中是该状态码对应的 HTTP 动词）

1	200 OK – [GET]: 服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）
2	201 CREATED – [POST/PUT/PATCH]: 用户新建或修改数据成功

¹ 参考阮一峰的博客：www.ruanyifeng.com/blog/2011/09/restful.html

² <http://www.informit.com/articles/article.aspx?p=1566460>

³ HTTP 状态码：<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

```
3 204 NO CONTENT – [DELETE]: 用户删除数据成功
4 400 INVALID REQUEST – [POST/PUT/PATCH]: 用户发出的请求有错误，服务器没
    有进行新建或修改数据的操作，该操作是幂等的
5 404 NOT FOUND – [*]: 用户发出的请求针对的是不存在的记录，服务器没有进行操作，
    该操作是幂等的
6 500 INTERNAL SERVER ERROR – [*]: 服务器发生错误，用户将无法判断发出的请求
    是否成功
```

接口调用链接的命名原则：/项目名称/模块名/接口名? 参数集合。由于每个接口皆返回状态码，错误消息、提示等公共信息，新建一个基类定义公共信息。

```
1 public class BaseModel
2 {
3     #region 错误码
4
5     private int _errorCode;
6
7     public int ErrorCode
8     {
9         get { return _errorCode; }
10        set { _errorCode = value; }
11    }
12
13    #endregion
14
15    #region 错误消息
16
17    private string _errorMessage;
18
19    public string ErrorMessage
20    {
21        get { return _errorMessage; }
22        set { _errorMessage = value; }
23    }
24
25    #endregion
26
27    #region 错误提示
28
29    private string _errorTip;
30
31    public string ErrorTip
32    {
33        get { return _errorTip; }
34        set { _errorTip = value; }
35    }
36
37    #endregion
38 }
```

之类继承基类，自定义 body 内容。

```

1  public class Explorer : BaseModel
2  {
3      #region Name
4
5      private discount_awardModel _body;
6
7      public discount_awardModel Body
8      {
9          get { return _body; }
10         set { _body = value; }
11     }
12
13     #endregion
14 }
```

实现数据返回。

```

1  public Explorer a(string a)
2  {
3      Explorer ex = new Explorer();
4      ex.ErrorCode = 2000;
5      ex.ErrorMessage = "OK";
6      discount_awardModel award = new discount_awardModel();
7      award.awardName = "熊猫玩具";
8      award.awardStatus = 1;
9      award.awardId = 1;
10     ex.Body = award;
11     return ex;
12 }
```

最终的返回效果如图2.1所示。

在 RESTful 架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的“集合”（collection），所以 API 中的名词也应该使用复数。

```
1 https://api.example.com/zoops
```

2.1.1 WCF 测试客户端

Windows Communication Foundation(WCF) 测试客户端 (WcfTestClient.exe) 是一个 GUI 工具，使用该工具，用户可以输入测试参数、将该输入提交给服务并查看服务发回的响应。当与 WCF 服务主机结合时，它可以提供完美的服务测试体验。the MSDN documentation tells you to navigate to the

```

1. {
2.     "ErrorCode": 2000,
3.     "ErrorMessage": "OK",
4.     "ErrorTip": null,
5.     "Body":
6.     {
7.         "awardContext": null,
8.         "awardId": 1,
9.         "awardName": "熊猫玩具",
10.        "awardOrder": 0,
11.        "awardStatus": 1,
12.        "awardTag": null,
13.        "awardTypeId": 0,
14.        "expireDate": 0,
15.        "picUrl": null,
16.        "ratio": null,
17.        "remaindCount": 0,
18.        "sellerId": 0,
19.        "totalCount": 0,
20.        "validEndTime": null,
21.        "validStartTime": null,
22.        "verifyUserId": null
23.     }
24. }

```

图 2.1: 接口数据返回

1 %SystemDrive%\Program Files\Microsoft Visual Studio 10.0\Common7\IDE

folder. BUT, if you're running 64-bit Windows, you won't find it there. You'll need to look in

1 %SystemDrive%\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE folder

instead.

2.1.2 WCF 和 Web Service 的异同

WCF is a replacement for all earlier web service technologies from Microsoft. It also does a lot more than what is traditionally considered as "web services".

WCF "web services" are part of a much broader spectrum of remote communication enabled through WCF. You will get a much higher degree of flexibility and portability doing things in WCF than through traditional ASMX because WCF is designed, from the ground up, to summarize all of the different distributed programming infrastructures offered by Microsoft. An endpoint in WCF can be communicated with just as easily over SOAP/XML as it can over TCP/binary and to change this medium is simply a configuration file mod. In theory, this reduces the amount of new code needed when porting or changing business needs, targets, etc.

ASMX is older than WCF, and anything ASMX can do so can WCF (and more). Basically you can see WCF as trying to logically group together all the different ways of getting two apps to communicate in the world of Microsoft; ASMX was just one of these many ways and so is now grouped under the WCF umbrella of capabilities.

Web Services can be accessed only over HTTP & it works in stateless environment, where WCF is flexible because its services can be hosted in different types of applications. Common scenarios for hosting WCF services are IIS, WAS, Self-hosting, Managed Windows Service.

The major difference is that Web Services Use XmlSerializer. But WCF Uses DataContractSerializer which is better in Performance as compared to XmlSerializer⁴.

2.1.3 WCF 的配置

WCF 配置下主要包含：bindings、Services、behaviors。behaviors 又包含 4 种 behavior：serviceBehaviors, endpointBehaviors, contractBehaviors 和 operationBehaviors。这些 behavior 接口能作为几乎所有其他扩展功能的入口点，在实现它们的过程中我们能方便地取得后者（其他扩展功能）的引用。如果浏览器中通过 Get 展示返回数据如 Json 和 Xml 等，endpointBehaviors 的配置如下代码片段所示。

```

1  <endpointBehaviors>
2      <behavior name="commonEndpoint">
3          <webHttp helpEnabled="true" />
4      </behavior>
5  </endpointBehaviors>
```

在项目中可以点击右键弹出菜单，选择”编辑 WCF 配置”，调取 svcConfigEditor.exe 程序对 WCF 的配置进行编辑，如图2.2所示。



图 2.2: 编辑 WCF 配置菜单

配置在 system.serviceModel 配置节中，包含 bindings、client。binding 的策略选择见图2.3。

WCF 提供了一种特殊的终结点——元数据⁵ 交换终结点（MEX 终结点-Metadata Ex-

⁴具体请参考：<http://www.codeproject.com/Articles/139787/What-s-the-Difference-between-WCF-and-Web-Services>

⁵元数据（Metadata），又称中介数据、中继数据，为描述数据的数据（data about data），主要是描述数据属性（property）的信息，用来支持如指示存储位置、历史数据、资源查找、文件记录等功能。元数据是关于数据的组织、数据域及其关系的信息，简言之，元数据就是关于数据的数据。

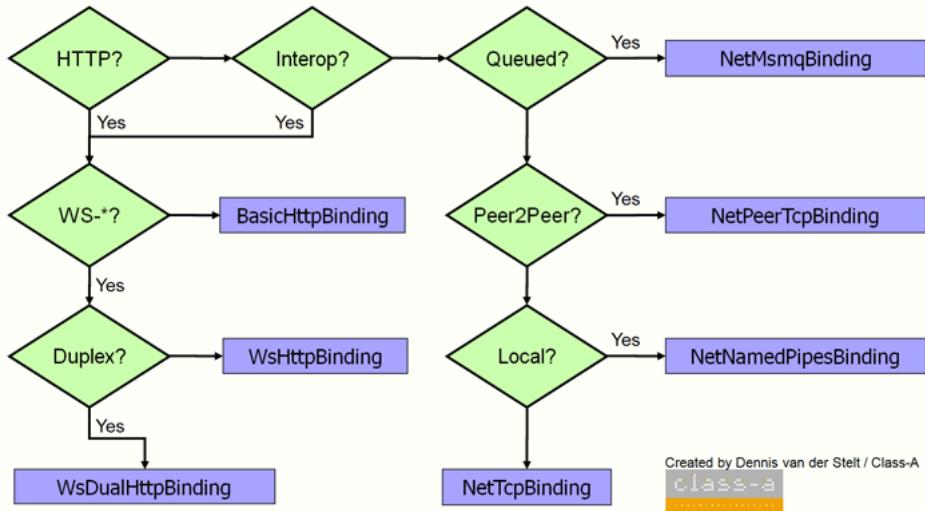


图 2.3: WCF 的 binding 策略选择

change)，通过它，服务就能够发布元数据。WCF 服务端和客户端通过共享元数据（包括服务协定、服务器终结点信息）在两个终结点上建立通道从而进行通信。我们通过手写代码（或配置）的方式为服务端编写了元数据信息，没有借助元数据交换就实现了通信。然而在实际应用中，元数据往往是很多的，而且重复编写元数据的工作也是不值得的，因此必然会用到元数据交换的方式让客户端获取元数据。

2.1.4 WCF 契约

服务契约定义了远程访问对象和可供调用的方法，数据契约则是服务端和客户端之间要传送的自定义数据类型。只有声明为 `DataContract` 的类型的对象可以被传递，且只有成员属性会被传递，成员方法不会被传递。WCF 对声明为 `DataContract` 的类型提供更加细节的控制，可以把一个成员排除在序列化 (Serialization) 范围以外，也就是说，客户端程序不会获得被排除在外的成员的任何信息，包括定义和数据。默认情况下，所有的成员属性都被排除在外，因此需要把每一个要传送的成员声明为 `DataMember`。`DataContract` 也支持 `Name\Namespace` 属性，如同 `ServiceContract`，`Name` 和 `Namespace` 可以自定义名称和命名空间，客户端将使用自定义的名称和命名空间对 `DataContract` 类型进行访问。

2.1.5 WCF 的调用

浏览部署完毕后的 WCF 页面，文件名以 `svc` 结尾。浏览此文件时会有一个网址，形如：<http://129.21.11.121:8018/WCFSERVICE/Tools/UploadFile.svc?wsdl>，将此网址添加进 Visual Studio 的 Web Service 引用即可。系统在引用后会自动添加形如下代码的配置并同时创建代理类，客户端需要一个代理类来访问服务。

```

1 <system.serviceModel>
2   <bindings>
3     <basicHttpBinding>

```

```

4     <binding name="BasicHttpBinding_IUploadFile" messageEncoding="Mtom" />
5   </basicHttpBinding>
6 </bindings>
7 <client>
8   <endpoint address="http://129.21.11.121:8018/WCFSERVICE/Tools/UploadFile.svc
9     ?wsdl"
10    binding="basicHttpBinding" bindingConfiguration="
11      BasicHttpBinding_IUploadFile"
12      contract="WCFFILESERVICE.IUploadFile" name="BasicHttpBinding_IUploadFile"
13    " />
14 </client>
15 </system.serviceModel>

```

MTOM(Message Transmission Optimization Mechanism) 编码⁶，如有必要，需手动将此段代码拷贝到当前程序正在使用的配置文件中，绑定(bindings)描述了服务的通信方式，其中 basicHttpBinding 用于最广泛的交互操作的第一代 Web 服务。所使用的传输协议是 HTTP 或 HTTPS，安全性由协议保证⁷。客户端调用服务如下代码片段2.1所示：

Listing 2.1: 调用 WCF 接口上传文件

```

1 #region 上传文件
2 /// <summary>
3 /// 上传文件
4 /// </summary>
5 /// <param name="extension"> 文件扩展名 </param>
6 /// <param name="path"> 服务器保存路径 </param>
7 /// <param name="source"> 本地路径 </param>
8 /// <param name="inputStream"> 文件流 </param>
9 public void UploadFileWithFileStream(string extension, string path, string source,
10           Stream inputStream)
11 {
12   try
13   {
14     UploadFileClient uploadFileClient = new UploadFileClient();
15     string fullPath = string.Empty;
16     string fileName = string.Empty;
17     bool isSuccess = true;
18     string errorMesssage = uploadFileClient.StreamToFile(extension, path, inputStream,
19               out fileName, out fullPath, out isSuccess);
20     if (!isSuccess)
21     {
22       logger.Error("Upload file failed!" + errorMesssage);
23     }

```

⁶MTOM (Message Transmission Optimization Mechanism) 编码，MTOM 是一种机制，用来以原始字节形式传输包含 SOAP 消息的较大二进制附件，从而使所传输的消息较小。使用 MTOM 的目的在于优化对较大的二进制负载的传输。对于较小的二进制负载来说，使用 MTOM 发送 SOAP 消息会产生显著的开销，但是，当这些负载增大到几千个字节时，该开销会变得微不足道。原因在于，正常文本 XML 使用 Base64 对二进制数据进行编码，这要求每三个字节对应四个字符，从而使得数据的大小增加三分之一。MTOM 能够以原始字节形式传输二进制数据，这会缩短编码/解码时间并生成较小的消息。与当今的业务文档和数字照片相比，几千个字节的阈值是非常小的。

⁷《C# 高级编程》(第三版) 第 43.5 节

```

22     uploadFileClient.Close();
23 }
24 catch (Exception e)
25 {
26     logger.Error("Upload file failed, path:" + source, e);
27 }
28 }
#endregion

```

此种调用方式一般不推荐，此访问操作需要得知访问的具体地址，加以服务引用即可，不足之处是可能出现序列化失败，值传递异常，加载资源大，需要生成一堆使用不了的文件，服务端是附加在 Web API 上的通过配置文件将 WCF 开放出来。注意最后调用服务实例的 Close 方法，将 WCF 服务的客户端关闭，否则此连接会在设置的会话（一般为 10 分钟）后才自动关闭。期间任何客户端也无法使用此服务。如果默认的连接数不能满足客户端的需要，可以增加连接数。配置文件如下：

```

1 <serviceThrottling maxConcurrentCalls="20" maxConcurrentSessions="20"
                     maxConcurrentInstances="30" />

```

说明：maxConcurrentCalls：最大并发数，默认为 16；maxConcurrentSessions：最大的会话数，主要针对于 PerSession 的情况，默认为 10；maxConcurrentInstances：最大实例数，默认为 26

2.1.6 Web Request 和 ajax 调用

WCF 接口需要满足在非.NET 客户程序中，可直接调用，同时在可通过 Ajax 的 Get 请求可获取 WCF 服务返回结果。满足如上要求之前，需要了解一下 RESTful 架构⁸。REST⁹这个词，是 Roy Thomas Fielding 在他 2000 年的博士论文中提出的。什么是 RESTful 架构：

- (1) 每一个 URI 代表一种资源；
- (2) 客户端和服务器之间，传递这种资源的某种表现层；
- (3) 客户端通过四个 HTTP 动词，对服务器端资源进行操作，实现“表现层状态转化（Representational State Transfer）”。

RESTful 架构，就是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，跨平台跨语言。所以正得到越来越多网站的采用¹⁰。客户端用到的手段，只能是 HTTP 协议。具体来说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源（也可以用于更新资源），PUT 用来更新资源，DELETE 用来删除资源。在 WCF 服务中添加 WebGet 属性，添加后方法的接口定义：

⁸更早的有 RPC (Remote Procedure Call Protocol) 和 SOAP (Simple Object Access Protocol)

⁹REST，即 Representational State Transfer 的缩写。阮一峰对这个词组的翻译是“表现层状态转化”。

¹⁰<http://www.ruanyifeng.com/blog/2011/09/restful.html>

```

1 [OperationContract]
2 [WebGet(UriTemplate = "/GetSellerListByPage/pageIndex={pageIndex}&pageSize={"
3   pageSize}&count={count}",
4   ResponseFormat = WebMessageFormat.Json,
5   BodyStyle = WebMessageBodyStyle.Wrapped)]
6 List<discount_sellerModel> GetSellerListByPage(int pageIndex, int pageSize, ref int count)
;
```

这里加了一个 WebGet 的 Attribute, 这将允许 WCF 服务直接通过地址调用。UriTemplate¹¹ 是一个表示统一资源标识符 (URI) 模板的类, 继承自 Object 基类。接口中一个方法中添加了 WebGet 之后, 其他方法也需要添加 WebGet 并将 WebMessageBodyStyle 设置为 Wrapped¹²。请求消息和回复消息分别是对操作方法输入参数和返回值 (输出参数和引用参数) 的封装, WebMessageBodyStyle 中的 Bare 表示请求消息和回复消息的主体部分仅仅包含针对输入参数和返回值 (输出参数和引用参数) 序列化后的内容, 而 Wrapped 则会在外面包装一个基于当前操作的“封套”。Bare 消息风格返回的 Json 串可直接在客户端反序列化, 而 Wrapped 方式返回的 Json 串需要在客户端定义 Model, 这是 Bare 消息风格的优势。但是 Bare 消息主体风格有一些限制, 对于 Bare 消息主体风格来说, 意味着对象被序列化后生成的 XML 或者 JSON 表示直接作为消息的主体, 所以只适用于单一对象。具体来说, 只有具有唯一输入参数的操作方法才能将请求消息的主题风格设置为 Bare。

```

1 [ServiceContract(Namespace = "http://www.artech.com/")]
2 public interface ICalculator
3 {
4   [WebInvoke(BodyStyle = WebMessageBodyStyle.Bare)]
5   double Add( double x, double y);
6 }
```

如上所示的是我们熟悉的计算服务的契约接口的定义。消息主体风格为 Bare 的操作方法 Create 具有两个输入参数 (x 和 y), 在对实现了该契约接口进行寄宿的时候就会抛出如下图所示的 InvalidOperationException 异常, 提示“约定 “ICalculator” 的操作 ‘Add’ 指定要序列化多个请求正文参数, 但没有任何包装元素。如果没有包装元素, 至多可序列化一个正文参数。请删除多余的正文参数, 或将 WebGetAttribute/WebInvokeAttribute 的 BodyStyle 属性设置为 Wrapped。由于回复参数是对返回值、引用参数和输出参数的封装, 所以当操作方法具有引用参数 (ref 关键字) 或者输出参数 (out 关键字) 时不能将回复消息的主体风格设置为 Bare。

```

1 [ServiceContract(Namespace = "http://www.artech.com/")]
2 public interface ICalculator
3 {
4   [WebInvoke(BodyStyle = WebMessageBodyStyle.WrappedRequest)]
5   void Add(double x, double y, out double result);
6 }
```

¹¹<https://msdn.microsoft.com/zh-cn/library/system.uritemplate%28v=vs.100%29.aspx>

¹²有 4 个枚举值, 分别为 Bare: Both requests and responses are not wrapped. Wrapped: Both requests and responses are wrapped. WrappedRequest: Requests are wrapped, responses are not wrapped. WrappedResponse: Responses are wrapped, requests are not wrapped.

同样以计算服务契约为例，现在我们通过如上的方式以输出参数的形式返回加法运算的结果，并将应用在操作方法上的 WebInvokeAttribute 特性的 BodyStyle 属性设置为 WrappedRequest，这意味着请求消息和回复消息分别采用 Wrapped 和 Bare 风格。当我们对实现了该契约接口的服务设施寄宿时会抛出下图所示的 InvalidOperationException 异常，并提示“约定 ‘ICalculator’ 的操作 ‘Add’ 至少指定一个响应正文参数不是操作的返回值。当 WebGetAttribute/WebInvokeAttribute 的 BodyStyle 属性设置为 Bare 时，只允许使用返回值。请删除多余的响应正文参数或将 BodyStyle 属性设置为 Wrapped”。pageIndex 等加中括号的内容代表在提交 Get 请求时传入的参数，参数之间用符号 & 进行分隔。修改配置文件为：

```
1  <system.serviceModel>
2  <services>
3    <service behaviorConfiguration="RR.API.WCFService.User.UserServiceBehavior"
4      name="RR.API.WCFService.User.UserService">
5      <endpoint address="" binding="webHttpBinding" contract="RR.API.WCFService.User.
6        IUserService" behaviorConfiguration="test" bindingConfiguration="basicWeb">
7        <identity>
8          <dns value="localhost" />
9        </identity>
10       </endpoint>
11     <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
12   </service>
13 </services>
14
15 <bindings>
16   <basicHttpBinding>
17     <binding closeTimeout="00:10:00" receiveTimeout="00:20:00" sendTimeout="00
18       :20:00" maxBufferSize="20000000" maxReceivedMessageSize="20000000">
19       <security mode="None"></security>
20     </binding>
21   </basicHttpBinding>
22   <webHttpBinding>
23     <binding name="basicWeb"/>
24   </webHttpBinding>
25 </bindings>
26
27 <behaviors>
28   <serviceBehaviors>
29     <behavior name="RR.API.WCFService.User.UserServiceBehavior">
30       <serviceMetadata httpGetEnabled="true"/>
31       <serviceDebug includeExceptionDetailInFaults="false"/>
32     </behavior>
33   </serviceBehaviors>
34
35   <endpointBehaviors>
36     <behavior name="test">
37       <webHttp/>
38     </behavior>
39   </endpointBehaviors>
40 </behaviors>
```

```

39  <serviceHostingEnvironment multipleSiteBindingsEnabled="true" />
40  </system.serviceModel>

```

WebHttpBinding 用于通过 HTTP 请求提供的服务, 它对于脚本客户端很有用, 如 ASP.NET AJAX¹³。在 WCF 实际运行的过程中, 访问的地址以配置文件配置的地址为准, 和实际引用的地址无关。配置完毕后在浏览器中输入形如如下链接:

```

1 http://129.11.21.18:8019/WCFService/Product/PublicProductService.svc/
2 GetSingleSellerAwardListByPage?pageIndex={PAGEINDEX}&pageSize={PAGESIZE}&
   count={COUNT}&sellerId={SELLERID}

```

将会看到访问后的结果。将 Web.config 中的 <webHttp/> 修改为 <webHttp helpEnabled="true"/> 将可以在浏览器页面中列举出可用接口, 并提供提交的数据样例。打开浏览器, 在地址栏输入 <http://IP:Port/Services/ShowerService.svc/help> 即可, 如图 2.4 所示。

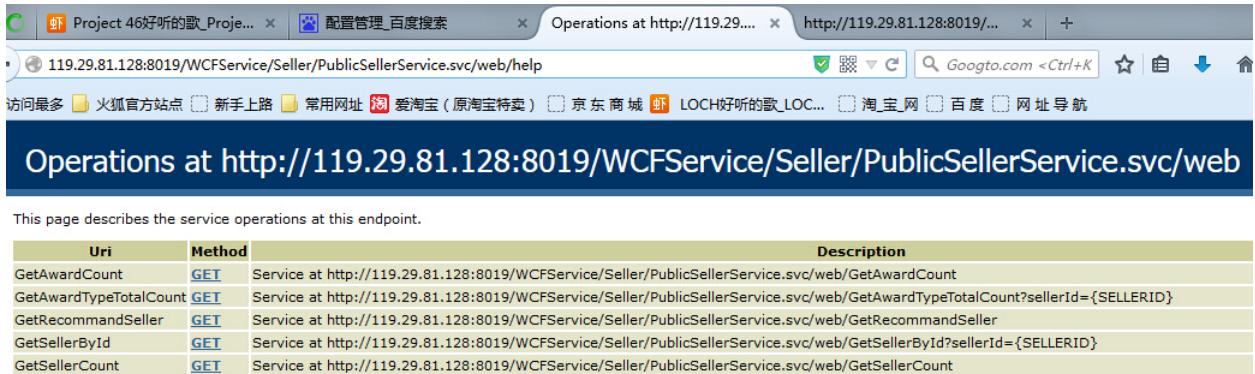


图 2.4: 查看 WCF 可用接口

因为 REST 是基于 HTTP 的, 所以对于 REST 的客户端的开发者, 无法像传统的 WebService 或其他的 WCF 服务通过引用 WSDL, 享受“奢侈”的代码生成, 而使用强类型的本地代理调用服务。开发者只能通过 Http Request 的组装, 但正因为这种直接的 HttpRequest 组装, 而使得客户端真正是语言无关的。WCF 通过 Web 调用时, 所用的绑定方式是 webHttpBinding。通过客户端引用方式调用时, 所用的绑定方式是 basicHttpBinding。在 service 配置节下添加不同的终节点 (endpoint), 针对 Web 调用, 终节点的配置如代码片段 2.2 所示。

Listing 2.2: Web 调用的终节点配置

```

1 <endpoint address="web" behaviorConfiguration="commonEndpoint" binding=
   "webHttpBinding" contract="RR.API.WCFService.Custom.IPublicBaseService"/>

```

在 web 调用时用如 <http://localhost:50590/WCFService/Custom/PublicBaseService.svc/web/DoWork?a={A}> 形式的 Url。同样在浏览器查看所有可用接口集合的时候也需要在 Url 中包含终节点地址名称, 形如 <http://localhost:50590/WCFService/Custom/PublicBaseService.svc/web/help>。

¹³<C# 高级编程 >43.5 节

`svc/web/help`, 其中 web 为终节点地址的名称。而通过客户端调用的还是原来的终节点配置, 如代码2.3所示。

Listing 2.3: 客户端引用调用 service 终节点配置

```
1 <endpoint binding="basicHttpBinding" contract="RR.API.WCFService.Custom.
    IPublicBaseService"></endpoint>
```

通过 Http Request 方式调用的链接和客户端引用方式调用的链接是不一样的, Http Request 方式调用的 Url 中需要加上 endpoint 的 Address 名称。通过 Http Request 而不通过引用的方式调用, 是因为引用的方式调用会在客户端生成大量代码, 自动生成许多文件, 给维护造成困难。另 WCF 的参数只有一个时, 可写成如图2.5所示的形式。

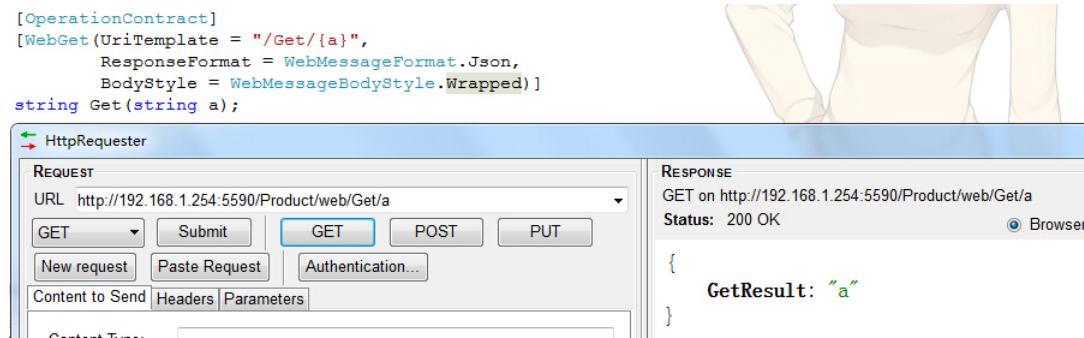


图 2.5: WCF 的 Url 参数风格

2.1.7 WCF REST AJax 跨域请求调用

网络应用程序, 分为前端和后端两个部分。当前的发展趋势, 就是前端设备层出不穷 (手机、平板、桌面电脑、其他专用设备……)。因此, 必须有一种统一的机制, 方便不同的前端设备与后端进行通信。这导致 API 构架的流行, 甚至出现“API First”的设计思想。RESTful API 是目前比较成熟的一套互联网应用程序的 API 设计理论 [2]。在以前, 前端和后端混杂在一起, 比如 JavaScript 直接调用同系统里面的一个 HttpHandler, 就不存在跨域的问题, 但是随着现代的这种多种客户端的流行, 比如一个应用通常会有 Web 端, App 端, 以及 WebApp 端, 各种客户端通常会使用同一套的后台处理逻辑, 即 API, 前后端分离的开发策略流行起来, 前端只关注展现, 通常使用 JavaScript, 后端处理逻辑和数据通常使用 WebService 来提供 json 数据。一般的前端页面和后端的 WebService API 通常部署在不同的服务器或者域名上。这样, 通过 ajax 请求 WebService 的时候, 就会出现同源策略的问题。

传统的 Ajax 请求只能获取在同一个域名下面的资源, 即只有协议 + 主机名 + 端口号 (如存在) 相同, 则允许相互访问。也就是说 JavaScript 只能访问和操作自己域下的资源, 不能访问和操作其他域下的资源。但是 HTML5 打破了这个限制, 允许 Ajax 发起跨域的请求。浏览器是可以发起跨域请求的, 比如你可以外链一个外域的图片或者脚本。但是 Javascript 脚本是不能获取这些资源的内容的, 它只能被浏览器执行或渲染。Ajax 支持跨域的解决方案目前就是 JSONP (JSON with Padding)。在使用用 Ajax 调用接口时出现如下提示:

```
1 同源策略禁止读取位于http://129.29.1.28:8019/Product/web/Get?{%20%22a%22:%20%22a
2 %22%20}&_=1441558531991 的远程资源。 (原因: CORS 头缺少 'Access-Control-
3 Allow-Origin') 。
```

正常使用 Ajax 会需要正常考虑跨域问题，所以伟大的程序员们又折腾出了一系列跨域问题的解决方案，如 JSONP、flash、ifame、xhr2 等等。CORS 定义一种跨域访问的机制，可以让 Ajax 实现跨域访问。CORS 允许一个域上的网络应用向另一个域提交跨域 Ajax 请求。实现此功能非常简单，只需由服务器发送一个响应标头即可。一个域是由 schema、host、port 三者共同组成，与路径无关。

所谓跨域，是指在 http://example-foo.com/域上通过 XMLHttpRequest 对象调用 http://example-bar.com/域上的资源。CORS 约定服务器端和浏览器在 HTTP 协议之上，通过一些额外 HTTP 头部信息，进行跨域资源共享的协商。服务器端和浏览器都必需遵循规范中的要求。

使用 CROS 实现 Ajax 跨域访问

CORS（跨域资源共享，Cross-Origin Resource Sharing）定义一种跨域访问的机制，可以让 Ajax 实现跨域访问。CORS 允许一个域上的网络应用向另一个域提交跨域 Ajax 请求。实现此功能非常简单，只需由服务器发送一个响应标头即可。跨域资源共享 CORS 与 JSONP 相比，无疑更为先进、方便和可靠。

1. JSONP 只能实现 GET 请求，而 CORS 支持所有类型的 HTTP 请求。
2. 使用 CORS，开发者可以使用普通的 XMLHttpRequest 发起请求和获得数据，比起 JSONP 有更好的错误处理。
3. JSONP 主要被老的浏览器支持，它们往往不支持 CORS，而绝大多数现代浏览器都已经支持了 CORS

CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE 浏览器不能低于 IE10。整个 CORS 通信过程，都是浏览器自动完成，不需要用户参与。对于开发者来说，CORS 通信与同源的 AJAX 通信没有差别，代码完全一样。浏览器一旦发现 AJAX 请求跨源，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感觉。因此，实现 CORS 通信的关键是服务器。只要服务器实现了 CORS 接口，就可以跨源通信。For WCF service you have to develop new behavior and include it in the endpoint configuration¹⁴:

客户端脚本如下所示：

```
1 <center style="margin-bottom:20px">
2   <a class="btn btn-block btn-lg btn-warning" id="input-verifycros">CROS实现
3     WCF跨域调用</a>
4   <div id="content"></div>
5 </center>
```

¹⁴http://enable-cors.org/server_wcf.html

调用服务的 Ajax 脚本：

```

1 var XMLHttpRequest=new XMLHttpRequest();
2 $(function () {
3     $("#input-verifycros").bind("click", function () {
4         //http://127.0.0.1:5509/my/WCFAjaxTest
5         var url = "http://129.19.1.138:7019/WCFService/Custom/ServiceCros.svc/web/
6         DoWork/a";
7         if (xmlHttpRequest) {
8             xmlHttpRequest.open('GET', url, true);
9             xmlHttpRequest.onreadystatechange = handler;
10            xmlHttpRequest.send();
11        }
12        else {
13            document.getElementById("content").innerHTML = "不能创建XMLHttpRequest";
14        }
15    });
}

```

XMLHttpRequest 是 Ajax 的基础，所有现代浏览器（IE7+、Firefox、Chrome、Safari 以及 Opera）均内建 XMLHttpRequest 对象。每当 readyState 改变时，就会触发 onreadystatechange 事件。readyState 属性存有 XMLHttpRequest 的状态信息。存有 XMLHttpRequest 的状态，从 0 到 4 发生变化：

- 0: 请求未初始化
- 1: 服务器连接已建立
- 2: 请求已接收
- 3: 请求处理中
- 4: 请求已完成，且响应已就绪

open 方法规定请求的类型、URL 以及是否异步处理请求。脚本 HttpRequest 调用的函数：

```

1 function handler(evtXHR) {
2     if (xmlHttpRequest.readyState == 4) {
3         console.log("HTTP status:" + xmlHttpRequest.status);
4         if (xmlHttpRequest.status == 200) {
5             var response = xmlHttpRequest.responseText;
6             document.getElementById("content").innerHTML = "结果：" + response;
7         } else {
8             console.log("不允许跨域请求");
9             document.getElementById("content").innerHTML = "不允许跨域请求。";
10        }
11    }
12    else {
13        document.getElementById("content").innerHTML += "<br/>执行状态 readyState:
14        " + xmlHttpRequest.readyState;
15    }
}

```

参考链接¹⁵。服务端的配置 Step 1:

```

1 <system.webServer>
2   <httpProtocol>
3     <customHeaders>
4       <add name="Access-Control-Allow-Origin" value="*" />
5     </customHeaders>
6   </httpProtocol>
7
8   <modules runAllManagedModulesForAllRequests="true"/>
9   <directoryBrowse enabled="true"/>
10 </system.webServer>
```

服务端的配置 Step 2: 在 system.ServiceModel 中的 standardEndpoint 增加配置。

```

1 <standardEndpoints>
2   <webHttpEndpoint>
3     <standardEndpoint crossDomainScriptAccessEnabled="true"/>
4   </webHttpEndpoint>
5 </standardEndpoints>
```

使用 JSONP 实现 Ajax 跨域访问

同源策略下，某个服务器是无法获取到服务器以外的数据，但是 html 里面的 img,iframe 和 script 等标签是个例外，这些标签可以通过 src 属性请求到其他服务器上的数据。而 JSONP 就是通过 script 节点 src 调用跨域的请求。在 WCF REST 中使用 Ajax 跨域请求调用 WCF 接口需要在服务端配置，同时在客户端调用时需要使用 jsonp 的方式。服务端的配置如下代码所示¹⁶:

```

1 <standardEndpoints>
2   <webHttpEndpoint>
3     <standardEndpoint crossDomainScriptAccessEnabled="true"/>
4   </webHttpEndpoint>
5 </standardEndpoints>
6 <bindings>
7   <webHttpBinding>
8     <binding crossDomainScriptAccessEnabled="true" />
9   </webHttpBinding>
10 </bindings>
```

同时在服务的相应终结点上添加 kind 属性，如下例子所示：

```

1 <services>
```

¹⁵<http://debugmode.net/2014/06/12/solved-access-control-allow-origin-error-in-wcf-rest-service/>

¹⁶<http://www.cnblogs.com/artechno/archive/2012/01/16/jsonp-wcf-rest.html>

```

2   <service name="Artech.WcfServices.Service.EmployeesService">
3     <endpoint kind="webHttpEndpoint"
4       address="http://127.0.0.1:3721/employees"
5       contract="Artech.WcfServices.Service.Interface.IEmployees"/>
6   </service>
7 </services>

```

客户端调用代码如下所示：

```

1 <script>
2 $(function () {
3   $("#input-verify").bind("click", function () {
4     //http://192.168.1.254:5508/my/WCFAjaxTest
5     var url = "http://129.9.1.138:8019/Product/web/Get/a";
6     jQuery.ajax({
7       url: url,
8       type: 'get',
9       dataType: 'jsonp',
10      success: function (data) {
11        alert(data);
12      },
13      error: function (xhr, status, error) {
14        console.log(xhr);
15        if (xhr === 'undefined' || xhr == undefined)
16          alert('undefined');
17        } else {
18          alert('object is there');
19        }
20        alert(status);
21        alert(error);
22      }
23    });
24  })
25 });
26 </script>

```

为了避免与 NVelocity 模板冲突，此处使用 `jQuery.ajax` 的写法，另一种写法为 `$.ajax`，两种写法没有本质的区别，`$` 即是 `Jquery` 的简写，`$` 只是 `jquery` 的代表符号，只有引入 `jquery.js` 后才可以这么用它，`$.ajax` 其实用的是 `jQuery.ajax`。在 `jQuery` 的源码里面有如下语句：

```

1 //Expose jQuery to the global object
2 window.jQuery = window.$ = jQuery;

```

许多 JavaScript 库使用 `$` 作为函数或变量名，`jQuery` 也一样。在 `jQuery` 中，`$` 仅仅是 `jQuery` 的别名，因此即使不使用 `$` 也能保证所有功能性。比如你想要将 `jQuery` 嵌入一个高度冲突的环境，假如我们需要使用 `jQuery` 之外的另一 `JavaScript` 库，我们可以通过调用 `$.noConflict()` 向该库返回控制权。

需要注意的是在进行 Ajax 调用的使用将 `dataType` 选项设置成“`jsonp`”，而不是“`json`”，这

是关键的地方。console.log 原先是 Firefox 的“专利”，严格说是安装了 Firebugs 之后的 Firefox 所独有的调试“绝招”。这一招，IE8 学会了，不过用起来比 Firebug 麻烦，只有在开启调试窗口(F12)的时候，console.log 才能出结果，不然就报错。console.log()，可以用来取代 alert() 或 document.write()。比如，在网页脚本中使用 console.log("Hello World")，加载时控制台就会自动显示如下内容。

2.1.8 WCF RESTful 接口传输文件

一般情况下文件传输有如下 4 种方式。

Base64 byte[]

一般来说，最简单的方式就是利用 byte[] 作为数据类型来设计你的 Web Service。byte[] 无论是作为一个简单参数或者你的一个自定义对象的成员变量都可以。

客户端把文件内容全部读出，填入一个 byte[]，然后经过 base64 的 encode (这个工作一般由 Web Service 框架来完成)，然后发送给服务端，服务端先经过 base64 的 decode (同样，一般由框架完成)，然后应用程序就可以拿到 byte[] 了。

这个方法比较简单，但是其问题也不小，就是不适合传输大文件。原因 base64 加密后的 byte[] 实际上是作为 SOAP 报文 XML 的一部分的，其字节数大约比加密前的字节数多三分之一。要命的是这些内容会被全部整体读入内存 (因为 XML 报文解析)，如果文件很大，那么内存的占用就很厉害，想象系统一般是多用户共用的，是不是很容易闹出 OOM (Out of Memory) 的问题？

SOAP Attachments

类似 Email 的附件，SOAP 也可以采用类似的方式添加附件，即二进制的文件保持不变，不需要改变编码方式以迁就 XML，其内容不出现在 SOAP 的 XML 报文中。不过这种方式下对编程人员有点麻烦，你需要用 SAAJ 编程 API 来操作 AttachmentPart。

MTOM

结合以上两种方式的优点，既可以保持二进制形式的文件内容，又保持相对简单的编程模式。然而，值得注意的是 MTOM 和 SOAP Attachments 这两种方式对大文件都也存在内存问题，即大文件的内容字节 bytes 也是在内存中存放着的，如果文件很大，很容易出 OOM(Out of Memory) 的问题。

Streaming Mode

顾名思义，Streaming Mode 就是流模式，对于大文件而言，其内容并不整体保存在内存中，系统拿到一小撮数据后交给应用层，应用程序把这小撮数据保存到文件中或者作另外的处理，然

后再跟系统要下一小撮数据。。。如此往复，直至所有数据处理完毕。这种模式是对付大文件，避免 OOM(Out of Memory) 问题的最好的手段。

WCF 上传文件代码。

```
1  /// <summary>
2  /// WCF REST 接口传输文件
3  /// </summary>
4  /// <param name="url">WCF REST 链接地址 </param>
5  /// <param name="fileFullLocalPath"> 上传文件在本地的全路径 </param>
6  public virtual string RESTUpload(string url, string fileFullLocalPath)
7  {
8      string jsonResult = string.Empty;
9      try
10     {
11         HttpClient httpClient = new HttpClient();
12         HttpContent httpContent = HttpContent.Create(File.OpenRead(fileFullLocalPath));
13         HttpResponseMessage httpResponse = httpClient.Post(url, httpContent);
14         httpResponse.EnsureStatusIsSuccessful();
15         jsonResult = httpResponse.Content.ReadAsStringAsync();
16     }
17     catch (Exception e)
18     {
19         logger.Error("Upload failed!", e);
20     }
21     return jsonResult;
22 }
```

服务端代码如下所示。

```
1 [OperationContract]
2 [WebInvoke(Method="POST",
3     UriTemplate = "UploadJpegImage",
4     BodyStyle = WebMessageBodyStyle.Wrapped,
5     ResponseFormat = WebMessageFormat.Json)]
6 JpegFileInfoResult UploadJpegImage(Stream fileStream);
7
8 /// <summary>
9 /// 保存文件
10 /// </summary>
11 /// <param name="fileStream"> 文件流 </param>
12 public JpegFileInfoResult UploadJpegImage(Stream fileStream)
13 {
14     JpegFileInfoResult jpegFileInfoResult = null;
15     JpegFileInfo jpegFileInfo = null;
16     var storePath = string.Empty;
17     var fileFullName = string.Empty;
18     try
19     {
20         if (System.Web.HttpContext.Current != null)
21         {
```

```

22     storePath = System.Web.HttpContext.Current.Server.MapPath("~/upload/");
23 }
24 else
25 {
26     storePath = Path.Combine(HttpRuntime.AppDomainAppPath, "\\upload\\");
27 }
28 if (!Directory.Exists(storePath))
29 {
30     Directory.CreateDirectory(storePath);
31 }
32 string filename = System.Guid.NewGuid().ToString() + ".jpeg";
33 fileFullName = Path.Combine(storePath, filename);
34 var bitmap = Bitmap.FromStream(fileStream);
35 bitmap.Save(fileFullName);
36 }
37 catch (Exception e)
38 {
39     logger.Error("WCF receive image encount an error,fullpath:" + fileFullName, e);
40 }
41 return jpegFileInfoResult;
42 }
```

2.1.9 WebGet 和 WebInvoke

WebInvoke 允许您指定哪个动词将被允许，将其设置为 POST。WebGet 要求客户端使用 GET 请求。在任何情况下，如果使用了错误的动词，则会收到“方法未被允许”。您正在使用浏览器，因此正在发送 GET 请求，所以一个仅支持 POST 的 WebInvoke 将拒绝该请求，而一个 WebGet 则会接受该请求。您可以在 WebInvoke 属性上指定 Method="GET" 来允许 GET，当然可以。

WebInvokeAttribute 会确定服务操作响应的 HTTP 方法。默认情况下，所有已应用 WebInvokeAttribute 的方法都会响应 POST 请求。Method 属性允许您指定不同的 HTTP 方法。如果您希望服务操作响应 GET，请改用 WebGetAttribute。

2.1.10 WCF 开启日志

在开发过程中需要诊断 WCF 各种未知问题，日志是最好的帮助，在 Configuration 中插入以下节点，WCF 中客户端开启日志的配置如下。

```

1 <system.diagnostics>
2   <sources>
3     <source name="System.ServiceModel.MessageLogging" switchValue="Warning,
4       ActivityTracing">
5       <listeners>
6         <add type="System.Diagnostics.DefaultTraceListener" name="Default"
7           >
```

```

6         <filter type="" />
7     </add>
8     <add name="ServiceModelMessageLoggingListener">
9         <filter type="" />
10    </add>
11    </listeners>
12  </source>
13  <source name="System.ServiceModel" switchValue="Warning, ActivityTracing"
14    propagateActivity="true">
15    <listeners>
16      <add type="System.Diagnostics.DefaultTraceListener" name="Default"
17        >
18          <filter type="" />
19      </add>
20      <add name="ServiceModelTraceListener">
21          <filter type="" />
22      </add>
23      </listeners>
24    </source>
25  </sources>
26  <sharedListeners>
27    <add initializeData="app_messages.svclog"
28      type="System.Diagnostics.XmlWriterTraceListener, System, Version=2.0.0.0,
29      Culture=neutral, PublicKeyToken=b77a5c561934e089"
30      name="ServiceModelMessageLoggingListener" traceOutputOptions="Timestamp"
31      >
32        <filter type="" />
33    </add>
34    <add initializeData="app_tracelog.svclog"
35      type="System.Diagnostics.XmlWriterTraceListener, System, Version=2.0.0.0,
36      Culture=neutral, PublicKeyToken=b77a5c561934e089"
37      name="ServiceModelTraceListener" traceOutputOptions="Timestamp"
38      >
39        <filter type="" />
40    </add>
41  </sharedListeners>
42  <trace autoflush="true" />
43</system.diagnostics>

```

在 System.serviceModel 中插入以下节点，如代码片段2.4所示。

Listing 2.4: WCF 日志 serviceModel 节点配置

```

1  <diagnostics wmiProviderEnabled="true" performanceCounters="All">
2    <messageLogging logMalformedMessages="true" logMessagesAtTransportLevel="true" />
3  </diagnostics>

```

配置生效后会在程序的目录下生成 app_messages.svclog 和 app_tracelog.svclog 两个文件.

2.1.11 WCF 引用安全

WCF 的安全体系主要包括三个方面：传输安全（Transfer Security）、授权或者访问控制（Authorization OR Access Control）以及审核（Auditing）。而传输安全又包括两个方面：认证（Authentication）和消息保护（Message Protection）。认证帮助客户端或者服务确认对方的真实身份，而消息保护则通过签名和加密实现消息的一致性和机密性。WCF 采用两种不同的机制来解决这三个涉及到传输安全的问题，我们一般将它们称为不同的安全模式，即 Transport 安全模式和 Message 安全模式。Windows Communication Foundation(WCF) 是 Microsoft 为构建面向服务的应用程序而提供的统一编程模型，WCF 访问权限控制可采用自定义用户名密码的验证方式。WCF 提供了如下三种方式来验证凭证中用户名是否和密码相符：

- *Windows*: 将用户名和密码映射为 Windows 帐号和密码，采用 Windows 认证；
- *MembershipProvider*: 利用配置的 MembershipProvider 验证用户名和密码；
- 自定义：通过继承抽象类 UsernamePasswordValidator，自定义用户名/密码验证器进行验证。

WCF 支持多种认证技术，例如 Windows 认证、X509 证书、Issued Tokens、自定义用户名密码认证等，在跨 Windows 域分布的系统中，用户名密码认证还是比较常用的，要实现用户名密码认证，就必须需要 X509 证书，为什么呢？因为我们需要 X509 证书这种非对称密钥技术来实现 WCF 在 Message 传递过程中的加密和解密，要不然用户名和密码就得在网络上明文传递！详细说明就是客户端把用户名和密码用公钥加密后传递给服务器端，当然，做个测试程序就没有必要去申请一个 X509 数字签名证书了，微软提供了一个 makecert.exe 的命令专门用来生成测试使用的 X509 证书的，那我们就来建立一个测试用的证书，在 cmd 下输入以下命令：

```

1 C:\Program Files (x86)\Windows Kits\8.1\bin\x64>makecert.exe -sr currentUser -ss
2 My -a sha1 -n CN=WCFCertServer -sky exchange -pe //服务端证书
3
4 C:\Program Files (x86)\Windows Kits\8.1\bin\x64>makecert.exe -sr currentUser -ss
5 My -a sha1 -n CN=WCFCertClient -sky exchange -pe //客户端证书

```

makecert.exe 程序存放在 C:\Program Files (x86) \Windows Kits\8.1\bin\x64 路径下，证书创建完毕后通过 mmc (Microsoft Manager Control) 命令打开控制台界面可看到证书，如图2.6所示。

-sr 参数指定的证书存储区中的注册表位置。为 currentUser 时指定注册版存储位置为 HKEY_CURRENT_USER. 为 localMachine 时指定注册版存储位置为 HKEY_LOCAL_MACHINE. 在访问 WCF 时若出现无法使用以下搜索标准找到 X.509 证书：*StoreName* “My”、*StoreLocation* “LocalMachine”、*FindType* “FindBySubjectName”、*FindValue* “jirisoft” 或者密钥集不存在错误时，是由于没有给读取证书的权限，如图2.7所示。

给 Everyone 账户添加读取的权限即可，管理证书的菜单只有证书所属为 LocalMachine 时才有。服务器端再用自己的私钥来解密，然后传递给相应的验证程序来实现身份验证。

C:\WINDOWS\microsoft.net\Framework64\v4.0.30319

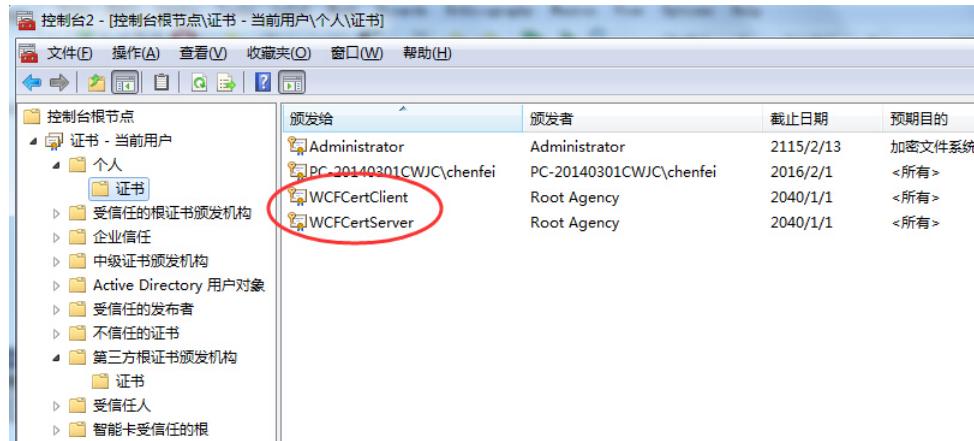


图 2.6: 创建 X.509 证书

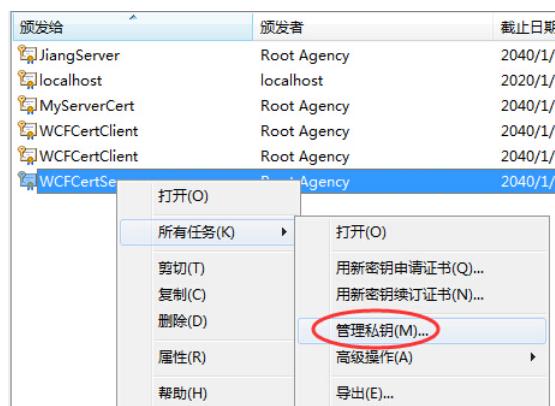


图 2.7: 管理 X.509 证书私钥

认证所需的数据表^{2.8}如图所示。

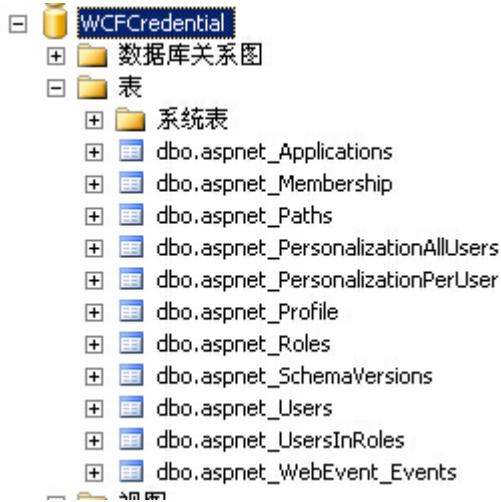


图 2.8: WCF 认证创建的数据表

在配置文件中添加数据库连接串：

```
1 <add name="WCFCredential" connectionString="Data Source=129.39.1.138,5002;Uid=
  WCFCredential;Pwd=WCFCredential!@#321;database=WCFCredential"/>
```

如果你有一个真正的 X509 证书，那么现在的代码就可以正常运行了。但是很不幸，我们的证书是测试用的，我们运行的时候出错： *X.509 certificate CN=MyServerCert 链生成失败*。所使用的证书具有无法验证的信任链。请替换该证书或更改 *certificateValidationMode*。已处理证书链，但是在不受信任提供程序信任的根证书中终止’，WCF 无法验证测试证书的信任链。那我们要做的就是绕过这个信任验证，具体做法如下：

WCF 安全之自定义 Windows 验证 应该说就采用的频率程度，集成 Windows 认证（IWA：Integrated Windows Authentication）是仅次于用户名/密码的认证方式。尤其是在基于 Windows 活动目录（AD： Active Directory）的 Intranet 应用来说，Windows 认证更是成为首选。微软几乎所有需要进行认证的产品或者开发平台都集成了 Windows 认证，比如 IIS，SQL Server，ASP.NET 等，当然，WCF 也不可能例外。基于 Windows 的身份验证服务端的 endpoint 配置如下：

```
1 <service behaviorConfiguration="CertificateBehavior" name="RR.API.WCFSERVICE.Custom.
  Service1">
  2   <endpoint behaviorConfiguration="" binding="wsHttpBinding"
  3     bindingConfiguration="CertificateBinding"
  4     name="Service1EndPoint"
  5     contract="RR.API.WCFSERVICE.Custom.IService1" />
  6
  7 </service>
```

behavior 配置如下：

```

1 <behavior name="CertificateBehavior">
2   <serviceMetadata httpGetEnabled="true" httpGetBinding="webHttpBinding"
3     httpGetBindingConfiguration="" />
4   <serviceDebug />
5   <serviceCredentials>
6     <serviceCertificate findValue="WCFCert1" storeLocation="LocalMachine"
7       x509FindType="FindBySubjectName" />
8   </serviceCredentials>
9 </behavior>

```

binding 的配置如下：

```

1 <wsHttpBinding>
2   <binding name="CertificateBinding" messageEncoding="Mtom">
3     <security>
4       <transport clientCredentialType="None" />
5       <message clientCredentialType="UserName" negotiateServiceCredential="false"/>
6     </security>
7   </binding>
8 </wsHttpBinding>

```

基于 Windows 的身份验证客户端的 endpoint 配置如下：

```

1 <endpoint address="http://192.168.1.254:5590/WCFService/Custom/Service1.svc"
2   binding="wsHttpBinding"
3   bindingConfiguration="Service1EndPoint"
4   behaviorConfiguration="CertificateBehavior"
5   contract="WCFValidateService.IService1"
6   name="Service1EndPoint">
7   <identity>
8     <certificate encodedValue="AwAAAAEAAAUAQABMqx3Nytjb4g0X+WAefU5NwHH10gAAAA
9       AQAAADgCAAAwggIOMIIB4qADAgECAhCMdbkl4fDgqkppsNbq0zC2MAkGBSs0AwIdBQAwFjEUMBI
10      GA1UEAxMLUm9vdCBZ2VuY3kwHhcNMTUwODA2MDkwNDUxWhcNMzkxMjMxMjM1OTU5WjATMREwDw
11      YDVQQDEwhXQOZDZXJ0MTCCASiWdQYJKoZIhvNAQEBBQADggEPADCCAQoCggEBALMOFmjx88RP
12      JUCGAdLOUBA2yB0/FiICDDHEagjCvKORWXsa+haON30EjZ5Nqv9jtq0xb2fgm/1FbyP23MaCp2
13      mL9McV7AFftu3m1/betkskUHZD5y/EUGzXFwyhGFezD5o6BFjn8zlcQYjU8aRIyFpQf0MeC062Z
14      tUd/92CyCsuzt8xsIXMm63i2yV46RhU5v0bgB/D40FVHuJUBbbrDYmRVqaUjIS2yq9DiX5+7oim
15      vz9CHQp6r3vnZ3pW8xcwE3zL/xEUNP+zs6ZLuOF/ad0SNYrDa0+EYAQ1FUVJfiZz35larGLDzd
16      ikqty4kJxHf7W/CxbiSaXMOgb2Fq8MCawEAAaNLMEmkwRwYDVR0BBEawPoAQEuQJLQYdHU8AjWEh
17      3BZkY6EYMBYxFDASBgNVBAMTC1Jvb3QgQWdlbmN5ghAGN2wAqgBkihHPuNSqXDXOMAkGBSs0AwI
18      dBQADQQAuCvXzvzI7Ec+VeU89Z09AKuLL9M5cEPUmri40z615FaqyIbA9LzvFp/nQ7fu1cPjK6C
19      0cqWWlk6vToLPWK8R" />
20   </identity>
21 </endpoint>

```

bindings 配置如下。

```

1 <wsHttpBinding>
2   <binding name="Service1EndPoint" messageEncoding="Mtom">

```

```

3   <security>
4     <message clientCredentialType="UserName" negotiateServiceCredential="false" />
5   </security>
6   </binding>
7 </wsHttpBinding>

```

behavior 的配置如下。

```

1 <behavior name="CertificateBehavior">
2   <clientCredentials>
3     <clientCertificate findValue="WCFCert1" storeLocation="LocalMachine"
4       x509FindType="FindBySubjectName" />
5     <serviceCertificate>
6       <defaultCertificate findValue="WCFCert1" storeLocation="LocalMachine"
7         x509FindType="FindBySubjectName" />
8       <authentication certificateValidationMode="None" trustedStoreLocation="
9         LocalMachine" />
10    </serviceCertificate>
11  </clientCredentials>
12 </behavior>

```

采用 Windows 用户名密码认证的调用方式如代码所示。

Listing 2.5: 认证方式调用 WCF 接口

```

1 Service1Client service1Client = new Service1Client();
2 service1Client.ClientCredentials.UserName.UserName = "Administrator";
3 service1Client.ClientCredentials.UserName.Password = "123";
4 string returnValue = service1Client.GetData(100);
5 service1Client.Close();

```

其中 Administrator 为登录 Windows 的用户名，123 为登录 Windows 的密码。

WCF 安全之自定义身份验证 在自定义 Windows 验证的基础上，重写 UserNamePasswordValidator 类。如下代码片段所示。

Listing 2.6: 自定义验证方式

```

1 public class WCFCustomValidator:UserNamePasswordValidator
2 {
3   public override void Validate(string userName, string password)
4   {
5     if(userName!="jiangxiaoqiang"||password!="jiangxiaoqiang")
6     {
7       throw new Exception("Unauthorized access!");
8     }
9   }
10 }

```

在 behavior 里面添加配置如下。

```

1 <behavior name="CertificateBehavior">
2   <serviceMetadata httpGetEnabled="true" httpGetBinding="webHttpBinding"
3     httpGetBindingConfiguration="" />
4   <serviceDebug />
5   <serviceCredentials>
6     <serviceCertificate findValue="WCFCert1" storeLocation="LocalMachine"
7       x509FindType="FindBySubjectName" />
8     <userNameAuthentication userNamePasswordValidationMode="Custom"
9       customUserNamePasswordValidatorType="RR.Labs.Utils.WCFCustomValidator,RR.
10      Labs" />
11   </serviceCredentials>
12 </behavior>

```

图形界面的配置如图所示。



图 2.9: 自定义验证界面配置

WCF 之 MemberShip 验证 在数据库中创建一个用于 WCF 认证的账户，如下代码所示。

Listing 2.7: MemberShip 创建用户

```

1 public void CreateUser()
2 {
3   if (Membership.FindUsersByName("jiangxiaoqiang").Count == 0)
4   {
5     Membership.CreateUser("jiangxiaoqiang", "jiangxiaoqiang", "
6       jiangtingqiang@gmail.com");
7   }
8 }

```

System.Web 节点的配置。

```

1 <system.web>
2   <compilation debug="true" targetFramework="4.0" />
3   <membership defaultProvider="SqlProvider">
4     <providers>
5       <clear />

```

```

6   <add name="SqlProvider"
7     type="System.Web.Security.SqlMembershipProvider"
8     connectionStringName="WCFCredential"
9     applicationName="demosite"
10    enablePasswordRetrieval="false"
11    enablePasswordReset="true"
12    requiresQuestionAndAnswer="false"
13    requiresUniqueEmail="true"
14    passwordFormat="Hashed" />
15  </providers>
16 </membership>
17 <httpRuntime maxRequestLength="20000000" executionTimeout="600"
18   requestValidationMode="2.0" />
</system.web>

```

2.1.12 WCF RESTful Security

在 SOAP 协议的 WCF 中，可以通过 SOAPHeader (MessageHeader) 来实现用户名密码的传输，早在 WebService 时代我们就这么用过了。在 REST WCF 中，我们可以利用 HttpHeaders 来完成这一目标。在每个服务契约里加上用户和密码的参数的设计肯定会成为你的噩梦。首先在服务端的服务中加入如^{2.8}方法用于校验 Header 的信息：如果 Header 中 Authorization 的字符串不是”123”那么就将返回 405 MethodNotAllowed 的错误。这个字符串的内容可以自定义，服务端根据某种规则检查这个字符串。

Listing 2.8: WCF 验证方法

```

1 private bool CheckAuthorization()
2 {
3   var ctx = WebOperationContext.Current;
4   var auth = ctx.IncomingRequest.Headers[HttpRequestHeader.Authorization];
5   if (string.IsNullOrEmpty(auth) || auth != "123")
6   {
7     ctx.OutgoingResponse.StatusCode = HttpStatusCode.MethodNotAllowed;
8     return false;
9   }
10  return true;
11 }

```

可以在服务提供的接口中调用此方法进行验证，如代码^{2.9}所示。

Listing 2.9: WCF 调用验证方法

```

1 public string GetData(string a)
2 {
3   if (!CheckAuthorization())
4   {
5     return "Access Failed!";
6   }
7   else

```

```

8  {
9      return "OK! You enter:" + a.ToString();
10 }
11 }
```

当服务接口较多时，不可能每个方法都调用 CheckAuthorization 来进行验证，以后的修改也会变得麻烦。为了避免此问题，采用自定义 HTTP 模块¹⁷（Custom HTTP Module）来对每个提交的 Request 请求的合法性进行验证。HTTP 自定义模块通常具有以下用途：第一就是安全。因为您可以检查传入的请求，所以 HTTP 模块可以在调用请求页、XML Web services 或处理程序之前执行自定义的身份验证或其他安全检查。在以集成模式运行的 Internet 信息服务(IIS)7.0 中，可以将 Forms 身份验证扩展到应用程序中的所有内容类型。在自定义模块的初始化方法中添加请求调用时触发的事件：

```

1 public void Init(HttpApplication context)
2 {
3     context.BeginRequest +=
4         (new EventHandler(this.Application_BeginRequest));
5 }
```

调用验证方法：

```

1 private void Application_BeginRequest(Object source, EventArgs e)
2 {
3     HttpApplication application = source as HttpApplication;
4     HttpContext context = application.Context;
5     if (context != null)
6     {
7         CheckAuthorization(context);
8     }
9 }
```

验证不通过时手动向客户端抛出错误：

```

1 private bool CheckAuthorization(HttpContext context)
2 {
3     string auth = context.Request.Headers["Authorization"];
4     if (string.IsNullOrEmpty(auth) || auth != "123")
5     {
6         throw new HttpException(404, "HTTP/1.1 404 Unauthorized");
7     }
8     return true;
9 }
```

HttpContext 封装了 ASP.NET 要处理的单次请求的所有信息。HttpContext 的生存周期：从客户端用户点击并产生了一个向服务器发送请求开始到服务器处理完请求并生成返回到客户端为止。针对每个不同用户的请求，服务器都会创建一个新的 HttpContext 实例直到请求结束，

¹⁷<https://msdn.microsoft.com/zh-cn/library/ms227673%28v=vs.100%29.aspx>

服务器销毁这个实例。HttpContext 类，它对 Request、Response、Server 等等都进行了封装，并保证在整个请求周期内都可以随时随地的调用。ASP.NET 中它还提供了很多特殊的功能。例如 Cache、还有 HttpContext.Item，通过它你可以在 HttpContext 的生存周期内提前存储一些临时的数据，方便随时使用。

采用 SHA1 签名认证

客户端通过 HttpWebRequest 提交 HTTP 请求，用 HttpWebResponse 接收请求，在 Request 的 Header 中附加签名的 Key-Value 信息。获取的签名字串还可以通过 MD5 加密一次，带签名的请求代码如2.10所示。

Listing 2.10: 客户端请求 WCF 接口

```

1  /// <summary>
2  /// GET 请求与获取结果 (带签名)
3  /// </summary>
4  /// <param name="url"> 请求的 WCF 地址 </param>
5  /// <param name="privateKey"> 生成签名的私钥 </param>
6  /// <param name="postDataParam"> 请求参数集合 </param>
7  public static string HttpGet(string url, string postDataParam, string privateKey)
8  {
9      string postUrl = url + (postDataParam == "" ? "?" : "?") + postDataParam;
10     HttpWebRequest request = (HttpWebRequest)WebRequest.Create(postUrl);
11     request.Method = "GET";
12     request.ContentType = "text/html; charset=UTF-8";
13     string signatureData = Encrypter.HashAndSignString(postUrl, privateKey);
14     request.Headers.Add("Authorization", signatureData);
15     HttpWebResponse response = (HttpWebResponse)request.GetResponse();
16     Stream myResponseStream = response.GetResponseStream();
17     StreamReader myStreamReader = new StreamReader(responseStream, Encoding.UTF8);
18     string resultString = myStreamReader.ReadToEnd();
19     myStreamReader.Close();
20     myResponseStream.Close();
21     return resultString;
22 }
```

url 的地址形如<http://192.168.1.254:5590/WCFService/Custom/Service1.svc/web/GetData>，postDataParam 形如 id=18 格式。服务端的验证方法如2.11所示。方法中将签名信息添加到请求头中，

Listing 2.11: 服务端验证请求

```

1  /// <summary>
2  /// 验证签名
3  /// </summary>
4  /// <param name="context">HttpContext 实例 </param>
5  /// <returns> 验证结果 </returns>
6  private bool CheckAuthorization(HttpContext context)
7  {
```

```

8   string signatureData = context.Request.Headers["Authorization"];
9   if (!string.IsNullOrEmpty(signatureData))
10  {
11      string publicKey = Developer.GetDeveloperPublicKeyByAppSecretKey("12");
12      string originalText = context.Request.Url.OriginalString;
13      bool sucess = SecurityUtil.VerifySigned(originalText, signatureData, publicKey);
14      if (!sucess)
15      {
16          throw new HttpException(404, "HTTP/1.1 404 Unauthorized");
17      }
18  }
19  else
20  {
21      //throw new HttpException(404, "HTTP/1.1 404 Unauthorized");
22      //context.Response.End();
23  }
24  return true;
25 }
```

在浏览服务时，需要将代码 throw new HttpException(404, "HTTP/1.1 404 Unauthorized")注释，否则报未找到资源错误。或者采用 context.Response.End() 语句结束此次请求。End 方法使 Web 服务器停止处理脚本并返回当前结果。文件中剩余的内容将不被处理。

WCF REST HTTPS

2.1.13 WCF 不使用 svc 文件

无 (.SVC) 文件服务激活(File-Less Activation)是 WCF 4.0 的新特性。在 system.ServiceModel 配置节下添加如下配置。

```

1 <serviceHostingEnvironment>
2   <serviceActivations>
3     <add relativeAddress="WCFSERVICE/Custom/RemoveSvcService.svc" service="RR.API.
4       WCFSERVICE.Custom.RemoveSvcService"/>
5   </serviceActivations>
6 </serviceHostingEnvironment>
```

service 为带命名空间的类的名称。service 和 factory 对用着原来定义在.svc 文件中 <%@ServiceHost> 指令的 Service 和 Factory 属性，而 relativeAddress 则表示服务相对服务寄宿的 IIS 站点的地址，该地址必须以.svc 为后缀。

2.1.14 WCF 自定义路由映射

WCF 自定义路由映射是将形如: `http://localhost/applicationname/Serivce.svc/Name` 的地址映射为形如: `http://localhost/applicationname/aa/Name` 的地址。首先在 Global.asax 中的 Application_Start 事件中调用以下方法:

```

1 public static void RegisterRoutes(RouteCollection routes)
2 {
3     routes.EnableFriendlyUrls();
4     WebServiceHostFactory factory = new WebServiceHostFactory();
5     RouteTable.Routes.Add(new ServiceRoute("aa", factory, typeof(ConferenceService)));
6 }

```

Global.asax 在程序第一次请求时访问，且只访问一次。使用 EnableFriendlyUrls 属性需要引用 Microsoft.AspNet.FriendlyUrls.dll。重定向成功后访问的页面如图2.10所示。



图 2.10: WCF Url 路由

原始的访问地址为:`http://192.168.1.254:5590/WCFService/Custom/RemoveSvcService.svc/web/help`，经过 Url 路由后的访问地址为: `http://192.168.1.254:5590/aa/web/help`。配置文件中添加如下配置。

```

1 <system.serviceModel>
2   <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
3 </system.serviceModel>

```

在添加路由之后无法访问初始页面，使用路由重置的方式有点类似 Web API，本身要出现元数据界面就是需要一个.svc，现在的很像一个一般处理程序或 Web API 所以没有界面可以预览。

自定义路由的类上添加如下内容：

```

1 [AspNetCompatibilityRequirements(RequirementsMode =
2   AspNetCompatibilityRequirementsMode.Allowed)]

```

2.1.15 WCF 传递 Cookie

这是一个服务器与服务器的通讯，客户端（web 应用）并不会自动发送 cookie 到 wcf。所以客户端还得做更多的工作核心在 IClientMessageInspector 这个接口，他有 BeforeSendRequest 和 AfterReceiveReply 两个方法。我们的目的是在 beforeSendRequest 时，在请求中加入 cookie

信息，在 AfterReceiveReply 方法中，从响应中获取要设置的 cookie，并真正设置到客户端。首先使用的绑定，必须允许 Cookie 传播

```

1 <wsHttpBinding>
2   <binding name="CertificateBinding"
3     messageEncoding="Mtom"
4     allowCookies="True">
5     <security>
6       <transport clientCredentialType="None" />
7       <message clientCredentialType="UserName" negotiateServiceCredential="false"/>
8     </security>
9   </binding>
10 </wsHttpBinding>
```

服务需要基于 ASP.NET 的激活进行 host。

```

1 <!--运行服务以ASP.NET激活模式激活服务-->
2 <serviceHostingEnvironment aspNetCompatibilityEnabled ="true"
  multipleSiteBindingsEnabled="true" />
```

服务的实现，基于属性的声明允许以 ASP.NET 的方式访问

```

1 [AspNetCompatibilityRequirements(RequirementsMode =
  AspNetCompatibilityRequirementsMode.Allowed)]
```

实现客户端的消息发送，响应拦截器。

```

1 public class CookieMessageInspector : IClientMessageInspector
2 {
3   public void AfterReceiveReply(ref Message reply, object correlationState)
4   {
5     return;
6   }
7
8   public object BeforeSendRequest(ref Message request, IClientChannel channel)
9   {
10    var cookie = "example";
11    HttpRequestMessageProperty httpRequestMessage;
12    object httpRequestMessageObject;
13    if (request.Properties.TryGetValue(HttpRequestMessageProperty.Name, out
14      httpRequestMessageObject))
15    {
16      httpRequestMessage = httpRequestMessageObject as
17      HttpRequestMessageProperty;
18      if (string.IsNullOrEmpty(httpRequestMessage.Headers["Cookie"]))
19      {
20        httpRequestMessage.Headers["Cookie"] = cookie;
21      }
22    }
23  }
```

```

22     {
23         httpRequestMessage = new HttpRequestMessageProperty();
24         httpRequestMessage.Headers.Add("Cookie", cookie);
25         request.Properties.Add(HttpRequestMessageProperty.Name, httpRequestMessage)
26         ;
27     }
28     return null;
29 }
```

实现 CookieBehavior。

```

1 public class CookieBehavior : IEndpointBehavior
2 {
3     public void AddBindingParameters(ServiceEndpoint endpoint,
4         BindingParameterCollection bindingParameters)
5     {
6         return;
7     }
8
9     public void ApplyClientBehavior(ServiceEndpoint endpoint, ClientRuntime
10        clientRuntime)
11     {
12         clientRuntime.MessageInspectors.Add(new CookieMessageInspector());
13     }
14
15     public void ApplyDispatchBehavior(ServiceEndpoint endpoint, EndpointDispatcher
16        endpointDispatcher)
17     {
18         return;
19     }
20
21     public void Validate(ServiceEndpoint endpoint)
22     {
23         return;
24     }
25 }
```

在服务器端接收 Cookie。

```

1 try
2 {
3     if (HttpContext.Current != null)
4     {
5         foreach (string s in HttpContext.Current.Request.Cookies.AllKeys)
6         {
7             }
8         }
9     else
10    {
```

```

11     logger.Info("Request is null!");
12 }
13 }
14 catch (Exception e)
15 {
16     logger.Error("aa", e);
17 }

```

回传 Cookie 时，需在服务端设置 Cookie，如下所示：

```
1 HttpContext.Current.Response.Cookies.Add(new HttpCookie("test123", "123"));
```

设置后回传的参数里会有 Set-Cookie，如图2.11所示。

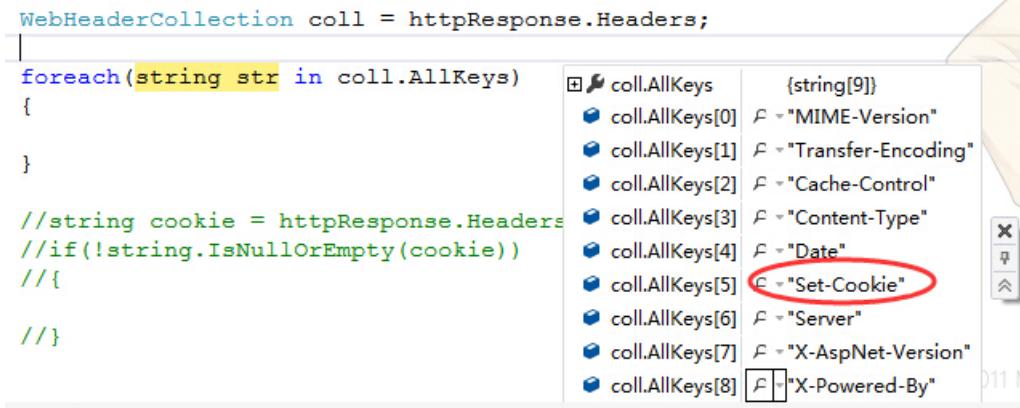


图 2.11: 回传参数中出现的 Set-Cookie

Tips: 在查看变量时，使变量锁定在当前页面也点击圆形的大头针图标。

2.1.16 WCF REST 传递 Cookie

在 WCF 里是可以使用 session 的。

在请求时添加如下语句：

```

1 CookieContainer Cookie = new CookieContainer();
2 request.CookieContainer = Cookie;

```

2.1.17 调取 HTTPS 类型的 Web Service

服务端的 Web Service 采用 Java 实现，在调取 HTTPS 类型的 Web Service 时跟调取普通的服务区别不大。在传入 HTTPS 类型的 URL 时提示：提供的 URI 方案 “https” 无效，应为 “http”。只需要将配置文件的安全配置节修改为 Transport 即可，Transport 代表采用 HTTPS 连接，如下代码片段所示。

表 2.1: Security Choice

Member Name	Description
None	The SOAP message is not secured during transfer. This is the default behavior.
Transport	Security is provided using HTTPS. The service must be configured with SSL certificates. The SOAP message is protected as a whole using HTTPS. The service is authenticated by the client using the service's SSL certificate. The client authentication is controlled through the ClientCredentialType.

```
1 <security mode="Transport" />
```

Transport seems to require HTTPS to encrypt credentials and throws an exception if there is no SSL. TransportCredentialsOnly will send the credentials in plain text and unencrypted and is recommended for testing ONLY! 所有的安全类型如表2.1所示：

详细可参考：[https://msdn.microsoft.com/en-us/library/system.servicemodel.basichttpsecuritymode\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/system.servicemodel.basichttpsecuritymode(v=vs.100).aspx)

2.2 WCF 异常处理

在 WCF 开发阶段，如果想在客户端访问 WCF 时看到服务端详细的错误信息，可增加配置。

```
1 <serviceBehaviors>
2   <behavior name="">
3     <serviceMetadata httpGetEnabled="true" />
4     <serviceDebug includeExceptionDetailInFaults="true" />
5   </behavior>
6 </serviceBehaviors>
```

如上，在 ServiceDebug 节点中增加 includeExceptionDetailInFaults，赋值为 true。Enterprise Lib 对 WCF 的支持中，Exception Block 中还特地有针对 WCF 程序异常处理的解决方案，而且满足以上说道的需求，即可记录异常，又可对异常信息进行封装。更重要的是，自动处理运行时的异常信息，不需要挨个方法的去写 Try catch。秉承企业库的优秀传统，大部分工作还是通过配置就可以完成了，非常好的解决方案。Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.dll

Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.WCF.dll

Microsoft.Practices.EnterpriseLibrary.Common.dll

Microsoft.Practices.ObjectBuilder2.dll

安装完毕后 Microsoft Enterprise Library 的目录在 C:\Program Files (x86)\Microsoft Enterprise Library 5.0\Bin 下面。

利用 Attribute 和 IErrorHandler 处理 WCF 全局异常

实现 IErrorHandler 接口，实现他的两个方法。

Listing 2.12: 实现 IErrorHandler 接口

```
1 public class GlobalExceptionHandler : IErrorHandler
2 {
3     /// <summary>
4     /// 日志实例
5     /// </summary>
6     private readonly log4net.ILog logger = log4net.LogManager.GetLogger(MethodBase.
    GetCurrentMethod().DeclaringType);
7
8     /// <summary>
9     ///
10    /// </summary>
11    /// <param name="error"></param>
12    /// <returns> 如果不应中止会话，则为 true；否则为 false。默认值为 false。</returns>
13    public bool HandleError(Exception error)
14    {
15        return true;
16    }
17
18    /// <summary>
19    ///
20    /// </summary>
21    /// <param name="error"></param>
22    /// <param name="version"></param>
23    /// <param name="fault"></param>
24    public void ProvideFault(Exception error, MessageVersion version, ref Message fault)
25    {
26        logger.Error("WCF异常", error);
27        /*定义抛给客户端的错误*/
28        var clientException = new FaultException(string.Format("WCF接口出错 {0}", error.
    TargetSite.Name));
29        MessageFault msgFault = clientException.CreateMessageFault();
30        fault = Message.CreateMessage(version, msgFault, clientException.Action);
31    }
32}
```

需要创建一个自定义的 Service Behaviour Attribute，让 WCF 知道当 WCF 任何异常发生的时候，通过这个自定义的 Attribute 来处理。实现这个需要继承 IServiceBehavior 接口，并在此类的构造函数里，我们获取到错误类型。一旦 ApplyDispatchBehavior 行为被调用时，通过 Activator.CreateInstance 创建错误 handler，并把这个错误添加到每个 channelDispatcher 中。channelDispatcher 中文叫信道分发器，当我们的 ServiceHost 调用 Open 方法，WCF 就会创

建我们的多个信道分发器（ChannelDispatcher），每个 ChannelDispatcher 都会拥有一个信道监听器（ChannelListener），ChannelListener 就有一直在固定的端口监听，等到 Message 的到来，调用 AcceptChannel 构建信道形成信道栈，开始对 Message 的处理。

Listing 2.13: 实现 IServiceBehavior 接口

```

1  public class GlobalExceptionHandlerBehaviourAttribute : Attribute, IServiceBehavior
2  {
3      private readonly Type _errorHandlerType;
4
5      public GlobalExceptionHandlerBehaviourAttribute(Type errorHandlerType)
6      {
7          _errorHandlerType = errorHandlerType;
8      }
9
10     public void AddBindingParameters(ServiceDescription serviceDescription,
11         ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
12         BindingParameterCollection bindingParameters)
13     {
14
15     }
16
17     public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
18         ServiceHostBase serviceHostBase)
19     {
20         var handler = (IErrorHandler)Activator.CreateInstance(_errorHandlerType);
21         foreach (ChannelDispatcherBase dispatcherBase in serviceHostBase.
22             ChannelDispatchers)
23         {
24             var channelDispatcher = dispatcherBase as ChannelDispatcher;
25             if (channelDispatcher != null)
26             {
27                 channelDispatcher.ErrorHandlers.Add(handler);
28             }
29         }
30     }
31
32 }
```

最后在 WCF 的实现类上，加上 GlobalExceptionHandlerBehaviour。

```

1 [GlobalExceptionHandlerBehaviour(typeof(GlobalExceptionHandler))]
2 public class PublicBaseService : IPublicBaseService
3 {
4     public string DoWorkWithReturnValue(string a)
5     {
```

```

6     return a;
7 }
8
9     public void DoWork(string a)
10    {
11        int ia = 1;
12        int b = 0;
13        int c = ia / b;
14    }
15 }
```

实现类中引发了一个除以 0 的异常，在全局中可以捕获到此异常，如此在实现的代码中可省去 try{}catchfinally 语句。

2.2.1 常见问题

安全包中没有可用的凭证 将服务端的 clientCredentialType 改为 UserName 方式，如代码所示。

```

1 <wsHttpBinding>
2   <binding name="NewBinding0" messageEncoding="Mtom">
3     <security>
4       <transport clientCredentialType="None" />
5       <message clientCredentialType="UserName" negotiateServiceCredential="false"/>
6     </security>
7   </binding>
8 </wsHttpBinding>
```

客户不能确定服务主体名称基于身份的目标地址 http://myServerUrl” SspiNegotiation / Kerberos 的目的。目标地址标识必须是 UPN 标识（如 acmedomain/alice）或 SPN 身份（如主机/ bobs-machine）。禁用服务端的 negotiateServiceCredential，如代码所示¹⁸。

```

1 <wsHttpBinding>
2   <binding name="NewBinding0" messageEncoding="Mtom">
3     <security>
4       <transport clientCredentialType="None" />
5       <message clientCredentialType="UserName" negotiateServiceCredential="false"/>
6     </security>
7   </binding>
8 </wsHttpBinding>
```

传入参数不能够和返回参数混合使用 部署 WCF 时遇到如下错误：

¹⁸ http://www.cnblogs.com/aritech/archive/2011/06/12/Authentication_043.html

“The operation could not be loaded because it has a parameter or return type of type System.ServiceModel.Channels.Message or a type that has MessageContractAttribute and other parameters of different types. When using System.ServiceModel.Channels.Message or types with MessageContractAttribute, the method must not use any other type of parameters” .

这个错误说明 WCF 的传入参数不能够和返回参数混合使用，比如传入的参数是基础类型，那么返回的参数也必须是基础类型，传入的参数是对象，返回的参数也只能够是对象。

主要存在如下 3 中混合：

- MixType1: Contract type and primitive types as operation parameters
- MixType2: Contract type as a parameter and primitive type as return type
- MixType3: Primitive type as a parameter and Contract type as return type

远程服务器返回了意外响应：(400) Bad Request 将远程服务调用的方法由 GET 改为 POST，如代码所示。

Listing 2.14: 改变调用方式

```

1 [OperationContract]
2 [WebInvoke(Method="POST",UriTemplate = "GetAwardCount",
3     ResponseFormat = WebMessageFormat.Json,
4     BodyStyle = WebMessageBodyStyle.Wrapped)]
5 string GetAwardCount();
```

没有终结点在侦听可以接受消息的 `http://192.168.1.254:5590/WCFService/Custom/PublicBaseService.svc`。这通常是由于不正确的地址或者 SOAP 操作导致的。如果存在此情况，请参见 `InnerException` 以了解详细信息。出现此问题一般是服务端和客户端所选的绑定不一致导致。

此工厂上启用了手动寻址，因此发送的所有消息都必须进行预寻址 添加 `behaviorConfiguration="webBehavior"`，如下代码所示。

```

1 <behaviors>
2   <endpointBehaviors>
3     <behavior name="webBehavior">
4       <webHttp/>
5     </behavior>
6   </endpointBehaviors>
7 </behaviors>
8 <client>
9   <endpoint name="employeeService"
10      address="http://127.0.0.1:3721/employees"
11      behaviorConfiguration="webBehavior"
```

```
12     binding="webHttpBinding"
13     contract="Artech.WcfServices.Service.Interface.IEmployees"/>
14 </client>
```

远程服务器返回了意外响应: (405) Method Not Allowed 配置问题, 一般通过客户端调用需要配置为 basicHttpBinding。

无法使用以下搜索标准找到 X.509 证书

元数据包含无法解析的引用

在服务端的 service 配置节下添加如下配置即可。

```
1 <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
```

响应消息的内容类型 text/html; charset=utf-8 与绑定 (multipart/related; type="application/xop+xml") 的内容类型不匹配。如果使用自定义编码器, 请确保正确实现 IsContentTypeSupported 方法。响应的前 1024 个字节为: “<html>

在浏览器中浏览地址试试能否成功浏览。

接口页面无法找到

WCF 无法访问路由页面

Chapter 3

ASP.NET Web API(Apache License 2.0)

ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework. 在调用 Web API 时，传入的参数名称需要和服务器参数名称一致，例如服务器的参数为 function (string value)，那么调用是也必须在 Url 的参数集合中以 value=1 的形式，名称必须一致。GlobalConfiguration.Configuration 中 GlobalConfiguration 类在 System.Web.Http.WebHost.dll 中。

3.1 Web Api 帮助文档

3.1.1 Controller 分类

Step 2: 继承 DefaultHttpControllerSelector The second part of the solution is to create an implementation of IHttpControllerSelector that actually uses the area name.

3.2 Web Api Area 支持

3.3 Web Api 路由

请求路径以字符串“api”开头的时候将访问 webAPI 的函数（注：至于为什么用 MapHttpRoute 而不是 MapRoute；为什么用 routeTemplate 而不是用 url 我们再以后的章节介绍）。因为 routeTemplate 中有了 controller，所以针对 api 的请求可以自动映射到指定的 controller 类。

3.4 参数传递

3.4.1 Web Api Json 传递

http://blog.csdn.net/besley/article/details/23955147?utm_source=tuicool&utm_medium=referral

3.4.2 错误处理 (Exception Handling in ASP.NET Web API)

You can customize how Web API handles exceptions by writing an exception filter. An exception filter is executed when a controller method throws any unhandled exception that is not an HttpResponseException exception. The HttpResponseException type is a special case, because it is designed specifically for returning an HTTP response. Exception filters implement the System.Web.Http.Filters.IExceptionFilter interface. The simplest way to write an exception filter is to derive from the System.Web.Http.Filters.ExceptionFilterAttribute class and override the OnException method.

Listing 3.1: Web API 错误处理

```

1 public class NotImplExceptionFilterAttribute : ExceptionFilterAttribute
2 {
3     public override void OnException(HttpActionExecutedContext context)
4     {
5         if (context.Exception is NotImplementedException)
6         {
7             context.Response = new HttpResponseMessage(HttpStatusCode.NotImplemented
8         );
9     }
10 }

```

3.4.3 消息处理 (HTTP Message Handlers in ASP.NET Web API)

A message handler is a class that receives an HTTP request and returns an HTTP response. Message handlers derive from the abstract `HttpMessageHandler` class.

Typically, a series of message handlers are chained together. The first handler receives an HTTP request, does some processing, and gives the request to the next handler. At some point, the response is created and goes back up the chain. This pattern is called a delegating handler¹.

3.4.4 Model 绑定 (Model Binding)

Model binding is the process that maps data from request and supply to Action methods parameters. Model binding takes places after the authorization filters have executed and model is ready to bind. Model binder works with Value providers to get and map data. By default there are four value providers that MVC framework uses, Form Data, Route Data, Query string and posted Files. You can create your own custom value provider to choose data from, such as Cookie Value provider as custom value provider.

Model Binder is implemented from `IModelBinder` interface. By default, MVC provides a very powerful model binder but you can create your own custom model binder for your specific project needs. Model 绑定不仅可以根据参数名（简单类型）从 `HTTPRouteData` 的 `value` 属性中提取同名的路由变量值作为参数值。在 ASP.NET MVC 中，用户请求道服务器的数据将被包装为 Model 数据对象，这个数据对象通常也被 View 用来提供显示的数据。在 ASP.NET MVC 中，提供了非常灵活的 Model 绑定机制，通过 `IModelBinder` 借口，定义了绑定 Model 数据的约定，并提供了一个接口的默认实现 `DefaultModelBinder`。在大多数情况下，仅仅通过 `DefaultModelBinder(System.Web.ModelBinding)` 就可以完成 Model 的绑定。默认情况下，ASP.NET MVC 使用 `DefaultModelBinder` 来绑定 Model 的数据。在传递 Action 参数的时候，ASP.NET MVC 按照如下顺序查找匹配的数据：

¹<http://www.asp.net/web-api/overview/advanced/http-message-handlers>

- Form 表单中的数据
- RouteData 中的数据
- QueryString 中的数据

3.5 Automated Documentation for REST APIs

3.5.1 安装 Swagger-UI

使用如下的命令安装 Swagger-UI。

```
1 Install –Package Swashbuckle
```

3.5.2 Swagger

The goal of Swagger™ is to define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interfaces have done for lower-level programming, Swagger removes the guesswork in calling the service. Swagger-UI 本身只提供在线测试功能，要集成它还需要告诉它本项目提供的各种服务和参数信息。

Listing 3.2: Swagger 链接

```
1 http://localhost:5509/swagger/ui/index
```

要让接口的内容显示，需要开启 Web API Tracing，在 WebApiConfig.cs 文件的 Rigester 方法中添加如下语句。

Listing 3.3: 开启 Tracing

```
1 config.EnableSystemDiagnosticsTracing();
```

需要注意的是 EnableSystemDiagnosticsTracing 为 HttpConfiguration 实例的扩展方法，需要引用动态链接库 System.Web.Http.Tracing，另外需要注意的是 Swagger 对文件 WebApiConfig.cs 的名字大小写敏感。测试页面虽然可以正常工作，但是接口说明和参数说明都是空着的，显得不够友好，但这些实际是可以从代码的 XML 注释中读取出来的。要实现这一效果，需要进行如下两步。首先，开放注释的 XML 文档的输出，如图3.1所示。

第二步就是将 WebApi 的注释 (Comments) 输出文档在 SwaggerConfig.cs 类中的 Register 方法进行指定，如下代码片段所示。

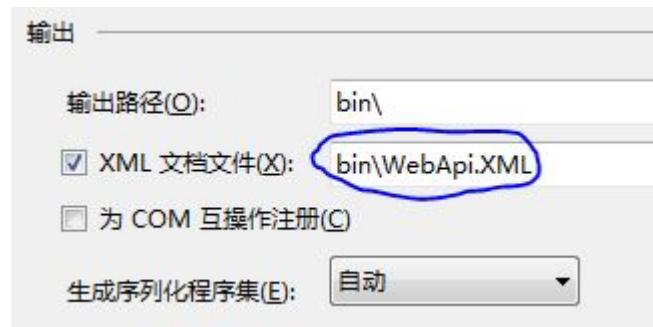


图 3.1: 输出注释

Listing 3.4: Swagger 指定注释文档的位置

```

1 c.IncludeXmlComments(XmlCommentsPath());
2
3 private static string GetXmlCommentsPath()
4 {
5     return System.String.Format(@"{0}\bin\WebApi.XML",System.AppDomain.
6         CurrentDomain.BaseDirectory);
7 }
```

WebApi 生成注释后的效果如图3.2所示，所有的 Api 文档目前的粒度为 Controller，Advert 为 Controller 的名称，GetAdvertise 为 Action 的名称，往后项目复杂后可以根据模块分离出 Area。

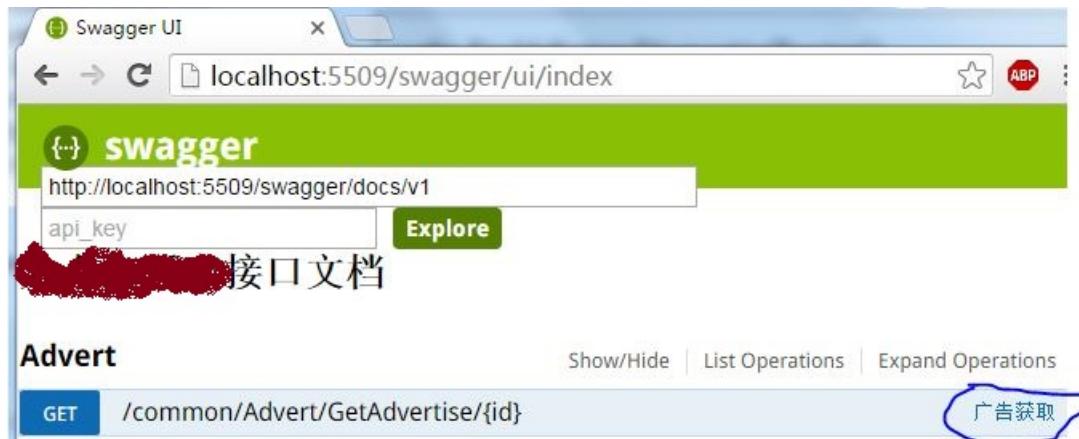


图 3.2: 注释的效果

Chapter 4

数据操作 (Data Operate)

4.1 生成二维码 (Generate QRCode)

生成二维码需要引用 ThoughtWorks.QRCode.dll 文件，使用里面的 QRCodeEncoder 类，初始化 QRCodeEncodeMode、QRCodeScale、QRCodeVersion、QRCodeErrorCorrect 属性。如下语句生成二维码：

Listing 4.1: 生成二维码

```

1 #region 创建二维码
2 /// <summary>
3 ///
4 /// </summary>
5 /// <param name="data"> 二维码内容 </param>
6 /// <returns></returns>
7 public string CreateQrCodeWithRelativePath(string data)
8 {
9     var savePath = string.Empty;
10    var qrCodeEncoder = new QRCodeEncoder
11    {
12        QRCodeEncodeMode = QRCodeEncoder.ENCODE_MODE.BYTE,
13        QRCodeScale = 12,
14        QRCodeVersion = 0,
15        QRCodeErrorCorrect = QRCodeEncoder.ERROR_CORRECTION.M
16    };
17    try
18    {
19        var fileName = Guid.NewGuid() + ".png";
20        var tempPath = Path.Combine(HttpRuntime.AppDomainAppPath, "qrcode\\");
21        if (!Directory.Exists(tempPath))
22        {
23            Directory.CreateDirectory(tempPath);
24        }
25        savePath = tempPath + fileName;
26        qrCodeEncoder.Encode(data, Encoding.UTF8).Save(savePath);
27    }
28    catch (Exception e)
29    {
30        PublicAttribute.Logger.Error("Generate QRCode encount an error", e);
31    }
32    return savePath;
33 }
34 #endregion

```

方法中获取 Web 程序当前运行的主目录的语句为：

```
1 HttpContext.Current.Server.MapPath("/")
```

但是此种方法获取路径并不安全，因为有时 HttpContext.Current 为 null，如定时器的回调、Cache 的移除通知、APM 模式下异步完成回调、主动创建线程或者将任务交给线程池来执

行时。所以尽量避免使用 MapPath, HttpRuntime.AppDomainAppPath 才是更安全的选择。此处获取的目录为: D:\项目\zouwo\RR.DataCommon\bin\Debug, 传入不同的参数获取不同的路径, 具体如下:

```

1 Server.MapPath("/") //返回应用程序根目录所在的位置 如 C:\Inetpub\wwwroot\
2 Server.MapPath("~/") //表示当前应用级程序的目录, 如果是根目录, 就是根目录, 如果是虚拟目
   录, 就是虚拟目录所在的位置 如: C:\Inetpub\wwwroot\Example\
3 注: 等效于Server.MapPath("~/").
4 Server.MapPath("./") //返回当前目录绝对路径
5 Server.MapPath("../") //返回上一级目录的绝对路径

```

推荐使用一种更加安全的方式, 代码如下:

```

1 string filePath=System.IO.Path.Combine(HttpRuntime.AppDomainAppPath,"\\qrcode\\a.
   png");

```

HttpRuntime 下的除了 WEB 中可以使用外, 非 WEB 程序也可以使用。而 HttpContext 则只能用在 WEB 中。因此, 在可能的情况下, 我们尽可能使用 HttpRuntime。

- Path.Combine 中如果其中一个参数为 null, 会抛出异常
- 如果指定的路径之一是零长度字符串, 则该方法返回其他路径。两个都是零长度字符串, 则返回的就是 string.Empty
- 如果 path2 包含绝对路径, 则该方法返回 path2
- path2 不能以\和/开头的字符串, 如果是这个字符串开头的, 则返回 path2

4.1.1 获取程序运行路径

有时需要从磁盘上读取配置文件, 此时需要指定配置文件的相对路径, 在 Windows Service 中获取当前的程序运行路径并获得配置文件相对路径如下代码片段所示。

```

1 var currentDirectory = AppDomain.CurrentDomain.BaseDirectory;
2 var configFilePath = currentDirectory + @"Config\quartz_jobs.xml";

```

4.2 序列化 (Serialization) 与反序列化 (Deserialization)

从有赞系统¹ 查询出的订单数据返回为 Json²串, Json 串的简化版本如下代码片段所示:

```

1 {
2   "response": {

```

¹有赞, 曾用名口袋通, 旨在为商户提供强大的微店铺和完整的微电商解决方案。是一个免费的微商城平台。有赞是一个以产品技术为主的团队, 2012 年开始一直专注于有赞这个产品, 所属公司为杭州起码科技有限公司。

²JavaScript Object Notation: JavaScript 对象标记

```

3     "total_results": 33,
4     "trades": [
5         {
6             "tid": "E231958349",
7             "num": 1,
8             "buyer_type": 1,
9             "seller_flag": "5",
10            "orders": [
11                {
12                    "oid": 14776,
13                    "num_iid": 3424234,
14                    "sku_id": 32221,
15                    "discount_fee": 0,
16                    "payment": 200.02,
17                    "buyer_messages": [
18                        {
19                            "title": "\u989c\u8272\u8981\u6c42",
20                            "content": "\u7ea2\u8272\u7684\u6216\u8005\u9ec4\u8272\
21                            u7684"
22                        },
23                        {
24                            "title": "\u989c\u8272\u8981\u6c42",
25                            "content": "\u7ea2\u8272\u7684\u6216\u8005\u9ec4\u8272\
26                            u7684"
27                    ],
28                }
29            }
30        }

```

此处需要将返回的 Json 串转化为对应的有赞系统的 Model，再将 Model 映射为本系统的 Model 持久化到数据库中。根据 Json 串的结构，定义 ResponseContainer、ResponseModel、TradeModel、OrderModel，每一个 Model 分别对应 Json 串的 response、trades、orders，在定义时需要注意每个属性的名称需要和 Json 串中的名称完全一致。每个 Model 的定义如下代码片段所示：

```

1 public class ResponseContainer
2 {
3     public ResponseModel response { get; set; }
4 }

```

```

1 public class ResponseModel
2 {
3     public int total_results { set; get; }
4     public List<TradeModel> trades{ set; get; }
5 }

```

如下代码片段是交易的 Model:

```

1 public class TradeModel
2 {
3     #region Attribute
4
5     #region 订单
6     private List<OrderModel> orders;
7
8     public List<OrderModel> Orders
9     {
10         get { return orders; }
11         set { orders = value; }
12     }
13     #endregion
14
15     #endregion
16 }
```

如下代码片段是订单的 Model:

```

1 public class OrderModel
2 {
3     #region Attribute
4
5     #region 价格
6     private string price;
7
8     public string Price
9     {
10         get { return price; }
11         set { price = value; }
12     }
13     #endregion
14 }
```

将 Json 转换为 Model 的代码如下代码片段所示, 通过调用 Newtonsoft.Json.dll 中的 JsonConvert 方法:

```

1 #region Json反序列化为ResponseModelContainer
2 /// <summary>
3 /// Json 反序列化为 ResponseModelContainer
4 /// </summary>
5 /// <param name="json"></param>
6 /// <returns></returns>
7 public static ResponseContainer MapResponseJsonToContainer(string json)
8 {
9     ResponseContainer container = new ResponseContainer();
10    try
11    {
```

```

12     container = JsonConvert.DeserializeObject<ResponseContainer>(json);
13 }
14 catch(Exception e)
15 {
16     logger.Error("Json转换过程中遇到错误",e);
17 }
18 return container;
19 }
#endregion

```

在 Json 与微信交互时，微信接收的 Json 的 key 只能小写（截止文章撰写时），而在 C# 中定义属性时采用驼峰命名法更合适，所以在传入 Json 请求时需要统一将 Json 的 key 转化为小写³。

Listing 4.2: Json 序列化时 key 转换为小写

```

1 public class LowercaseJsonSerializer
2 {
3     private static readonly JsonSerializerSettings Settings = new JsonSerializerSettings
4     {
5         ContractResolver = new LowercaseContractResolver()
6     };
7
8     public static string SerializeObject(object o)
9     {
10         return JsonConvert.SerializeObject(o, Formatting.Indented, Settings);
11     }
12
13     public class LowercaseContractResolver : DefaultContractResolver
14     {
15         protected override string ResolvePropertyName(string propertyName)
16         {
17             return propertyName.ToLower();
18         }
19     }
20 }

```

使用小写转换方法：

Listing 4.3: 转换实例

```

1 var json = LowercaseJsonSerializer.SerializeObject(new { Foo = "bar" });
2 // "foo": "bar"

```

4.2.1 序列化 (Serialization) 字典值

程序有时需要动态的字典类型 (Key-Value) 的配置项，一般的序列化方法不支持字典 (Dictionary) 类型的序列化，可使用 DataContractSerializer 进行序列化，输出的 XML 文件需

³参考链接:<http://stackoverflow.com/questions/6288660/net-ensuring-json-keys-are-lowercase#>

要是可读格式 (Readable, 不是所有的配置都是一行), 可以采用 XmlWriterSettings 进行设置。

Listing 4.4: 将字典序列化为 XML 文件

```

1  /// <summary>
2  /// 序列化辅助方法
3  /// </summary>
4  /// <typeparam name="T"></typeparam>
5  /// <param name="instance"> 需要序列化的对象 </param>
6  /// <param name="fileName"> 保存的文件名 (全路径) </param>
7  private static void Serialize<T>(T instance, string fileName)
8  {
9      var serializer = new DataContractSerializer(typeof(T));
10     /*输出为良好缩进的格式化的XML数据*/
11     var xmlWriteSettings = new XmlWriterSettings { Indent = true };
12     using (var writer = XmlWriter.Create(fileName, xmlWriteSettings))
13     {
14         serializer .WriteObject(writer, instance);
15     }
16 }
```

4.2.2 反序列化 XML

4.2.3 XML 增加内容

SelectSingleNode 方法总是返回为 null, 原因就在于上面的 xml 文档中使用了命名空间, 当 xml 中定义了命名空间时, 在查找节点的时候需要使用下面的方法。

4.2.4 XML 删除内容

4.3 数据转换

4.3.1 将 DataTable 转为 List

在从数据库中获取数据库的时候, 我们经常会返回一个 DataTable 类型, 然后将其转换为 List 集合。代码如下:

Listing 4.5: 将 DataTable 转换为 List

```

1  #region 将DataTable转换为List
2  /// <summary>
3  ///
4  /// </summary>
5  /// <typeparam name="T"></typeparam>
6  /// <param name="dt"></param>
7  /// <returns></returns>
8  public static List<T> ConvertDataTableToList<T>(DataTable dt) where T : class,new()
```

```

9  {
10 var propertyList = new List< PropertyInfo >();
11 Type type = typeof(T);
12 Array.ForEach(
13     type.GetProperties(),
14     PropertyInfo =>
15     {
16         if (dt.Columns.IndexOf(p.Name) != -1)
17         {
18             propertyList.Add(propertyInfo);
19         }
20     });
21 var objectList = new List< T >();
22 foreach (DataRow row in dt.Rows)
23 {
24     var singleObject = new T();
25     propertyList.ForEach(
26         PropertyInfo =>
27         {
28             if (row[p.Name] != DBNull.Value)
29             {
30                 PropertyInfo.SetValue(singleObject, row[propertyInfo.Name], null);
31             }
32         });
33     objectList.Add(singleObject);
34 }
35 return objectList;
36 }
37 #endregion

```

此处在转换的过程中使用了泛型，使用泛型可以有如下方面的优势：

- 泛型可以使代码更加简洁、清晰
- 提升程序的性能
- 类型安全

其中 T 代表任意类型 (枚举除外)，ConvertDataTableToList <T>，说明我们在调用这个方法的时候，同时要赋予方法名一个类型值，这个类型要和它的返回值类型一致 (泛型是类型安全的)，Where：用于限制 T 的条件，例如 where T : class,new() 表示 T 只能是一个类，或者一个类型对象。此方法在转换的过程中，会出现类型之间转换失败的问题，在设置值的时候将值转换成实际的类型。SetValue 语句改为：

```

1 PropertyInfo.SetValue(objectT, Convert.ChangeType(row[propertyInfo.Name], PropertyInfo.
    PropertyType), null);

```

ChangeType(Object, TypeCode) 方法将 Object 更改为 TypeCode 参数指定的类型（如果可能），ChangeType 往往用在不知道当前类型应当是什么的情况下进行类型的转换，如有个泛

型方法

```
1 T GetObject<T>(string str)
```

要求从 string 类型转换为指定的 T 类型，此时只能

```
1 return (T)Convert.ChangeType(str, typeof(T));
```

因为 str 显式转为 T 肯定是不行的，只能 ChangeType。

4.3.2 Double 转换

double.TryParse 和 double.Parse 两者最大的区别是，如果字符串格式不满足转换的要求，Parse 方法将会引发一个异常；TryParse 方法则不会引发异常，它会返回 false，同时将 result 置为 0。实际上，早期的 FCL⁴ 中并没有提供 TryParse 方法，那时只能调用 Parse 方法，如果转换类型失败，则要将值设定为一个初始值，同时必须要捕获异常，代码如下所示：

```
1 string str = string.Empty;
2 double d;
3 try
4 {
5     d = double.Parse(str);
6 }
7 catch (Exception ex)
8 {
9     d = 0;
10}
```

Convert.ToDouble 调用此方法 ToDouble(Char) 始终引发 InvalidCastException。

4.3.3 List 与 string 的转换

在开发中经常会用 List<string> 来保存一组字符串，比如下面这段代码：

```
1 List<string> studentNames = new List<string>();
2
3 studentNames.Add("John");
4 studentNames.Add("Mary");
5 studentNames.Add("Rose");
```

可是有时候，我们要从中获取一个字符串，字符串的内容就是集合中的内容，但是要用逗号隔开，下面的办法可以实现：

```
1 string.Join(", ", studentNames.ToArray());
```

⁴Framework Class Library, 即 Framework 类库。

输出 John, Mary, Rose。

4.3.4 获取集合的列

```

1 var questionIdCollectionList = from qids in questionnaire.QuestionsModel.
2   QuestionModelList
3   select qids.ID;
4 string questionIdCollection = string.Join(", ", questionIdCollectionList.ToArray());

```

4.3.5 去除 List 中重复的对象

List 的 Distinct 扩展方法 要实现对象的相等比较，需要实现 IEquatable<T>，或单独写一个类实现 IEqualityComparer<T> 接口。

```

1 public class WantGoPlaceCompare : IEqualityComparer<tour_wantToPlacesModel>
2 {
3   public bool Equals(tour_wantToPlacesModel x, tour_wantToPlacesModel y)
4   {
5     return x.objId == y.objId;
6   }
7
8   public int GetHashCode(tour_wantToPlacesModel obj)
9   {
10    return obj.objId.GetHashCode();
11  }
12}

```

Distinct 函数在比较时，先比较的 GetHashCode 值，扩展函数 Distinct 在内部使用了一个 Set<T> 的类来帮助踢掉重复数据，而这个内部类使用的是 hash 表⁵的方式存储数据，所以会调用到我们自定义类的 GetHashCode 函数，如果返回的 hashCode 值不等，它就不会再调用 Equals 方法进行比较了。使用 List 的 Distinct 方法来剔除重复数据如下代码片段所示：

```

1 //去除 List 重复对象
2 wantGoIds = wantGoIds.Distinct(new WantGoPlaceCompare()).ToList();

```

4.3.6 List 中的元素排序

使用 Linq 进行排序 如下的代码按照司机出发时间 (Starttime) 降序 (descending) 排列。

```

1 var sortedDriverActionInfos = from items in driverActionInfos orderby items.Starttime
2   descending select items;

```

⁵散列表 (Hash table, 也叫哈希表)，是根据关键码值 (Key value) 而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

4.3.7 Unix 时间戳格式转换

经常发现很多地方使用一个时间戳⁶表示时间。比如：1370838759 表示 2013 年 6 月 10 日 12:32:39。而更多地方使用如 2013-6-10 12:32:39 格式的时间，代码4.6实现两种格式之间的互相转换。

Listing 4.6: 时间戳与 DateTime 转换

```

1  /// <summary>
2  /// 时间戳转为 DateTime 格式时间
3  /// </summary>
4  /// <param name="timeStamp">Unix 格式时间戳 </param>
5  /// <returns>DateTime 格式时间 </returns>
6  private DateTime StampToDate(string timeStamp)
7  {
8      DateTime dateTimeStart = TimeZone.CurrentTimeZone.ToLocalTime(new DateTime
9          (1970, 1, 1));
10     long totalSecondTime = long.Parse(timeStamp + "0000000");
11     TimeSpan toNow = new TimeSpan(totalSecondTime);
12     return dateTimeStart.Add(toNow);
13 }
14 /// <summary>
15 /// DateTime 时间格式转换为 Unix 时间戳格式
16 /// </summary>
17 /// <param name="time"> 需要转换的 DateTime 时间 </param>
18 /// <returns>Unix 格式时间戳 </returns>
19 private long DateTimeToStamp(System.DateTime time)
20 {
21     System.DateTime startTime = TimeZone.CurrentTimeZone.ToLocalTime(new System.
22         DateTime(1970, 1, 1));
23     return (long)(time - startTime).TotalSeconds;
}

```

4.3.8 Linq 操作

Linq 分页 使用 Linq 分页，如下代码所示：

```

1  #region Linq获取OrderModel集合
2  /// <summary>
3  ///
4  /// </summary>
5  /// <param name="PageSize"></param>
6  /// <param name="CurPage"></param>

```

⁶时间戳，又叫 Unix Stamp. 从 1970 年 1 月 1 日 (UTC/GMT 的午夜) 开始所经过的秒数，不考虑闰秒。

```

7  ///<param name="objs"></param>
8  ///<returns></returns>
9  private List<shop_orderModel> QueryByPage(int PageSize, int CurPage, List<
10   shop_orderModel> objs)
11 {
12     var query = from oneItem in objs select oneItem;
13     return query.Take(PageSize * CurPage).Skip(PageSize * (CurPage - 1)).ToList();
14 }
#endregion

```

用以上方法确实能够得到分页的目的，但是当数据量大的情况下查询数据还是挺慢的，因为我们是先把所有数据取出来后在进行分页的，而不是在数据库内就分页。分页仅仅是把传入的对象按照要求进行排列，可以不用关心具体的分页对象，此方法可以做如下改进：

```

1  #region Linq获取sellerModel集合
2  ///<summary>
3  ///
4  ///</summary>
5  ///<param name="PageSize"></param>
6  ///<param name="CurPage"></param>
7  ///<param name="objs"></param>
8  ///<returns></returns>
9  private List<T> QueryByPage<T>(int PageSize, int CurPage, List<T> objs)
10 {
11     var query = from oneItem in objs select oneItem;
12     return query.Take(PageSize * CurPage).Skip(PageSize * (CurPage - 1)).ToList();
13 }

```

采用泛型的方式做到分页时与传入的具体对象无关。Linq 中的 take 方法从序列开头传回指定的连接项目数目。skip 方法跳过序列中指定数量的元素，然后返回剩余的元素。

Linq 非空判断 有时在用 Linq 时，需要确定搜索的元素不为空，Linq 如下：

```

1  var resultRows = (from DataGridViewRow row in gridAlarmInfo.Rows.Cast<
2   DataGridViewRow>().Where(row => row.Cells[12].Value != null)
3   where row.Cells[12].Value.ToString() == e.SerialNo.ToString()
4   select row).ToList();

```

Linq 中的 Any 和 All 方法 LINQ 提供了两个布尔方法：Any() 和 All()，它们可以快速确定对于数据而言，某个条件是 true 还是 false。Any 用于判断集合中是否有元素满足某一条件；不延迟。（若条件为空，则集合只要不为空就返回 True，否则为 False）。有 2 种形式，分别为简单形式和带条件形式，下面是一个示例。

```

1  var wantGoResult = from query in alreadyGoneModel
2  where query.played == 1 && query.collectionStatus == 1
3  select query;

```

```
4 var isContains = wantGoResult.Any<tour_wantToPlacesModel>();
```

以上代码通过 Linq 语句根据条件查询一个集合 alreadyGoneModel，使用 any 方法（返回 bool）获得这个集合中是否有满足条件的元素（tour_wantToPlacesModel）。All 方法则确定某一集合中是否所有的元素都满足某一特定条件。以上语句可以简化为：

```
1 var containsAlreadyGone = alreadyGoneModel.Any(alreadyGone => alreadyGone.played ==  
1 && alreadyGone.collectionStatus == 1);
```

取出查询出来的实体直接用 foreach 循环结果即可。

Dictionary 转换为 List

```
1 var vechicleList = vechicleDicitonary.Select(vechicle => vechicle.Value).ToList();
```

Linq 查询

```
1 from prod in db.Products  
2 select prod.UnitPrice  
3 .Min()
```

4.3.9 分页查询语句

SQL Server 可用如下语句进行分页查询，此种分页语句适合 SQL Server 2005 及以上版本：

```
1 SELECT TOP 页大小 *  
2 FROM  
3 (  
4     SELECT ROW_NUMBER() OVER (ORDER BY id) AS RowNumber,* FROM  
5         table1  
6 ) A  
6 WHERE RowNumber > 页大小*(页数-1)
```

页大小：每页的行数；页数：第几页。使用时，请把“页大小”和“页大小*(页数-1)”替换成数字。一般情况下分页过程为：传入分页查询 SQL，在数据库中获取到分页后的结果，结果以 DataTable 的形式返回，再将 DataTable 转换为 List 集合展示在前台页面。ROW_NUMBER() OVER() 为开窗函数，在开窗函数出现之前存在着很多用 SQL 语句很难解决的问题，很多都要通过复杂的相关子查询或者存储过程来完成。为了解决这些问题，在 2003 年 ISO SQL 标准加入了开窗函数，开窗函数的使用使得这些经典的难题可以被轻松的解决。目前在 MS SQL Server、Oracle、DB2 等主流数据库中都提供了对开窗函数的支持。与聚合函数一样，开窗函数也是对行集组进行聚合计算，但是它不像普通聚合函数那样每组只返回一个值，开窗函数可以为每组返回多个值，因为开窗函数所执行聚合计算的行集组是窗口。在 ISO SQL 规定了这样的函数为开窗函数，在 Oracle 中则被称为分析函数，而在 DB2 中则被称为 OLAP 函数。

开窗函数的调用格式为：

函数名(列) OVER(选项)

OVER 关键字表示把函数当成开窗函数而不是聚合函数。SQL 标准允许将所有聚合函数用做开窗函数，使用 OVER 关键字来区分这两种用法。开窗函数 COUNT(*) OVER() 对于查询结果的每一行都返回所有符合条件的行的条数。OVER 关键字后的括号中还经常添加选项用以改变进行聚合运算的窗口范围。如果 OVER 关键字后的括号中的选项为空，则开窗函数会对结果集中的所有行进行聚合运算。

4.3.10 事务

在代码中使用事务需要 TransactionScope 类，TransactionScope 存在于 System.Transactions 命名空间中，它是从 Framework 2.0 开始引入的一个事务管理类，它也是微软推荐使用的一个事务管理类。在 TransactionScope 的构造函数中会自动创建了一个新的 LTM(轻量级事务管理器)，并通过 Transaction.Current 隐式把它设置为环境事务。在使用隐式事务时，事务完成前程序应该调用 TransactionScope 的 Complete() 方法，把事务提交，最后利用 Dispose() 释放事务对象。若执行期间出现错误，事务将自动回滚。在使用事务时操作系统需要启动 MSDTC⁷服务，可通过命令 net start msdtc 启动。使用如下代码所示。

```

1  using (TransactionScope transactionScope = new TransactionScope())
2  {
3      try
4      {
5          //事务操作
6          transactionScope.Complete();
7      }
8      catch (Exception e)
9      {
10         logger.Error("", e);
11     }
12 }
13 return success;

```

事务操作完毕调用 Complete() 方法进行提交，遇到错误未执行 Complete() 时会自动回滚。

在抽取奖品的过程中，为了防止多人同时抽奖（并发）时造成数据混乱，保证抽取的奖品记录在高并发的情况下不被其他进程修改，添加了行级锁，sql 语句写法如下：

```

1  SELECT *
2  FROM <TABLENAME>
3  WITH (ROWLOCK,UPDLOCK,READPAST)
4  WHERE CONDITION

```

ROWLOCK 表示使用行级锁，而不使用粒度更粗的页级锁和表级锁。UPDLOCK 读取表时使用更新锁，而不使用共享锁，并将锁一直保留到语句或事务的结束。UPDLOCK 的优点是

⁷msdtc.exe 是微软分布式传输协调程序。该进程调用系统 Microsoft Personal Web Server 和 Microsoft SQL Server。该服务用于管理多个服务器。msdtc:Microsoft Distributed Transaction Coordinator

允许您读取数据（不阻塞其它事务）并在以后更新数据，同时确保自从上次读取数据后数据没有被更改。READPAST 跳过锁定行。此选项导致事务跳过由其它事务锁定的行（这些行平常会显示在结果集内），而不是阻塞该事务，使其等待其它事务释放在这这些行上的锁。如果第一次查询出行 1，在第一次操作未结束时，如有第二次查询会跳过第一次查询的行，也即是第二次查询出的结果集中不包含第一次查询的行记录。READPAST 锁提示仅适用于运行在提交读隔离级别的事务，并且只在行级锁之后读取。仅适用于 SELECT 语句。

Granularity and isolation level and mode are orthogonal.

Granularity = what is locked = row, page, table (PAGLOCK, ROWLOCK, TABLOCK)

Isolation Level = lock duration, concurrency (HOLDLOCK, READCOMMITTED, REPEATABLEREAD, SERIALIZABLE)

Mode = sharing/exclusivity (UPDLOCK, XLOCK)

”combined” eg NOLOCK, TABLOCKX

4.3.11 随机查询

随机查询在 SQL 中的写法为：

```

1 select top 2 *
2 from discount_seller
3 where isRecommend=1
4 order by newid()
```

此查询语句的作用为在标中符合 isRecommend=1 的结果集中随机选择 2 个结果。

4.3.12 自定义异常 (Custom Exception)

在 C# 中所有的异常类型都继承自 System.Exception，也就是说，System.Exception 是所有异常类的基类。总起来说，其派生类分为两种：一是 SystemException 类，所有的 CLR 提供的异常类型都是由 SystemException 派生。 ApplicationException 类：由用户程序引发，用于派生自定义的异常类型，一般不直接进行实例化。在程序中可以简单的进行异常的抛出，在自定义异常类中统一进行异常的处理，而不需要在类中进行日志的记录，也算是封装变化点的实现方式之一（因为记录日志的工具可能发生变化，那样就不需要到每一个类中去修改关于日志的代码，在异常类中统一进行修改即可），自定义异常如代码片段4.7所示。

Listing 4.7: 自定义异常

```

1 public class RestfulNullDataException : Exception
2 {
3     public RestfulNullDataException()
4     {
5     }
6 }
```

```

7
8     public RestfulNullDataException(string message)
9     {
10         PublicAttribute.Logger.Error(message);
11     }
12
13     public RestfulNullDataException(string message, Exception inner)
14         : base(message, inner)
15     {
16         PublicAttribute.Logger.Error(message, inner);
17     }
18 }
```

4.3.13 根据经纬度计算距离

根据地球之间 2 点的经纬度计算 2 点之间的距离，Google 提供的方法，公式为：

$$S = 2 \arcsin \sqrt{\sin^2 \left(\frac{a}{2} \right) + \cos(Lat1) \times \cos(Lat2) \times \sin^2 \frac{b}{2}} \times 6378.1$$

公式中经纬度均用弧度表示，角度到弧度的转化应该是很简单的了吧，若不会，依然请参考这个这个经纬度算距离的工具；Lat1 Lng1 表示 A 点纬度和经度，Lat2 Lng2 表示 B 点纬度和经度（不要弄错顺序）； $a = \text{Lat1} - \text{Lat2}$ 为两点纬度之差 $b = \text{Lng1} - \text{Lng2}$ 为两点经度之差；6378.137 为地球半径，单位为公里；计算出来的结果单位为公里；计算的代码如下所示：

```

1 #region 根据经纬度计算2点之间距离
2 /// <summary>
3 /// 角度到弧度的转化
4 /// </summary>
5 /// <param name="d"></param>
6 /// <returns></returns>
7 private static double rad(double d)
8 {
9     return d * Math.PI / 180.0;
10 }
11
12 /// <summary>
13 ///
14 /// </summary>
15 /// <param name="lat1">A 点纬度 </param>
16 /// <param name="lng1">A 点经度 </param>
17 /// <param name="lat2">B 点纬度 </param>
18 /// <param name="lng2">B 点经度 </param>
19 /// <returns> 返回距离 (千米 km) </returns>
20 public static double GetDistance(double lat1, double lng1, double lat2, double lng2)
21 {
22     double a = rad(lat1) - rad(lat2);
23     double b = rad(lng1) - rad(lng2);
24     double distance = 2 * Math.Asin(Math.Sqrt(Math.Pow(Math.Sin(a / 2), 2) +
```

```

25     Math.Cos(radLat1) * Math.Cos(radLat2) * Math.Pow(Math.Sin(b / 2), 2)));
26     distance = distance * EARTH_RADIUS;
27     distance = Math.Round(distance * 10000) / 10000;
28     return distance;
29 }
30 #endregion

```

其中地球半径常数为：

```

1 /// <summary>
2 /// 地球半径（单位：km）
3 /// </summary>
4 private const double EARTH_RADIUS = 6378.137;

```

验证是否计算准确数据集。

```

1 /*
2  * 石桥铺地铁站坐标：106.484957,29.532313(精度，纬度)
3  * 歇台子地铁站坐标：106.498388,29.535303(精度，纬度)
4  * 距离：1.35km
5 */

```

4.3.14 为数组元素添加单引号

在做 SQL 查询时，有时需要将字符串变量置于单引号中作为查询条件，快速将字符串数组添加单引号如代码片段4.8所示。

Listing 4.8: 为字符串数组添加单引号

```

1 string condition = string.Empty;
2 int[] a = { 12, 13, 14, 15 };
3 condition = a.Aggregate(condition, (current, i) => current + ("'" + i + "'", ")).Trim(',');
4 this.Text = condition;
5
6 输出结果为：'12','13','14','15'

```

这里是将数组、集合中的元素拼接为带引号的字符串，方便放入 SQL 的 in 操作符中，这个语法可以做一些复杂的聚合运算，例如累计求和，累计求乘积。它接受 2 个参数，一般第一个参数是称为累积数（默认情况下等于第一个值），而第二个代表了下一个值。第一次计算之后，计算的结果会替换掉第一个参数，继续参与下一次计算。

4.3.15 获取 foreach 循环索引

在 C# 开发中往往使用 foreach 循环语句来代替 for 循环语句。foreach 比 for 更加简洁高效。for 语句直接就存在索引变量，但在实际操作中，使用 foreach 有时需要用到索引。最容易

想到的解决方法是在 foreach 语句外面定义索引变量，然后在 foreach 语句内自加，以此获取索引。例如：

Listing 4.9: foreach 获取当前索引号

```

1 int i = 0;
2 foreach(var item in arr)
3 {
4     i++;
5     item ....
6 }
```

这样是实现了，但是简单地使用 IndexOf 函数就可以获取到索引值，更加简洁，如代码片段 4.10：

Listing 4.10: foreach 获取当前索引号

```

1 foreach(var item in arr)
2 {
3     int index = arr.IndexOf(item); //index 为索引值
4     item ....
5 }
```

还可以使用 Lambda 表达式获取索引号，如代码片段 4.11。

Listing 4.11: foreach 获取当前索引号

```

1 foreach (var roadLineModel in roadLineModels.Select((currentModel, currentIndex) => new
2             { currentModel, currentIndex }))
3 {
4     @(roadLineModel.currentIndex)//当前索引号
5     @(roadLineModel.currentModel.lineId)//当前实体
6 }
```

4.3.16 byte 操作

不論是 Socket or Serial 都是要使用 byte[] 格式，所以學會轉 byte[] 是寫程式需要處理的第一步。将 byte[] 轉换为 string：

```

1 var str = System.Text.Encoding.Default.GetString(result);
```

将 byte[] 轉换为十六进制 (Hexadecimal) 字符串：

```

1 public static string ByteArrayToString(byte[] byteArr)
2 {
3     string hex = BitConverter.ToString(byteArr);
4     return hex.Replace("-", "");
5 }
```

将十六进制字符转换为 byte[], 如下代码片段所示:

```

1 public byte[] StringToByteArray(string hex) {
2     return Enumerable.Range(0, hex.Length)
3         .Where(x => x % 2 == 0)
4         .Select(x => Convert.ToByte(hex.Substring(x, 2), 16))
5         .ToArray();
6 }
```

将整型 int 转换为 byte[], 也可以适用于将 ushort 类型转换为 byte[], 阵列索引值越低, 存放高位元组, 这称为 Big Endian, 字串由左至右阅读, 转换后, 阵列索引值越低, 存放低位元组, 这称为 Little Endian, 使用 BitConverter 转换默认情况下为 Little Endian, Little Endian 和 Big Endian 只是排列分布顺序颠倒而已, 就看 Server 吃什么样的顺序, 网络字节序: TCP/IP 各层协议将字节序定义为 Big-Endian, 因此 TCP/IP 协议中使用的字节序通常称之为网络字节序。我们可以透过 Array.Reverse 方法来颠倒阵列的顺序, 如下代码片段所示:

```

1 byte[] bytes = BitConverter.GetBytes(i);
2 //如果是小端, 转化为大端, 因为 TCP/IP 定义的是大端 (Big-Endian)
3 if(BitConverter.IsLittleEndian) Array.Reverse(hex)
```

互联网使用的网络字节顺序采用大端模式进行编址, 而主机字节顺序根据处理器的不同而不同, 如 PowerPC 处理器使用大端模式, 而 Pentium 处理器使用小端模式。The order of bytes in the array returned by the GetBytes method depends on whether the computer architecture is little-endian or big-endian. 将 12 位手机号转换为 6 字节数组 BCD 码:

```

1 public byte[] ConvertToBcd6(string mobileNo)
2 {
3     var mobileArray = new byte[6];
4     if (mobileNo.Length != 12) return mobileArray;
5     for (var i = 0; i < 6; i++)
6     {
7         mobileArray[i] = Convert.ToByte(mobileNo.Substring(i * 2, 2), 16);
8     }
9     return mobileArray;
10}
```

Decimal 转换为 BCD 较通用的将 Decimal 转换为 BCD.

```

1 private IEnumerable<Byte> DecimalToBcd(Decimal value)
2 {
3     Byte currentByte = 0;
4     Boolean odd = true;
5     while (value > 0)
6     {
7         if (odd)
8             currentByte = 0;
9             Decimal rest = value % 10;
```

```

10     value = (value - rest) / 10;
11     currentByte |= (Byte)(odd ? (Byte)rest : (Byte)((Byte)rest << 4));
12     if (!odd)
13         yield return currentByte;
14     odd = !odd;
15   }
16   if (!odd)
17     yield return currentByte;
18 }
```

4.3.17 获取字符串中的数字

获取字符串中的数字方法如下：

```

1 public string GetNumberFromString(string mixedString)
2 {
3     var resultNumber = string.Empty;
4     for (var i = 0; i < mixedString.Length; i++)
5     {
6         if (char.IsNumber(mixedString, i))
7         {
8             resultNumber += mixedString.Substring(i, 1);
9         }
10     else
11     {
12         if (mixedString.Substring(i, 1) == ",")
13         {
14             resultNumber += mixedString.Substring(i, 1);
15         }
16     }
17 }
18 return resultNumber;
19 }
```

4.4 Windows 服务 (Windows Service)

实现从有赞系统定时将订单数据导入本地数据库，导入的同时，对购买商品的用户发送短信通知，具体的整体流程如图27.4所示：

4.4.1 Demo

服务需要继承自 ServiceBase 类，服务的应用程序入口点如下代码片段所示：

```

1 static void Main()
2 {
```

```

3  ServiceBase[] ServicesToRun;
4  ServicesToRun = new ServiceBase[]
5  {
6      new OrderImportService()
7  };
8
9  ServiceBase.Run(ServicesToRun);
10 }

```

其中 OrderImportService 为实现服务的类。

4.4.2 调试服务

可以通过如下命令操作 Windows 服务。

```

1 -----安装Windows服务-----
2 C:\Windows\Microsoft.NET\Framework64\v4.0.30319>InstallUtil.exe -i D:\项目\zouwo\
   RRInterop\bin\Debug\RRInterop.exe
3
4 -----卸载Windows服务-----
5 C:\Windows\Microsoft.NET\Framework64\v4.0.30319>InstallUtil.exe -u D:\项目\zouwo\
   RRInterop\bin\Debug\RRInterop.exe
6
7 -----启动Windows服务-----
8 sc start OrderImportService
9
10 -----停止Windows服务-----
11 sc stop OrderImportService

```

其中利用了 InstallUtil 工具。该工具的路径一般为: C:/Windows/Microsoft.NET/, 命令参数 i 即是 install 的缩写, u 即是 uninstall 的缩写。sc 为 Service Controller 的缩写。与 net.exe 实用程序相比, sc.exe 实用程序的功能更强大, 可以检查服务的实际状态, 或者配置、删除以及添加服务。

4.4.3 修改配置文件

修改配置文件的方法为:

```

1 #region change configuration file
2 /// <summary>
3 /// change configuration file
4 /// </summary>
5 /// <param name="createdTime"></param>
6 public void ChangeConfiguration(string createdTime)
7 {
8     var exePath = System.AppDomain.CurrentDomain.BaseDirectory + @"\"RRInterop.exe";

```

```

9  var appDomainConfigFile = AppDomain.CurrentDomain.SetupInformation.
10 ConfigurationFile;
11 var config = ConfigurationManager.OpenExeConfiguration(exePath);
12 var appSettings = (AppSettingsSection)config.GetSection("appSettings");
13 appSettings.Settings.Remove("queryTime");
14 appSettings.Settings.Add("queryTime", createdTime);
15 config.Save();
16 ConfigurationManager.RefreshSection("configuration");
17 }
#endregion

```

保存后总是出现一个新文件 App.config.config, StackOverFlow 上的解释⁸是:

This is because OpenExeConfiguration takes the exe name as input and automatically looks for a config file name exe.config (This is the default format).

是因为修改配置文件时程序寻找的默认文件名格式为 exe.config 进行修改, 这里要修改配置文件实际需要修改命名格式为**程序名称 +exe.config** 文件, 且保存时自动添加 config 后缀, 所以 exePath 写入程序名称即可。在获取程序运行路径 exePath 时, 应调用方法 System.AppDomain.CurrentDomain.BaseDirectory。调用 System.Environment.CurrentDirectory⁹ 时获取的是路径 C:\System\Windows32。

4.4.4 读取配置文件

读取配置文件时出现错误: 未将对象引用设置到对象的实例。可以通过如下语句查看当前配置文件所在路径:

```

1 AppDomain domain = AppDomain.CurrentDomain;
2 string configFilePath = domain.SetupInformation.ConfigurationFile;

```

此处使用单元测试工具 NUnit, 启动项目为 NUnit 的界面, 所使用的配置文件并不是 bin 目录下的配置文件。配置文件的输出的路径为 D:/项目 zouwo/RRInterop/bin/Debug/Project1.config。可知是 Project1.config 配置文件中没有连接串导致的此问题。另在 CS 类型项目运行中配置文件信息默认从启动程序 exe 的配置文件中获取, 例如 RRMail.WxPrint.Dal 的配置信息从 RRMail.WxPrint.exe.config 中读取。

4.5 字符串格式化 (Format)

```

1 //250,000.00, 默认保留 2 位小数, 如果为 0:N1, 则默认保留 1 位小数, N 为 Number 的缩写
2 string result=string.Format("{0:N}", 250000);

```

⁸详见《细信息，请参考》链接: <http://stackoverflow.com/questions/30836963/why-the-program-generate-a-new-configuration-file>

⁹Gets or sets the fully qualified path of the current working directory.

4.6 参数验证

4.6.1 代码契约 (Code Contract)

Microsoft 在.NET 4.0 中正式引入契约式编程库。契约式编程是一种相当不错的编程思想，它不但可以使开发人员的思维更清晰，而且对于提高程序性能很有帮助。值得一提的是，它对于并行程序设计也有莫大的益处。

Contract 的基本使用包括 Requires 和 Ensures，Requires 在方法开始时检查初始条件是否满足，通常用来做参数验证。Ensures 方法用来在方法结束时检查执行结果是否符合预期，比如可以放在 Property set 方法的末尾检查 Property 是否被正确设置。插件里的 ccrewrite 工具会将 Contract 方法编译成有效的检查代码分别注入函数体的首尾。所以即使你把 Contract.Ensures 检查放在函数开头部分（这也是推荐做法），编译之后这部分逻辑依然会出现在函数末尾，检查函数结束条件是否满足。

需要注意的是，如果想要在 Debug 和 Release Build 都使用运行时验证功能，则需要在项目设置为 Debug 和 Release 编译时，分别设置打开 Runtime check。

Contract 有一个很酷的 feature，就是在接口里定义一些检查，要求所有的实现都满足这些检查条，这样就不用在接口的每个实现里分别定义相同的检查逻辑了，非常的优雅，也符合 Declaration Programming 的初衷，如下代码所示。

```

1 public void Invoke(string input)
2 {
3     Contract.Ensures(!string.IsNullOrEmpty(input));
4 }
```

此处明确调用此方法的前置条件是参数 input 不能为空，如果传入了非法的参数，则会引发 System.Diagnostics.Contracts._ContractsRuntime.ContractException 类型的异常，程序可以在合适的地点处理异常。

4.6.2 验证小时分钟

```

1 #region 验证 (小时-分钟时间格式)
2 /// <summary>
3 ///
4 /// </summary>
5 /// <param name="shutdownTime"></param>
6 /// <returns></returns>
7 [TestCase("19:23", Result = true)]
8 [TestCase("09:00", Result = true)]
9 [TestCase("25:55", Result = false)]
10 [TestCase("00:00", Result = true)]
11 [TestCase("03:77", Result = false)]
12 [TestCase("", Result = false)]
```

```

13 | public static bool IsValidShutdownTimeFormat(string shutdownTime)
14 |
15 | {
16 |     return Regex.IsMatch(shutdownTime, @"^(([0-1]\d)|(2[0-4])):[0-5]\d$");
17 | }
#endregion

```

验证采用了正则表达式 (Regular Expression)，^ 匹配开头，\$ 匹配结尾，[0-1]\d 匹配 0 到 19 之间的数字，2[0-4] 匹配 20 到 23 之间的数字。[0-5]\d 匹配 0 到 59 之间的数字。

4.6.3 生成 GUID

注意：这里的 D, N, B, P 是不区分大小写的，如果传入空字符串，则使用的默认的 D 类型，其它情况都会报异常。

```

1 var guid = Guid.NewGuid();
2
3 //10244798-9a34-4245-b1ef-9143f9b1e68a
4 Console.WriteLine(guid.ToString("D")); //D:hyphens, 连字符, 连 (字) 号
5
6 //102447989a344245b1ef9143f9b1e68a
7 Console.WriteLine(guid.ToString("N")); //N:digits, 数字 ( digit 的名词复数) 手指, 足趾
8
9 //{{10244798-9a34-4245-b1ef-9143f9b1e68a}
10 Console.WriteLine(guid.ToString("B")); //B:brace
11
12 //{{(10244798-9a34-4245-b1ef-9143f9b1e68a)
13 Console.WriteLine(guid.ToString("P")); //P:parentheses

```

4.7 代码生成 (Code Generate)

4.7.1 CSharpCodeProvider

Listing 4.12: CSharpCodeProvider 代码自动生成

```

1 /// <summary>
2 /// 编译源代码
3 /// </summary>
4 /// <param name="sourceCodeContent"></param>
5 /// <param name="outAssemblyPath">DLL 输出目录 </param>
6 /// <param name="referencedAssemblies"> 编译时需引用的程序集 </param>
7 /// <returns></returns>
8 public static bool CompileUtil(string sourceCodeContent, string outAssemblyPath,
9                                IEnumerable<string> referencedAssemblies)
10 {
11     var compileSuccess = true;
12     var compileInfo = string.Empty;

```

```

12  var csharpCodeProvider = new CSharpCodeProvider();
13  var codeCompiler = csharpCodeProvider.CreateCompiler();
14  var compilerParameters = new CompilerParameters();
15  foreach (var singleReference in referencedAssemblies)
16  {
17      //添加引用
18      compilerParameters.ReferencedAssemblies.Add(singleReference);
19  }
20  compilerParameters.GenerateExecutable = false;
21  compilerParameters.GenerateInMemory = false;
22  compilerParameters.OutputAssembly = outAssemblyPath;
23  var compilerResult = codeCompiler.CompileAssemblyFromSource(compilerParameters,
24      sourceCodeContent);
25  if (compilerResult.Errors.HasErrors)
26  {
27      foreach (CompilerError compilerError in compilerResult.Errors)
28      {
29          compileInfo += compilerError.ErrorText;
30      }
31      compileSuccess = false;
32      LogHelper.Logger.Error("Compile error:" + compileInfo);
33  }
34  return compileSuccess;
}

```

4.7.2 CodeDOMProvider

4.8 配置

4.8.1 ini

INI 文件格式是某些平台或软件上的配置文件的非正式标准，以节 (section) 和键 (key) 构成，常用于微软 Windows 操作系统中。这种配置文件的文件扩展名多为 INI，故名。INI 是英文“初始化” (initialization) 的缩写。正如该术语所表示的，INI 文件被用来对操作系统或特定程序初始化或进行参数设置。

4.8.2 自动注册 COM 组件

```

1 #region 是否已注册控件
2 /// <summary>
3 ///
4 /// </summary>
5 /// <param name="classId"></param>
6 /// <returns></returns>
7 public static void IsDllRegisteredSuccess(string[] classId)

```

```
8  {
9      foreach (var currentClassId in classId)
10     {
11         var registerKey = Registry.ClassesRoot.OpenSubKey("CLSID\\" + currentClassId);
12         if (registerKey != null) continue;
13         var i = DllRegisterServer();
14         if (i < 0)
15         {
16             Log.Logger.Error("Control registered failed");
17         }
18         else
19         {
20             Log.Logger.Info("Control registered success");
21         }
22     }
23 }
24 #endregion
```


Chapter 5

Web

Web 应用这些年来变得越来越强大，但相比于桌面应用能够完全访问计算机硬件，Web 应用还有一些差距。

1999 HTML

2000 CGI

2001 asp

2002 xhtml

2003 flash

2004 ajax

2005 php perl java

2006 now-RIA

5.0.1 URL 规范

URL 规范化 (url normalization) 其实就是一个标准化 URL 的过程，其实也就是将一个 URL 转化为一个符合规范的等价 URL(如`http://www.cnblogs.com/shuchao`转化为`http://www.cnblogs.com/shuchao/`)，这样程序可以确定这两个 URL 是等价的。URL 规范化用于搜索引擎可以减少对页面的重复索引，同时也可以减少爬虫的重复抓取。浏览器端识别用户是否访问过一个 URL 也需要使用 URL 规范化。Url 的组成为：

```

1 protocol://hostname[:port]/path/[;parameters][?query-string]\#fragment\\
2 协议://主机名[:端口]/ 路径/[;参数] [?查询]\#Fragment

```

- protocol(协议) 表示访问资源和服务的协议。例如 http,ftp,mailto 和 file 等。
- hostname(主机名) 表示资源所在主机的完全限定域名，例如 www.schoolfree.com.cn。
- port(端口) 表示协议使用的 TCP 端口号，具体的端口号是可选的，省略端口号时前面的冒号也要省略，即表示采用“最著名的端口”。HTTP 协议的常用端口为 80。一般的 web 服务器一个 ip 地址的 80 端口只能正确对应一个网站，处理一个域名的访问请求。而 web 服务器在不使用多个 ip 地址和端口的情况下，如果需要支持多个相对独立的网站就需要一种机制来分辨同一个 ip 地址上的不同网站的请求，这就出现了主机头绑定的方法。在 IIS 中，每个 web 站点都具有唯一的、由三个部分组成的标识，用来接收和响应请求：
 - (1) ip 地址；
 - (2) 端口号；
 - (3) 主机头名。
- path(路径) 表示资源的目录/文件路径名。
- query-string 表示发送给 http 服务器的数据。
- search 查找元素表示 URL 中传递的查询字符串。查询字符串是通过 QUERY_STRING 环境变量传递给 ASP 程序的参数。
- hash 散列符元素表示指定的文件偏移量，包括散列符 (#) 加上该文件偏移量相关的位置点名。

5.0.2 Uri

URI、URL、URN 三者的关系如图5.1所示。

URL: (Uniform/Universal Resource Locator 的缩写，统一资源定位符)。

URI: (Uniform Resource Identifier 的缩写，统一资源标识符)。

关系：



图 5.1: URI、URL、URN 关系图

URI 属于 URL 更低层次的抽象，一种字符串文本标准。就是说，URI 属于父类，而 URL 属于 URI 的子类。URL 是 URI 的一个子集。二者的区别在于，URI 表示请求服务器的路径，定义这么一个资源。而 URL 同时说明要如何访问这个资源 (<http://>)。

5.0.3 CommonJS

CommonJS 是一种规范，NodeJS 是这种规范的实现。CommonJS 是一个不断发展的规范，CommonJS 有很多实现，其中不乏很多大名鼎鼎的项目，比如说：Apache 的 CouchDB 和 node.js 等。但这些项目大部分只实现了 CommonJS 的部分规范。

5.1 通信

5.1.1 端口

22 SSH 为 Secure Shell 的缩写，由 IETF 的网络工作小组（Network Working Group）所制定；SSH 为建立在应用层和传输层基础上的安全协议。SSH 是目前较可靠，专为远程登录会话和其他网络服务提供安全性的协议。利用 SSH 协议可以有效防止远程管理过程中的信息泄露问题。SSH 最初是 UNIX 系统上的一个程序，后来又迅速扩展到其他操作平台。SSH 在正确使用时可弥补网络中的漏洞。SSH 客户端适用于多种平台。几乎所有 UNIX 平台—包括 HP-UX、Linux、AIX、Solaris、Digital UNIX、Irix，以及其他平台，都可运行 SSH。

5.1.2 Internet Information Service

IIS 运行时使用的 3 份配置文件如下：

- 1 应用程序配置文件: E:\OneDrive\Source Code\Renrenmall\Tour\trunk\RR.Web.CCN.Tour\web.config
- 2 主机配置文件: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\aspnet.config
- 3 C:\Windows\Microsoft.NET\Framework64\v4.0.30319\config\machine.config 的计算机配置文件。

在 IIS 5.x 中，存在如下两个方面的问题：其一是 ISAPI(Internet Server Application Programming Interface) 动态链接库被加载到 InetInfo.exe 进程中，它和工作进程 (aspnet.exe) 之

表 5.1: IIS Application Mappings for aspnet_isapi.dll

Extension	Resource Type
.asax	ASP.NET application files. Usually global.asax.
.ascx	ASP.NET user control files.
.ashx	HTTP handlers, the managed counterpart of ISAPI extensions.
.asmx	ASP.NET web services.
.aspx	ASP.NET web pages.
.axd	ASP.NET internal HTTP handlers.

间是一种典型的跨进程通信方式，虽然采用命名管道，但是仍然会带来性能的瓶颈。为了解决这个问题，IIS 6.0 将 ISAPI 动态链接库直接加载到工作进程 (aspnet.exe) 中。其二是所有的 ASP.NET 应用运行在相同的进程 (aspnet_wp.exe) 的不同程序域中，基于应用程序域的隔离不能从根本上解决一个应用程序对另一个应用程序的影响，更多时候需要不同 Web 运行在不同的进程中。为了解决这个问题，在 IIS 6.0 中引入了应用程序池 (Application Pool) 的机制。可以为一个或者多个应用创建一个应用程序池，每一个应用程序池对应一个独立的工作进程 (w3wp.exe)，所以运行在不同应用程序池中的 Web 应用提供基于进程级别的隔离机制 [1]，这也解释了为什么有时 Windows 进程中会同时运行多个 w2wp.exe。原则上，我们可以通过 IIS 访问置于虚拟目录下的所有 Resource，这部仅仅包含一些静态资源文件，比如图片、纯 Html 文件、CSS、JS 等等，也包含一些需要动态执行的文件，比如 aspx, asmx 等等，我们还可以将 Remoting 和 WCF Service Host 到 IIS 下。对于这些静态的文件，IIS 直接提取对应的文件将其作为 Http Response 返回给 Client，但是对于这些需要进一步处理的动态执行的文件，IIS 必须将 Request 进一步传递给对应的处理程序，待处理程序执行完毕获得最终的 Http Response 通过 IIS 返回给 Client。对于 IIS 来说，这些处理程序通过 ISAPI Extension 来体现。对于基于 ASP.NET 的 Resource，其对应的 ISAPI Extension 为 ASP.NET ISAPI，通过一个 aspnet_isapi.dll 承载。IIS 的 Metadata database 维护着一个称为 ISAPI Extension Mapping 的数据表，负责将不同类型的 Resource 映射到对应的 ISAPI Extension。常见映射关系如表5.1所示。

IIS 与操作系统

IIS5.1 集成于 Windows XP 操作系统中。

IIS6.0 集成于 Windows Server 2003 操作系统中。

IIS7.0 集成于 Windows Vista 和 Windows Server 2008 操作系统中。

IIS7.5 集成于 Windows 7 和 Windows Server 2008 R2 操作系统中。

IIS 的工作进程

IIS5.1 上的 ASP.NET 程序运行在 aspnet_wp.exe 进程下。服务器同一时间只能开启一个 aspnet_wp.exe 进程，所有的应用程序都运行在该进程中，也就是说，如果由于一个应用程

序导致 aspnet_wp.exe 崩溃，所有的应用程序都将不可用。IIS6.0 上的 ASP.NET 程序运行在 w3wp.exe 进程下。为了防止出现上述的一个应用程序崩溃导致所有应用程序都不可用的情况，IIS6.0 引入了应用程序池的概念。一般情况下，一个应用程序池对应一个 w3wp.exe 进程，但是当启用 Web Garden 时，一个应用程序池就可对应多个 w3wp.exe 进程了。一个应用程序池可托管一个或多个 ASP.NET 程序。IIS7.0 的工作进程同 IIS6.0。IIS7.5 的工作进程同 IIS6.0。

IIS 的运行账户

IIS5.1 的工作进程 aspnet_wp.exe 运行在 ASPNET 账户下。ASPNET 账户隶属于 Users 用户组，该用户组下的用户无法对操作系统进行有意或无意的改动，但可以运行经过验证的应用程序。Users 用户组除了拥有 ASPNET 账户外，还拥有 Authenticated Users 用户组和 INTERACTIVE 用户组。由于内置账户 LocalSystem 也是 aspnet_wp.exe 的默认运行账户，而 LocalSystem 账户拥有几乎全部的操作系统访问权限，所以这就导致了严重的安全隐患。

IIS6.0 的工作进程 w3wp.exe 运行在 NETWORK SERVICE 用户组下。为了加强安全性，IIS 的工作进程默认运行在新的内置的 NETWORK SERVICE 用户组下，该用户组只拥有低级的用户访问权限。NETWORK SERVICE 隶属于 IIS_WPG 用户组。默认情况下，IIS_WPG 用户组只拥有最小化的权限集合用于开启和运行进程。如果要设计一个特殊的用户账户用于运行网站，最简单的方法便是将用户账户作为 IIS_WPG 用户组成员。IIS_WPG 用户组除了拥有 NETWORK SERVICE 用户组外，还拥有 IWAM_ 计算机名账户、Service 用户组和 SYSTEM 用户组。IWAM_ 计算机名账户是启动 IIS 进程账户，是用于启动进程外应用程序的 IIS 的内置账户。在 IIS6.0 中还有一个账户是 IUSR_ 计算机名，这个账户是 IIS 来宾账户，是用于匿名访问 IIS 的内置账户，该账户同时隶属于 Guests 用户组。在 IIS6.0 中，无论默认的应用程序池还是新建的应用程序池都运行在 NETWORK SERVICE 用户组下。所有的 Web 应用程序都运行在相同的权限下，那么一个在应用程序池 A 中的应用程序可以读取应用程序池 B 的配置信息，甚至有权访问属于应用程序池 B 的应用程序的内容文件。虽然创建新的应用程序池以及为它们配置自定义账号的任务足够简单，但是随着时间的推移，管理这些账号却并不那么轻松。

在 IIS7.0 里，系统自动为各应用程序新建一个应用程序池。默认情况下，应用程序池被配置为以 NETWORKSERVICE 账号运行。而当工作进程被创建时，IIS7.0 会向 NETWORKSERVICE 安全令牌注入一个特殊的唯一标识该应用程序池的 SID（通常是应用程序池的名称）。IIS7.0 还会为工作进程创建一个配置文件，并且将文件的 ACL 设置为仅允许应用程序池唯一的 SID 访问。这么做的结果就是：一个应用程序池的配置将无法被别的应用程序池读取。顺便提醒一下，你可以更改内容文件的 ACL，从而允许应用程序池唯一的 SID 进行访问而不是 NETWORKSERVICE 账号。这可以阻止应用程序池 A 中的某个应用程序读取应用程序池 B 中某应用程序的内容文件。对于 Web 服务器来说，使用哪个账号作为匿名访问的身份凭证是关系进程身份的重要问题。IIS6 依赖于一个本地账户——IUSR_ 计算机名，将其作为匿名用户登录的身份凭证。IIS7 则使用了一个全新的内置账号——IUSR，这是一个特殊的账户，用户不能使用该账户进行本地登录，它没有密码，所以任何基于密码破解的攻击对它是无效的。由于 IUSR 账号总是拥有相同的 SID，所以它的相关 ACL 在 WindowsServer2008 的机器之间是可传递的。如果 IUSR 账号不适合我们的应用场景的话（也就是说，如果匿名请求需要经身份验证的网络访问的话），可以关闭匿名用户账号，IIS7 将为匿名请求使用工作者进程身份。同样全新的还有内

置的 IIS_IUSRS 组，这个用户组取代了原先的 IIS_WPG 组。在 IIS6 里，IIS_WPG 组提供了运行一个工作者进程所需的最小权限，而且必须手动地将账号添加到该组，从而为一个工作者进程提供定制的身份凭证。IIS_IUSRS 组为 IIS7 提供了类似的角色，但是不必特意将账号添加到该组。取而代之的是，当账号被指派为某一应用程序池的身份凭证时，IIS7 会自动将这些账号收入到 IIS_IUSRS 组。并且和 IUSR 账号一样，IIS_IUSRS 组也是内置的，所以在所有的 Windows Server 2008 机器上，它总是具有相同的名称和 SID，这就让 ACL 以及其它配置在 WindowsServer2008(以及 WindowsVista) 机器之间是完全可迁移的。

5.1.3 经典模式 (Classic Model) 和集成模式 (Integrated Model)

这种经典的托管模式有很多缺点，其一，扩展繁琐，开发人员要自己编写程序运行所需要的 ISAPI 过滤器，但对于一般的程序员来说这是很困难的；其二，执行效率低，在执行 ISAPI 过滤时要退出请求管道运行应用程序实现过滤，增加了应用程序个数，浪费了内存空间；其三，安全性差。其中最主要的是安全问题，它在运行时要退出管道，查找映射，如果被影射的 DLL 文件不存在，则不能向下执行，导致执行出错（常见的错误如：映射出错）。虽然这种托管模式已被修改，但是 IIS 7.0 继续提供了这种模式，主要是考虑到程序兼容性问题，如果在集成模式下不能使用，可以迁移到经典模式中。

要想深入理解集成管道模式的好处，首先我们要了解集成的概念，集成是说把某一模块能够作为处理的一个步骤加入其中，而且这种集成是非常灵活的，开发人员可以任意指定要集成的托管代码，如：ASP.NET、PHP 等。此时在看集成的优势就会发现，集成模式不仅增强了 IIS 的灵活性，而且使得 HTTP 请求模式更加安全。其一，集成模块的托管代码可以由开发人员自行制定，并且模块可以成为管道的组成部分；其二，它去除了复杂的 ISAPI 的映射托管过程，在请求时可以直接由相应的编译环境来执行请求，使得编译过程更加安全。ASP.NET 的性能之所以能够得到改善，也是因为 ASP.NET 应用程序集成到管道中，在请求时不再需要退出管道并加载 ISAPI 进程来处理 ASP.NET 代码，然后再返回到管道为客户提供响应信息。如果我们把 ASP.NET 集成到 IIS 管道中，那么此时 ASP.NET 就成为 IIS7.0 的核心。可以在管道中处理 ASP.NET 文件，这样可以在处理过程的任意一个步骤使用 ASP.NET 代码。因为 ASP.NET 已经成为管道的一部分了。所以，诸如身份验证之类的 ASP.NET 功能也可以用于处理非 ASP.NET 内容（如：StaticFile、Image、HTML 等）。每个请求都可以由 IIS 和 ASP.NET 进行处理，而不必考虑其所属类型。

5.1.4 Configuring an ASP.NET 4 Application to Auto-Start

在 IIS 更新之后的首次 Request 会等待较长时间，ASP.NET 4 中有一个叫“自动启动 (auto-start)”的新特性，可以较好地解决这个场景，可以在 IIS 7.5（随 Windows 7 和 Windows Server 2008 R2 发布）上运行 ASP.NET 4 时使用。这个自动启动特性提供了一个可控制的方式来启动一个应用工作进程，初始化 ASP.NET 应用，然后接受 HTTP 请求。To use the ASP.NET 4 auto-start feature, you first configure the IIS “application pool” worker process that the application runs within to automatically startup when the web-server first loads. You can do this by opening up the IIS 7.5 applicationHost.config file

```
C:\Windows\System32\inetsrv\config\applicationHost.config
```

and by adding a startMode="AlwaysRunning" attribute to the appropriate <applicationPools> entry:

```
<add name="WebApi" managedRuntimeVersion="v4.0" startMode="AlwaysRunning" />
```

5.1.5 CGI(Common Gateway Interface)

最早的 Web 服务器简单地响应浏览器发来的 HTTP 请求，并将存储在服务器上的 HTML 文件返回给浏览器，也就是静态 html。事物总是不断发展，网站也越来越复杂，所以出现动态技术。但是服务器并不能直接运行 php, asp 这样的文件，自己不能做，外包给别人吧，但是要与第三做个约定，我给你什么，然后你给我什么，就是把请求参数发送给你，然后我接收你的处理结果给客户端。那这个约定就是 Common Gateway Interface，简称 CGI。这个协议可以用 vb, c, php, python 来实现。CGI 只是接口协议，根本不是什么语言。从图5.2可以看到流程。

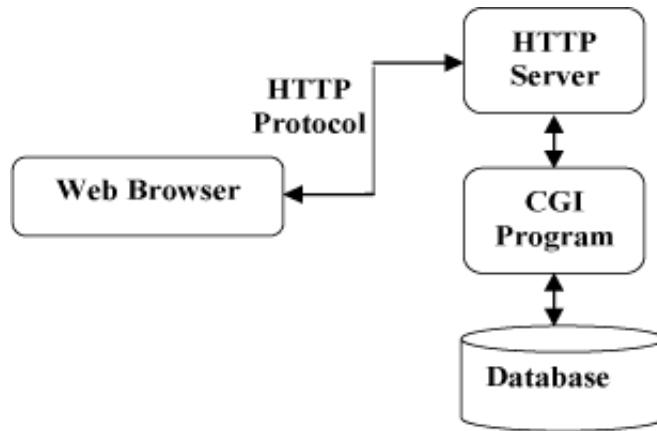


图 5.2: URI、URL、URN 关系图

CGI 程序通过标准输入 (STDIN) 和标准输出 (STDOUT) 来进行输入输出。此外 CGI 程序还通过环境变量来得到输入，操作系统提供了许多环境变量，它们定义了程序的执行环境，应用程序可以存取它们。Web 服务器和 CGI 接口又另外设置了一些环境变量，用来向 CGI 程序传递一些重要的参数。CGI 的 GET 方法还通过环境变量 QUERY-STRING 向 CGI 程序传递 Form 中的数据。CGI 工作原理：每当客户请求 CGI 的时候，WEB 服务器就请求操作系统生成一个新的 CGI 解释器进程 (如 php-cgi.exe)，CGI 的一个进程则处理完一个请求后退出，下一个请求来时再创建新进程。当然，这样在访问量很少没有并发的情况也行。可是当访问量增大，并发存在，这种方式就不适合了。于是就有了 FastCGI。FastCGI 像是一个常驻 (long-live) 型的 CGI，它可以一直执行着，只要激活后，不会每次都要花费时间去 fork 一次 (这是 CGI 最为人诟病的 fork-and-execute 模式)。一般情况下，FastCGI 的整个工作流程是这样的：

1. Web Server 启动时载入 FastCGI 进程管理器 (IIS ISAPI 或 Apache Module)

2.FastCGI 进程管理器自身初始化，启动多个 CGI 解释器进程 (可见多个 php-cgi) 并等待来自 Web Server 的连接。

3. 当客户端请求到达 Web Server 时，FastCGI 进程管理器选择并连接到一个 CGI 解释器。Web server 将 CGI 环境变量和标准输入发送到 FastCGI 子进程 php-cgi。

4.FastCGI 子进程完成处理后将标准输出和错误信息从同一连接返回 Web Server。当 FastCGI 子进程关闭连接时，请求便告处理完成。FastCGI 子进程接着等待并处理来自 FastCGI 进程管理器 (运行在 Web Server 中) 的下一个连接。在 CGI 模式中，php-cgi 在此便退出了。

PHP-FPM 与 Spawn-FCGI

Spawn-FCGI 是一个通用的 FastCGI 管理服务器，它是 lighttpd 中的一部份，很多人都用 Lighttpd 的 Spawn-FCGI 进行 FastCGI 模式下的管理工作。但是有缺点，于是 PHP-fpm 就是针对于 PHP 的，Fastcgi 的一种实现，他负责管理一个进程池，来处理来自 Web 服务器的请求。目前，PHP-fpm 是内置于 PHP 的¹。

5.1.6 MIME

MIME(Multipurpose Internet Mail Extensions) 多用途互联网邮件扩展类型。是设定某种扩展名的文件用一种应用程序来打开的方式类型，当该扩展名文件被访问的时候，浏览器会自动使用指定应用程序来打开。多用于指定一些客户端自定义的文件名，以及一些媒体文件打开方式。它是一个互联网标准，扩展了电子邮件标准，使其能够支持：非 ASCII 字符文本；非文本格式附件（二进制、声音、图像等）；由多部分（multiple parts）组成的消息体；包含非 ASCII 字符的头信息（Header information）。这个标准被定义在 RFC² 2045、RFC 2046、RFC 2047、RFC 2048、RFC 2049 等 RFC 中。MIME 改善了由 RFC 822 转变而来的 RFC 2822，这些旧标准规定电子邮件标准并不允许在邮件消息中使用 7 位 ASCII 字符集以外的字符。正因如此，一些非英语字符消息和二进制文件，图像，声音等非文字消息原本都不能在电子邮件中传输（MIME 可以）。Internet 中有一个专门组织 IANA(The Internet Assigned Numbers Authority)³ 来确认标准的 MIME 类型，但 Internet 发展的太快，很多应用程序等不及 IANA 来确认他们使用的 MIME 类型为标准类型。因此他们使用在类别中以 x-开头的方法标识这个类别还没有成为标准，例如：x-gzip，x-tar 等。MIME 规定了用于表示各种各样的数据类型的符号化方法。此外，在万维网中使用的 HTTP 协议中也使用了 MIME 的框架，标准被扩展为互联网媒体类型。MINE 中的 Content-Type 表明信息类型，缺省值为“text/plain”。它包含了主要类型（primary type）和次要类型（subtype）两个部分，两者之间用“/”分割。主要类型有 9 种，分别是 application、audio、example、image、message、model、multipart、text、video。每一种主要类型下面又有许多种次要类型，常见的有：

¹<http://www.cnblogs.com/liuzhang/p/3929198.html#undefined>

²Request For Comments (RFC)，是一系列以编号排定的文件。文件收集了有关互联网相关信息，以及 UNIX 和互联网社区的软件文件。目前 RFC 文件是由 Internet Society (ISOC) 赞助发行。基本的互联网通信协议都有在 RFC 文件内详细说明。RFC 文件还额外加入许多的论题在标准内，例如对于互联网新开发的协议及发展中所有的记录。因此几乎所有的互联网标准都有收录在 RFC 文件之中。

³IANA(The Internet Assigned Numbers Authority，互联网数字分配机构) 是负责协调一些使 Internet 正常运作的机构。同时，由于 Internet 已经成为一个全球范围的不受集权控制的全球网络，为了使网络在全球范围内协调，存在对互联网一些关键的部分达成技术共识的需要，而这就是 IANA 的任务。

表 5.2: 常用 MIME 类型对照表

文件扩展名	Content-Type(Mime-Type)
.* (二进制流, 不知道下载文件类型)	application/octet-stream
.mp4	video/mpeg4

```

1 text/plain: 纯文本, 文件扩展名.txt
2 text/html: HTML文本, 文件扩展名.htm和.html
3 image/jpeg: jpeg格式的图片, 文件扩展名.jpg
4 image/gif: GIF格式的图片, 文件扩展名.gif
5 audio/x-wave: WAVE格式的音频, 文件扩展名.wav
6 audio/mpeg: MP3格式的音频, 文件扩展名.mp3
7 video/mpeg: MPEG格式的视频, 文件扩展名.mpg
8 application/zip: PK-ZIP格式的压缩文件, 文件扩展名.zip

```

常见的 MIME 类型对照表如表5.2所示, 详细的 MIME 类型对照表参见<http://tool.oschina.net/commons>。

ashx

一般处理程序 (HttpHandler) 是 .NET 众多 web 组件的一种, ashx 是其扩展名。一个 HttpHandler 接受并处理一个 http 请求, 类比于 Java 中的 servlet。类比于在 Java 中需要继承 HttpServlet 类。在 net 中需要实现 IHttpHandler 接口, 这个接口有一个 IsReusable 成员, 一个待实现的方法 ProcessRequest(HttpContext ctx)。程序在 processRequest 方法中处理接受到的 Http 请求。成员 IsReusable 指定此 IHttpHandler 的实例是否可以被用来处理多个请求。.ashx 程序适合产生供浏览器处理的、不需要回发处理的数据格式, 例如用于生成动态图片、动态文本等内容。

Generic Web handler (*.ashx, extension based processor) is the default HTTP handler for all Web handlers that do not have a UI and that include the @WebHandler directive.

其实用.aspx 文件可以完全实现.ashx 的功能, .ashx 比着.aspx 的优点是性能高一些, 它免去了.aspx 页面的控件解析以及页面处理过程。它特别适合于生成动态图片, 生成动态文本之类, 实现某一具体功能的操作。平时查看 ashx 双击时 Visual Studio 自动定位到 ashx.cs 文件, 如果想要查看 ashx 文件本身的内容可选中文件点击鼠标右键, 有一个“查看标记”的菜单, 点击即显示 ashx 文件本身的内容。如下代码片段所示:

```

1 <%@ WebHandler Language="C#" CodeBehind="basics.ashx.cs" Class="XINLG.Weixin._codes.my.basics" %>

```

cshtml

Razor is a view engine for ASP.NET MVC, and also a template engine. Razor code and ASP.NET inline code (code mixed with markup) both get compiled first and get turned into a temporary assembly before being executed. Thus, just like C# and VB.NET both compile to IL which makes them interchangeable, Razor and Inline code are both interchangeable.

5.1.7 虚拟目录 (Virtual Directories)

A virtual directory is a directory name (also referred to as path) that you specify in IIS and map to a physical directory on a local or remote server. The directory name then becomes part of the application's URL, and users can request the URL from a browser to access content in the physical directory, such as a Web page or a list of additional directories and files. If you specify a different name for the virtual directory than the physical directory, it is more difficult for users to discover the actual physical file structure on your server because the URL does not map directly to the root of the site.

In IIS 7 and above, each application must have a virtual directory, which is named the root virtual directory, and which maps the application to the physical directory that contains the application's content. However, an application can have more than one virtual directory. For example, you might use a virtual directory when you want your application to include images from another location in the file system, but you do not want to move the image files into the physical directory that is mapped to the application's root virtual directory.

By default, IIS uses configuration from Web.config files in the physical directory to which the virtual directory is mapped, as well as in any child directories in that physical directory. If you do not want to use Web.config files in child directories, specify false for the allowSubDirConfig attribute on the virtual directory.

Optionally, when you need to specify credentials and a method to access the virtual directory, you can specify values for the username, password, and logonMethod attributes.

虚拟目录⁴可将大量的静态内容独立到一个系统盘符，构建简单的分布式应用。可网络节点分布，提升硬盘 IO 性能。可以映射到网络不同的硬盘，要知道 IO 的瓶颈，就是单块硬盘的极限，通过映射到不同的硬盘，性能的提升点就是：单块硬盘的极限 *N 块硬盘。而这一切的扩展，只是简单的虚拟目录映射，再移动相应的文件，而程序上，并不需要动刀，简单就完成文件的分布式存储。可以横向扩展，可以通过不停的加独立硬盘，方便性的提升性能。

⁴<http://www.iis.net/learn>

5.2 HTTP

5.2.1 浏览器的请求流程

DNS 解析 导航的第一步是通过访问的域名找出其 IP 地址。

301 重定向 为什么服务器一定要重定向而不是直接发会用户想看的网页内容呢？这个问题有好多有意思的答案。其中一个原因跟搜索引擎排名有关。你看，如果一个页面有两个地址，就像 `http://www.igoro.com/` 和 `http://igoro.com/`，搜索引擎会认为它们是两个网站，结果造成每一个的搜索链接都减少从而降低排名。而搜索引擎知道 301 永久重定向是什么意思，这样就会把访问带 www 的和不带 www 的地址归到同一个网站排名下。还有一个是用不同的地址会造成缓存友好性变差。当一个页面有好几个名字时，它可能会在缓存里出现好几次。

5.2.2 完整的 HTTP 请求流程

HTTP 通信机制是在一次完整的 HTTP 通信过程中，Web 浏览器与 Web 服务器之间将完成下列 7 个步骤，此处只是介绍高层的步骤，具体的细节需更多篇幅：

1. 建立 TCP 连接 在 HTTP 工作开始之前，Web 浏览器首先要通过网络与 Web 服务器建立连接，该连接是通过 TCP 来完成的，该协议与 IP 协议共同构建 Internet，即著名的 TCP/IP 协议族，因此 Internet 又被称作是 TCP/IP 网络。HTTP 是比 TCP 更高层次的应用层协议，根据规则，只有低层协议建立之后才能进行更高层协议的连接，因此，首先要建立 TCP 连接，一般 TCP 连接的端口号是 80。建立 TCP 连接需要经过“3 次握手（Three-way Handshake）”，具体如下：
 - SYN_SEND 客户端发送一个带 SYN (seq=j, 这里 j 的值为 0) 标志的 TCP 报文到服务器，并进入 SYN_SEND 状态，等待服务器确认，这是三次握手过程中的报文 1，如图所示。
 - SYN_RECV 服务器收到 syn 包，必须确认客户的 SYN (ack=j+1)，同时自己也发送一个 SYN 包 (syn=k)，即 SYN+ACK 包，此时服务器进入状态。
 - ESTABLISHED 客户端收到服务器的 SYN + ACK 包，向服务器发送确认包 ACK(ack=k+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。
2. Web 浏览器向 Web 服务器发送请求命令 一旦建立了 TCP 连接，Web 浏览器就会向 Web 服务器发送请求命令。例如：GET/sample/hello.jsp HTTP/1.1。
3. Web 浏览器发送请求头信息 浏览器发送其请求命令之后，还要以头信息的形式向 Web 服务器发送一些别的信息，之后浏览器发送了一空白行来通知服务器，它已经结束了该头信息的发送。

192.168.1.108	202.102.81.102	[TCP Retransmission] 56908-80 [SYN] Seq=0 win=8192 Len=0 MSS=1460 WS=4
202.102.81.102	192.168.1.108	80-56908 [SYN, ACK] Seq=0 Ack=1 win=14600 Len=0 MSS=1440
192.168.1.108	202.102.81.102	56908-80 [ACK] Seq=1 Ack=1 win=64800 Len=0
192.168.1.108	202.102.81.102	POST /p/pcdn/i.php?v=282079619183 HTTP/1.0 (application/octet-stream)
202.102.81.102	192.168.1.108	80-56908 [ACK] Seq=1 Ack=367 win=15544 Len=0
202.102.81.102	192.168.1.108	HTTP/1.1 200 OK
192.168.1.108	202.102.81.102	56908-80 [RST, ACK] Seq=367 Ack=1345 win=0 Len=0
⊕ Frame 974: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0		
⊕ Ethernet II, Src: Azurewave_9a:b6:01 (00:25:d3:9a:b6:01), Dst: Tp-LinkT_8f:1d:0e (e4:d3:32:8f:1d:0e)		
⊕ Internet Protocol Version 4, Src: 192.168.1.108 (192.168.1.108), Dst: 202.102.81.102 (202.102.81.102)		
⊕ Transmission Control Protocol, Src Port: 56908 (56908), Dst Port: 80 (80), Seq: 0, Len: 0		

图 5.3: TCP3 次握手

4. Web 服务器应答 客户机向服务器发出请求后，服务器会客户机回送应答，HTTP/1.1 200 OK，应答的第一部分是协议的版本号和应答状态码。服务器最基本的功能就是响应客户端的资源请求。服务器首先会侦听 80 端口，来了 http 请求，就根据请求进行处理，请求一个图片那就根据路径找到资源发回，请求静态 html 页面也是如此，如果请求的是像 php 这样的动态页面应该先调用 php 编译器（或是解释器吧）生成 html 代码，然后返回给客户端。
5. Web 服务器发送应答头信息 正如客户端会随同请求发送关于自身的信息一样，服务器也会随同应答向用户发送关于它自己的数据及被请求的文档。
6. Web 服务器向浏览器发送数据 Web 服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以 Content-Type 应答头信息所描述的格式发送用户所请求的实际数据。
7. Web 服务器关闭 TCP 连接 一般情况下，一旦 Web 服务器向浏览器发送了请求数据，它就要关闭 TCP 连接，然后如果浏览器或者服务器在其头信息加入了这行代码：Connection:keep-alive TCP 连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。从 HTTP/1.1 起，默认都开启了 Keep-Alive，保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输 HTTP 数据的 TCP 连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive 不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如 Apache）中设定这个时间，IIS 7 默认是 120 秒。

5.2.3 Http 请求在 IIS 中的处理流程

进入的 HTTP Web 请求最先由 IIS Web 服务器接收到，它在此请求基于 ASP.NET 已注册处理的扩展名传送到 ASP.NET ISAPI 上。HTTP 运行期首先创建一个 HttpContext 对象的实例，HttpContext 类由 HttpRuntime 对象实例化，接着该对象会经历请求生存期的各个阶段，它包含了当前正在处理的请求信息，接着创建在处理逻辑中涉及到的所有其他组件都可以使用的上下文对象。HttpContext 实例提供了对请求对象（HttpRequest 类的实例）和响应对象（HttpResponse 类的实例）的访问。

IIS 7.0 在 Kernel Mode 下, http.sys 接收到一个基于 Aspx 的 HttpRequest, 然后它会根据 IIS 中的 Metabase 查看该基于该 Request 的 Application 属于哪个 Application Pool, 如果该 Application Pool 不存在, 则创建之。否则直接将 request 发到对应 Application Pool 的 Queue 中。每个 Application Pool 对应着一个 Worker Process: w3wp.exe, 毫无疑问他是运行在 User Mode 下的。在 IIS Metabase 中维护着 Application Pool 和 worker process 的 Mapping。WAS (Web Administrative service) 根据这样一个 mapping, 将存在于某个 Application Pool Queue 的 request 传递到对应的 worker process(如果没有, 就创建这样一个进程)。在 worker process 初始化的时候, 加载 ASP.NET ISAPI, ASP.NET ISAPI 进而加载 CLR。最后的流程就和 IIS 5.x 一样了: 通过 AppManagerAppDomainFactory 的 Create 方法为 Application 创建一个 Application Domain; 通过 ISAPIRuntime 的 ProcessRequest 处理 Request, 进而将流程进入到 ASP.NET Http Runtime Pipeline。

5.2.4 HttpRuntime

Asp.net 有四大核心对象: HttpContext, HttpRequest, HttpResponse, HttpRuntime。IIS 所收到的对某 Microsoft ASP.NET 页面的每个请求都被移交给 ASP.NET HTTP 管线 (Pipline)。HTTP 管线由一系列托管对象组成, 这些对象按顺序处理该请求, 并完成从 URL 到普通 HTML 文本的转换。HTTP 管线的入口点是 HttpRuntime 类。ASP.NET 基础结构为辅助进程中所承载的每个 AppDomain 创建此类的一个实例请注意, 该辅助进程为当前正在运行的每个 ASP.NET 应用程序维护一个不同的 AppDomain。

要激活 HTTP 管道, 可以创建一个 HttpRuntime 类的新实例, 然后调用其 ProcessRequest 方法。一个完整的页面请求会包括下面的流程: 首先被 WWW 服务器截获 (inetinfo.exe 进程), 该进程首先判断页面后缀, 然后根据 IIS 中配置决定调用具体的扩展程序。aspx 就会调用 aspnet_isapi.dll, 然后由 aspnet_isapi.dll 发送给 w3wp.exe (iis 工作者进程, IIS6.0 中叫做 w3wq.exe, IIS5.0 中叫做 aspnet_wp.exe)。接下来在 w3wp.exe 调用.NET 类库进行具体处理, 顺序如下: ISAPIRuntime, HttpRuntime, HttpApplicationFactory, HttpApplication, HttpModule, HttpHandlerFactory, HttpHandler。

ISAPIRuntime: 主要作用是调用一些非托管代码生成 HttpWorkerRequest 对象, HttpWorkerRequest 对象包含当前请求的所有信息, 然后传递给 HttpRuntime

HttpRuntime: 根据 HttpWorkerRequest 对象生成 HttpContext, HttpContext 包含 request、response 等属性, HttpRuntime 初始化 HttpContext 如代码片段5.1所示。

Listing 5.1: 初始化 HttpContext

```

1  private void ProcessRequestInternal(HttpWorkerRequest wr)
2  {
3      HttpContext context;
4      try
5      {
6          context = new HttpContext(wr, false);
7      }
8      catch
9      {
}

```

```

10    wr.SendStatus(400, "Bad Request");
11    wr.SendKnownResponseHeader(12, "text/html; charset=utf-8");
12    byte[] bytes = Encoding.ASCII.GetBytes("<html><body>Bad Request</body></html>");
13    wr.SendResponseFromMemory(bytes, bytes.Length);
14    wr.FlushResponse(true);
15    wr.EndOfRequest();
16    return;
17 }
18 }
```

400 Bad Request 是客户端请求与语法错误，不能被服务器所理解，无法正确初始化 HttpContext。初始化 HttpRuntime 完成后再调用 HttpApplicationFactory 来生成 IHttpHandler，调用 HttpApplication 对象执行请求

HttpApplicationFactory: 生成一个 HttpApplication 对象，如代码片段5.2所示。

Listing 5.2: 生成 HttpApplication

```

1 IHttpHandler applicationInstance = HttpApplicationFactory.GetApplicationInstance(context);
2 if (applicationInstance == null)
3     throw new HttpException(SR.GetString("Unable_create_app_object"));
```

HttpApplication: 进行 HttpModule 的初始化，HttpApplication 创建针对此 Http 请求的 HttpContext 对象 HttpModule: 当一个 HTTP 请求到达 HttpModule 时，整个 ASP.NET Framework 系统还并没有对这个 HTTP 请求做任何处理，也就是说此时对于 HTTP 请求来讲，HttpModule 是一个 HTTP 请求的“必经之路”，所以可以在这个 HTTP 请求传递到真正的请求处理中心（HttpHandler）之前附加一些需要的信息在这个 HTTP 请求信息之上，或者针对截获的这个 HTTP 请求信息作一些额外的工作，或者在某些情况下干脆终止满足一些条件的 HTTP 请求，从而可以起到一个 Filter 过滤器的作用。

HttpHandlerFactory: 把用户 request 转发到 HttpHandlerFactory，再由 HttpHandlerFactory 实例化 HttpHandler 对象。

HttpHandle:Http 处理程序，处理页面请求。从上面看出 HttpRuntime 其中有一个 ProcessRequest 方法 public static void ProcessRequest(HttpWorkerRequest wr); //驱动所有 ASP.NET Web 处理执行。

5.2.5 HttpContext

HttpContext 基于 HttpApplication 的处理管道，由于 HttpContext 对象贯穿整个处理过程，所以，可以从 HttpApplication 处理管道的前端将状态数据传递到管道的后端，完成状态的传递任务。HttpContext 的生命周期从服务器接收的 HTTP 请求开始到反应发送回客户端结束。在 WebForm 或类库（包括 MVC）项目中，通过 Current 静态属性，就能够获得 HttpContext 的对象。在 MVC 的 Controller 中使用的 Request 实际上是 HttpContext 的 Request 属性。

```

1 //输出: /Home/Index
2 string rawUrl = Request.RawUrl;
```

```

3 //输出: localhost
4 string host = Request.Url.Host;
5 //输出: http://localhost/Home/Index
6 string concatUrl = "http://" + host + rawUrl;
7 //输出: http://localhost:57699/Home/Index
8 string url = Request.Url.ToString();
9 //输出: http://localhost:57699/Home/Index
10 string originalString = Request.Url.OriginalString;

```

5.2.6 HttpModule

HttpModule 是类似于过滤器的作用，可以没有，也可以有任意个，每一个都可以订阅管道事件中的任意个事件，在每个订阅的事件中可自定义功能实现。由于 HttpModule 的个数可以有多个，我们可以定义 HttpModule 实现类，然后再 web.config 中增加配置项，就可以实现多个 HttpModule 同时订阅管道事件了。

HTTP 模块是一个在每次针对应用程序发出请求时调用的程序集。HTTP 模块作为 ASP.NET 请求管道的一部分调用，它们能够在整个请求过程中访问生命周期事件。HTTP 模块使您可以检查传入和传出的请求并根据该请求进行操作。HTTP 模块通常具有以下用途：

- **安全** 您可以检查传入的请求，所以 HTTP 模块可以在调用请求页、XML Web Services 或处理程序之前执行自定义的身份验证或其他安全检查。
- **统计信息和日志记录** HTTP 模块是在每次请求时调用的，所以，您可以将请求统计信息和日志信息收集到一个集中的模块中，而不是收集到各页中。
- **自定义的页眉或页脚** 您可以修改传出响应，所以可以在每一个页面或 XML Web Services 响应中插入内容，如自定义的标头信息。

5.2.7 HttpHandler

HttpHandler 是 HTTP 请求的处理中心，真正地对客户端请求的服务器页面做出编译和执行，并将处理过后的信息附加在 HTTP 请求信息流中再次返回到 HttpModule 中。HttpHandler 与 HttpModule 不同，一旦定义了自己的 HttpHandler 类，那么它对系统的 HttpHandler 的关系将是“覆盖”关系。

5.2.8 HTTP 结构

HTTP 协议大家都知道是规定了以 ASCII 码传输，建立在 TCP、IP 协议之上的应用层规范，规范内容把 HTTP 请求分为 3 个部分：状态行，请求头，请求体。所有的方法、实现都是围绕如何运用和组织这三部分来完成的。HTTP 使用统一资源标识符（Uniform Resource Identifiers, URI）来传输数据和建立连接。一旦建立连接后，数据消息就通过类似 Internet 邮件所使用的格式 [RFC5322] 和多用途 Internet 邮件扩展（MIME）[RFC2045] 来传送。客户端

发送一个 HTTP 请求到服务器的请求消息包括以下格式：请求行（request line）、请求头部（header）、空行和请求数据 4 个部分组成，一个简单的请求如下代码片段所示：

```
1 GET /hello.txt HTTP/1.1❶
2 User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3❷
3 Host: www.example.com❸
4 Accept-Language: en, mi
```

- ❶ 此行为状态行，依次为请求的方法、请求的资源、协议版本，接下来几行的内容为请求头，请求头以 Key-Value 的形式展现，也可以添加自定义请求头，例如在需要授权认证的 API 中将加密信息添加到请求头中
- ❷ User-Agent 头域的内容包含发出请求的用户信息
- ❸ Host 头域指定请求资源的 Internet 主机和端口号，必须表示请求 url 的原始服务器或网关的位置。HTTP/1.1 请求必须包含主机头域，否则系统会以 400 状态码返回

HTTP 的头域包括通用头、请求头、响应头和实体头四个部分。每个头域由一个域名，冒号（：）和域值三部分组成。通用头部是客户端和服务器都可以使用的头部，可以在客户端、服务器和其他应用程序之间提供一些非常有用的一般功能，如 Date 头部。请求头部是请求报文特有的，它们为服务器提供了一些额外信息，比如客户端希望接收什么类型的数据，如 Accept 头部。响应头部便于客户端提供信息，比如，客服端在与哪种类型的服务器进行交互，如 Server 头部。实体头部指的是用于应对实体主体部分的头部，比如，可以用实体头部来说明实体主体部分的数据类型，如 Content-Type 头部。

HTTP 通用头

通用头域包含请求和响应消息都支持的头域，通用头域包含缓存头部 Cache-Control、Pragma 及信息性头部 Connection、Date、Transfer-Encoding、Upgrade、Via。

1、Cache-Control

Cache-Control 指定请求和响应遵循的缓存机制。在请求消息或响应消息中设置 Cache-Control 并不会修改另一个消息处理过程中的缓存处理过程。请求时的缓存指令包括 no-cache、no-store、max-age、max-stale、min-fresh、only-if-cached，响应消息中的指令包括 public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age。各个消息中的指令含义如下：

no-cache：指示请求或响应消息不能缓存，实际上是可以存储在本地缓存区中的，只是在与原始服务器进行新鲜度验证之前，缓存不能将其提供给客户端使用。

no-store：缓存应该尽快从存储器中删除文档的所有痕迹，因为其中可能会包含敏感信息。

max-age：缓存无法返回缓存时间长于 max-age 规定秒的文档，若不超规定秒浏览器将不会发送对应的请求到服务器，数据由缓存直接返回；超过这一时间段才进一步由服务器决定是返回新数据还是仍由缓存提供。若同时还发送了 max-stale 指令，则使用期可能会超过其过期时间。

min-fresh：至少在未来规定秒内文档要保持新鲜，接受其新鲜生命期大于其当前 Age 跟 min-fresh 值之和的缓存对象。

max-stale：指示客户端可以接收过期响应消息，如果指定 max-stale 消息的值，那么客户端可以接收过期但在指定值之内的响应消息。

only-if-cached：只有当缓存中有副本存在时，客户端才会获得一份副本。

Public：指示响应可被任何缓存区缓存，可以用缓存内容回应任何用户。

Private：指示对于单个用户的整个或部分响应消息，不能被共享缓存处理，只能用缓存内容回应先前请求该内容的那个用户。

打开新窗口时，如果指定 cache-control 的值为 private、no-cache、must-revalidate，那么打开新窗口访问时都会重新访问服务器。而如果指定了 max-age 值，那么在此值内的时间里就不会重新访问服务器，例如：Cache-control: max-age=5 表示当访问此网页后的 5 秒内再次访问不会去服务器。在 ASP 中，可以通过 Response 对象的 Expires、ExpiresAbsolute 属性控制 Expires 值；通过 Response 对象的 CacheControl 属性控制 Cache-control 的值，例如：Response.ExpiresAbsolute = "#2000-1-1#" 指定绝对的过期时间，这个时间用的是服务器当地时间，会被自动转换为 GMT 时间 Response.Expires = 20' 指定相对的过期时间，以分钟为单位，表示从当前时间起过多少分钟过期。Response.CacheControl = "no-cache" Expires 值是可以通过在 Internet 临时文件夹中查看临时文件的属性看到的。

2、Pragma

Pragma 头域用来包含实现特定的指令，最常用的是 Pragma:no-cache。在 HTTP/1.1 协议中，它的含义和 Cache-Control:no-cache 相同。

3、Connection

Connection 表示是否需要持久连接。如果 Servlet 看到这里的值为 “Keep-Alive”，或者看到请求使用的是 HTTP 1.1 (HTTP 1.1 默认进行持久连接)，它就可以利用持久连接的优点，当页面包含多个元素时（例如 Applet，图片），显著地减少下载所需要的时间。要实现这一点，Servlet 需要在应答中发送一个 Content-Length 头，最简单的实现方法是：先把内容写入 ByteArrayOutputStream，然后在正式写出内容之前计算它的大小。

Close：告诉 WEB 服务器或者代理服务器，在完成本次请求的响应后，断开连接，不要等待本次连接的后续请求了。

Keepalive：告诉 WEB 服务器或者代理服务器，在完成本次请求的响应后，保持连接，等待本次连接的后续请求。

Keep-Alive：如果浏览器请求保持连接，则该头部表明希望 WEB 服务器保持连接多长时间（秒），如 Keep-Alive: 300。

4、Date

Date 头域表示消息发送的时间，服务器响应中要包含这个头部，因为缓存在评估响应的新鲜度时要用到，其时间的描述格式由 RFC822 定义。例如，Date:Mon, 31 Dec 2001 04:25:57 GMT。Date 描述的时间表示世界标准时，换算成本地时间，需要知道用户所在的时区。

5、Transfer-Encoding

WEB 服务器表明自己对本响应消息体（不是消息体里面的对象）作了怎样的编码，比如是否分块（chunked），例如：Transfer-Encoding: chunked

6、Upgrade

它可以指定另一种可能完全不同的协议，如 HTTP/1.1 客户端可以向服务器发送一条 HTTP/1.0 请求，其中包含值为“HTTP/1.1”的 Upgrade 头部，这样客户端就可以测试一下服务器是否也使用 HTTP/1.1 了。

7、Via

列出从客户端到 OCS 或者相反方向的响应经过了哪些代理服务器，他们用什么协议（和版本）发送的请求。

当客户端请求到达第一个代理服务器时，该服务器会在自己发出的请求里面添加 Via 头部，并填上自己的相关信息，当下一个代理服务器收到第一个代理服务器的请求时，会在自己发出的请求里面复制前一个代理服务器的请求的 Via 头部，并把自己的相关信息加到后面，以此类推，当 OCS 收到最后一个代理服务器的请求时，检查 Via 头部，就知道该请求所经过的路由。例如：Via: 1.0 236-81.D07071953.sina.com.cn:80 (squid/2.6.STABLE13)

5.2.9 HTTP 协议-压缩 (HTTP Protocol-Compress)

HTTP 压缩是指：Web 服务器和浏览器之间压缩传输的“文本内容”的方法。HTTP 采用通用的压缩算法，比如 gzip 来压缩 HTML, Javascript, CSS 文件。能大大减少网络传输的数据量，提高了用户显示网页的速度。当然，同时会增加一点点服务器的开销。HTTP 压缩，在 HTTP 协议中，其实是内容编码的一种。在 HTTP 协议中，可以对内容（也就是 body 部分）进行编码，可以采用 gzip 这样的编码。从而达到压缩的目的。也可以使用其他的编码把内容搅乱或加密，以此来防止未授权的第三方看到文档的内容。HTTP 压缩，其实就是 HTTP 内容编码的一种，HTTP 压缩和 HTTP 内容编码两个概念容易混淆。HTTP 定义了一些标准的内容编码类型，并允许用扩展的形式添加更多的编码。Content-Encoding header 就用这些标准化的代号来说明编码时使用的算法。Content-Encoding 值：

gzip	表明实体采用 GNU zip 编码
compress	表明实体采用 Unix 的文件压缩程序
deflate	表明实体是用 zlib 的格式压缩的
identity	表明没有对实体进行编码。当没有 Content-Encoding header 时，就默认为这种情况

gzip, compress, 以及 deflate 编码都是无损压缩算法，用于减少传输报文的大小，不会导致信息损失。其中 gzip 通常效率最高，使用最为广泛。HTTP 压缩对纯文本可以压缩至原内容的 40%，从而节省了 60% 的数据传输。Gzip 压缩是在一个文本文件中找出类似的字符串，并临时替换他们，使整个文件变小。这种形式的压缩对 Web 来说非常适合，因为 HTML 和 CSS 文件通常包含大量的重复的字符串，例如空格，标签。浏览器在发送请求时告知可以接收的压缩编

码类型，如图5.4所示，由图可知浏览器可接受的压缩编码类型为 gzip,deflate,sdch。现在最好禁用 deflate 压缩，因为现在浏览器对 DEFLATE 压缩处理不好，早期浏览器对 DEFLATE 压缩描述混乱⁵，deflate 有可能直接穿过某些防火墙产品，给骇客留下弱点。

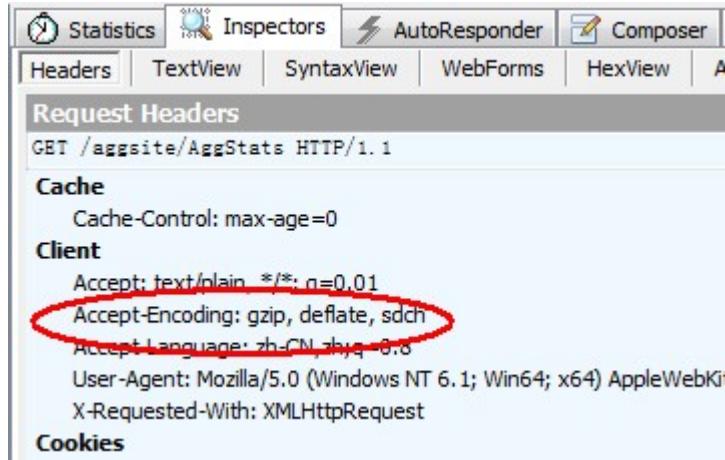


图 5.4: 告知可接受的压缩编码类型

在启用了 gzip 压缩的网站响应头中会有如下语句：

```
1 Content-Encoding: gzip
```

5.2.10 HTTP ETag

ETag 全称 Entity Tag，用来标识一个资源。在具体的实现中，ETag 可以是资源的 hash 值，也可以是一个内部维护的版本号。但不管怎样，ETag 应该能反映出资源内容的变化，这是 Http 缓存可以正常工作的基础。在 HTTP1.1 规范中，新增了一个 HTTP 头信息：ETag。对 Web 开发者来说，它是一个非常重要的信息。它是用作缓存使用的两个主要的头信息之一（另一个是 Expires）。除此之外，在 REST 架构中，它还可以用于控制并发操作（上节中已经大致介绍 AtomPub 中控制并发的流程）。ETag：是实体标签 (Entity Tag) 的缩写。ETag 一般不以明文形式相应给客户端。在资源的各个生命周期中，它都具有不同的值，用于标识出资源的状态。当资源发生变更时，如果其头信息中一个或者多个发生变化，或者消息实体发生变化，那么 ETag 也随之发生变化。在 HTTP1.1 协议中并没有规范如何计算 ETag。ETag 值可以是唯一标识资源的任何东西，如持久化存储中的某个资源关联的版本、一个或者多个文件属性，实体头信息和校验值、(CheckSum)，也可以计算实体信息的散列值。有时候，为了计算一个 ETag 值可能有比较大的代价，此时可以采用生成唯一值等方式（如常见的 GUID）。无论怎样，服务都应该尽可能的将 ETag 值返回给客户端。客户端不用关心 ETag 值如何产生，只要服务在资源状态发生变更的情况下将 ETag 值发送给它就行。OutgoingResponse 类中设置 ETag 值如代码5.3所示：

Listing 5.3: 生成 ETag

```
1 public void SetETag(string entityTag)
```

⁵<http://www.webkaka.com/blog/archives/DEFLATE-Obsolete-Compression-Format.html>

```

2  {
3      this.ETag = OutgoingWebResponseContext.GenerateValidEtagFromString(entityTag);
4  }
5
6  public void SetETag(int entityTag)
7  {
8      this.ETag = OutgoingWebResponseContext.GenerateValidEtag(entityTag);
9  }
10
11 public void SetETag(long entityTag)
12 {
13     this.ETag = OutgoingWebResponseContext.GenerateValidEtag(entityTag);
14 }
15
16 public void SetETag(Guid entityTag)
17 {
18     this.ETag = OutgoingWebResponseContext.GenerateValidEtag(entityTag);
19 }

```

在 REST 架构下，ETag 值可以通过 Guid、整数、长整形、字符串四种类型的参数传入 SetETag 方法，WCF 服务发回给客户端的 HTTP 响应头中就包含了 ETag 值。另外 OutgoingResponse 类也有字符串属性：ETag 直接给它赋值也能在 HTTP 响应头中写入 ETag 值。按照 HTTP 标准，Last-Modified 只能精确到秒级。ETag 的出现可以很好的解决这个问题。在用途上，ETag 常与 If-None-Match 或者 If-Match 一起，由客户端通过 HTTP 头信息（包括 ETag 值）发送给服务端处理。ETag 有两种类型：强 ETag(strong ETag) 与弱 ETag(weak ETag)。

强 ETag 表示形式：“22FAA065-2664-4197-9C5E-C92EA03D0A16”。

弱 ETag 表现形式：w/“22FAA065-2664-4197-9C5E-C92EA03D0A16”。

强、弱 ETag 类型的出现与 Apache 服务器计算 ETag 的方式有关。Apache 默认通过 FileEtag 中 FileEtag INode Mtime Size 的配置自动生成 ETag（当然也可以通过用户自定义的方式）。假设服务端的资源频繁被修改（如 1 秒内修改了 N 次），此时如果有用户将 Apache 的配置改为 MTime，由于 MTime 只能精确到秒，那么就可以避免强 ETag 在 1 秒内的 ETag 总是不同而频繁刷新 Cache（如果资源在秒级经常被修改，也可以通过 Last-Modified 来解决）

5.2.11 HTTP Proxy

Web 代理 (proxy) 服务器是网络的中间实体。代理位于 Web 客户端和 Web 服务器之间，扮演“中间人”的角色。HTTP 的代理服务器即是 Web 服务器又是 Web 客户端。Fiddler 是以代理 web 服务器的形式工作的，它使用代理地址：127.0.0.1，端口：8888。当 Fiddler 退出的时候它会自动注销代理，这样就不会影响别的程序。

- **代理翻墙** 很多人都喜欢用 Facebook，看 YouTube。但是我们在天朝，天朝有 The Great of Fire Wall（长城防火墙），屏蔽了这些好网站。怎么办？通过代理来跳墙，就可以访问了。

- **匿名访问** 经常听新闻，说“某某某”在网络上发布帖子，被跨省追缉了。假如他使用匿名的代理服务器，就不容易暴露自己的身份了。HTTP 代理服务器的匿名性是指：HTTP 代理服务器通过删除 HTTP 报文中的身份特性（比如客户端的 IP 地址，或 cookie, 或 URI 的会话 ID），从而对远端服务器隐藏原始用户的 IP 地址以及其他细节。同时 HTTP 代理服务器上也不会记录原始用户访问记录的 log(否则也会被查到)。
- **通过代理上网** 比如局域网不能上网，只能通过局域网内的一台代理服务器上网。
- **代理缓存，加快上网速度** 大部分代理服务器都具有缓存的功能，就好像一个大的 cache，它有很大的存储空间，它不断将新取得数据存储到它本地的存储器上，如果浏览器所请求的数据在它本机的存储器上已经存在而且是最新的，那么它就不重新从 Web 服务器取数据，而直接将存储器上的数据传给用户的浏览器，这样就能显著提高浏览速度。
- **儿童过滤器** 很多教育机构，会利用过滤器代理来阻止学生访问成人内容。

5.2.12 HTTP Tunnel

Http 隧道分为两种：1. 不使用 CONNECT 的隧道 2. 使用 CONNECT 的隧道

不使用 CONNECT 的隧道，实现了数据包的重组和转发。也就是在我们使用 Proxy 的时候，然后发起 Http 请求，使用的就是非 CONNECT 的隧道。在这种情况下，在 Proxy 收到来自客户端的 Http 请求之后，会重新创建 Request 请求，并发送到目标服务器，也就是图中的 Content Server。当目标服务器返回 Response 给 Proxy 之后，Proxy 会对 Response 进行解析，然后重新组装 Response，发送给客户端。所以，在不使用 CONNECT 方式建立的隧道，Proxy 有机会对客户端与目标服务器之间的通信数据进行窥探，而且有机会对数据进行串改。

而对于使用 CONNECT 的隧道则不同。当客户端向 Proxy 发起 Http CONNECT Method 的时候，就是告诉 Proxy，先在 Proxy 和目标服务器之间先建立起连接，在这个连接建立起来之后，目标服务器会返回一个回复给 Proxy，Proxy 将这个回复转发给客户端，这个 Response 是 Proxy 跟目标服务器连接建立的状态回复，而不是请求数据的 Response。在此之后，客户端跟目标服务器的所有通信都将使用之前建立起来的建立。这种情况下的 Http 隧道，Proxy 仅仅实现转发，而不会关心转发的数据。

这也是为什么在使用 Proxy 的时候，Https 请求必须首先使用 Http CONNECT 建立隧道。因为 Https 的数据都是经过加密的，Proxy 是无法对 Https 的数据进行解密的，所以只能使用 CONNECT，仅仅对通信数据进行转发。

所以，实际上所有的网络库（例如 libcurl, chromium_net）在实现的时候，如果使用了 Proxy，在请求 Https 协议的资源时，首先是使用 CONNECT 方法建立 Http 隧道，等收到目标服务器建立成功的回复之后，开始做 SSL/TLS 握手，然后进行数据传输。

虽然 Http CONNECT 隧道安全，能够使得客户端的数据实现“翻墙”，但是 Http CONNECT 容易失控，由于数据都是加密的，Proxy 跟目标服务器建立 TCP 连接之后，之后的所有数据都服用这个连接。但是 Http CONNECT 隧道不仅仅是可以建立 Https 443 端口的连接，实际上可以建立任何端口的连接，这也使得 Proxy 变得非常危险，所以，一般的服务器都会对 Http CONNECT 隧道进行控制，只能实现 Https 443 端口的隧道通信。

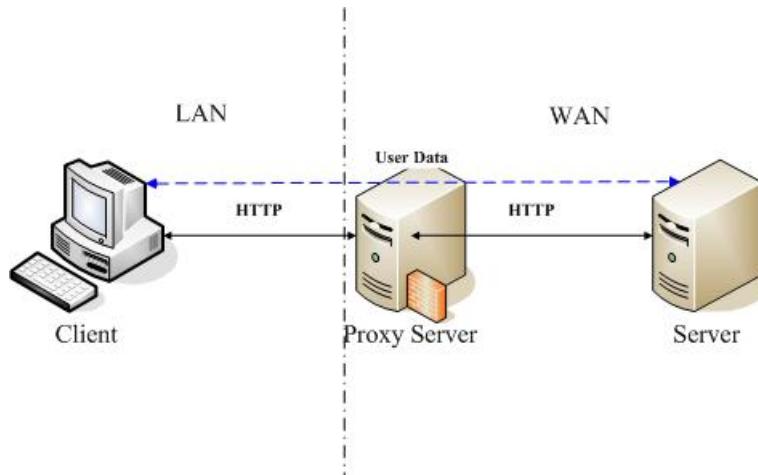


图 5.5: HTTP 隧道

HttpTunnel(也叫 Http 隧道, Http 穿梭), 是这样一种技术: 它用 HTTP 协议在要通信的 Client 和 Server 建立起一条 “Tunnel”, 然后 Client 和 Server 之间的通信, 都是在这条 Tunnel 的基础之上。HttpTunnel 通常被用在受限的网络环境中, 比如在 NAT⁶(Network address translation) 环境中的 Client, 受防火墙限制的环境中的 Client 等, 在这样的环境中, Client 不能直接连接到公网 (WAN) 的 Server, 这时候就可以通过 HttpTunnel 技术, 来解决上述问题。上图是 HttpTunnel 技术的基本原理, 它基本的工作过程主要分为以下几个步骤: (1) Client 向 ProxyServer 发送要连接到 Server 的请求 (Http 协议) (2) Proxy Server 向实际的 Server 发送连接请求 (Http 协议) (3) 上述两步成功后, 就相当于在 Client 和 Server 之间存在了一条连通的 Tunnel (如上图中的蓝色虚线所示) (4) 后续 Client 和 Server 就可以直接进行数据的收发, 协议由 Client 和 Server 自己约定, 与 HttpTunnel 无关

5.2.13 REST 方式 POST、GET

Get 请求的数据会附在 URL 之后 (就是把数据放在 HTTP 协议头中), 如果数据是英文字母或数字, 原样发送, 如果是空格, 转换为 +, 如果是中文或其他字符, 则直接把字符用 BASE64 加密, 得到如: %E4%BD%A0%E5%A5%BD 类似格式, 其中%XX 中的 X 为该符号以 16 进制表示的 ASCII. GET 方式通过在网络地址附加参数来完成数据的提交, 比如在地址 <http://www.google.com/webhp?hl=zh-CN> 中, 前面部分 <http://www.google.com/webhp> 表示数据提交的网址, 后面部分 hl=zh-CN 表示附加的参数, 其中 hl 表示一个键 (key), zh-CN 表示这个键对应的值 (value)。POST 把数据放置在 HTTP 包的包体中。POST 方式通过在页面内容中填写参数的方法来完成数据的提交, 参数的格式和 GET 方式一样, 是类似于 hl=zh-CN&newwindow=1 这样的结构。程序代码如5.4所示:

⁶Network Address Translation (NAT) is a methodology of remapping one IP address space into another by modifying network address information in Internet Protocol (IP) datagram packet headers while they are in transit across a traffic routing device. The technique was originally used for ease of rerouting traffic in IP networks without renumbering every host. It has become a popular and essential tool in conserving global address space allocations in face of IPv4 address exhaustion by sharing one Internet-routable IP address of a NAT gateway for an entire private network.

Listing 5.4: REST 方式 POST, 不带验证

```

1 #region 从WCF接口POST数据-不带验证
2 /// <summary>
3 /// POST 请求与获取结果 (带验证)
4 /// </summary>
5 /// <param name="url"> 请求的 WCF 地址 </param>
6 /// <param name="postDataParam"> 请求参数集合 (key=value&key=value)</param>
7 public static string HttpPost(string url, string postDataParam,string authorization)
8 {
9     string postUrl = url + (postDataParam == "" ? "" : "?") + postDataParam;
10    HttpWebRequest request = (HttpWebRequest)WebRequest.Create(postUrl);
11    request.Method = "POST";
12    //添加验证
13    request.Headers.Add("Authorization", authorization);
14    byte[] bytes = Encoding.ASCII.GetBytes(postDataParam);
15    request.ContentLength = bytes.Length;
16    request.ContentType = "text/html;charset=UTF-8";
17    Stream requestStream=request.GetRequestStream();
18    requestStream.Write(bytes, 0, (int)bytes.Length);
19    requestStream.Close();
20    HttpWebResponse response = request.GetResponse() as HttpWebResponse;
21    if (response != null)
22    {
23        Stream responseStream = response.GetResponseStream();
24        if (responseStream != null)
25        {
26            StreamReader myStreamReader = new StreamReader(responseStream, Encoding.
27 UTF8);
28            resultString = myStreamReader.ReadToEnd();
29            myStreamReader.Close();
30            responseStream.Close();
31        }
32        return resultString;
33    }
34 #endregion

```

Get 请求浏览器或 OS 对其数据长度有限制, 太长的数据无法用 Get 操作. POST 服务器也有相应的限制.

1)IIS6 的 POST 大小为 200KB, 每个表单域 100KB.

2)IIS6 最大请求头 16KB.

3)IIS6 默认上传文件的大小是 4MB.

服务端获取 Get 请求参数用 Request.QueryString. 获取 POST 请求用 Request.Form. JSP 用 request.getParameter(). 另 POST 的安全性比 Get 高.

5.2.14 HTTP Request and HTTP Response

一个 HTTP 请求报文由请求行 (request line)、请求头部 (header)、空行和请求数据 4 个部分组成。HTTP 协议的请求方法有 GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE、CONNECT。最常见的一种请求方式，当客户端要从服务器中读取文档时，当点击网页上的链接或者通过在浏览器的地址栏输入网址来浏览网页的，使用的都是 GET 方式。GET 方法要求服务器将 URL 定位的资源放在响应报文的数据部分，回送给客户端。使用 GET 方法时，请求参数和对应的值附加在 URL 后面，利用一个问号 (“?”) 代表 URL 的结尾与请求参数的开始，传递参数长度受限制。例如，/index.jsp?id=100&op=bind，这样通过 GET 方式传递的数据直接表示在地址中，所以我们可以把请求结果以链接的形式发送给好友。以用 google 搜索 domety 为例，Request 格式如下：

```

1 GET /search?hl=zh-CN&source=hp&q=domety&aq=f&oq= HTTP/1.1
2 Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel,
           application/vnd.ms-powerpoint,
3 application/msword, application/x-silverlight, application/x-shockwave-flash, */
4 Referer: <a href="http://www.google.cn/">http://www.google.cn/</a>
5 Accept-Language: zh-cn
6 Accept-Encoding: gzip, deflate
7 User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
              2.0.50727; TheWorld)
8 Host: <a href="http://www.google.cn">www.google.cn</a>
9 Connection: Keep-Alive
10 Cookie: PREF=ID=80a06da87be9ae3c:U=f7167333e2c3b714:
11 NW=1:TM=1261551909:LM=1261551917:S=ybYcq2wpfefs4V9g;
12 NID=31=ojj8d-IygaEtSxLgaJmqSjVhCspkviJrB6omjamNrSm8lZhKy_
13 yMfO2M4QMRKcH1g0iQv9u-2hfBW7bUFwVh7pGaRUb0RnHcJU37y-
14 FxlRugatx63JL7CWMD6UB_O_r

```

可以看到，GET 方式的请求一般不包含“请求内容”部分，请求数据以地址的形式表现在请求行。地址链接如下：

```

1 <a href="http://www.google.cn/search?hl=zh-CN&source=hp&q=domety&aq=f&oq=">http://
   www.google.cn/search?hl=zh-CN&source=hp
2 &q=domety&aq=f&oq=</a>

```

地址中“?”之后的部分就是通过 GET 发送的请求数据，我们可以在地址栏中清楚的看到，各个数据之间用“&”符号隔开。显然，这种方式不适合传送私密数据。另外，由于不同的浏览器对地址的字符限制也有所不同，一般最多只能识别 1024 个字符，服务器对地址字符同样也有相应的限制⁷，为了让所有的用户都能正常浏览，我们的 URL 最好不要超过 IE 的最大长度限制 (2038 个字符)，当然，如果 URL 不直接提供给用户，而是提供给程序调用，侧这时的长度就只受 Web 服务器影响了。所以如果需要传送大量数据的时候，也不适合使用 GET 方式。

⁷具体可以参考链接：<http://www.cnblogs.com/henryhappier/archive/2010/10/09/1846554.html>

5.2.15 HTTP 状态码 (HTTP Status Code)

Response 消息中的第一行叫做状态行，由 HTTP 协议版本号，状态码，状态消息三部分组成。状态码用来告诉 HTTP 客户端，HTTP 服务器是否产生了预期的 Response. HTTP/1.1 中定义了 5 类状态码，状态码由三位数字组成，第一个数字定义了响应的类别。

2XX

提示信息 - 表示请求已被成功接收，继续处理

2XX 成功 - 表示请求已被成功接收，理解，接受

204 No Content HTTP 的 204(No Content) 响应，就表示执行成功，但是没有数据，浏览器不用刷新页面。也不用导向新的页面。当有一些服务，只是返回成功与否的时候，可以尝试使用 HTTP 的状态码来作为返回信息，而省掉多余的数据传输，比如 REST 中的 DELETE 和如上所述的查询式 Ajax 请求。

206 Partial Content 首先你需要知道文件大小以及远程服务器是否支持 HTTP 206 请求。使用 curl 命令可以查看任意资源的 HTTP 头，使用下面的 curl 命令可以发送一个 HEAD 请求：

```

1 curl -I http://192.168.1.222:8085/tour/images/beginPage_1.jpg
2
3 HTTP/1.1 200 OK
4 Content-Length: 311837
5 Content-Type: image/jpeg
6 Last-Modified: Fri, 22 Jan 2016 10:48:07 GMT
7 Accept-Ranges: bytes
8 ETag: "de59761255d11:0"
9 Server: Microsoft-IIS/7.5
10 X-Powered-By: ASP.NET
11 Date: Fri, 29 Jan 2016 08:44:21 GMT

```

Accept-Ranges: bytes - 该响应头表明服务器支持 Range 请求，以及服务器所支持的单位是字节（这也是唯一可用的单位）。我们还能知道：服务器支持断点续传，以及支持同时下载文件的多个部分，也就是说下载工具可以利用范围请求加速下载该文件。Accept-Ranges:none 响应头表示服务器不支持范围请求。

Content-Length: 311837 - Content-Length 响应头表明了响应实体的大小，也就是真实的图片文件的大小是 311837 字节 (304K)。服务器已经成功处理了部分 GET 请求。类似于 FlashGet 或者迅雷这类的 HTTP 下载工具都是使用此类响应实现断点续传或者将一个大文档分解为多个下载段同时下载。该请求必须包含 Range 头信息来指示客户端希望得到的内容范围，并且可能包含 If-Range 来作为请求条件。响应必须包含如下的头部域：Content-Range 用以指示本次响应中返回的内容的范围；如果是 Content-Type 为 multipart/byteranges 的多段下载，则每一 multipart 段中都应包含 Content-Range 域用以指示本段的内容范围。假如响应中包含

Content-Length, 那么它的数值必须匹配它返回的内容范围的真实字节数。Date ETag 和/或 Content-Location, 假如同样的请求本应该返回 200 响应。Expires, Cache-Control, 和/或 Vary, 假如其值可能与之前相同变量的其他响应对应的值不同的话。假如本响应请求使用了 If-Range 强缓存验证, 那么本次响应不应该包含其他实体头; 假如本响应的请求使用了 If-Range 弱缓存验证, 那么本次响应禁止包含其他实体头; 这避免了缓存的实体内容和更新了的实体头信息之间的不一致。否则, 本响应就应当包含所有本应该返回 200 响应中应当返回的所有实体头部域。假如 ETag 或 Last-Modified 头部不能精确匹配的话, 则客户端缓存应禁止将 206 响应返回的内容与之前任何缓存过的内容组合在一起。任何不支持 Range 以及 Content-Range 头的缓存都禁止缓存 206 响应返回的内容。

3XX

301 Moved Permanently 301 代表永久性转移 (Permanently Moved), 301 重定向是网页更改地址后对搜索引擎友好的最好方法, 只要不是暂时搬移的情况, 都建议使用 301 来做转址。

302 Found Found: 重定向, 新的 URL 会在 Response 中的 Location 中返回, 浏览器将会自动使用新的 URL 发出新的 Request。例如在 IE 中输入, <http://www.google.com>. HTTP 服务器会返回 302, IE 取到 Response 中 Location header 的新 URL, 又重新发送了一个 Request. 302 代表暂时性转移 (Temporarily Moved), 在前些年, 不少 Black Hat SEO 曾广泛应用这项技术作弊。各大主要搜索引擎均加强了打击力度, 像 Google 对 BMW 德国网站的惩罚。即使网站客观上不是 spam, 也很容易被搜寻引擎误判为 spam 而遭到惩罚。

304 Not Modified Not Modified: 代表上次的文档已经被缓存了, 还可以继续使用, 刷新一个网页时, Fiddler 会捕获 304 类型的封包, 代表此资源可重复使用。

4XX

400 400 是一种是 HTTP 状态码, 400 Bad Request。是在打开网页时浏览器返回到客户端的一种状态码。显示在客户端的也就是 400 页面。400 页面是当用户在打开网页时, 返回给用户界面带有 400 提示符的页面。其含义是你访问的页面域名不存在或者请求错误。主要有两种形式: 1、bad request 意思是“错误的请求”; 2、invalid hostname 意思是“不存在的域名”。通常只用 Windows 主机才会出现这样的字样, 如果是 Linux 主机, 会显示不同的错误提示。bad request invalid hostname 出现这个错误的原因是某个域名绑定到了某个主机上, 而该主机却没有绑定这个域名, 所以 IIS 就返回了这个提示信息。

401 一般来说该错误消息表明您首先需要登录 (输入有效的用户名和密码)。如果你刚刚输入这些信息, 立刻就看到一个 401 错误, 就意味着, 无论出于何种原因您的用户名和密码其中之一或两者都无效 (输入有误, 用户名暂时停用, 等)。

4XX 客户端错误 - 请求有语法错误或请求无法实现。400 Bad Request 客户端请求与语法错误, 不能被服务器所理解。403 Forbidden 服务器收到请求, 但是拒绝提供服

务。404 Not Found 请求资源不存在（输错了 URL）。比如在 IE 中输入一个错误的 URL, http://www.cnblogs.com/tesdf.aspx。

5XX

5XX 服务器端错误 - 服务器未能实现合法的请求。500 Internal Server Error 服务器发生了不可预期的错误。503 Server Unavailable 服务器当前不能处理客户端的请求，一段时间后可能恢复正常。

503 服务不可用是的一种状态，那么在服务器 503 错误出现了之后，大家不必担心的，服务器或许就是正在维护或者暂停了，你可以联系一下服务器空间商。还有的时候 cpu 占用的频率大导致的。查看当前网站属于哪个应用程序，检查应用程序池是否处于停止状态。

5.2.16 获取客户端 IP

```

1  /// <summary>
2  /// 获取客户端 ip
3  /// </summary>
4  /// <returns></returns>
5  public static string GetCurrentIP()
6  {
7      string userHostAddress = string.Empty;
8      try
9      {
10         //如果客户端使用了代理服务器，则利用 HTTP_X_FORWARDED_FOR 找到客户端
11         // IP 地址
12         userHostAddress = HttpContext.Current.Request.ServerVariables["
13             HTTP_X_FORWARDED_FOR"].ToString().Split(',')[0].Trim();
14         //否则直接读取 REMOTE_ADDR 获取客户端 IP 地址
15         if (string.IsNullOrEmpty(userHostAddress))
16         {
17             userHostAddress = HttpContext.Current.Request.ServerVariables["REMOTE_ADDR"];
18         }
19         //前两者均失败，则利用 Request.UserHostAddress 属性获取 IP 地址，但此时无法确定该
20         // IP 是客户端 IP 还是代理 IP
21         if (string.IsNullOrEmpty(userHostAddress))
22         {
23             userHostAddress = HttpContext.Current.Request.UserHostAddress;
24         }
25         //最后判断获取是否成功，并检查 IP 地址的格式（检查其格式非常重要）
26         if (!string.IsNullOrEmpty(userHostAddress) && IsIP(userHostAddress))
27         {
28             return userHostAddress;
29         }
30     }
31     catch (Exception e)
32     {
33         logger.Error("Get current IP encount an error", e);
34     }
35 }
```

```

31     }
32
33     return userHostAddress;
34 }
```

在 Web 开发中，我们大多都习惯使用 HTTP 请求头中的某些属性来获取客户端的 IP 地址，常见的属性是 REMOTE_ADDR、HTTP_VIA 和 HTTP_X_FORWARDED_FOR。这三个属性的含义，大概是如此：REMOTE_ADDR：该属性的值是客户端跟服务器“握手”时候的 IP。如果使用了“匿名代理”，REMOTE_ADDR 将显示代理服务器的 IP。X-Forwarded-For：是用来识别通过 HTTP 代理或负载均衡方式连接到 Web 服务器的客户端最原始的 IP 地址的 HTTP 请求头字段。X-Forwarded-For 的有效性依赖于代理服务器提供的连接原始 IP 地址的真实性，因此，X-Forwarded-For 的有效使用应该保证代理服务器是可信的，比如可以通过建立可信服务器白名单的方式。

5.2.17 placeholder 属性

placeholder 属性是 HTML5 中为 input 添加的。在 input 上提供一个占位符，文字形式展示输入字段预期值的提示信息 (hint)，该字段会在输入为空时显示。

```

1 <input type="text" name="UserName" class="form-control" placeholder="用户唯一ID"/>
```

5.3 调试 (Debug)

WCF 接口调试是在路由配置文件中设置，传入适当的参数。如下所示：<http://localhost:15973>ShowInfo>ShowAwardingListFC?openId=oM8-3t39DFln91EdJCvUPAIHqbF8&accountId=34>。

页面的 JavaScript 脚本调试可以打开 FireDebug，打开脚本 Tab 页，在左侧的窗口中设置断点，如图5.6所示。右侧的断点 tab 页可查看当前设置的所有断点。

按 F11 进行单步进入调试，按 F12 进行单步跳过调试。在使用 Visual Studio 调试网页时，如果网页有微小的改动而你又不想重新启动新的调试界面，那么可以利用 Visual Studio 的“浏览器链接 (Browser Link)”功能，如图5.7所示，每次启动页面时会在网页尾部添加一段 Javascript 脚本，定时监听 Visual Studio 的刷新指令。

当修改页面之后只需要点击刷新符号，链接到 Visual Studio 的浏览器即可即时刷新页面，刷新链接浏览可以快速的看到页面修改结果，刷新是刷新链接浏览器的所有页面。在调试 MVC 的 Action 时，没有 View 一样可以调试，只需要浏览器链接指向 Action 即可，如果有参数带上相应参数。

```

49 <input id="questionType" type="hidden" />
50
51 <script type="text/javascript">
52
53 window.onload = function () {
54     var isHaveRecord = $("#isHaveRecord").val();
55     if (isHaveRecord) {
56         SetQuestionAnswer();
57     }
58
59
60 $(document).ready(function () {
61     $("#submit").bind("click", function () {
62         console.log("提交中...");
63         var userphone = $("#userPhone").val();
64         if (userphone == "" || userphone == null || userphone.length != 11) {
65             alert("请输入正确的手机号码!"); return;
66         } else {

```

图 5.6: Firefox 调试页面 JavaScript 脚本

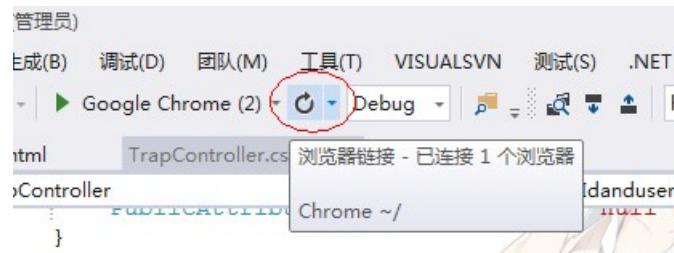


图 5.7: Visual Studio 中刷新浏览器

5.3.1 Chrome Javascript 条件断点 (Condition Breakpoint)

在开发过程中 Javascript 报空引用错误，报错的地方有十几个循环，在第 7 次循环是出现错误，那么此时需要快速定位错误的话，条件断点是一个非常有用的助手。在 Google Chrome 中按 F12 打开开发者调试工具，在资源（Sources）下找到需要调试的 Javascript 脚本，在断点位置的右键菜单中选择“Edit Breakpoint...”可以设置触发断点的条件，就是写一个表达式，表达式为 true 时才触发断点，此处是循环条件等于 6。新增条件断点如图5.8所示，当断点新增了条件之后，断点标识的颜色会变为浅黄色（Google Chrome canary 48.0.2564.0(64-bit)）。

在调试 Javascript 时，脚本有可能是经过压缩的，此时显示仅有一行，为了便于阅读，可点击调试器下方的大括号格式化压缩后的 Javascript 脚本，如图5.9所示。

5.3.2 SlidesJS 改变滑动间隔

使用 SlideJS⁸ 设置界面图片的滑动时间间隔如下代码所示：

```

1  $("#slides").slidesjs({
2      width: 1080,
3      height: 500,
4      play:

```

⁸SlidesJS is a responsive slideshow plug-in for jQuery (1.7.1+) with features like touch and CSS3 transitions.

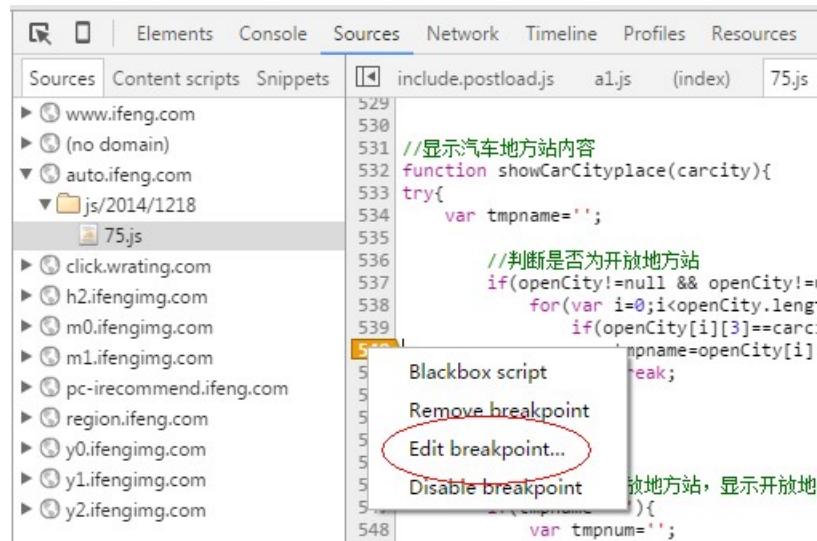


图 5.8: Javascript 设置条件断点

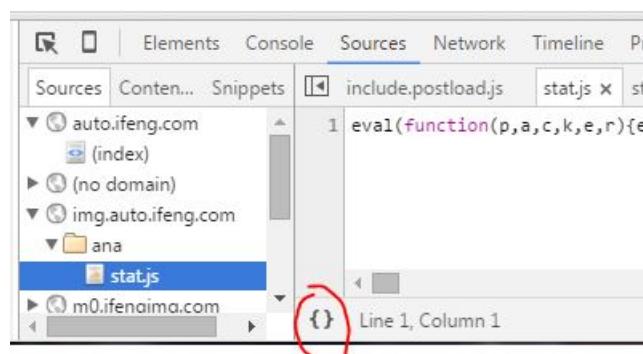


图 5.9: 格式化 Javascript 脚本

```

5     {
6         auto:true,
7         interval:10000
8     }
9 });

```

其中 interval 设置幻灯片的滑动的间隔时间，单位为毫秒，这里设置的是 10 秒。

5.3.3 IIS 6.0 中 MVC 项目部署

将 MVC4 网站部署到 IIS6.0，部署完成后在网站的属性页面点击“配置”，如图5.10所示：

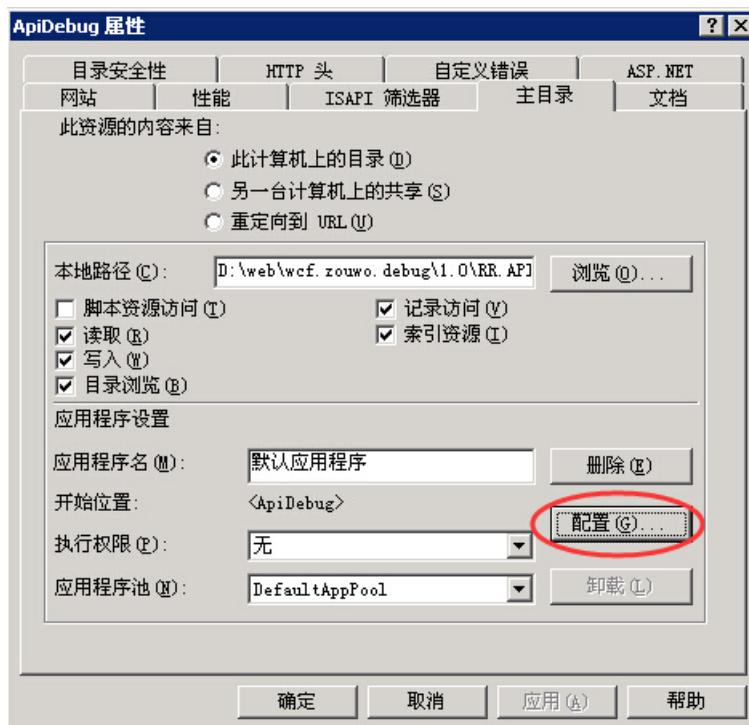


图 5.10: IIS6 配置 MVC4

弹出的“应用程序配置”页面，映射下通配符应用程序映射中，点击“插入”按钮，在弹出的界面中填写 aspnet_isapi.dll 路径：

```
C:\WINDOWS\Microsoft.NET\Framework64\v4.0.30319\aspnet_isapi.dll
```

如图5.11所示。注意需要将“确认文件是否存在”项勾选掉。ISAPI(Internet Server Application Programming Interface) 是一套本地的 (Native) Win32 API, 是 IIS 和其他动态 Web 应用或平台之间的纽带。ISAPI 支持 ISAPI 扩展 (ISAPI Extension) 和 ISAPI 筛选 (ISAPI Filter)，前者是真正处理 HTTP 请求的接口，后者则可以在 HTTP 请求真正被处理之前查看、修改、转发或拒绝请求，比如 IIS 可以利用 ISAPI 筛选进行请求的验证。如果检测到某个 HTTP 请求的资源为静态资源 (如: .html、.img、.text、.xml 等)，IIS 会将文件的内容直接响应给客户端，对于动态资源 (如: .aspx、.asp、.php 等)，IIS 则通过扩展名从 IIS 的脚本映射 (Script

Map) 中找到相应的 ISAPI 动态链接库, aspnet_isapi.dll 会被加载, ASP.NET ISAPI 随后会创建 ASP.NET 的工作进程。IIS 进程与工作进程间通过命名管道 (Name Pipes) 进行通信 [1]。

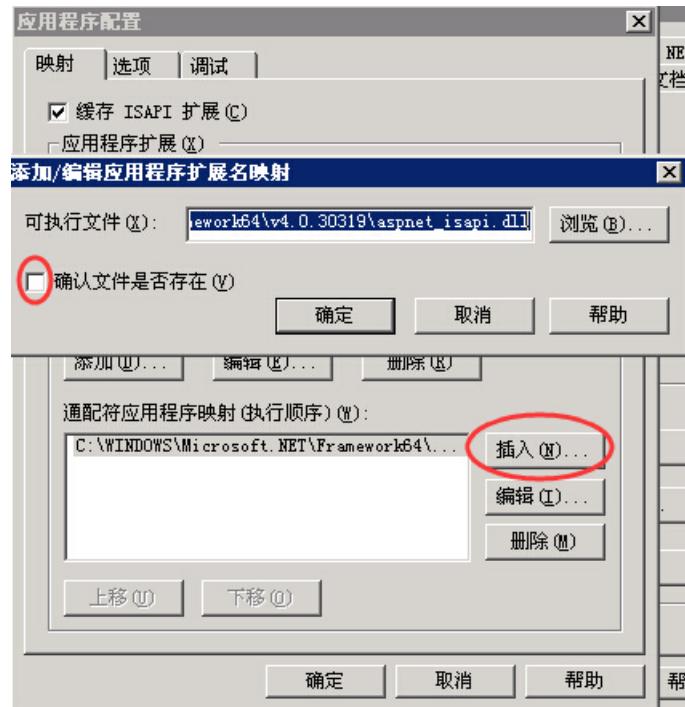


图 5.11: IIS6 配置 MVC4 应用程序配置

从以上叙述能够大致明白为什么需要手动配置 aspnet_isapi.dll。需要注意的是, 在添加 aspnet_isapi.dll 路径时, 尽量在“浏览”的弹出窗体中手动选择, 以避免复制粘贴时由于大小写等不一致造成问题。如果出现此问题: Compilation Error: [No relevant source lines]。那么可以设置应用程序池属性的标识选项卡中的预定义账户, 如图5.12所示。如果页面中提示 403 Forbidden, 可以将应用程序池标志中的预定义账户设置为“本地系统”。

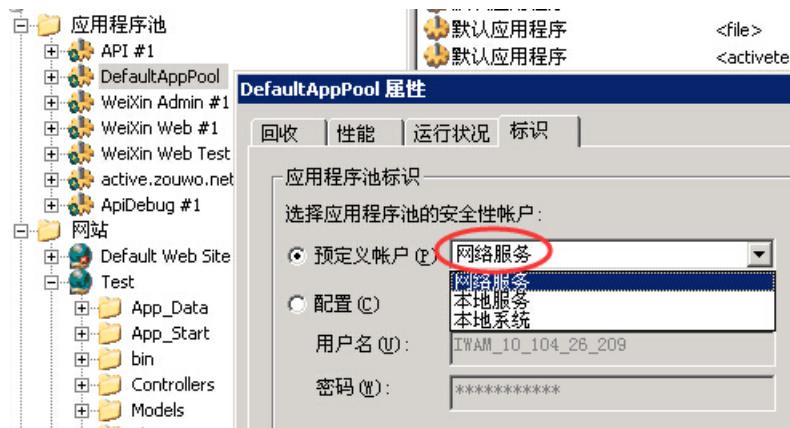


图 5.12: 应用程序池标识设置

在部署完毕后浏览网页时可能出现错误, 如代码5.5所示进行设置, 可使 IIS 返回详细的错误信息, 便于进一步诊断错误。On 表示在本地和远程用户都会看到自定义错误信息。Off 禁用自定义错误信息, 本地和远程用户都会看到详细的错误信息。RemoteOnly 表示本地用户将看到

详细错误信息，而远程用户将会看到自定义错误信息。当我们访问 asp.net 应用程时所使用的机器和发布 asp.net 应用程序所使用的机器为同一台机器时成为本地用户，反之则称之为远程用户。在开发调试阶段为了便于查找错误 Mode 属性建议设置为 Off，而在部署阶段应将 Mode 属性设置为 On 或者 RemoteOnly，以避免这些详细的错误信息暴露了程序代码细节从而引来黑客的入侵。

Listing 5.5: 自定义错误设置

```

1 <system.web>
2   <customErrors mode="Off">
3     </customErrors>
4 </system.web>
```

5.3.4 HTML

 标签被用来组合文档中的行内元素（inline element），也称作内联元素，例如：

```

1 <div class="content-text">今日剩余抽奖次数: <span id="lastCount"></span>次</
    div>
```

在这里 span 标签被用来替换页面中的可变元素。span 是一个标准的行内元素。一个行内元素可以在段落中 像这样 包裹一些文字而不会打乱段落的布局。a 元素是最常用的行内元素，它可以被用作链接。与行类元素类似的是块级元素，div 是一个标准的块级元素。一个块级元素会新开始一行并且尽可能撑满容器。其他常用的块级元素包括 p、form 和 HTML5 中的新元素：header、footer、section 等等。

```

1 <form name="form" method="post">
2   <center>
3     <textarea id="code" name="qrcode" cols="60"></textarea>
4   </center>
5   <center>
6     <input type="submit" class="btn btn-block btn-lg btn-danger" value="点击验
        证" />
7   </center>
8 </form>
```

提交页面通过添加 form 标签，指定 form 的 method 为 post，后台通过 Request.Form["qrcode"] 的形式接收参数，其中 qrcode 为文本框的名称。

5.3.5 AJax 跨域请求

获取服务器上面的打印数据的 JavaScript 代码：

```

1 function getPrintList(id, isRefresh) {
2   var url = api_path + "/weixin/printdata?id=" + id;
```

```

3   url += isRefresh ? "&an=refresh" : "";
4   var obj = $(".print-list ul");
5   $.ajax({
6       type: "get",
7       async: true,
8       url: url,
9       dataType: "jsonp",
10      jsonp: "callbackparam",
11      jsonpCallback: "success_jsonpCallback",
12      success: function (pjson) {
13          var tpl = '<li data-id="{guid}" step="{step}">打印机空闲...</p>");
19              return;
20          }
21          for (var i = 0; i < json.length; i++) {
22              var waitingSeconds = i * print_speed;
23              var stime = "";
24              if (waitingSeconds >= 60 && waitingSeconds % 60 > 0) {
25                  stime = waitingSeconds % 60 + "秒"
26              }
27              var time = waitingSeconds >= 60 ? parseInt(waitingSeconds / 60) + "分" +
28                stime + "后" : waitingSeconds + "秒后";
29              if (waitingSeconds == 0) {
30                  time = "请稍候...";
31              }
32              lihtml = tpl.replace("{url}", json[i].imageUrl).replace("{date}", time).replace
33                ("{guid}", json[i].guid).replace("{step}", json[i].step) + lihtml;
34          }
35          var printQueueNum = pjson[0].count - 1;
36          lihtml = '<li class="more"><p>.....</p><span></span><p>共' +
37            printQueueNum + '人排队</p></li>' + lihtml;
38          obj.html(lihtml);
39          obj.find("li [step='5']").addClass("active").find("p").text("正在打印...");
```

其中 dataType 为 jsonp, JSONP(JSON with Padding) 是一个非官方的协议, 它允许在服务器端集成 Script tags 返回至客户端, 通过 javascript callback 的形式实现跨域访问 (这仅仅是 JSONP 简单的实现形式)。由于同源策略⁹的限制, XMLHttpRequest 只允许请求当前源 (域

⁹同源策略 (Same Origin Policy) 是一种约定, 它是浏览器最核心也最基本的安全功能, 如果缺少了同源策略, 则浏览器的正常功能可能都会受到影响。可以说 Web 是构建在同源策略基础之上的, 浏览器只是针对同源策略的一种实现。所谓同源是指, 域名, 协议, 端口相同。

名、协议、端口) 的资源, 为了实现跨域请求, 可以通过 script 标签实现跨域请求, 然后在服务端输出 JSON 数据并执行回调函数, 从而解决了跨域的数据请求。

Ajax¹⁰向后台提交数据, 表单代码如下:

```

1 <div class="container">
2   <center style="margin-bottom:20px">
3     <input type="tel" name="qrcode" id="qrcode" />
4   </center>
5   <center>
6     <input id="inputbyhand" type="button" class="btn btn-block btn-lg btn-
danger" onclick="btsave();" value="手动输入验证" />
7   </center>
8 </div>
```

这里是当点击界面上手动输入按钮时, 通过 POST 方式将二维码值传入后台。第一个 input 的 type 为 tel 时, 可以在移动端直接调出数字输入的界面。第二个由 onclick 事件调取脚本中的 btsave 方法, 方法如下代码所示。

```

1 function btsave() {
2   var code = $("#qrcode").val();
3   jQuery.ajax({
4     url: '/my/qrcode',// 跳转到 action
5     data: { "qrcode": code },
6     type: 'post',
7     cache: false,
8     dataType: 'json',
9     success: function (data) {
10       alert(data.msg);
11       $("#qrcode").val('');
12     },
13     error: function () {
14       alert("异常! ");
15     }
16   });
17 }
```

由于使用的是 NVlocity 模板, Ajax 请求会与模板中的请求写法冲突, 所以此处的 Ajax 请求写法为 jQuery.ajax 而不是 \$.ajax, 获取后台的 AjaxResult.msg 返回信息用 data.msg。

5.3.6 文件上传

HTML5 上传

```

1 <form enctype="multipart/form-data" id="form1" action="/QiuJiFangJiaoHui/
OriginalUploadImage" method="post">
```

¹⁰AJAX = Asynchronous JavaScript and XML (异步的 JavaScript 和 XML)。, 在 2005 年, Google 通过其 Google Suggest 使 Ajax 变得流行起来。

```

2   <input type="hidden" id="userId" name="userId" value="@ ViewData["userId"]" /
3     >
4   <input type="hidden" id="activityId" name="activityId" value="@ ViewData["
5     activityId"]" />
6   <div>
7     <input type="file" style="width:220px;margin:20px auto;" id="
8       userUploadFile" name="filename" accept="image/*" class='btn btn-warning' /
9     >
10    <a href="#">
11      
13    </a>
14  </div>
15 </form>

```

```

1 // 上传图片
2 public JsonResult OriginalUploadImage(int userId, int activityId)
3 {
4   Stream fileStream = Request.Files["filename"].InputStream;
5 }

```

拖放上传

此处的文件上传在 MVC 结构项目中实现，拖放上传在服务器端的接收代码如片段所示。

```

1 // 上传图片
2 public ActionResult Upload()
3 {
4   try
5   {
6     var requestFiles = Request.Files①;
7     foreach (string uploadFiles in requestFiles)
8     {
9       string filename = Guid.NewGuid() + ".jpg";
10      string path = Path.Combine(Server.MapPath("~/App_Data"), filename);
11      var requestFilesInstance = Request.Files[uploadFiles];
12      if (requestFilesInstance != null)
13      {
14        requestFilesInstance.SaveAs(path);
15        ViewBag.Message = "File(s) uploaded successfully";
16      }
17      else
18      {
19        _logger.Error("requestFilesInstance is null");
20      }
21    }
22  }
23  catch (Exception e)
24  {

```

```

25     _logger.Error("Upload file failed!", e);
26 }
27
28     return RedirectToAction("Index");
29 }
```

- ① Gets the collection of files uploaded by the client, in multipart MIME format. The file collection is populated only when the HTTP request Content-Type value is "multipart/form-data".

multipart/form-data 的请求头必须包含一个特殊的头信息：Content-Type，且其值也必须规定为 multipart/form-data，同时还需要规定一个内容分割符用于分割请求体中的多个 post 的内容，内容分割符在 Fiddler 中捕获到如图5.13所示。

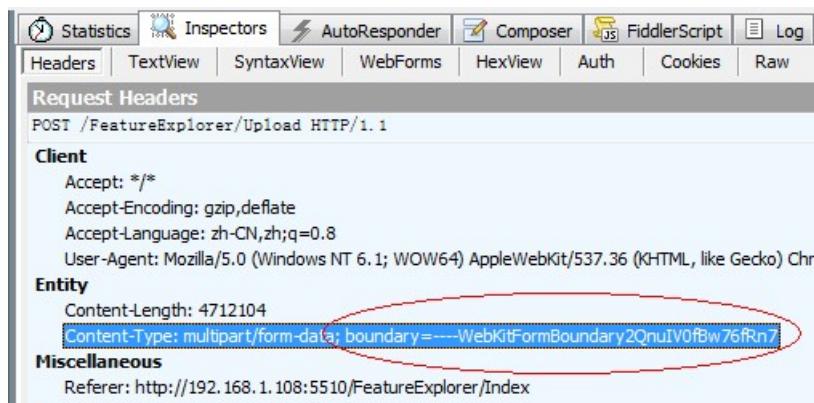


图 5.13: multipart/form-data 请求的内容分割符

从图中可以看出，分隔符的内容为：——WebKitFormBoundary2QnuIV0fBw76fRn7 (Google Chrome Browser)。在分隔符后就是原生的图片内容，如图5.14所示。图片数据流的最后以分隔符为标记。multipart/form-data 的请求体也是一个字符串，不过和 post 的请求体不同的是它的构造方式，post 是简单的 name=value 值连接，而 multipart/form-data 则是添加了分隔符等内容的构造体。如文件内容和文本内容自然需要分割开来，不然接收方就无法正常解析和还原这个文件了。

上传图片时会提示超过了最大请求长度，原因是由于 asp.net 默认最大上传文件大小为 4M，运行超时时间为 90s。在 Web.config 中添加如下配置可以改变这个默认值。

```

1 <system.web>
2     <httpRuntime maxRequestLength="4096" executionTimeout="3600" />
3 </system.web>
```

前端的页面监测用户的拖放操作，并利用 HTML5 的 FormData 进行文件的提交。

```

1 if (tests.dnd) {
2     holder.ondragover = function () { this.className = "hover"; return false; };
```

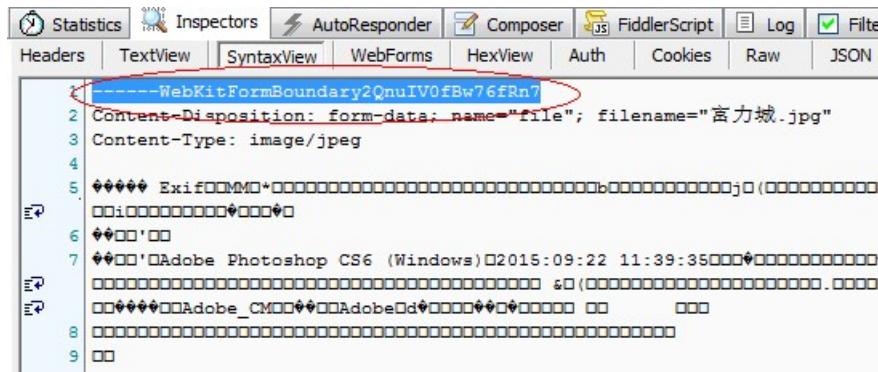


图 5.14: multipart/form-data 请求的内容分割符标记图片流数据

```

3 holder.ondragend = function () { this.className = ""; return false; };
4 holder.ondrop = function (e) {
5     this.className = "";
6     e.preventDefault();
7     readfiles(e.dataTransfer.files);
8 }
9 } else {
10     fileupload.className = "hidden";
11     fileupload.querySelector('input').onchange = function ()
12         readfiles(this.files);
13     };
14 }
```

XMLHttpRequest Level 2 添加了一个新的接口——FormData。利用 FormData 对象，我们可以通过 JavaScript 用一些键值对来模拟一系列表单控件，我们还可以使用 XMLHttpRequest 的 send() 方法来异步的提交表单。与普通的 Ajax 相比，使用 FormData 的最大优点就是我们可以异步上传二进制文件。

5.4 Cookie

大多数浏览器支持最大为 4096 字节的 Cookie。由于这限制了 Cookie 的大小，最好用 Cookie 来存储少量数据，或者存储用户 ID 之类的标识符。用户 ID 随后便可用于标识用户，以及从数据库或其他数据源中读取用户信息。浏览器还限制站点可以在用户计算机上存储的 Cookie 的数量。大多数浏览器只允许每个站点存储 20 个 Cookie；如果试图存储更多 Cookie，则最旧的 Cookie 便会被丢弃。有些浏览器还会对它们将接受的来自所有站点的 Cookie 总数作出绝对限制，通常为 300 个。通过前面的内容，我们了解到 Cookie 是用于维持服务端会话状态的，通常由服务端写入，在后续请求中，供服务端读取。下面本文将按这个过程看看 Cookie 是如何从服务端写入，最后如何传到服务端以及如何读取的。

会话 Cookie(Session Cookie) 不设置过期时间，则表示这个 Cookie 生命周期为浏览器会话期间，只要关闭浏览器窗口，Cookie 就消失了。生命周期为浏览器会话期。一般不保存在硬盘上

而是保存在内存里。

持久性 Cookie(Persistent Cookie) 设置了过期时间，浏览器就会把 Cookie 保存到硬盘上，关闭后再次打开浏览器，这些 Cookie 依然有效直到超过设定的过期时间。保存在用户硬盘上面，同一浏览器可以获取。

HTTP 协议是无状态的，同一个客户端的这次请求和上次请求是没有对应关系，对 HTTP 服务器来说，它并不知道这两个请求来自同一个客户端。为了解决这个问题，Web 程序引入了 Cookie 机制来维护状态。Cookie 是由服务器端生成，发送给 User-Agent（一般是浏览器），浏览器会将 Cookie 的 key/value 保存到某个目录下的文本文件内，下次请求同一网站时就发送该 Cookie 给服务器（前提是浏览器设置为启用 Cookie）。Cookie 是可以被 Web 服务器设置的字符串，并且可以保存在浏览器中。如图5.15所示，当浏览器访问了页面 1 时，web 服务器设置了一个 Cookie，并将这个 Cookie 和页面 1 一起返回给浏览器，浏览器接到 Cookie 之后，就会保存起来，在它访问页面 2 的时候会把这个 Cookie 也带上，Web 服务器接到请求时也能读出 Cookie 的值，根据 Cookie 值的内容就可以判断和恢复一些用户的信息状态。



图 5.15: Cookie HTTP 保持状态

在 Google Chrome 的 Cookie 存储路径为：

```
1 C:\Documents and Settings\Administrator\Local Settings\Application Data\Google\Chrome
    \User Data
```

Mozilla Firefox 的存储路径为：

```
1 C:\Documents and Settings\Administrator\AppData\Local\Mozilla\Firefox\Profiles\
    d4l4tvw6.default\OfflineCache
```

Google Chrome 和 Firefox 的 Cookie 存放在 sqlite 数据库中，也可以使用 sqlite 数据库工具直接打开文件进行查看，不过由于密码是 BLOB 格式存储的，不能看到明文密码，Cookie 名称和值可以由服务器端开发自己定义，对于 JSP 而言也可以直接写入 jsessionid，这样服务器可以知道该用户是否合法用户以及是否需要重新登录等，服务器可以设置或读取 Cookies 中包含信息，借此维护用户跟服务器会话中的状态。Cookies 最典型的应用是判定注册用户是否已经登录网站，用户可能会得到提示，是否在下一次进入此网站时保留用户信息以便简化登录手续，这些都是 Cookies 的功用。另一个重要应用场合是“购物车”之类处理。用户可能会在一段时间内在同一家网站的不同页面中选择不同的商品，这些信息都会写入 Cookies，以便在最后付款时

提取信息。通过上面这个例子，可以看到 Cookie 是很重要的，识别是否是登陆用户，就是通过 Cookie。假如截获了别人的 Cookie 是否可以冒充他人的身份登陆呢？当然可以，这就是一种黑客技术叫 Cookie 欺骗。利用 Cookie 欺骗，不需要知道用户名密码。就可以直接登录，使用别人的账户做事。有两种方法可以截获他人的 Cookie：

1. 通过 XSS¹¹脚本攻击即跨站脚本攻击 (Cross Site Scripting)，获取他人的 Cookie.
2. 想办法获取别人电脑上保存的 Cookie 文件 (这个比较难)。

5.4.1 编辑 Cookie

在 Google Chrome 中编辑 Cookie 需要安装 Edit This Cookie 插件，在 FireFox 中可通过 Firebug 编辑 Cookie，或者添加 Cookie，如图5.16所示。



图 5.16: 采用 FireDebug 编辑 Cookie

需要注意 Cookie 的有效时间，如果有效时间小于等于当前时间时需要调整有效时间，否则添加的 Cookie 会因为时间超出而立即失效，在编辑 Cookie 指定域的时候最好指定网址，而不是 IP 地址 + 端口的形式，因为 Cookie 在端口支持方面至今未有标准，Cookie 可以说是不分端口 (Cookies do not provide isolation by port)，Cookie 跟站点（域名）对应，例如同一个主机端口 8889 保存的 Cookie 在其他的端口同样能够读取¹²。

For historical reasons, cookies contain a number of security and privacy infelicities. For example, a server can indicate that a given cookie is intended for "secure" connections, but the Secure attribute does not provide integrity in the presence of an active network attacker. Similarly, cookies for a given host are shared across all the ports on that host, even though the usual "same-origin policy" used by web browsers isolates content retrieved via different ports.

Cookies do not provide isolation by port. If a cookie is readable by a service running on one port, the cookie is also readable by a service running on another port of the same server. If a cookie is writable by a service on one port, the cookie is also writable by a service running on another port of the same server. For this reason, servers SHOULD NOT both run mutually distrusting services on different ports of the same host and use cookies to store security sensitive information.

¹¹为不和层叠样式表 (Cascading Style Sheets, CSS) 的缩写混淆，故将跨站脚本攻击缩写为 XSS

¹²<http://stackoverflow.com/questions/1612177/are-http-cookies-port-specific>

5.5 Session

Session 又称为会话状态，是 Web 系统中最常用的状态，用于维护和当前浏览器实例相关的一些信息。举个例子来说，我们可以把已登录用户的用户名放在 Session 中，这样就能通过判断 Session 中的某个 Key 来判断用户是否登录，如果登录的话用户名又是多少。Session 对于每一个客户端（或者说浏览器实例）是“人手一份”，用户首次与 Web 服务器建立连接的时候，服务器会给用户分发一个 SessionID 作为标识。SessionID 是一个由 24 个字符组成的随机字符串。用户每次提交页面，浏览器都会把这个 SessionID 包含在 HTTP 头中提交给 Web 服务器，这样 Web 服务器就能区分当前请求页面的是哪一个客户端。Session 存储在服务器端，一般为了防止在服务器的内存中（为了高速存取），Session 在用户访问第一次访问服务器时创建，需要注意只有访问 JSP、Servlet 等程序时才会创建 Session，只访问 HTML、IMAGE 等静态资源并不会创建 Session，可调用 `request.getSession(true)` 强制生成 Session。服务器会把长时间没有活动的 Session 从服务器内存中清除，此时 Session 便失效。Tomcat 中 Session 的默认失效时间为 20 分钟。虽然 Session 保存在服务器，对客户端是透明的，它的正常运行仍然需要客户端浏览器的支持。这是因为 Session 需要使用 Cookie 作为识别标志。HTTP 协议是无状态的，Session 不能依据 HTTP 连接来判断是否为同一客户，因此服务器向客户端浏览器发送一个名为 JSESSIONID 的 Cookie，它的值为该 Session 的 id（也就是 `HttpSession.getId()` 的返回值）。Session 依据该 Cookie 来识别是否为同一用户。Session 机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。当程序需要为某个客户端的请求创建一个 Session 的时候，服务器首先检查这个客户端的请求里是否已包含了一个 session 标识 - 称为 session id，如果已包含一个 session id 则说明以前已经为此客户端创建过 Session，服务器就按照 session id 把这个 Session 检索出来使用（如果检索不到，可能会新建一个），如果客户端请求不包含 session id，则为此客户端创建一个 Session 并且生成一个与此 Session 相关联的 session id，session id 的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个 session id 将被在本次响应中返回给客户端保存。保存这个 session id 的方式可以采用 Cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发送给服务器。一般这个 Cookie 的名字都是类似于 SEESIONID，而。比如 WebLogic 对于 Web 应用程序生成的 Cookie:

```
1 JSESSIONID=ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng  
!-145788764
```

它的名字就是 JSESSIONID，由于 Cookie 可以被人为的禁止，必须有其他机制以便在 Cookie 被禁止时仍然能够把 session id 传递回服务器。经常被使用的一种技术叫做 URL 重写，就是把 session id 直接附加在 URL 路径的后面，附加方式也有两种，一种是作为 URL 路径的附加信息，表现形式为

```
1 http://..../xxx;jsessionid=  
ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764
```

另一种是作为查询字符串附加在 URL 后面，表现形式为

```
1 http://..../xxx?jsessionid=
```

ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764

这两种方式对于用户来说是没有区别的，只是服务器在解析的时候处理的方式不同，采用第一种方式也有利于把 session id 的信息和正常程序参数区分开来。为了在整个交互过程中始终保持状态，就必须在每个客户端可能请求的路径后面都包含这个 session id。

5.5.1 Session 的优点与限制

Session 的优点是它可以自动的进行页面间的参数传递。不需要表单传递、不需要超链接传递，只需要在后台传递。而且设了一次 session 之后，就可以在每一个页面使用，真的非常方便。如果网站很大，一不小心设了两个同名的 session，就会造成错误，尤其是象 session(“id”) 这类变量，很容易出错。session 会占用系统资源，而且在 20 分钟内没有连接的情况下才会自动消失。所以一旦大量使用，将会对系统造成严重影响。

5.5.2 Session 的生命周期

Session 是在用户第一次访问网站的时候创建的，那么 Session 是什么时候销毁的呢？Session 使用一种平滑超时的技术来控制何时销毁 Session。默认情况下，Session 的超时时间（Timeout）是 20 分钟，用户保持连续 20 分钟不访问网站，则 Session 被收回，如果在这 20 分钟内用户又访问了一次页面，那么 20 分钟就重新计时了，也就是说，这个超时是连续不访问的超时时间，而不是第一次访问后 20 分钟必过时。这个超时时间同样也可以通过调整 Web.config 文件进行修改。别太相信 Session 的 Timeout 属性，如果你把它设置为 24 小时，则很难相信 24 小时之后用户的 Session 还在。Session 是否存在，不仅仅依赖于 Timeout 属性，以下的情况都可能引起 Session 丢失（所谓丢失就是在超时以前原来的 Session 无效）。

- **bin 目录中的文件被改写** asp.net 有一种机制，为了保证 dll 重新编译之后，系统正常运行，它会重新启动一次网站进程，这时就会导致 Session 丢失，所以如果有 access 数据库位于 bin 目录，或者有其他文件被系统改写，包括创建新文件、修改文件名、修改文件内容、删除文件、修改目录名、删除目录就会导致 Session 丢失，新建目录 IIS 不会重启。
- **SessionID 丢失或者无效** 如果你在 URL 中存储 SessionID，但是使用了绝对地址重定向网站导致 URL 中的 SessionID 丢失，那么原来的 Session 将失效。如果你在 Cookie 中存储 SessionID，那么客户端禁用 Cookie 或者 Cookie 达到了 IE 中 Cookie 数量的限制（每个域 20 个），那么 Session 将无效。
- **如果使用 InProc 的 Session，那么 IIS 重启将会丢失 Session** 同理，如果使用 StateServer 的 Session，服务器重新启动 Session 也会丢失。
- **进程用户名更改权限后** 例如 Network Service 更改权限后也会导致 IIS 重新启动，Session 会丢失。

一般来说，如果在 IIS 中存储 Session 而且 Session 的 Timeout 设置得比较长，再加上 Session 中存储大量的数据，非常容易发生 Session 丢失的问题。最后，Session 的安全性怎么样

呢？我们知道，Session 中只有 SessionID 是存储在客户端的，并且在页面每次提交的过程中加入 HTTP 头发送给服务器。SessionID 只是一个识别符，没有任何内容，真正的内容是存储在服务器上的。总的来说安全性还是可以的，不过笔者建议你不要使用 cookieless 和 SqlServer 模式的 Session。把 SessionID 暴露在 URL 中，把内容存储在数据库中可能会发生攻击隐患。

5.5.3 网页嵌入视频

网页嵌入视频代码如下：

```

1 <div>
2   <center style="margin-bottom:20px">
3     <embed src="http://player.youku.com/player.php/sid/XNzA1ODUyNjYw/v.swf"
4       allowfullscreen="true"
5       quality="high"
6       width="480"
7       height="400"
8       align="middle"
9       allowscriptaccess="always"
10      type="application/x-shockwave-flash" />
11  </center>
12 </div>

```

例如网页中嵌入 FSF 的介绍视频：

```

1 <video src="//static.fsf.org/nosvn/FSF30-video/FSF_30_720p.webm" controls width="
640" height="390"></video>

```

5.6 缓存 (Cache)

网页缓存由 HTTP 消息头中的 Cache-control 控制，常见取值有 private、no-cache、max-age、must-revalidate 等，默認為 private。浏览器会自动缓存网页的 css.js 等文件，有时为了强制浏览器进行缓存刷新，可修改 css 文件的链接，例如添加版本号等等。如更新了服务器上的 example.css 文件，希望浏览器立即使用新的样式，可将链接改成如下形式。

```

1 <link rel="stylesheet" type="text/css" href="http://static.zouwo.com/lottery/1.02/
css/lottery2.css?v=1.0.1" />

```

其中？v=1.0.1 是新加入的内容，此时 css 链接与本地不一致，浏览器会下载新的 css 样式表。根据标准，到目前为止，HTML5 一共有 6 种缓存机制，有些是之前已有，有些是 HTML5 才新加入的。

- 浏览器缓存机制
- Dom Storage (Web Storage) 存储机制

浏览器缓存机制是指通过 HTTP 协议头里的 Cache-Control (或 Expires) 和 Last-Modified (或 Etag) 等字段来控制文件缓存的机制。这两个字段配置的是本地缓存，如果缓存命中（当前时间小于资源的过期时间）那么浏览器是不会发起 HTTP 请求而是直接加载缓存资源。这 (Expires) 是 WEB 中较早的缓存机制，是在 HTTP 1.0 协议中实现的，它描述的是一个绝对时间，由服务器返回，用 GMT 格式的字符串表示。有点不同于 Dom Storage、AppCache 等缓存机制，但本质上是一样的。可以理解为，一个是协议层实现的，一个是应用层实现的。

5.6.1 Request 缓存相关首部字段

Cache-Control Expires 和 Cache-Control 同时存在时，Cache-Control 优先级高于 Expires，Cache-Control 在 HTTP 1.1 中加入的，用于控制文件在本地缓存有效时长。最常见的，比如服务器回包：Cache-Control:max-age=600 表示文件在本地应该缓存，且有效时长是 600 秒（从发出请求算起）。在接下来 600 秒内，如果有请求这个资源，浏览器不会发出 HTTP 请求，而是直接使用本地缓存的文件。许多人以为，出于安全考虑，浏览器不会在本地保存 HTTPS 缓存。实际上，只要在 HTTP 头中使用特定命令，HTTPS 是可以缓存的。Firefox 默认只在内存中缓存 HTTPS。但是，只要头命令中有 Cache-Control: Public，缓存就会被写到硬盘上。只要 HTTP 头允许这样做，所有版本的 IE 都缓存 HTTPS 内容。比如，如果头命令是 Cache-Control: max-age=600，那么这个网页就将被 IE 缓存 10 分钟。IE 的缓存策略，与是否使用 HTTPS 协议无关。（其他浏览器在这方面的行为不一致，取决于你使用的版本）

- no-cache 不直接使用缓存，始终向服务器发起请求。表示必须先与服务器确认返回的响应是否被更改，然后才能使用该响应来满足后续对同一个网址的请求。因此，如果存在合适的验证令牌 (ETag)，no-cache 会发起往返通信来验证缓存的响应，如果资源未被更改，可以避免下载。
- no-store 禁止缓存任何响应，也就是说每次用户请求资源时，都会向服务器发送一个请求，每次都会下载完整的响应。
- max-age 缓存过期时间，是一个时间数值，比如 3600 秒，设置为 0 的时候效果等同于 no-cache
- private(default) 浏览器可以缓存 private 响应，但是通常只为单个用户缓存，因此，不允许任何代理服务器对其进行缓存。比如，用户浏览器可以缓存包含用户私人信息的 HTML 网页，但是 CDN 不能缓存。
- s-maxage 给缓存代理用的指令，对直接返回资源的 server 无效，当 s-maxage 生效时，会忽略 max-age 的值
- only-if-cached 若有缓存，则只使用缓存，若缓存文件出问题了，请求也会出问题

Last-Modified 是标识文件在服务器上的最新更新时间。下次请求时，如果文件缓存过期，浏览器通过 If-Modified-Since 字段带上这个时间，发送给服务器，由服务器比较时间戳来判断文件是否有修改。如果没有修改，服务器返回 304 告诉浏览器继续使用缓存；如果有修改，则返

回 200，同时返回最新的文件。Cache-Control 通常与 Last-Modified 一起使用。一个用于控制缓存有效时间，一个在缓存失效后，向服务查询是否有更新。Cache-Control 还有一个同功能的字段：Expires。Expires 的值一个绝对的时间点，如：Expires: Thu, 10 Nov 2015 08:45:11 GMT，表示在这个时间点之前，缓存都是有效的。Expires 是 HTTP1.0 标准中的字段，Cache-Control 是 HTTP1.1 标准中新加的字段，功能一样，都是控制缓存的有效时间。当这两个字段同时出现时，Cache-Control 是高优化级的。分布式系统里多台机器间文件的 Last-Modified 必须保持一致，以免负载均衡到不同机器导致比对失败。

Etag(Entity Tag) Etag 也是和 Last-Modified 一样，对文件进行标识的字段。不同的是，Etag 的取值是一个对文件进行标识的特征字串。在向服务器查询文件是否有更新时，浏览器通过 If-None-Match 字段把特征字串发送给服务器，由服务器和文件最新特征字串进行匹配，来判断文件是否有更新。没有更新回包 304，有更新回包 200。Etag 和 Last-Modified 可根据需求使用一个或两个同时使用。两个同时使用时，只要满足其中一个条件，就认为文件没有更新。Last-Modified 需要向服务器发起查询请求，才能知道资源文件有没有更新。虽然服务器可能返回 304 告诉没有更新，但也还有一个请求的过程。对于移动网络，这个请求可能是比较耗时的。有一种说法叫“消灭 304”，指的就是优化掉 304 的请求。抓包发现，带 if-Modified-Since 字段的请求，如果服务器回包 304，回包带有 Cache-Control:max-age 或 Expires 字段，文件的缓存有效时间会更新，就是文件的缓存会重新有效。304 回包后如果再请求，则又直接使用缓存文件了，不再向服务器查询文件是否更新了，除非新的缓存时间再次过期。分布式系统尽量关闭掉 ETag(每台机器生成的 ETag 都会不一样)。

另外，Cache-Control 与 Last-Modified 是浏览器内核的机制，一般都是标准的实现，不能更改或设置。以 QQ 浏览器的 X5 为例，Cache-Control 与 Last-Modified 缓存不能禁用。缓存容量是 12MB，不分 HOST，过期的缓存会最先被清除。如果都没过期，应该优先清最早的缓存或最快到期的或文件大小最大的；过期缓存也有可能还是有效的，清除缓存会导致资源文件的重新拉取。

还有，浏览器，如 X5，在使用缓存文件时，是没有对缓存文件内容进行校验的，这样缓存文件内容被修改的可能。

分析发现，浏览器的缓存机制还不是非常完美的缓存机制。完美的缓存机制应该是这样的：

1. 缓存文件没更新，尽可能使用缓存，不用和服务器交互
2. 缓存文件有更新时，第一时间能使用到新的文件
3. 缓存的文件要保持完整性，不使用被修改过的缓存文件
4. 缓存的容量大小要能设置或控制，缓存文件不能因为存储空间限制或过期被清除

以 X5 为例，第 1、2 条不能同时满足，第 3、4 条都不能满足。在实际应用中，为了解决 Cache-Control 缓存时长不好设置的问题，以及为了“消灭 304”，Web 前端采用的方式是：在要缓存的资源文件名中加上版本号或文件 MD5 值字串，如 common.d5d02a02.js，common.v1.js，同时设置 Cache-Control:max-age=31536000，也就是一年。在一年时间内，资源文件如果本地

有缓存，就会使用缓存；也就不会有 304 的回包。如果资源文件有修改，则更新文件内容，同时修改资源文件名，如 common.v2.js，html 页面也会引用新的资源文件名。通过这种方式，实现了：缓存文件没有更新，则使用缓存；缓存文件有更新，则第一时间使用最新文件的目的。即上面说的第 1、2 条。第 3、4 条由于浏览器内部机制，目前还无法满足。因为 Cache-Control 与 Expires 的作用一致，Last-Modified 与 ETag 的作用也相近。但它们有以下区别，如图5.17所示：

现在默认浏览器均默认使用 HTTP 1.1，所以 Expires 和 Last-Modified 的作用基本可以忽略，具备 Cache-Control 和 Etag 即可。

Pragma 用来做缓存过期判断，它可以取值 no-cache，这是一个 http1.0 遗留的字段，当它和 Cache-Control 同时存在的时候，会被 Cache-Control 覆盖

if-match/if-none-match 用来做资源更新判断，这个指令会把缓存中的 Etag 传给服务器，服务器用它来和服务器端的资源 Etag 进行对比，若不一致则证明资源被修改了，需要响应请求为 200 OK

if-modified-since 用来做资源更新判断，这个指令会把文件的上一次缓存中的文件的更新时间传给服务器，服务器判断文件在这个时间点后是否被修改，如果被修改过则需要响应请求为 200 OK

5.6.2 Response 缓存相关首部字段

5.6.3 System.Web.Caching

System.Web.Caching 命名空间提供用于缓存服务器上常用数据的类。这包括 Cache 类，该类是一个使您可以存储任意数据对象（如哈希表和数据集）的词典。它还为这些对象提供到期功能，并提供使您可以添加和移除对象的方法。您还可以添加依赖于其他文件或缓存项的对象，



图 5.17: HTTP Cache-Control 和 Etag 区别

并在从 Cache 中移除对象时执行回调以通知应用程序。将页面与 Controller 和 Action 对应的数据存储在 XML 配置文件中。

5.7 浏览器渲染 (Browser Render)

Javascript 的使用一定要在 Javascript 的引用之后，否则会出现未定义 (undefined) 错误，例如页面中的 JavaScript 脚本使用了如下语句：

```
1 config.backUrl="http://www.baidu.com"
```

但是 config 对象在另一个 JavaScript 脚本中，那么引用脚本一定要在加载脚本之前。`<script>` 标签引入脚本有三种情况：

立即执行 `<script src="a.js"><script src="b.js">` 顺序：保证先后顺序。解析：HTML 解析器遇到它们时，将阻塞（停止解析标签后的文档），待脚本下载完成并执行后，继续解析标签之后的文档。

推迟执行 `<script defer src="a.js"> <script defer src="b.js">` 顺序：保证先后顺序。解析：HTML 解析器遇到它们时，不阻塞（脚本将被异步下载），待文档解析完成之后，执行脚本。

尽快执行 `<script async src="a.js"><script async src="b.js">` 顺序：不保证先后顺序。解析：HTML 解析器遇到它们时，不阻塞（脚本将被异步下载，一旦下载完成，立即执行它），并继续解析之后的文档。

HTML 页面加载和解析流程：

- 用户输入网址（假设是个 html 页面，并且是第一次访问），浏览器向服务器发出请求，服务器返回 html 文件
- 浏览器开始载入 html 代码，发现 `<head>` 标签内有一个 `<link>` 标签引用外部 CSS 文件
- 浏览器又发出 CSS 文件的请求，服务器返回这个 CSS 文件
- 浏览器继续载入 html 中 `<body>` 部分的代码，并且 CSS 文件已经拿到手了，可以开始渲染页面了
- 浏览器在代码中发现一个 `` 标签引用了一张图片，向服务器发出请求。此时浏览器不会等到图片下载完，而是继续渲染后面的代码
- 服务器返回图片文件，由于图片占用了一定面积，影响了后面段落的排布，因此浏览器需要回过头来重新渲染这部分代码
- 浏览器发现了一个包含一行 Javascript 代码的 `<script>` 标签，赶快运行它

- Javascript 脚本执行了这条语句, 它命令浏览器隐藏掉代码中的某个 `<style>(style.display="none")`。杯具啊, 突然就少了这么一个元素, 浏览器不得不重新渲染这部分代码

9. 终于等到了 `</html>` 的到来, 浏览器泪流满面……10. 等等, 还没完, 用户点了一下界面中的“换肤”按钮, Javascript 让浏览器换了一下 `<link>` 标签的 CSS 路径。11. 浏览器召集了在座的各位 `<div>` 们, “大伙儿收拾收拾行李, 咱得重新来过……”, 浏览器向服务器请求了新的 CSS 文件, 重新渲染页面。

reflow 几乎是无法避免的。现在界面上流行的一些效果, 比如树状目录的折叠、展开 (实质上是元素的显示与隐藏) 等, 都将引起浏览器的 reflow。鼠标滑过、点击……只要这些行为引起了页面上某些元素的占位面积、定位方式、边距等属性的变化, 都会引起它内部、周围甚至整个页面的重新渲染。reflow 问题是可优化的, 我们可以尽量减少不必要的 reflow。例子中的 `` 图片载入问题, 这其实就是一个可以避免的 reflow——给图片设置宽度和高度就可以了。这样浏览器就知道了图片的占位面积, 在载入图片前就预留好了位置。另外, 有个和 reflow 看上去差不多的术语: repaint, 中文叫重绘。如果只是改变某个元素的背景色、文字颜色、边框颜色等等不影响它周围或内部布局的属性, 将只会引起浏览器 repaint。repaint 的速度明显快于 reflow (在 IE 下需要换一下说法, reflow 要比 repaint 更缓慢)。



图 5.18: 浏览器渲染 TimeLine

5.7.1 重绘 (Redraw) 与重排 (Reflow)

浏览器从下载文档到显示页面的过程是个复杂的过程, 这里包含了重绘和重排。各家浏览器引擎的工作原理略有差别, 但也一定规则。简单讲, 通常在文档初次加载时, 浏览器引擎会解析 HTML 文档来构建 DOM 树, 之后根据 DOM 元素的几何属性构建一棵用于渲染的树。渲染树的每个节点都有大小和边距等属性, 类似于盒子模型 (由于隐藏元素不需要显示, 渲染树中并不包含 DOM 树中隐藏的元素)。当渲染树构建完成后, 浏览器就可以将元素放置到正确的位置了, 再根据渲染树节点的样式属性绘制出页面。由于浏览器的流布局, 对渲染树的计算通常只需要遍历一次就可以完成。但 table 及其内部元素除外, 它可能需要多次计算才能确定好其

在渲染树中节点的属性，通常要花 3 倍于同等元素的时间。这也是为什么我们要避免使用 table 做布局的一个原因。

重绘 (Redraw) 是一个元素外观的改变所触发的浏览器行为，例如改变 visibility、outline、背景色等属性。浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。重绘不会带来重新布局，并不一定伴随重排。**重排 (Redraw)** 是更明显的一种改变，可以理解为渲染树需要重新计算。下面是常见的触发重排的操作：

1. DOM 元素的几何属性变化

当 DOM 元素的几何属性变化时，渲染树中的相关节点就会失效，浏览器会根据 DOM 元素的变化重新构建渲染树中失效的节点。之后，会根据新的渲染树重新绘制这部分页面。而且，当前元素的重排也许会带来相关元素的重排。例如，容器节点的渲染树改变时，会触发子节点的重新计算，也会触发其后续兄弟节点的重排，祖先节点需要重新计算子节点的尺寸也会产生重排。最后，每个元素都将发生重绘。可见，重排一定会引起浏览器的重绘，一个元素的重排通常会带来一系列的反应，甚至触发整个文档的重排和重绘，性能代价是高昂的。

2. DOM 树的结构变化

当 DOM 树的结构变化时，例如节点的增减、移动等，也会触发重排。浏览器引擎布局的过程，类似于树的前序遍历，是一个从上到下从左到右的过程。通常在这个过程中，当前元素不会再影响其前面已经遍历过的元素。所以，如果在 body 最前面插入一个元素，会导致整个文档的重新渲染，而在其后插入一个元素，则不会影响到前面的元素。

3. 获取某些属性

浏览器引擎可能会针对重排做了优化。比如 Opera，它会等到有足够的数量的变化发生，或者等到一定的时间，或者等一个线程结束，再一起处理，这样就只发生一次重排。但除了渲染树的直接变化，当获取一些属性时，浏览器为取得正确的值也会触发重排。这样就使得浏览器的优化失效了。这些属性包括：offsetTop、offsetLeft、offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight、getComputedStyle() (currentStyle in IE)。所以，在多次使用这些值时应进行缓存。此外，改变元素的一些样式，调整浏览器窗口大小等等也将触发重排。开发中，比较好的实践是尽量减少重排次数和缩小重排的影响范围。例如：

- 将多次改变样式属性的操作合并成一次操作。
- 将需要多次重排的元素，position 属性设为 absolute 或 fixed，这样此元素就脱离了文档流，它的变化不会影响到其他元素。例如有动画效果的元素就最好设置为绝对定位。
- 在内存中多次操作节点，完成后再添加到文档中去。例如要异步获取表格数据，渲染到页面。可以先取得数据后在内存中构建整个表格的 html 片段，再一次性添加到文档中去，而不是循环添加每一行。
- 由于 display 属性为 none 的元素不在渲染树中，对隐藏的元素操作不会引发其他元素的重排。如果要对一个元素进行复杂的操作时，可以先隐藏它，操作完成后显示。这样只在隐藏和显示时触发 2 次重排。

- 在需要经常获取那些引起浏览器重排的属性值时，要缓存到变量。

5.8 网页跳转 (Page Redirect)

在 JavaScript 中的写法，虽然 window.location 和 window.location.href 都可以达到跳转的效果，但是建议使用 window.location.href，具体原因参考以下注解¹³。

```
1 window.location = "http://www.baidu.com";
2 window.location.href = "http://www.baidu.com";①
```

- ① Like as has been said already, location is an object. But that person suggested using either. But, you will do better to use the .href version. Objects have default properties which, if nothing else is specified, they are assumed. In the case of the location object, it has a property called .href. And by not specifying ANY property during the assignment, it will assume "href" by default. This is all well and fine until a later object model version changes and there either is no longer a default property, or the default property is changed. Then your program breaks unexpectedly.

window.location 对象用于获得当前页面的地址 (URL)，并把浏览器重定向到新的页面。window.location 对象在编写时可不使用 window 这个前缀。location.href 属性返回当前页面的 URL。

5.8.1 Redirect

网页重定向在.NET 中的写法如下代码片段所示 (属于服务器端重定向)。

```
1 System.Web.HttpContext.Current.Response.Redirect("http://www.baidu.com", true);
```

第一个参数为需要重定向到的网络链接，第二个参数为是否结束此次会话(endResponse)，默认为 true。Redirect 在跳转时 url 如果写成 Home/Index，那么将在当前链接后附加 Home/Index 路径。例如当前链接为：http://192.168.1.1/Detail，那么重定向后的路径为

```
1 http://192.168.1.1/Detail/Home/Index
```

如果写成/Home/Index(注意 Home 前面多一个斜杠)，那么跳转的链接为

```
1 http://192.168.1.1/Home/Index
```

在后台链接会自动跳转到指定的 URL 地址。服务器端并没有返回请求页面的 html 数据，而是 Response 了一个 302 Found，并在 Location 中给出了目标 URL，这就是在告诉浏览器：

¹³注解的来源：<http://stackoverflow.com/questions/2383401/javascript-setting-location-href-versus-location>

请重新发出一个 HTTP 请求，所请求的 URL 为返回的 URL。浏览器于是按照吩咐，重新发出了一个 http 的请求。这就是服务器 Redirect 重定向的过程。当服务器执行到 Response.Redirect 语句时，会立即中断页面的生命周期，直接向客户端返回信息，让客户端进行重定向操作。从 Redirect(重定向) 的流程可以看出，Redirect 比较耗时，它仅用于从当前物理服务器开发跳转到其它服务器。如果只是在本服务器开发内页面跳转请使用 Server.Transfer 语法，这样会减少很多没有必要的客户端重定向。如果网站指定是永久性转移则返回 HTTP 301 状态码，如果是临时性转移，则返回 HTTP 302 状态码，两者仅仅是状态码不相同而已，都是同样的实现原理，从源码5.6 (System.Web.dll) 中可以看出。

Listing 5.6: Redirect 实现片段

```

1 this.StatusCode = permanent ? 301 : 302;
2 this.RedirectLocation = url;
3 url = !UriUtil.IsSafeScheme(url) ? HttpUtility.HtmlAttributeEncode(HttpUtility.UrlEncode(
   url)) : HttpUtility.HtmlAttributeEncode(url);
4 this.Write("<html><head><title>Object moved</title></head><body>\r\n");
5 this.Write("<h2>Object moved to <a href=\"" + url + "\">here</a>.</h2>\r\n");
6 this.Write("</body></html>\r\n");

```

5.8.2 Meta Refresh

跳转有时还需要在指定的时长后进行跳转，可用 Javascript 的 setInterval 和 setTimeOut 方法。有时前 2 种方式会失效，如在机顶盒内置浏览器中（应该是写法错误导致），那么可以采用如下语句进行指定时间后的跳转。

```

1 <meta http-equiv="refresh" content="5;url=/ShaPingBa/Index" />

```

其中 url 为跳转的目标地址，5 表示 5 秒后跳转。注意这种方式在 2000 年前比较流行，通过网页中的 meta 指令，在特定时间后重定向到新的网页，如果延迟的时间太短（约 5 秒之内），会被搜索引擎判断为 spam，遭到惩罚。在 MVC 中还可以使用 RedirectToAction 进行网页的跳转，此方法有多个重载。

5.8.3 Server.Transfer

```

1 Server.Transfer("WebForm2.aspx",True)

```

Server.Transfer 方法的另一个参数——“preserveForm”。如果你设置这个参数为 True，那么 querystring 和任何 form 变量都会同时传递到你定位的页面。如果要将执行流程转入同一 Web 服务器的另一个 ASPX 页面，应当使用 Server.Transfer 而不是 Response.Redirect，因为 Server.Transfer 能够避免不必要的网络通信，从而获得更好的性能和浏览效果。页面 A 跳转到页面 B，同时页面处理的控制权也进行移交，在跳转过程中 Request,Session 等保存的信息不变，浏览器的 URL 仍保存 A 的 URL 信息。Server.Transfer 的重定向请求在服务器端进行，客户端不知晓服务器执行了页面转换，因此 URL 保持不变，Server.Transfer 属于服务器端跳转。

Server.Transfer 页面跳转的效率比 Response.Redirect 高；且由于在服务器上执行，可以兼容任何浏览器，但是只能在 IIS 服务器下运行。在 MVC 下的写法为：

```

1 //使用 TransferRequest 效率更高, 不需 304 回包, 浏览器链接不变
2 HttpContext.Server.TransferRequest("~/ Houses/IndexNew");

```

跳转后浏览器的 Url 不变，跳转后的页面可以使用跳转前的页面的参数。

5.8.4 URL 锚点 (Fragment URLs)

页面之所以能定位到锚点所在位置，都是因为 URL 地址中的锚链的作用，而不是点击行为。最好的证据就是，当重新载入带有锚链的页面时，锚点依然会被定位。# 代表网页中的一个位置。其右面的字符，就是该位置的标识符。在第一个 # 后面出现的任何字符，都会被浏览器解读为位置标识符。这意味着，这些字符都不会被发送到服务器端¹⁴。

5.9 技巧 (Little Tricks)

5.9.1 查看 Cookie 登录密码

在 Firefox 中可以查看所有网站 Cookie 保存的密码，在 Firefox 选项-> 安全里面，如图5.19所示。

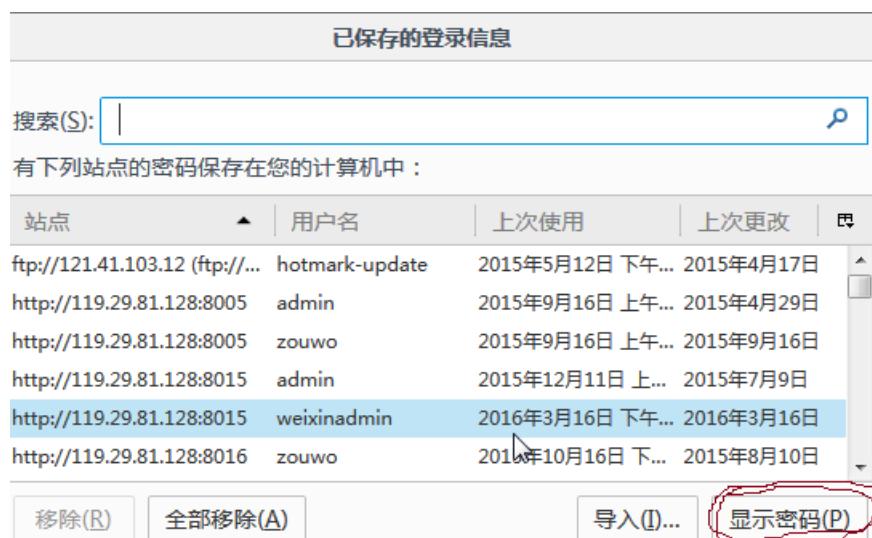


图 5.19: 在 Firefox 中查看网站的登录密码

¹⁴http://www.ruanyifeng.com/blog/2011/03/url_hash.html

5.9.2 刷新当前页面 (Refresh Current Page)

微信开发过程中，由于微信浏览器 Android 下没有刷新按钮，每次修改数据需要重新进入修改页面比较麻烦，可在每个页面中添加一个刷新按钮，直接点击按钮刷新，部署后将按钮注释即可，在 MVC 的布局页中添加刷新标签，如代码5.7所示。

Listing 5.7: 添加刷新按钮便于调试

```

1 <body style="background-color:#f3f3f3;">
2   <div class="s_part1"><a href="javascript:window.location.reload()">Refresh</
3     a></div>
4   @RenderBody()
5 </body>

```

也可以判断当前程序集是发布状态还是调试状态，根据程序集状态来判断是否需要添加用于调试的刷新按钮。

Listing 5.8: 获取程序集是否为调试状态

```

1 #region Is Debug Model
2 /// <summary>
3 ///
4 /// </summary>
5 /// <param name="dllPath">Assembly path</param>
6 /// <returns></returns>
7 public static bool IsDebugged(string dllPath)
8 {
9   var assembly = Assembly.LoadFile(Path.GetFullPath(dllPath));
10  return assembly.GetCustomAttributes(false).OfType<DebuggableAttribute>().Any(
11    debuggableAttribute => debuggableAttribute.IsJITTrackingEnabled);
12 }
13 #endregion

```

dllPath 为程序集的路径，获取程序集的路径如下代码所示。

Listing 5.9: 获取程序集路径

```

1 var applicationPath = Assembly.GetExecutingAssembly().Location;
2 var currentApplicationPath = GetType().Assembly.Location;

```

判断程序集是调试 (Debug) 选项编译还是发布 (Release) 选项编译也可以观察反编译程序集的 Debuggable 值，用 ILSpy 分别打开不同编译选项的程序集，可以看到调试编译和发布编译 DebuggingModes 值不同，如下片段所示。

Listing 5.10: 查看程序集调试模式

```

1 #打开 Release 模式编译的程序集
2 [assembly: Debuggable(DebuggableAttribute.DebuggingModes.
3   IgnoreSymbolStoreSequencePoints)]
4 #打开 Debug 模式编译的程序集
5 [assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default | DebuggableAttribute.

```

```
DebuggingModes.DisableOptimizations | DebuggableAttribute.DebuggingModes.  
IgnoreSymbolStoreSequencePoints | DebuggableAttribute.DebuggingModes.  
EnableEditAndContinue)]
```

序列点是用于指示调试器用户将希望能够来指代唯一的 Microsoft 中间语言 (MSIL) 代码中的位置，例如用于设置断点。JIT 编译器可确保它不会将在两个不同的序列点 MSIL 编译成单个的本机指令。默认情况下，JIT 编译器将检查符号存储区的其他序列点列表的程序数据库 (PDB) 文件中。但是，加载 PDB 文件要求该文件可用并且具有负面影响。从 2.0 版开始，编译器可以发出“隐式序列点”在 MSIL 代码流通过使用 MSIL “nop” 说明进行操作。此类编译器应设置 IgnoreSymbolStoreSequencePoints 标志来通知公共语言运行时不会加载 PDB 文件。

VPN 网络访问

在使用重庆有线 VPN 服务器时，由于远程 VPN 服务器限制只能与特定的用于开发的服务器进行通信，导致连接 VPN 的同时本机无法访问互联网。在 VPN 的属性设置中勾选掉“在远程网路上使用默认网关”选项即可解决此问题，如图5.20所示。

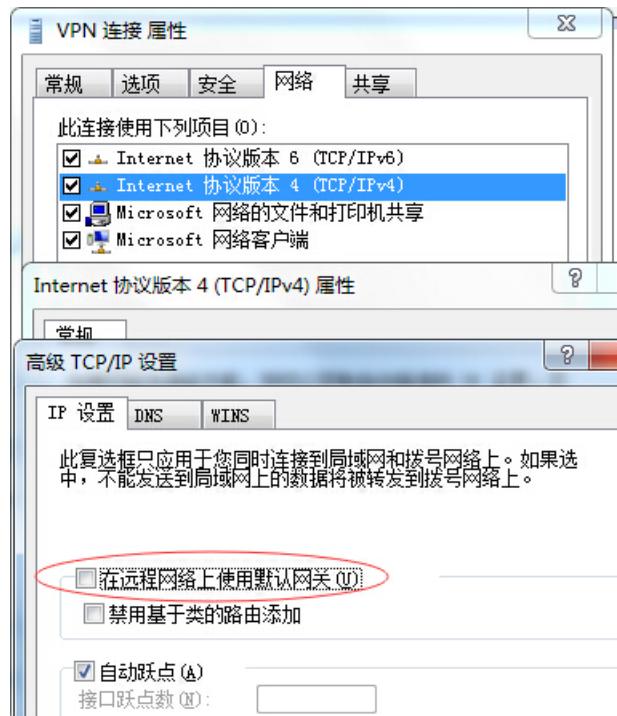


图 5.20: VPN 默认使用本机网关

勾选的优劣 好处：无需添加静态路由即可访问 VPN 服务后面的局域网资源。不足：VPN 拨上号后，本地无法上网，需要在 VPN 服务器上开通“到 Internet 的 VPN 路由通道”才行。此外，即使开通了 VPN 路由通道，通过 VPN 上网速度可能会比较慢（没有本地线路快）。评点：需要访问远程局域网时，进行 VPN 拨号，不需要时断开，但不适合“本地上网”和“访问远程局域网”同时进行的场合。

不勾选的优劣 好处: 不改变本地默认网关, VPN 拨号后本地上网不受影响, 仍然走本地线路。不足: 访问远程局域网资源时, 需要手动添加静态路由。点评: 手动添加静态路由比较麻烦, 可以通过批处理文件来辅助完成。综合以上分析, 得出以下结论: 如果 VPN 客户端不需要访问远程局域网, 只需要拨入客户端之间可以相互访问, 则不用勾选“默认网关”; 如果需要, 但是又不想手动添加静态路由, 则可以采用第 2 种方案。

将 VPN 连接快捷方式拖动到启动项中, 并将 VPN 连接属性-选项-“提示名称密码、和证书等”选项取消勾选, 可在开机时自动连接 VPN, 避免使用 VPN 时却发现 VPN 被占用导致无法连接 VPN 的尴尬, 宁愿占着茅坑不用, 也不要内急时找不到茅坑, Windows 7 的启动文件夹为:

```
1 C:\Users\Administrator\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\
    Startup.
```

5.9.3 URL 特殊参数传递

要把一个字符串作为 URL 参数传递给另一个页面, 必须要保证传递的字符串是正确的, 能够被目标页面正确识别。但由于 URL 传递时会造成某些冲突。比如, 你要传递的字符串也是一个带有参数的 URL, 如需要传递的参数是如下形式的字符串:

```
1 http://www.oyksoft.com/a.asp?b=1&c=2&d=3
```

要把这个 URL 字符串传递给 `http://www.oyksoft.com/e.asp`, 参数为 url。如果不作任何处理, 那么就是这样:

```
1 http://www.oyksoft.com/e.asp?url=http://www.oyksoft.com/a.asp?b=1&c=2&d=3
```

这样是很混乱的, 它肯定会把 c 和 d 认为是传参, 而解析出的 url 参数为

```
1 http://www.oyksoft.com/a.asp?b=1
```

这样理解当然是错误的了, 所以我们必须对原字符串进行重新编码。在 ASP 和 Javascript 里面, 有 escape 和 unescape 可以解决问题, 现在已不推荐使用。注意 escape 方法不能够用来对统一资源标示码 (URI) 进行编码。对其编码应使用 encodeURI 和 encodeURIComponent 方法。例如调用另一个网页完成支付功能, 用户支付完毕后再回到指定的网页, 那么就需要将指定的网页编码后以参数的形式传递给支付页, 支付页再进行回调, 编码并跳转到指定页如下代码片段所示。

```
1 var callbackUrl = "http://www.oyksoft.com/a.asp?b=1&c=2&d=3";
2 var encodeCallbackUrl = encodeURIComponent(callbackUrl);
3 var payUrl = "http://www.oyksoft.com/a.asp?url=" + encodeCallbackUrl;
4 window.location.href = payUrl;
```

在服务端以下形式进行解码:

```
1 var decodeUrl = Server.UrlDecode("http%3A%2F%2Fwww.oyksoft.com%2Fa.asp%3Fb%3D1%26c%3D2%26d%3D3");
2 var decodeUrl1 = HttpUtility.UrlDecode("http://www.oyksoft.com/a.asp?b%3D1%26c%3D2%26d%3D3");
```

encodeURI 和 encodeURIComponent 都是 ECMA-262 标准中定义的函数，所有兼容这个标准的语言（如 JavaScript, ActionScript）都会实现这两个函数。它们都是用来对 URI (RFC-2396) 字符串进行编码的全局函数，但是它们的处理方式和使用场景有所不同。为了解释它们的不同，我们首先需要理解 RFC-2396 中对于 URI 中的字符分类。

保留字符 (reserved characters) 这类字符是 URI 中的保留关键字符，它们用于分割 URI 中的各个部分。这些字符是：“;” | “/” | “?” | “:” | “@” | “&” | “=” | “+” | “\$” | “,”

Mark 字符 (mark characters) 这类字符在 RFC-2396 中特别定义，但是没有特别说明用途，可能是和别的 RFC 标准相关。这些字符是：“-” | “_” | “.” | “!” | “” | “*” | “”” | “(” | “)”

基本字符 (alphanum characters) 这类字符是 URI 中的主体部分，它包括所有的大写字母、小写字母和数字

在介绍完上面三类字符串后，我们就非常容易来解释 encodeURI 和 encodeURIComponent 函数的不同之处了：

encodeURI: 该函数对传入字符串中的所有非（基本字符、Mark 字符和保留字符）进行转义编码 (escaping)。所有的需要转义的字符都按照 UTF-8 编码转化成为一个、两个或者三个字节的十六进制转义字符 (%xx)。例如，字符空格“ ”转换成为“%20”。在这种编码模式下面，需要编码的 ASCII 字符用一个字节转义字符代替，在\0080 和\007ff 之间的字符用两个字节转义字符代替，其他 16 为 Unicode 字符用三个字节转义字符代替。

encodeURIComponent: 一般对 URL 参数进行编码，该函数处理方式和 encodeURI 只有一个不同点，那就是对于保留字符同样做转义编码。例如，字符“:”被转义字符“%3A”代替之所以有上面两个不同的函数，是因为我们在写 JS 代码的时候对 URI 进行两种不同的编码处理需求。encodeURI 可以用来对完整的 URI 字符串进行编码处理。而 encodeURIComponent 可以对 URI 中一个部分进行编码，从而让这一部分可以包含一些 URI 保留字符。这在我们日常编程中是十分有用的。比如下面的 URI 字符串：

```
1 http://www.mysite.com/send-to-friend.aspx?url=http://www.mysite.com/product.html
```

在这个 URI 字符串中。send-to-friend.aspx 页面会创建 HTML 格式的邮件内容，里面会包含一个链接，这个链接的地址就是上面 URI 字符串中的 url 值。显然上面的 url 值是 URI 中的一个部分，里面包含了 URI 保留关键字符。我们必须调用 encodeURIComponent 对它进行编码后使用，否则上面的 URI 字符串会被浏览器认为是一个无效的 URI。正确的 URI 应该如下：

```
1 http://www.mysite.com/send-to-friend.aspx?url=http%3A%2F%2Fwww.mysite.com%2Fproduct.html
```

有一个小细节需要注意，因为 HTML 有许多保留的转义字符（Escape Sequence），也叫字符实体（Character Entity）。所以在后台向前台传值时如果字符中带有转义字符的话，会显示字符的实体名称（Entity Name），例如后台向前台传递“&”符号时，前台 HTML 页面显示为“&”（包括 Javascript 的输出），如果需要显示成“&”符号，在 Razor 视图中使用 @Html.Raw(str) 的写法，str 是带有转义字符（Escape Sequence）的字符流。

5.9.4 前端技巧

iPad 上的 Safari 总会把长串数字识别为电话号码，文字变成蓝色，点击还会弹出菜单添加到通讯录。在做问卷调查的时，房价金额会自动识别为电话号码。Safari 有个私有 meta 属性可以解决这个问题：

```
1 <meta name="format-detection" content="telephone=no" />
```

5.9.5 监视 log

一个进程在运行，并在不断的写 log，你需要实时监控 log 文件的更新（一般是 debug 时用），怎么办，不断的打开，关闭文件吗？不用，至少有两个方法，来自两个很常用的命令：

tail -f log.txt，另外一个进程在写 log，而你用 tail，就可以实时的打印出新的内容

less log.txt，然后如果要监控更新，按 F，如果要暂停监控，可以 CTRL+C，这样就可以上下翻页查看，要继续监控了再按 F 即可。这个功能要比 tail 更强。

在 Windows 中使用 tail 和 less 命令需要安装 Cygwin，将安装路径 C:/Cygwin/bin 添加到 Windows 环境变量中。

5.9.6 网站统计

Piwik 是一套基于 Php+MySQL 技术构建，能够与 Google Analytics 相媲美的开源网站访问统计系统。Piwik 可以给你详细的统计信息，比如网页浏览人数，访问最多的页面，搜索引擎关键词等等，并且采用了大量的 AJAX/Flash 技术，使得在操作上更加便易。

5.9.7 页面统计 (Pageview Tracking)

5.9.8 断点续传 (Resume Broken Transfer)

206 状态码代表服务器已经成功处理了部分 GET 请求（只有发送 GET 方法的 request，web 服务器才可能返回 206），应用场景：

1. FlashGet, 迅雷或者 HTTP 下载工具都是使用 206 状态码来实现断点续传

2. 将以个大文档分解为多个下载段同时下载比如, 在线看视频

实例: 一些流媒体技术比如在线视频, 可以边看边下载。就是使用 206 来实现的。

5.9.9 ASP.NET 应用程序与页面生命周期

ASP.NET 应用程序的生命周期以浏览器向 Web 服务器 (对于 ASP.NET 应用程序, 通常为 IIS) 发送请求为起点, 直至将请求结果返回至浏览器结束。在这个过程中, 首先我们需要了解 ASP.NET 请求的 2 个大致的步骤。其次我们将详细了解'httphandler ', 'httpmodule 和 asp.net 页面对象 (Page) 中不同的事件的执行顺序, 逻辑。二个步骤的过程:

asp.net 请求处理, 2 步的过程如图5.21所示, 用户发送一个请求到 IIS 服务器:

1、asp.net 创建一个运行时, 可以处理请求。换句话说, 它创建应用程序对象, 请求, 响应和上下文对象处理请求。

2、运行时一旦被创建, 请求处理, 通过一系列的事件处理模块, Handler 处理和页面对象。简称 MHPM (Module, handler, page and Module event)。

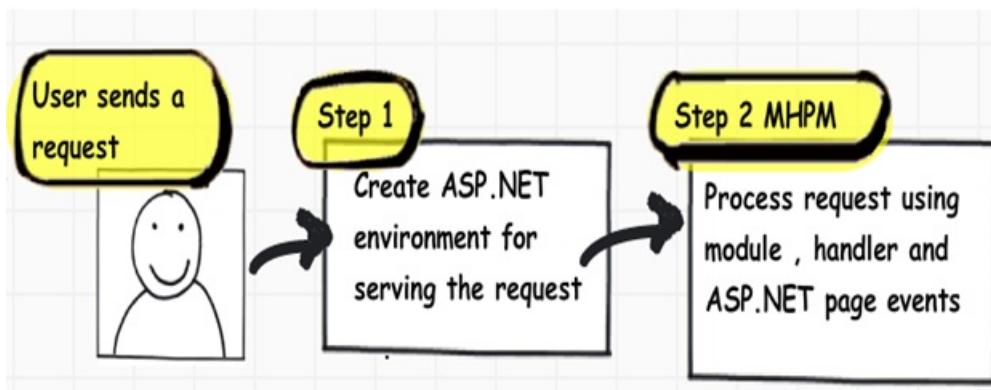


图 5.21: IIS 处理请求过程

<http://www.cnblogs.com/suizhouqiwei/archive/2012/08/15/2637775.html>

5.10 页面适应多屏

5.10.1 适应手机浏览

使用 meta 标签 这也是普遍使用的方法, 理论上讲使用这个标签是可以适应所有尺寸的屏幕的, 但是各设备对该标签的解释方式及支持程度不同造成了不能兼容所有浏览器或系统。

```
1 <meta name="viewport" content="width=device-width,initial-scale=1.0, minimum-
scale=1.0, maximum-scale=1.0, user-scalable=no"/>
```

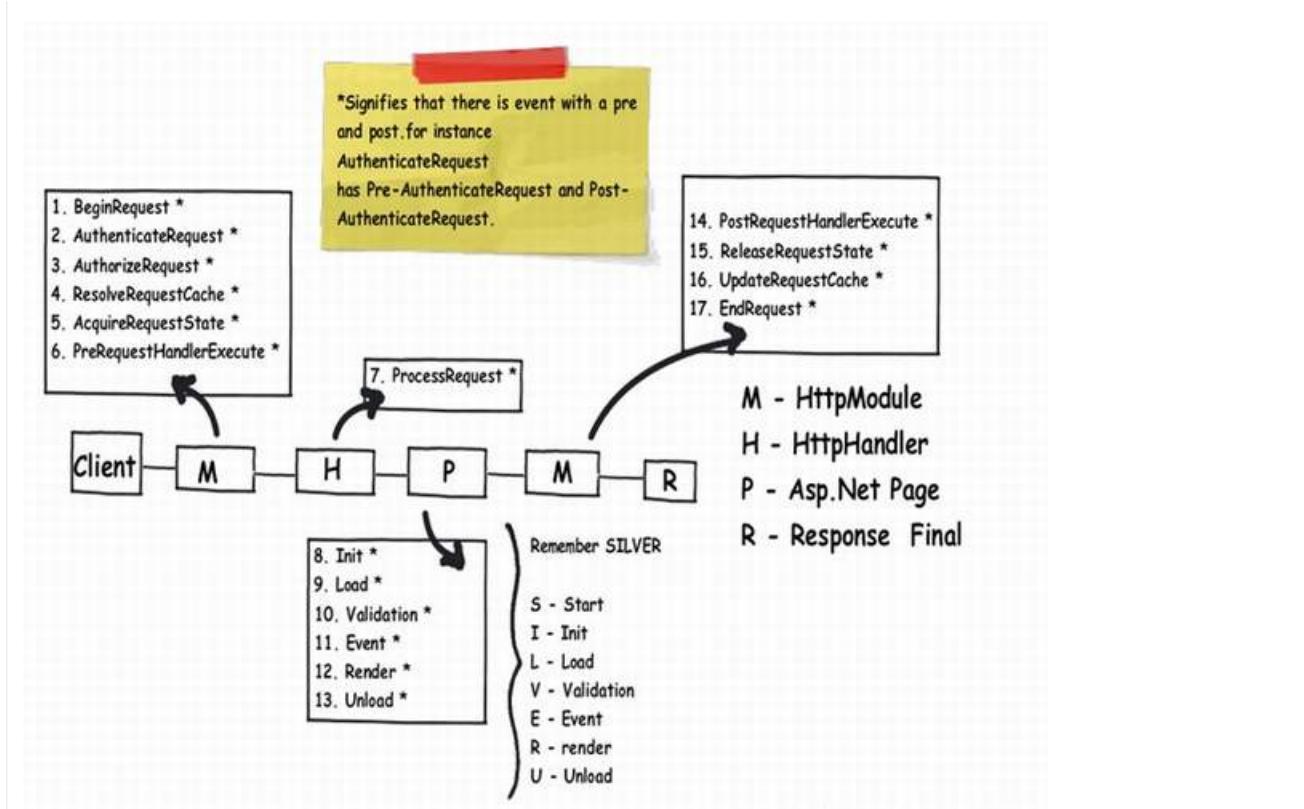


图 5.22: ASP.NET 生命周期

5.11 性能 (Performance)

5.11.1 网页优化 (Website Optimized)

css 文件放在 `<head>` 里面，js 文件尽量放在页面的底部 因为请求 js 文件是很花费时间，如果放在 `<head>` 里面，就会导致页面的 DOM 树呈现需要等待 js 文件加载完成。这也就是为什么很多网站的源码里面看到引用的文件放在最后的原因。

CDN 静态内容（比如图片、CSS、JavaScript、以及其他 cookie 无关的网页内容）都应该放在一个不需要使用 Cookie 的独立域名之上。因为域名之下如果有 Cookie，那么客户端向该域名发出的每次 Http 请求，都会附上 Cookie 内容。这里的一个好方法就是使用“内容分发网络”（Content Delivery Network，CDN）

favicon.ico 确保网站根目录下有 favicon.ico 文件，因为即使网页中根本不包括这个文件，浏览器也会自动发出对它的请求。所以如果这个文件不存在，就会产生大量的 404 错误，消耗光你的服务器的带宽。

不要在 HTML 中缩放图片

把 CSS 放到顶部

5.11.2 利用反向代理服务器加速和保护应用

如果 Web 应用运行在一台独立的电脑上，性能问题的解决方案是显而易见的：换一台更快的电脑，里面加上更多的处理器、内存、快速磁盘阵列等等。然后在这台新电脑上运行 WordPress 服务、Node.js 应用、Java 应用等等，会比以前快很多。（如果应用需要访问服务器，方案还是很简单：换两台更快的电脑，用更快速的连接把它们连接起来。）

但电脑速度可能不是问题所在。通常 Web 应用运行缓慢，是由于电脑一直在不同的任务间切换：同成千上万的客户交互、访问磁盘上的文件、执行应用代码和其它的任务。应用服务器可能会因为下面这些问题而崩溃——内存耗尽、把很多的数据从内存交换到磁盘上、以及很多请求都在等待一个类似磁盘 I/O 的单个任务。

你应该采用一种完全不同的方式，而不是升级硬件：增加一个反向代理服务器来分担这些任务。这台反向代理服务器设置在运行应用的电脑之前，用来处理网络流量。只有这台反向代理服务器直接连到网络上，它和应用服务器通过一个快速的内部网络进行通信。

利用这台反向代理服务器，应用服务器就不用等着和 Web 应用的用户进行交互，它可以专注在建立网页，并通过反向代理服务器把它们发送到网络上。因为应用服务器不必再等待客户的响应，所以能以最优的速度运行。

增加一台反向代理服务器也增加了 Web 服务器的弹性。如果一台服务器过载了，很容易增加另一台同类型的服务器。如果一台服务器宕机，也很容易把它换掉。

因为反向代理服务器带来的灵活性，它也成为了许多其它性能提升方法的先决条件，比如：

负载均衡——反向代理服务器上运行一个负载均衡器，把流量平均分配给一堆应用服务器。由于负载均衡器的引入，在增加应用服务器时可以完全不用修改应用程序。

缓存静态文件——直接请求的文件，比如图片或者代码文件，可以存在反向代理服务器上，并直接发给客户端，这样可以更快地提供服务，分担了应用服务器的负载，可以让应用执行得更快。

保护网站——反向代理服务器可以设置较高的安全级别，通过监控进快速识别和响应攻击，这样就可以把应用服务器保护起来。

NGINX 软件是专门设计用做反向代理服务器的，具有上述这些附加功能。NGINX 利用事件驱动处理的方法，比其它传统的服务器更加高效。NGINX Plus 增加了更多反向代理的高级功能和支持，包含应用程序健康检查、特定请求路由和高级缓存等

<http://blog.jobbole.com/94962/>

5.11.3 增加一个负载均衡器

增加一个负载均衡器是一个相对简单的改动，而且会大幅度地改善网站的性能和安全性。你可以利用负载均衡器把业务分配给一些服务器，而不是建造一台更大更强的 Web 核心服务器。

就算应用程序编写得很烂或者扩展性很差，负载均衡器都能提升用户体验而不需要任何其它的改动。

负载均衡器首先是一个反向代理服务器——它接收网络流量，并把请求转交给另一个服务器。一个窍门就是让负载均衡器支持两台以上的应用服务器，利用一个选择算法在服务器间分配请求。最简单的方法就是轮询，每个新请求发送给列表中的下一台服务器。其它方法包括把请求发送给活动连接数量最少的服务器。NGINX Plus 可以在同一台服务器上维持一个给定的用户会话，这个功能被称为会话持久性。

负载均衡器可以极大地改善性能，因为它们避免让一台服务器过载，而其它服务器却处于空闲的状态。它们也很容易扩展 Web 服务器的能力，增加相对便宜的服务器并确保它们物尽其用。

负载均衡可以运用在很多协议上，包含 HTTP、HTTPS、SPDY、HTTP/2、WebSocket、FastCGI、SCGI、uwsgi、memcache，还有一些应用程序，包含基于 TCP 的应用和 L4 协议。分析 Web 应用使用了什么技术和性能落后在什么地方。

同一台服务器或者用于负载均衡的服务器，还能处理其他任务，包含 SSL 终端、支持客户端使用的 HTTP/1/x 和 HTTP/2、以及缓存静态文件。

NGINX 通常被用于负载均衡：想了解更多，请参考这些资料，一篇介绍性的文章、一篇关于配置的文章、一本电子书和相关的网络课程和相关文档。我们的商业版本（NGINX Plus），支持更多负载均衡的特殊功能，比如基于服务器响应时间的路由规划，和基于微软 NTLM 协议的负载均衡。（译者注：NTLM 是 NT LAN Manager 的缩写，NTLM 是 Windows NT 早期版本的标准安全协议。）

5.11.4 缓存静态和动态内容

5.12 Electron

5.12.1 安装

全局安装 electron

```
1 # Install the 'electron' command globally in your $PATH
2 npm install electron-prebuilt -g
```

在安装时提示如下错误，Downloading electron-v1.2.1-win32-x64.zip Error: connect ETIMED-OUT 54.231.18.161:443，直接 ping IP：54.231.18.161 失败，到环境变量中将下载地址改成淘宝的镜像 ID 即可：

```
1 变量名: ELECTRON_MIRROR
2 变量值: http://npm.taobao.org/mirrors/electron/
```

安装打包工具 electron-packager

```
1 npm install -g electron-packager
```

查看 npm 已经安装的包：

```
1 npm ls
2 npm ls -g
```

NodeJS 的安装目录在 D:/Program Files/nodejs.

5.12.2 第一个程序

Error: Cannot find module 'app' 确全局模块的默认安装位置: npm root -g 接着查看全局模块的默认搜索路径:

```
1 node
2 global.module.paths
```

5.12.3 打包

5.13 常见问题

IIS 6 中 log4net 不写日志

IIS 6 默认运行在 Network Service 账户下，尝试添加 Network Service 账户对日志文件夹的读写权限。

HTTP 错误 500.23 - Internal Server Error 检测到在集成的托管管道模式下不适用的 ASP.NET 设置。

取消集成模式验证配置：

```
1 <system.webServer>
2   <validation validateIntegratedModeConfiguration="false" />
3 </system.webServer>
```

HTTP 403 - 您无权查看该网页

每次浏览网页的时候只显示目录，或者提示你无权查看该网页。在配置里添加了 asp-net_isapi.dll 后，问题解决，如图5.23所示。

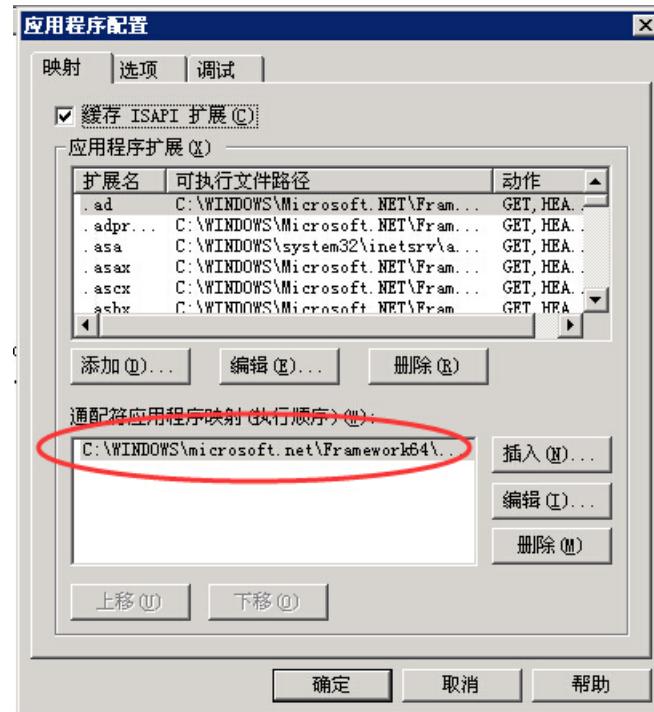


图 5.23: HTTP 403 您无权查看该网页

HTTP 错误 404.2 - Not Found 由于 Web 服务器上的“ISAPI 和 CGI 限制”列表设置，无法提供您请求的页面。

如图5.24所示：

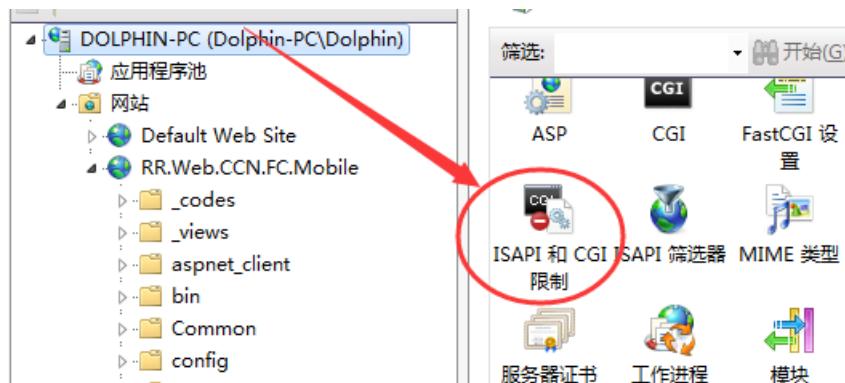


图 5.24: HTTP 错误 404.2

打开 IIS7.5，左侧选择根节点，在功能视图中找到“ISAPI 和 CGI 限制”并打开，将网站应用程序池对应.NET Framework 版本设置为允许即可。

aAjax 提交 500 错误

在 aAjax 提交后返回 500 错误，后台调试未发现问题，可将请求 URL 贴入浏览器地址查看具体信息，打开 Firebug，切换到网络选项卡，查看请求失败链接的请求头信息和响应头信息，

如图5.25所示。



图 5.25: aAjax 提交 500 错误

将画红框部分贴入浏览器中访问即可发现更加详细的后台错误信息。

5.13.1 Bad Request (Invalid Hostname)

总结页面出现 Bad Request (Invalid Hostname) 的原因：1. 如果确定域名已经解析生效，但是仍然不能访问，出现 Bad Request (Invalid Hostname). 那么这就可能是您没有绑定该域名的原因 2. 确定端口使用的是 80 端口

Chapter 6

Windows Form

6.1 资源文件 (Resource)

6.1.1 简介

资源文件顾名思义就是存放资源的文件。资源文件在程序设计中有着自身独特的优势，他独立于源程序，这样资源文件就可以被多个程序使用。同时在程序设计的时候，有时出于安全或者其它方面因素的考虑，把重要东西存放在资源文件中，也可以达到保密、安全的效果。资源文件中一般存三种类型的数据：byte 流 (byte[])、对象 (object) 和字符串 (string)。对于一些纯文件的信息可以用 string 类型来保存，对于图片 (Image)、图标 (Icon) 等用 object 来保存，其它的可以用 byte 流来保存。System.Resources 命名空间中有大量的类和方法来处理资源文件。

6.1.2 资源文件的分类

资源文件可以分为两类，一类是以.resx 为后缀名的文件，一类是以.resources 为后缀名的文件。二者的区别在于：

1.resx 虽然是以 resx 结尾的文件，但是它却是 XML 格式的文件，你可以用记事本等工具直接打开它修改里面的东西；而 resources 是二进制的文件，相对来说安全性更好一些。

2.resources 作为内嵌资源，在指定路径正确的前提下，可以在程序中直接引用；而 resx 虽然也是内嵌资源，但它却是要依附.cs 文件存在的。也就是说它是作为 winform 窗体的一个描述性资源存在的，要想在程序中直接使用它，在解决方案中必须有与它同名（只是名字相同，后缀名不同）的.cs 文件存在。

3. 可以利用 CSC 命令把 resx 文件转换成 resources 文件。RESGEN.EXE Litware-Strings.resx LitwareStrings.resources. 注意变量环境为 framework1.1。

6.1.3 利用资源文件来做多国语言版本

如何使用 VS 的 IDE 来制作多国语言版本。每一个 Form1.cs 文件都有一或多个相应的 resx 文件作为附属资源。他们的命名规则为 Form1.cs 的资源文件为 Form1.resx, Form1.zh-CHS.resx, Form1.zh-CHT.resx 等, 其中 Form1.resx 是缺省的窗体资源文档, 其它是在不同语言环境要使用的资源文档, 其中 Form1.zh-CHS.resx 是中文简体系统, Form1.zh-CHT.resx 是中文繁体系统。关于命名可不是随便起的, 可以参见 msdn 中关于不同地区的命名规则。

6.1.4 MouseDown 事件后无法触发 MouseDoubleClick

.NET 控件的单双击事件的触发过程, 就是先触发单击过程, 再触发双击过程这个是.NET 已有控件的固定机制, 这个无法改变。事件引发的顺序: MouseDown 事件。Click 事件。MouseClick 事件。MouseUp 事件。MouseDown 事件。DoubleClick 事件。MouseDoubleClick 事件。MouseUp 事件。

6.1.5 指定 DllImport 文件目录

引用的 dll 都放在根目录下, 随着项目的日益增大, 根目录下充满了各种各样的 dll, 非常的不美观。如果能够把 dll 按照想要的目录来存放, 那么系统就美观多了, 有时需要引用 Win32 的 Native dll, 无法通过定义私有目录的方式定义 dll, 此时可以通过指定程序的环境变量的方式定义 dll 的搜索目录, 添加环境变量的代码如下片段所示。

```

1 public static void AddEnvironmentPaths(IEnumerable<string> paths)
2 {
3     var path = new[] { Environment.GetEnvironmentVariable("PATH") ?? string.Empty };
4     var newPath = string.Join(Path.PathSeparator.ToString(), path.Concat(paths));
5     Environment.SetEnvironmentVariable("PATH", newPath);
6 }
```

这样添加环境变量对系统的环境变量没有影响。

6.1.6 窗口句柄

在 Windows 中, 句柄是一个系统内部数据结构的引用。例如当你操作一个窗口, 或说是一个 Delphi 窗体时, 系统会给你一个该窗口的句柄, 系统会通知你: 你正在操作 142 号窗口, 就此你的应用程序就能要求系统对 142 号窗口进行操作——移动窗口、改变窗口大小、把窗口最小化等等。实际上许多 Windows API 函数把句柄作为它的第一个参数, 如 GDI (图形设备接口) 句柄、菜单句柄、实例句柄、位图句柄等, 不仅仅局限于窗口函数。换句话说, 句柄是一种内部代码, 通过它能引用受系统控制的特殊元素, 如窗口、位图、图标、内存块、光标、字体、菜单等。

6.1.7 Windows 弹出窗体

将最小化窗体弹出并置于最前端。

```
1 frmBatchSendMessage.Show();
2 if (frmBatchSendMessage.WindowState == FormWindowState.Minimized)
3 {
4     frmBatchSendMessage.WindowState = FormWindowState.Normal;
5 }
6 frmBatchSendMessage.Activate();
```

MessageBox MessageBox 的关闭按钮在 MessageBoxButtons 为 YesNo 枚举类型时不可用，可以采用 MessageBoxButtons.OKCancel，如果想让 MessageBox 始终置于窗体的最前端，MessageBoxOptions 使用 DefaultDesktopOnly 属性。如代码所示：

```
1 var result = MessageBox.Show(@"系统检测到新版本，是否升级", @"更新提示",
2                             MessageBoxButtons.OKCancel, MessageBoxIcon.Information,
3                             MessageBoxDefaultButton.Button1, MessageBoxOptions.DefaultDesktopOnly); // MB_TOPMOST
```

6.2 常见问题

6.2.1 关于嵌入互操作类型 (Embed Interop Types)

在项目开发中引用了一个 dll，编译器显示警告如下：

由于存在对由程序集“AxInterop.videocx3Lib.dll”创建的程序集的间接引用，因此创建了对嵌入的互操作程序集“Interop.videocx3Lib.dll”的引用。请考虑更改其中一个程序集的“嵌入互操作类型”属性。

当按照提示更改 AxInterop.videocx3Lib.dll 的嵌入互操作类型后出现错误：

无法嵌入来自程序集“AxInterop.videocx3Lib.dll”的互操作类型，因为它缺少“ImportedFromTypeLibAttribute”特性或“PrimaryInteropAssemblyAttribute”特性

排查发现此程序集和另一个 ActiveX 组件存在冲突，更改 ActiveX 组件的嵌入互操作类型后问题得以解决。使用 COM 互操作程序集，而不要求该程序集在运行时必须存在。目的是减轻将 COM 互操作程序集与您的应用程序一起部署的负担。”嵌入互操作类型”中的嵌入就是引进、导入的意思，类似于 c# 中 using，c 中 include，Java 的 Import 的作用，目的是告诉编译器是否要把互操作类型引入。”互操作类型”实际是指一系列 Com 组件的程序集，是公共运行库中库文件，类似于编译好的类，接口等。”嵌入互操作类型”设定为 true，实际上就是不引入互操作集（编译时候放弃 Com 程序集），仅编译用户代码的程序集。而设定为 false 的话，实际就是需要从互操作程序集中获取 COM 类型的类型信息。当 COM 互操作在最初版本的.NET

Framework 中引入时，就确立了主互操作程序集 (PIA) 的概念。引入此概念，是为了解决在组件之间共享 COM 对象的难题。例如：如果您有一些不同的互操作程序集，分别定义了一个 Excel Worksheet，则我们无法在组件之间共享这些 Worksheet，因为它们具有不同的.NET 类型。PIA 通过只存在一次而解决了这个难题：所有客户端都使用它，因此.NET 类型始终是匹配的。尽管 PIA 在理论上是个好主意，但在实际部署中却被证明是个大麻烦，因为它只有一份，而有多个应用程序可能会尝试安装或卸载它。而由于 PIA 通常很大，事情更复杂了。Office 在默认 Office 安装方式中并未部署它们，用户只需通过使用 TLBIMP 来创建自己的互操作程序集，即可轻松绕过这一个程序集系统。因此，现在为了扭转这种局面，发生了两件事：对于两个结构相同且共享相同识别特征（名称、GUID 等）的 COM 互操作类型，运行时能够聪明地将其看作同一个.NET 类型。C# 编译器利用这一点的方式是在编译时直接在您自己的程序集中重现互操作类型，因此不再要求在运行时存在该互操作程序集。通过将引用上的“嵌入式互操作类型”属性设置为 true，告诉编译器为您将互操作类型嵌入到 Visual Studio 中。由于 C# 团队希望这种方法成为引用 COM 程序集的首选方法，因此在默认情况下，Visual Studio 会将添加到 C# 项目中的任何新互操作引用的此属性设置为 True(这里不准确，经过测试添加的 dll 嵌入互操作类型的值为 False)。Beginning with the .NET Framework 4, the common language runtime supports embedding type information for COM types directly into managed assemblies, instead of requiring the managed assemblies to obtain type information for COM types from interop assemblies. Because the embedded type information includes only the types and members that are actually used by a managed assembly, two managed assemblies might have very different views of the same COM type. Each managed assembly has a different Type object to represent its view of the COM type. The common language runtime supports type equivalence between these different views for interfaces, structures, enumerations, and delegates.

6.2.2 Class not registered

Class not registered (Exception from HRESULT: 0x80040154 (REGDB_E_CLASSNOTREG))

6.2.3 试图加载格式不正确的程序

在 Manage Code 中加载 Native dll 程序时出错，错误如下：

1 “System.BadImageFormatException” 类型的第一次机会异常在 MVSP.Common.dll 中发生
2 其他信息: 试图加载格式不正确的程序。 (异常来自 HRESULT:0x8007000B)

程序在 64 位的机器上就会用运行为 64 位，而 64 程序是不能加载 32 位 dll 的。dll 文件是在 64 位开发环境下编译的，而你现在的调用程序是的 32 位，所以无法调用。

6.2.4 Dictionary 索引超出数组界限

在向 Dictionary 中添加项时提示此错误，原因是程序是多线程，Dictionary 不是线程安全的，解决方案是添加 lock 语句。

```
1 lock (DataManager.VehicleLatestAlarmInfoList)
2 {
3     if (!DataManager.VehicleLatestAlarmInfoList.ContainsKey(vehicleInfo.VehicleID))
4     {
5         DataManager.VehicleLatestAlarmInfoList.Add(vehicleInfo.VehicleID, DateTime.Now);
6     }
7     else
8     {
9         DataManager.VehicleLatestAlarmInfoList[vehicleInfo.VehicleID] = DateTime.Now;
10    }
11 }
```

```
1 import string
```

6.2.5 System.ComponentModel.Design.ExceptionCollection

在 Visual Studio 打开设计器时提示此错误，The real problem of the "ExceptionCollection" being thrown is that when there is a WSOD (White Screen of Darn) indicating a designer load issue, the designer gets unloaded. In your project, the RussTabStrip is throwing exceptions. These get caught by our unload method and get displayed in the dialog box you see. So, to fix this you should:

- 1) Start a second instance of visual studio
- 2) go to the Tools menu, "Attach to process", select the 'devenv.exe' process, and click the 'attach' button.
- 3) In the Debug/Exceptions menu
- 4) Open the designer with the debugger attached
- 5) The second visual studio will break on your error.

Chapter 7

ASP.NET MVC(Apache License 2.0)

MVC 模型以低耦合、可重用、可维护性高等众多优点已逐渐代替了 WebForm 模型。

7.1 MVC 权限

7.1.1 MVC 登录验证

有时有的网页需要用户登录才能够操作，所以需要进行用户是否登录的验证。基本思路是用户登录后 Cookie 里存放一个值，当再次请求时判断此值是否存在，存在就代表已经登录。编写过滤器 ActionExecuteFilter 继承 ActionFilterAttribute 在每一个 Action 执行之前判断用户是否已经登录：

```
1 public class ActionExecuteFilter : ActionFilterAttribute
2 {
3     public override void OnActionExecuting(ActionExecutingContext filterContext)
4     {
5         //OnActionExecuting: 正要准备执行 Action 的时候但还未执行时执行
6         var userId = filterContext.HttpContext.Request.Cookies["userId"];
7         if (userId != null) return;
8         var controllerName = (string)filterContext.RouteData.Values["controller"];
9         if (controllerName != "Login")
10        {
11            //跳转到登陆页
12            filterContext.Result = new RedirectResult("/Login/Index");
13        }
14    }
15 }
```

用户登录时设置 Cookie，下一次请求时 Cookie 将通过 HTTP 发送到服务端。

```
1 public JsonResult SignIn(string input1, string input2)
2 {
3     var loginSuccess = false;
4     if (input1 == "doc" && input2 == "doc")
5     {
6         var cookie = new HttpCookie("userId", "1");
7         Response.Cookies.Add(cookie);
8         loginSuccess = true;
9     }
10    return new JsonResult { Data = loginSuccess, JsonRequestBehavior =
11        JsonRequestBehavior.AllowGet };
12 }
```

7.2 MVC 生命周期

7.2.1 ASP.NET MVC 的处理流程

当我们访问 `http://localhost:8080/Home/Index` 这个地址的时候，请求首先被 `UrlRoutingModule` 截获，截获请求后，从 `Routes` 中得到与当前请求 URL 相符合的 `RouteData` 对象，将 `RouteData` 对象和当前 URL 封装成一个 `RequestContext` 对象，然后从 `RequestContext` 封装的 `RouteData` 中得到 Controller 名字，根据 Controller 的名字，通过反射创建控制器对象，这个时候控制器才真正被激活，最后去执行控制器里面对应的 action。

处理流程:1. 用户发起请求—>`UrlRoutingModule` 获取请求—>`MvcRouteHandler.GetHttpHandler()`
—>`MvcHandler.ProcessRequest()`

2.`UrlRoutingModule` 获取浏览器发起的请求后将 `RouteData` 与 `HttpContext` 合并成为 `RequestContext` 传递到 `IRouteHandler` 接口，`IRouteHandler` 接口的实现类 `MvcRouteHandler` 接口到 `RequestContext` 参数，返回一个 `MvcHandler` 对象，并且为这个对象赋值 `RequestContext`

MvcHandler 对象 根据 Controller 的名字正确的实例化了一个 Controller 对象后。回到 `MvcHandler(System.Web.Mvc.MvcHandler)` 的 `BeginProcessRequest` 方法，可以看到，当得到 Controller 对象之后，首先判断它是不是 `IAsyncController`，如果是则会创建委托用来异步执行。通常情况下，我们都是继承自 Controller 类，这不是一个 `IAsyncController`，于是会直接执行 Controller 的 `Execute` 方法。`Execute` 方法是在 Controller 的基类 `ControllerBase` 中定义的，这个方法除去一些安全检查，初始化了 `ControllerContext` (包含了 `ControllerBase` 和 `Request` 的信息)，核心是调用了 `ExecuteCore(System.Web.Mvc.Controller.ExecuteCore)` 方法。根据 `RequestContext` 参数解析出 `RouteData` 以及 `HttpContext`，根据 `RouteData` 来查找出 Controller 以及对象的 Action 及其 Parameters。在每次执行 action 时 (`System.Web.Mvc.ControllerActionInvoker` 类下的 `InvokeAction` 方法) 都会找出过滤器一并执行，核心语句为：

```
1 FilterInfo filters = this.GetFilters(controllerContext, action);
```

这就是过滤器的执行时机，在应用程序启动的时候全局注册过滤器，在每次 Request 的时候执行过滤器，对于权限控制，日志记录等需要面向切面 (AOP:Aspect Oriented Programming) 的业务来说是非常合适的。

5.`Controller.Execute()` 方法处理流程：查找 Action—> 获取 Action—> 调用 `ActionResult(Abstract 方法)` 的 `ActionResult.ExecuteResult()` 方法

6.`ActionResult.ExecuteResult()` 方法

获取到 `IView` 对象，—> 根据 `Iview` 对象的页面路径获取到具体的 `Page`，—> 调用 `IView.RenderView()` 方法显示页面，`IView` 对象中存储的是页面的路径地址，最终通过页面引擎 (View Engine) 使用该路径生成具体的页面类，`ViewPage(System.Web.Mvc.ViewPage)` 是实现了 `IView` 接口的对象。

7, 最终页面就可以正确的显示。

ViewPage.RenderPartialView() 显示.ascx 文件或者是 ViewPage.RenderView() 显示.aspx 文件。现在 MVC 3 中使用的是 Razor 视图引擎, 和 WebFormViewEngine 一样的处理流程下面附上 Pro Asp.net MVC 的作者的一副图, 如图7.1所示。

7.2.2 ASP.NET 请求生命周期 (ASP.NET MVC Request Life Cycle)

RouteTable(路由表) 的创建

MVC 路由表的创建在站点开始启动之后, 将站点的所有路由缓存到内存中, 当截获 Request 请求时, 从内存中进行匹配。

UrlRoutingModule 请求拦截

The request is first intercepted by the UrlRoutingModule which is a HTTP Module. It is this module that decides whether the request would be handled by our MVC application. UrlRoutingModule selects the first matching route.

- 3) Routing engine 确定 route
- 4) route handler 创建相关的 IHttpHandler 实例
- 5) IHttpHandler 实例确定 Controller(控制器)
- 6) Controller 执行
- 7) 一个视图引擎创建
- 8) 视图呈现

7.2.3 路由忽略

在 MVC 的路由配置中, 有如下语句:

```
1 routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

表示对于 *.axd 类型的链接将不进行路由。axd files are typically implemented as HTTP Handlers. They don't exist as an ASP.NET web page, but rather as a class that implements the IHttpHandler interface. *.axd 类型的链接主要用于获得嵌入到 dll 中的资源, The answer is WebResource.axd. WebResource. axd is an HTTP Handler that is part of the .NET Framework that does one thing and one thing only –it is tasked with getting an embedded resource out of a DLL and returning its content. What DLL to go to and what embedded resource to take are specified through the querystring. For instance, a request to www.yoursite.com/WebResource.axd?d=EqSMS …&t=63421 …might return a particular

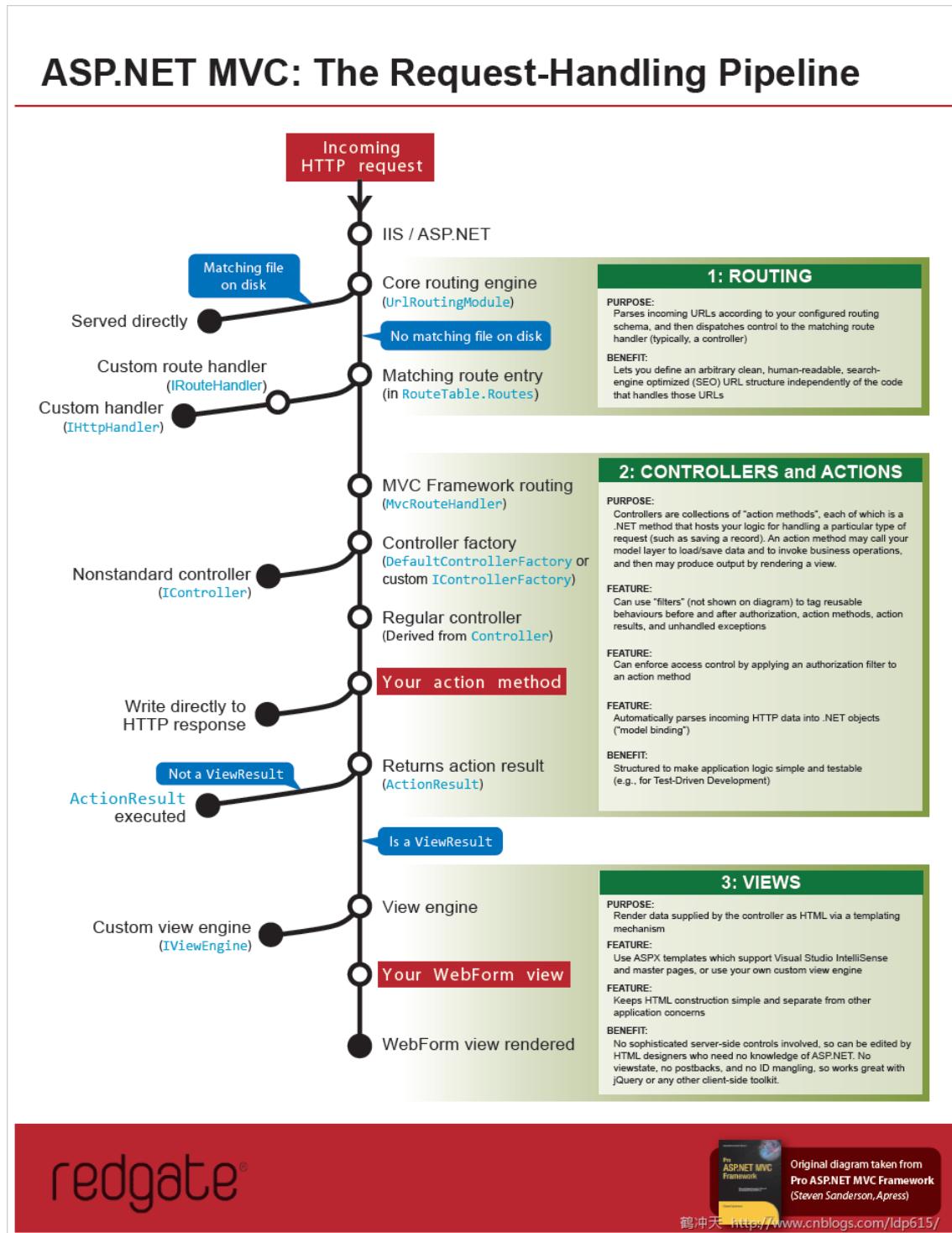


图 7.1: MVC 中 HTTP 请求的处理过程

snippet of JavaScript embedded in a particular assembly. The d querystring parameter contains encrypted information that specifies the assembly and resource to return; the t querystring parameter is a timestamp and is used to only allow requests to that resource using that URL for a certain window of time. 详情参考<http://scottonwriting.net/sowblog/archive/2010/10/28/just-where-is-webresource-axd.aspx>。

7.2.4 Controller 的激活过程 (Process of Controller Activation)

RouteData 从 Url 中获取 Controller 的名称 (具体可参见 Route 类的 GetRouteData 方法), MVC 从 RouteData(在 C:/Windows/Microsoft.NET/Framework/v4.0.30319 文件夹下的 System.Web.dll 中) 中获取 Controller 的名称, 通过 ControllerFactory 传入 Controller 名称激活 Controller。第一步由 Controller 的名称获取 Controller 的类型实例。代码如片段7.1所示:

Listing 7.1: 由 Controller 名称获取类型实例

```

1  private Type GetControllerTypeWithinNamespaces(RouteBase route, string controllerName,
2  											 HashSet<string> namespaces)
3  {
4  	this.ControllerTypeCache.EnsureInitialized(this.BuildManager);
5  	//由 Controller 名称和命名空间获取类型实例
6   ICollection<Type> controllerTypes = this.ControllerTypeCache.GetControllerTypes(
7  	controllerName, namespaces);
8  	switch (controllerTypes.Count)
9  	{
10   	case 0:
11    	return (Type) null;
12   	case 1:
13    	return Enumerable.First<Type>((IEnumerable<Type>) controllerTypes);
14   	default:
15    	throw DefaultControllerFactory.CreateAmbiguousControllerException(route,
16    	controllerName, controllerTypes);
17   }
18 }
```

ControllerFactory 的作用是创建为请求提供服务的 Controller 实例; 在获取实例的过程中利用了反射, 由类的名称获取类型实例就是典型的反射。反射可以增加程序的可扩展性与灵活性, 同时也会降低程序的性能。简单的由类的名称获取类型实例以及类的实例示例代码如片段7.2所示:

Listing 7.2: 反射获取类的实例示例

```

1 //定义一个类, 包含一个公共的名称属性
2 public class Demo
3 {
4  	public string Name = "Dolphin";
5 }
6 //通过反射获取类型实例
7 
```

```

8  Type demoTypeInstance = Type.GetType("MvcApplication1.Controllers.Demo,
9      MvcApplication1");
10 {
11     //获取 Demo 类的实例
12     Demo demoInstance = (Demo)Activator.CreateInstance(demoTypeInstance);
13     string name = demoInstance.Name;//name 的值为:Dolphin
14 }

```

在 GetType 方法中，参数由命名空间和程序集组合而成，中间以逗号隔开，逗号前是命名空间和类名，逗号后是程序集的名称。Type 为 System.Reflection 功能的根，也是访问元数据的主要方式。使用 Type 的成员获取关于类型声明的信息，如构造函数、方法、字段、属性（Property）和类的事件，以及在其中部署该类的模块和程序集。MVC 的激活过程即是如此过程，在 MVC 中通过类型实例激活类实例如代码片段7.3所示。

Listing 7.3: MVC 获取类的实例

```

1 public IController Create(RequestContext requestContext, Type controllerType)
2 {
3     try
4     {
5         return (IController) (this._resolverThunk().GetService(controllerType) ?? Activator.
6             CreateInstance(controllerType));
7     }
8     catch (Exception ex)
9     {
10        throw new InvalidOperationException(string.Format((IFormatProvider) CultureInfo.
11             CurrentCulture, MvcResources.DefaultControllerFactory_ErrorCreatingController,
12             new object[1]
13             {
14                 (object) controllerType
15             }), ex);
16     }
17 }

```

ActionInvoker 的作用是寻找并调用 Action 方法。

7.3 MVC 页面传值

7.3.1 (View → Controller)

MVC 中通过 Ajax 的 POST 方式向 Controller 传值如下代码所示：

```

1 $(function () {
2     $("#debuggerSubmit").click(function () {
3         $.post(
4             "/User/ GetUserAwardTypeCountDebugger",

```

```

5   {
6     action: "post",
7     UserID: $("UserID").val(),
8     UserName: $("AwardType").val()
9   },
10  function (data) {
11    alert("OK");
12  });
13});
14);

```

UserID 将页面控件 ID 为 UserID 的控件值传入后台的 Controller 页面，在 Controller 页面的接收如下代码片段所示（post 采用 Request.Form 的方式获取值，服务端获取 POST 方式传递的参数采用此种写法，服务端获取 Get 请求参数用 Request.QueryString）：

```

1 public ActionResult GetUserAwardTypeCountDebugger()
2 {
3   int userID = Convert.ToInt32(Request.Form["UserID"]);
4   return Content("OK");
5 }

```

使用 ajaxSubmit ajaxSubmit(obj) 方法是 jQuery 的一个插件 jquery.form.js 里面的方法，所以使用此方法需要先引入这个插件。

```

1 $(".right").bind("click", function () {
2   $("form[method='post']").submit();
3 });
4
5 //bind()和delegate()都是由on()实现
6 $(document).on("submit", "form[method='post']", function () {
7   $(this).ajaxSubmit(function (m) {
8     var json = m;
9     if (!json.status) {
10       alert(json.message);//出错，弹出错误提示
11     } else {
12       var arrData = json.data.split("!");
13       var calllbackUrl = "http://m.tour.zouwo.com/Order/Index?orderid=" + json.
14       data;
15       var url = "@(PayUrl)id=" + arrData[0] + "&url=" + calllbackUrl + "";
16       window.location.href = url;
17     }
18   });
19   return false;
20 });

```

后端定义相应的模型（Model），通过模型绑定的方式接收前端的传值。on() 的描述：

```

1 .on( events [, selector ] [, data ], handler(eventObject))

```

事件冒泡通过大量祖先元素会导致较大的性能损失。而使用.on() 方法，事件只会绑定到 \$() 函数的选择符表达式匹配的元素上（上面我的例子中，为了简单绑定到了 document），因此可以精确地定位到页面中的一部分，而事件冒泡的开销也可以减少。

POST 传值 (View 与 Controller 互传)

在 MVC 控制器传递多个 Model 到视图，可使用 ViewData, ViewBag, PartialView, TempData, ViewModel, Tuple。PartialView 是对于哪些需要重复使用的视图部分，提取出来作为部分视图。

View 向 Controller 传值如下代码所示：

```

1 <script>
2     $(function () {
3         $("#debuggerSubmit").click(function () {
4             $.ajax({
5                 type: 'POST',
6                 async: false,
7                 url: "/User/ GetUserAwardTypeCountDebugger",
8                 data:
9                 {
10                     action: "post",
11                     UserID: $("#UserID").val(),
12                     UserName: $("#AwardType").val()
13                 },
14                 success: function (data) {
15                     $("#ajax_msg").html(data);
16                 }
17             });
18         });
19     });
20 </script>
```

其中 url 中的 User 为 View 的 cshtml 页面的上级文件夹名称， GetUserAwardTypeCountDebugger 为 Controller 中相应的方法的名称。在 Controller 中接收参数和回传参数如下代码所示：

```

1 [HttpPost]
2 public JsonResult GetUserAwardTypeCountDebugger()
3 {
4     int userID = Convert.ToInt32(Request.Form["UserID"]);
5     string a = "回传值";
6     return new JsonResult { Data=a, JsonRequestBehavior= JsonRequestBehavior.
7     AllowGet};
}
```

回传值通过 JsonResult 类的 Data 属性进行回传，在前端 cshtml 界面中通过 ajax 调用的 success 进行参数的接收。方法的 HttpPost 属性代表此方法只处理 POST 请求，防止无意间调用，相当于增加了方法的使用约束，增加安全性。可以添加 ValidateAntiForgeryToken 属性防止跨站请求伪造 CSRF (Cross-site Request Forgery) 攻击，相应的在页面的开头需要添加 @Html.AntiForgeryToken()。

title form 元素中的子元素可以直接使用，在 Controller 中可以直接通过 Request.Form 直接获取元素的值。form 的写法如下：

```

1 <form name="" action="/Product/PickAward" method="post" enctype="multipart/
    form-data">
2   <p>
3     <label>用户ID: </label>
4     <input type="text" id="userId" name="userId" class="form-control"
            placeholder="用户 ID"/>
5   </p>
6 </form>
```

form 元素注明为 post，action 为 Controller 中相应的方法的名称。在 Controller 中接收参数通过如下语句接收参数：

```
1 int userId = int.Parse(Request.Form["userId"].ToString());
```

由 Controller 向 View 传递参数可在服务端采用 return View(model) 语句，在前端页面通过 Model 进行接收。

```

1 @model DBCCN.Model.discount_sellerModel
2 @using RR.Web.CCN.FC.Common
3 @{
4   ViewBag.Title = "";
5   Layout = "~/Views/Shared/_Layout.cshtml";
6   string staticPath = PublicAttribute.staticPath;
7   string themeVersion = PublicAttribute.themeVersion;
8 }
```

MVC 传递全局变量

通过 Session 和 Cookie 传值。

7.3.2 PageRouteHandler vs MvcRouteHandler

地址路由功能并不是 MVC 特有的，在 ASP.NET 中为 PageRouteHandler，在 MVC 中为 MvcRouteHandler。对于调用 RouteCollection 的 GetRouteData 获得的 RouteData 对象，其 RouteHandler 来源于匹配的 Route 对象。对于通过调用 RouteCollection 的 MapPageRoute 方

法注册的 Route 来说，它的 RouteHandler 是一个类型为 PageRouteHandler 对象。由于调用 MapPageRoute 方法的目的在于实现请求地址与某个.aspx 页面文件之间的映射，所以我们最终还是要创建的 Page 对象还处理相应的请求，所以 PageRouteHandler 的 GetHttpHandler 方法最终返回的就是针对映射页面文件路径的 Page 对象。ASP.NET MVC 的 Route 对象是通过调用 RouteCollection 的扩展方法 MapRoute 方法进行注册的，它对应的 RouteHandler 是一个类型为 MvcRouteHandler 的对象。如下面的代码片断所示，MvcRouteHandler 用于获取处理当前请求的 HttpHandler 是一个 MvcHandler 对象。MvcHandler 实现对 Controller 的激活、Action 方法的执行以及对请求的相应，毫不夸张地说，整个 MVC 框架实现在 MvcHandler 之中。

```

1 public class MvcRouteHandler : IRouteHandler
2 {
3     IHttpHandler IRouteHandler.GetHttpHandler(RequestContext requestContext)
4     {
5         return GetHttpHandler(requestContext);
6     }
7 }
```

如果 url 是 /home/index?id=3 直接 Request 就 ok。但是如果路由设定为 :controller /action /id url 是 /home/index /3 这时想在页面 View 中获取参数 id 的值，该怎么获取？

一种方式可利用 Action 获取到参数值后，用 ViewData 传到 View 中例如 Controllers 中的 phonelist 这样定义 public ActionResult phonelist(int id) ViewData["id"] = id; return View();

其实，有更加简单的方式，只要在 view 中这样获取就可以：

```
<%=Html.ViewContext.RouteData.Values["id"]%>
```

就算没有 id 的参数也不会报错。同样：

```

1 //写法1
2 <%=Request.RequestContext.RouteData.Values["id"] %>
3 //写法2
4 <%=Html.ViewContext.RouteData.Route.GetRouteData(Html.ViewContext.HttpContext).
    Values["id"]%>
```

也可以取到。

注：在用户控件中是无法直接访问到 RouteData，RouteData 是 Page 对象中的属性，所以需要在用户控件中使用 this.Page.RouteData 来获取参数使用 this.Page.RouteData.Values["id"] 来获取参数的值

7.4 MVC 异常处理 (MVC Error Handling)

在 MVC 里面，万一在行为方法里面抛出了什么异常的，而那个行为方法或者控制器有用上 HandleError 过滤器的，异常的信息都会在某一个视图显示出来，这个显示异常信息的视图默认是在 Views/Shared/Error。在应用程序启动的时候就已经注册了全局过滤器，HandleErrorAttribute 就是系统自带的异常过滤器。在这注册的全局过滤器，可以不用到每个 Controller 或者是每个 Action 去声明，直接作用于全局了，即可以捕捉整个站点的所有异常。MVC 异常处理一般可以在 Action 级别处理，也就是当前 Action，也可以在 Controller 级别处理，包括当前 Controller 和所有 Controller，还可以在应用程序级别处理，也就是 Application_Error，如果都不进行处理，异常将抛给用户。

7.4.1 Application_Error

程序中发生的所有异常，都可以在这里捕获。但是有一个不足的地方，那就是，如果我们在那里捕获异常之后，需要显示错误页面，那么浏览器上的地址是会改变的，已经被重定向了。也就是说，浏览器上显示的地址，并不是真正发生错误的地址，而是仅仅是显示错误信息的页面地址。这个不足，跟在 Web.config 的 customError 中配置错误页面是一样的。很多时候，我们是希望在访问某个 URL 后发生了错误，显示错误页面之后，URL 还是不变。因此，在这里处理异常并不能满足这种需求。在 Global.asax 代码片段 7.4 代码是在程序级别捕获错误。

Listing 7.4: MVC 应用程序级别错误处理

```

1 void Application_Error(object sender, EventArgs e)
2 {
3     var exception = Server.GetLastError().GetBaseException();
4     var errorDetail = new StringBuilder();
5     errorDetail.Append("\r\n" + DateTime.Now.ToString("yyyy.MM.dd HH:mm:ss"));
6     errorDetail.Append("\r\n. 客户信息: ");
7     var serverIp = Request.ServerVariables.Get("HTTP_X_FORWARDED_FOR");
8     var remoteAddr = Request.ServerVariables.Get("Remote_Addr").Trim();
9     var ipAddress = (serverIp != null) ? serverIp.Trim() : remoteAddr;
10    errorDetail.Append("\r\n\tIp: " + ipAddress);
11    errorDetail.Append("\r\n\t浏览器: " + Request.Browser.Browser);
12    errorDetail.Append("\r\n\t浏览器版本: " + Request.Browser.MajorVersion);
13    errorDetail.Append("\r\n\t操作系统: " + Request.Browser.Platform);
14    errorDetail.Append("\r\n. 错误信息: ");
15    errorDetail.Append("\r\n\t页面: " + Request.Url);
16    errorDetail.Append("\r\n\t错误信息: " + exception.Message);
17    errorDetail.Append("\r\n\t错误源: " + exception.Source);
18    errorDetail.Append("\r\n\t异常方法: " + exception.TargetSite);
19    errorDetail.Append("\r\n\t堆栈信息: " + exception.StackTrace);
20    errorDetail.Append("\r\n-----");
21    PublicAttribute.Logger.Error(errorDetail);
22    // 处理完及时清理异常
23    Server.ClearError();
24    // 跳转至友好的出错页面
25    Response.Redirect("~/error/index");

```

```
26 }
```

7.4.2 OnException 继承 HandleErrorAttribute

MVC 的特点就是，每一个请求都对应一个 Controller，当在某个 Controller 中发生异常时，我们都可以在 OnException 中捕获。但是，如果根本就找不到这个 Controller，那么这种错误 OnException 就无能为力了。举个例子：假设有个 Controller 叫 About，当访问 <http://host/About/Index> 发生错误时，是可以在 OnException 中捕获的。但如果我访问 <http://host/Aboute/Index> 的时候，因为根本不存在 Aboute 控制器，所以这种错误是无法在 OnException 捕获的，如果要捕获这种错误，只能在 Application_Error 中处理。虽然 OnException 不能捕获所有的错误，但是，它可以解决 Application_Error 错误页面重定向的问题，在显示错误页面的时候，URL 保持不变。在 Web.config 文件中添加配置如代码片段7.5所示。

Listing 7.5: 配置 MVC 错误处理页面

```
1 <System.Web>
2   <customErrors mode="On" defaultRedirect="/Error/Index" />
3 </System.Web>
```

如果在当前 Controller 中重写 OnException 方法，那么当前 Controller 中的 Action 抛出异常后交由当前 Controller 中重写的 OnException 方法处理，如果继承 HandleErrorAttribute 并重写 OnException 方法，那么可以处理所有 Controller 抛出的异常，如代码片段7.6所示。

Listing 7.6: 自定义 Handler 错误处理

```
1 public class CustomExceptionFilter : HandleErrorAttribute
2 {
3   public override void OnException(ExceptionContext filterContext)
4   {
5     if (!filterContext.ExceptionHandled)
6     {
7       //错误处理代码, 如日志记录、跳转到错误页等
8     }
9     base.OnException(filterContext);
10  }
11 }
```

7.5 脚本压缩和合并

打包 (Bundling) 及压缩 (Minification) 指的是将多个 js 文件或 css 文件打包成单一文件并压缩的做法，如此可减少浏览器需下载多个文件案才能完成网页显示的延迟感，同时通过移除 JS/CSS 文件案中空白、批注及修改 JavaScript 内部函数、变量名称的压缩手法，能有效缩小文件案体积，提高传输效率，提供使用者更流畅的浏览体验。在 ASP.NET MVC 4 中可以使用

BundleTable 绑绑多个 css 文件和 js 文件，以提高网络加载速度和页面解析速度。更为重要的是通过捆绑可以解决 IE 浏览器的 31 个 CSS 文件连接的限制。在做 ASP.NET 项目时很多时候会使用一些开源的 javascript 控件。无形中增加了 css 和 javascript 文件的引用。如果手工将这些 css 文件合并将给将来版本升级造成很大的麻烦。于是，我们只好小心翼翼的处理这些 css 文件在页面中的引用。ASP.NET 绑绑是 ASP.NET 4.5 的新功能，是 System.Web.Optimization 命名空间下。他提供了一些 ASP.NET 运行性能方面的优化，比如，一个页面可能有很多 CSS/JS/图片，通过灵活的应用 BundleTable 类，他可以帮你将文件合并压缩代码优化成一个最理想的文件，然后输出到客户端，从而提高了浏览器下载速度。在 Web.config 文件中 compilation 节点设置 debug 的值可以开启或关闭压缩和合并功能。在下面的 XML 中，debug 设置值为 true，可以禁用脚本压缩和合并功能。

```

1 <system.web>
2   <compilation debug="true" />
3   <!-- Lines removed for clarity. -->
4 </system.web>
```

注册 Bundle.

```

1 public static void RegisterBundles(BundleCollection bundles)
2 {
3     //添加多个 JS 合并且压缩
4     bundles.Add(new ScriptBundle("~/Script/common").Include("~/Script/jquery.js", "~/Script/common.js"));
5     //添加多个 css 合并且压缩
6     bundles.Add(new StyleBundle("~/Style/common").Include("~/Style/common.css", "~/Style/header.css"));
7     //启用压缩/合并
8     BundleTable.EnableOptimizations = true;
9 }
```

在视图中引入压缩脚本。

```

1 <head>
2   <meta name="viewport" content="width=device-width" />
3   <title>View1</title>
4   <!--引用css-->
5   @Styles.Render("~/Style/common")
6   <!--引用JS-->
7   @Scripts.Render("~/Script/index")
8 </head>
```

7.6 参数的传递

在 MVC 中多个参数可以以竖线 (|) 分割，作为一个参数传递到另一个 Action 中，在 Action 中以 Split 函数取出每个参数。也可以通过形如代码所示的链接传递参数，注意这里的链

接的特殊写法，? 和/符号一个都不能少。

```
1 window.location.href = "/FeatureExplorer/Param/?userId=1&groupId=1";
```

在后端的 Action 中接收参数如代码所示。

```
1 public ActionResult Param(string userId, string groupId)
2 {
3     string str1 = userId;
4     string str2 = groupId;
5     return View();
6 }
```

7.6.1 ActionResult

```
1 System.Object
2 System.Web.Mvc.ActionResult
3 System.Web.Mvc.ContentResult
4 System.Web.Mvc.EmptyResult
5 System.Web.Mvc.FileResult
6 System.Web.Mvc.HttpStatusCodeResult
7 System.Web.Mvc.JavaScriptResult
8 System.Web.Mvc.JsonResult
9 System.Web.Mvc.RedirectResult
10 System.Web.Mvc.RedirectToRouteResult
11 System.Web.Mvc.ViewResultBase
```

定义的是 ActionResult，为什么返回值会是 View 呢？其实这个 View 的类型是 ActionResult 的子类 ViewResult。

7.6.2 MVC 中 Razor 语法

在某些情况下，你需要明确地呈现一些值，这些值可以通过 HTML 标签去控制它在页面中呈现的格式。那么您可以使用 Html.Raw 语法，以确保该值的编码。

Text output will generally be HTML encoded. Using Html.Raw allows you to output text containing html elements to the client, and have them still be rendered as such. Should be used with caution, as it exposes you to cross site scripting vulnerabilities.

代码可以如下所示。

```
1 <div class="info-content">
2     @Html.Raw(((Model as discount_sellerModel).sellerIntroduce)❶
3 </div>
```

① 内容会以 HTML 格式进行解析，可以在内容用 HTML 标签控制前台的展示格式

布局页面还有节（Section）的概念，也就是说，如果某个视图模板中定义了一个节，那么可以把它单独呈现出来，用法如下，在母板视图中添加 head 节。

```

1 //母板页中定义StyleCss节
2 @RenderSection("StyleCss")
3
4 //定义head节
5 @RenderSection("head")

```

当然还要在视图中定义节，否则会出现异常：

```

1 //在子页面中使用母板页中定义的StyleCss节，内容会呈现在母板页定义节的位置
2 @section StyleCss{
3     <link href="@newStaticServer/tour/jiulongpo-tour.css" rel="stylesheet" type="
4         text/css" />
5 }
6 @section head{
7     //do
8 }

```

为了防止因缺少节而出现异常，可以给 RenderSection() 提供第 2 个参数：

```
1 @RenderSection("SubMenu", false)
```

或

```

1 @if (IsSectionDefined("SubMenu"))
2 {
3     @RenderSection("SubMenu", false)
4 }
5 else
6 {
7     <p>SubMenu Section is not defined!</p>
8 }

```

7.6.3 ViewData、ViewBag、PartialView、TempData、ViewModel、Tuple

- 如果传递的是“小数据”，我们想到 ViewBag, ViewData
- 如果基于 View 的 Model，我们想到针对该 View 设计 View Model
- 如果视图的某个部分需要被重复使用，就把之提炼出来，成为一个 Partial View
- 当需要跨 controller，跨 action 传递，我们想到 TempData
- 如果传递的是“小数据”，又不想使用 View Model，可以考虑 Tuple

7.6.4 JsonRequestBehavior

```

1 public JsonResult QueryOrderStatus(string orderPayNumber)
2 {
3     return new JsonResult
4     {
5         Data = orderStatus,
6         JsonRequestBehavior = JsonRequestBehavior.AllowGet
7     };
8 }

```

JsonResult 在最后 return Json(list,JsonRequestBehavior.AllowGet) 中增加了第二个参数 JsonRequestBehavior.AllowGet， 默认是 JsonRequestBehavior.DenyGet。之所以我们要在这里设置允许 HTTP GET 请求，是因为 ASP.NET MVC 为了预防一个网站信息泄漏的漏洞，所以默认是禁止客户端的 HTTP GET 请求的。这是一个很出名的漏洞，名字是：JSON 劫持漏洞，所以建议 AJAX 还是使用 POST 请求来获取数据，防止重要的信息被恶意攻击者窃取。允许 GET 请求可能会导致用户在某一网站中仍处于已登录状态时访问另一个网站。这可能会生成导致信息泄漏的安全漏洞。

By default, the ASP.NET MVC framework does not allow you to respond to an HTTP GET request with a JSON payload. If you need to send JSON in response to a GET, you'll need to explicitly allow the behavior by using JsonRequestBehavior.AllowGet as the second parameter to the Json method. However, there is a chance a malicious user can gain access to the JSON payload through a process known as JSON Hijacking. You do not want to return sensitive information using JSON in a GET request.

7.6.5 过滤器 (Filter) 记录访问日志

MVC 支持的过滤器类型有四种，分别是：Authorization(授权),Action (行为) ,Result (结果) 和 Exception (异常)。

```

1 public class ActionRecordFilter : ActionFilterAttribute
2 {
3     public override void OnActionExecuted(ActionExecutedContext filterContext)
4     {
5         var actionMethod = filterContext.Controller
6             .GetType()
7             .GetMethod(filterContext.ActionDescriptor.ActionName);
8         if (actionMethod.ReturnType == typeof(ViewResult))
9         {
10             //ViewResult, 记录当前页面访问数据
11         }
12     }
13 }

```

此方法在每次 Action 执行完毕后都会执行，可在全局记录访客日志信息。有因为 ActionResult 分为多种， JsonResult 等可不进行记录，可根据 ActionResult 的不同类型采取不同的记录动作。

7.7 部分视图

在使用 Ajax 提交时，即使不使用布局 (Layout)，也要指定 Layout=null。

```
1 @{
2     Layout = Request.IsAjaxRequest() ? null : "~/Views/Shared/_Layout.cshtml";
3 }
```

7.8 常见错误

7.8.1 找不到编译动态表达式所需的一种或多种类型。是否缺少引用？

在 MVC 中的 cshtml 页面中， ViewBag 处报错，如图所示。

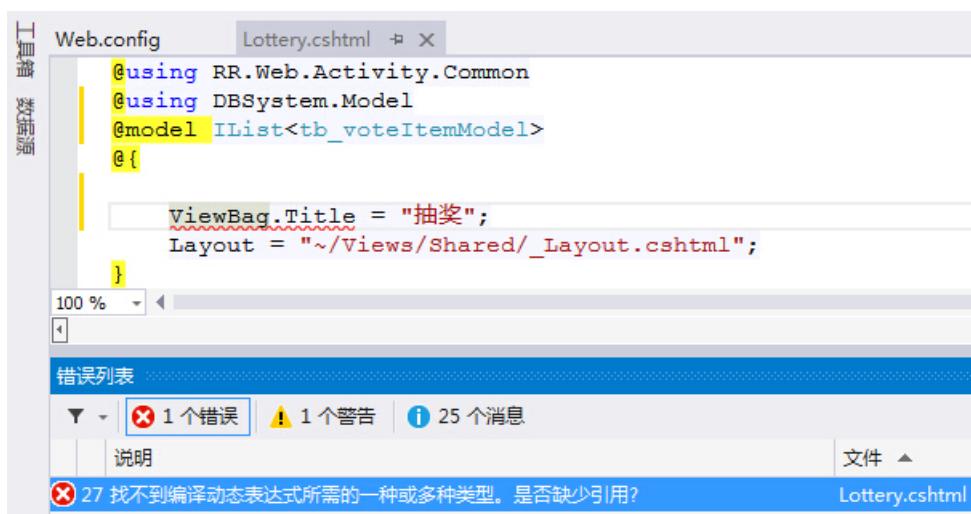


图 7.2: ViewBag 错误

首先查看是否已经引用 Microsoft.CSharp.dll，在配置文件的 System.Web 配置节中添加指定目标框架的相关配置，如代码 7.7 所示。因为 ViewBag 是.NET Framework 4.0 中才有，命名空间为 System.Web.Mvc.dll。由于 compilation 配置节中未指定 targetFramework，估计是程序默认认为是比 4.0 的版本更低，才出现此错误。

Listing 7.7: 配置 MVC 目标框架

```
1 <system.web>
2     <compilation debug="true" targetFramework="4.0" />
3 </system.web>
```

compilation 配置节，用于配置 ASP.NET 用于编译应用程序的所有编译设置。target-Framework 指定网站的目标.NET Framework 的版本，如果省略此特性，则目标版本将由 Web.config 文件中的其他设置以及与网站相关联的 IIS 应用程序池确定。具体更多内容参见：<https://msdn.microsoft.com/zh-cn/library/vstudio/s10awwz0%28v=VS.100%29.aspx>。

7.8.2 指定的转换无效

在从接口接受到 Object 类型数据后，需要转换为指定的泛型类型，提示指定的转换无效，此时需要反序列化返回的对象。

```

1  /// <summary>
2  /// </summary>
3  /// <param name="relativeUrl"> 请求地址 </param>
4  /// <param name="param"> 请求地址 </param>
5  /// <returns></returns>
6  public static T SaveObject<T>(string relativeUrl, string param)
7  {
8      var restUrl = PublicAttribute.WebApiPath + relativeUrl;
9      var restResult = (Message) PublicAttribute.ApiClient.POST(restUrl, param);
10     if (restResult == null)
11         throw new RestfulNullDataException("接口返回空,url:" + restUrl + param);
12     if (restResult.status)
13     {
14         //不能直接强转为泛型 T，需要反序列化
15         return JsonConvert.DeserializeObject<T>(restResult.data.ToString());
16     }
17     PublicAttribute.Logger.Error("SaveObjects failed,url:" + restUrl + param + ",",
18         message:" + restResult.message);
19     return default(T);
}

```

其中 restResult.data 为 Object 类型，使用 JsonConvert 反序列化为指定类型。

7.8.3 未能加载文件或程序集“RR.Web.CCN.Tour”或它的某一个依赖项。 试图加载格式不正确的程序。

检查不能加载的程序集生成选项的目标平台是否是 Any CPU，在 64 位程序集直接加载单个的特殊的 x86 程序集会出现此错误。

Chapter 8

JavaScript

JavaScript 语句和 JavaScript 变量都对大小写敏感。JavaScript 的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript 的主要用途是与用户互动，以及操作 DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。比如，假定 JavaScript 同时有两个线程，一个线程在某个 DOM 节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？所以，为了避免复杂性，从一诞生，JavaScript 就是单线程，这已经成了这门语言的核心特征，将来也不会改变。为了利用多核 CPU 的计算能力，HTML5 提出 Web Worker 标准，允许 JavaScript 脚本创建多个线程，但是子线程完全受主线程控制，且不得操作 DOM。所以，这个新标准并没有改变 JavaScript 单线程的本质。

8.0.1 规范 (Specification)

表示区块起首的大括号，不要另起一行 Javascript 会自动添加句末的分号，导致一些难以察觉的错误。

1. 调用函数的时候，函数名与左括号之间没有空格
2. 函数名与参数序列之间，没有空格
3. 所有其他语法元素与左括号之间，都有一个空格
4. 不要省略句末的分号
5. 不要使用 with 语句
6. 不要使用”相等”（==）运算符，只使用”严格相等”（==）运算符，JS 数据类型分为 6 类：Undefined Null String Number Boolean Object。Javascript 有两个表示”相等”的运算符：“相等”（==）和”严格相等”（==）。因为”相等”运算符会自动转换变量类型，造成很多意想不到的情况，建议使用严格相等
7. 不要将不同目的语句，合并成一行
8. 所有变量声明都放在函数的头部
9. 不要使用 new 命令，改用 Object.create() 命令 这种做法的问题是，一旦你忘了加上 new，myObject() 内部的 this 关键字就会指向全局对象，导致所有绑定在 this 上面的变量，都变成全部变量。
10. 构建函数的函数名，采用首字母大写（InitialCap）；其他函数名，一律首字母小写
11. 不要使用自增（++）和自减（-）运算符，用 += 和 -= 代替
12. 总是使用大括号表示区块

8.0.2 进入页面时执行 JavaScript

两种方法实现在 HTML 页面加载完毕后运行某个 JavaScript 脚本。

```

1 <script type="text/javascript">
2   /*JavaScript原生写法*/
3   window.onload = function () {
4     //Run code when page load...
5   }
6
7   /*jQuery写法*/
8   $(window).load = function () {
9     //Run code when page load...
10  }
11 </script>

```

当一个文档完全下载到浏览器中时，才会触发 window.onload 事件。这意味着页面上的全部元素对 JavaScript 而言都是可以操作的，也就是说页面上的所有元素加载完毕才会执行。window 对象是 JavaScript 层级中的顶层对象。window 对象会在 <body> 或 <frameset> 每次出现时被自动创建。window 对象表示一个浏览器窗口或一个框架。在客户端 JavaScript 中，window 对象是全局对象，所有的表达式都在当前的环境中计算。也就是说，要引用当前窗口根本不需要特殊的语法，可以把那个窗口的属性作为全局变量来使用。例如，可以只写 document，而不必写 window.document。同样，可以把当前窗口对象的方法当作函数来使用，如只写 alert()，而不必写 window.alert()。除了上面列出的属性和方法，window 对象还实现了核心 JavaScript 所定义的所有全局属性和方法。window 对象的 window 属性和 self 属性引用的都是它自己。当你想明确地引用当前窗口，而不仅仅是隐式地引用它时，可以使用这两个属性。除了这两个属性之外，parent 属性、top 属性以及 frame[] 数组都引用了与当前 window 对象相关的其他 window 对象。以下为 jQuery 方法，需要引用 jQuery 文件。

```

1 <script type="text/javascript">
2   $(document).ready(function(){
3     //Run code when page load...
4   });
5 </script>

```

会在 DOM 完全就绪并可以使用时调用。虽然这也意味着所有元素对脚本而言都是可以访问的，但是，并不意味着所有关联的文件都已经下载完毕。换句话说，当 HTML 下载完成并解析为 DOM 树之后，代码就会执行。

8.0.3 一道 Javascript 试题

```

1 function Foo() {
2   getName = function () { alert (1); };
3   return this;
4 }
5 Foo.getName = function () { alert (2);};
6 Foo.prototype.getName = function () { alert (3);};
7 var getName = function () { alert (4);};
8 function getName() { alert (5);}

```

```

9 //请写出以下输出结果:
10 Foo.getName();//2
11 getName();//4
12 Foo().getName();//1
13 getName();//1
14 new Foo.getName();//2
15 new Foo().getName();//3
16 new new Foo().getName();//3

```

第一问的 Foo.getName 是访问 Foo 函数上存储的静态属性。第二问，直接调用 getName 函数。既然是直接调用那么就是访问当前上文作用域内的叫 getName 的函数，所以跟 1 2 3 都没什么关系。此题有无数面试者回答为 5。此处有两个坑，一是变量声明提升，二是函数表达式。

变量声明提升 所有声明变量或声明函数都会被提升到当前函数的顶部。

函数表达式 var getName 与 function getName 都是声明语句，区别在于 var getName 是函数表达式，而 function getName 是函数声明。

8.0.4 Javascript 日期

Javascript 的 Date 对象用于处理日期和时间，如下代码片段。

```
1 var myDate=new Date();
```

Date 对象会自动把当前日期和时间保存为其初始值。

8.0.5 jQuery Rotate 插件实现旋转

```

1 var rotateFn = function (awards, angles, txt) {
2     $('#rotate').stopRotate();
3     $('#rotate').rotate(
4         angle: 0,//起始角度
5         animateTo: angles + 1800,//结束的角度
6         duration: 8000,//转动时间
7         callback: function () {
8             //回调函数
9         }
10    );
11};

```

8.0.6 JavaScript 调试

http://www.ruanyifeng.com/blog/2011/03/firebug_console_tutorial.html

8.0.7 匹配手机号码

```

1 if (!userphone.match(/^1[3|4|5|8][0-9]\d{4,8}$/)) {
2     tanc4_show("请输入正确的手机号码!"); return;
3 }
```

8.0.8 JavaScript 清空数组

splice

```

1 var ary = [1,2,3,4];
2 ary.splice(0,ary.length);
3 console.log(ary); // 输出 [], 空数组, 即被清空了
```

`splice()` 方法可删除从 `index` 处开始的零个或多个元素，并且用参数列表中声明的一个或多个值来替换那些被删除的元素。第一个参数为起始位置，第二个参数为删除指定长度元素。返回值是一个由所移除的元素组成的新 `Array` 对象。

length 赋值为 0

```

1 var ary = [1,2,3,4];
2 ary.length = 0;
3 console.log(ary); // 输出 [], 空数组, 即被清空了
```

赋值为 []

```

1 var ary = [1,2,3,4];
2 ary = []; // 赋值为一个空数组以达到清空原数组
```

这里其实并不能说是严格意义的清空数组，只是将 `ary` 重新赋值为空数组，之前的数组如果没有引用在指向它将等待垃圾回收。`length` 赋值为 0 保留了数组其它属性，赋值为 `[]` 则未保留。很多人认为 `length` 赋值为 0 的效率很高些，因为仅仅是给 `length` 重新赋值了，而方式 3 则重新建立个对象。经测试恰恰是赋值为 `[]` 的效率高。

8.0.9 数组是否有包含关系

判断两个数组之间是否有包含关系，`mutexArr` 代表具有互斥元素的数组，`checkedArr` 代表当前选择的元素构成的数组，要求选择的元素不能包含互斥元素。

```

1 function contains1(mutexArr, checkedArr) {
```

```

2  var i = mutexArr.length;
3  var k = 0;
4  while (i > 0) {
5      var existsValue = $.inArray(mutexArr[i - 1], checkedArr);
6      if (existsValue != -1) {
7          k++;
8      }
9      i--;
10 }
11 if ((k == mutexArr.length || k > mutexArr.length) && k > 1) {
12     return true;
13 }
14 return false;
15 }
```

需要使用 jQuery 元素是否包含在数组中的方法 `inArray`，`inArray` 方法确定第一个参数在数组中的位置，从 0 开始计数（如果没有找到则返回 -1）。也可以使用库函数 `Underscore.js`¹ 中的 `intersection` 方法，如代码所示。

```

1 function contains2(mutexArr, checkedArr) {
2     var arr = __.intersection(mutexArr, checkedArr);
3     if (arr.length > 1) {
4         return true;
5     }
6     return false;
7 }
```

`___.intersection(*arrays)` 返回传入 arrays（数组）交集。结果中的每个值是存在于传入的每个 arrays（数组）里。

8.0.10 XMLHttpRequest

8.0.11 获取元素文本

以 `span` 元素为例，获取元素文本（此处是 `aaaa`）可使用如下方式。

```

1 <span id="testid">aaaa</span>
2
3 /*方式1：从起始位置到终止位置的内容，但它去除Html标签*/
4 var x1 = document.getElementById("testid").innerText;
5
6 /*方式2：从对象的起始位置到终止位置的全部内容,包括Html标签*/
7 var x2 = document.getElementById("testid").innerHTML;
8
```

¹项目主页：<http://underscorejs.org/>。Underscore provides over 100 functions that support both your favorite workaday functional helpers: `map`, `filter`, `invoke` —as well as more specialized goodies: `function binding`, `javascript templating`, `creating quick indexes`, `deep equality testing`, and so on.

```

9  /*jQuery写法*/
10 var x3 = $("#testid").text();
11 var x4 = $("#testid").val();

```

innerHTML 会包含 HTML 页面中的换行符等，如果标签写法如下

```

1 <div id="demo">
2 <div>

```

那么通过 `document.getElementById("demo").innerHTML` 取出的内容其长度为 1。In Javascript, new lines are represented by a single "\n" character. Textarea values might have the character combos "\n \r" in the text box, but once they are pulled into Javascript, "\n \r" becomes JUST "\n". When replacing out new line characters, treat them as you would any other character. They are not special, they just have special notation.

```

//用中文分号替换英文分号、中英文逗号或者回车
function ReplaceSeparator(mobiles) {
    var i;
    var result = "";
    var c;
    for (i = 0; i < mobiles.length; i++) {
        c = mobiles.substr(i, 1);
        if (c == ";" || c == "," || c == "，" || c == "\n")
            result = result + ";";
        else if (c != "\r")
            result = result + c;
    }
    return result;
}

```

8.0.12 序列化 (Serialize) 与反序列化 (Deserialize)

在针对参数较多的使用场景中，将参数序列化为 Json 可以大大简化代码的编写和逻辑的处理。JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. Douglas Crockford 是 Web 开发领域最知名的技术权威之一，ECMA JavaScript 2.0 标准化委员会委员。被 JavaScript 之父 Brendan Eich 称为 JavaScript 的大宗师 (Yoda)，是 JSON、JSLint、JSMin 和 ADSafe 的创造者。

在数据传输流程中，Json 是以文本，即字符串的形式传递的，而 JS 操作的是 JSON 对象，所以，JSON 对象和 JSON 字符串之间的相互转换是关键，如下代码片段由 Json 字符串转化为 Json 对象。

```

1 var str1 = "name": "cxh", "sex": "man";

```

```

2 var obj = eval('(' + str1 + ')');
3 alert("eval:" + obj.name);

```

eval() 函数是 js 自带的，JavaScript 原生的写法反序列化 Json 为对象，会产生安全和性能问题，不推荐使用。IE8(兼容模式),IE7 和 IE6 也可以使用 eval() 将字符串转为 JSON 对象，但不推荐这些方式，这种方式不安全 eval 会执行 json 串中的表达式。

```

1 var str1 = '{"name": "cxh", "sex": "man"}';
2
3 /*
4 由JSON字符串转换为JSON对象,使用jQuery封装的方法
5 */
6 var objParseJson = jQuery.parseJSON(str1);
7 alert("objParseJson:" + objParseJson.name);
8
9 /*
10 由JSON字符串转换为JSON对象
11 */
12 var objJsonParse = JSON.parse(str1);
13 alert("objJsonParse:" + objJsonParse.name);
14
15 /*
16 将对象转化为Json, IE7和IE6没有JSON对象
17 */
18 var jsonParam = JSON.stringify(order);

```

8.1 Ajax

8.1.1 Ajax 提交原生写法

Ajax 的基础是 XMLHttpRequest， XMLHttpRequest 是一个浏览器接口，使得 Javascript 可以进行 HTTP(S) 通信。

```

1 function saveUserPhone(mobile) {
2     var xhr = new XMLHttpRequest();
3     xhr.onreadystatechange = function () {
4         if (xhr.readyState === 4) {
5             if (xhr.status === 200) {
6                 var response=xhr.responseText;
7             } else {
8                 xhr.abort();
9                 alert("服务器忙，请稍后重试");
10            }
11        }
12    }
13    xhr.open("GET", "/Active/SaveUserPhone?phone=" + mobile, true);
14    xhr.send(null);

```

```
15 }
```

最后一个参数 asynch 表示是否采用异步方法，true 为异步，false 为同步。

8.1.2 Ajax 提交

Ajax 提交的语法如下：

```
1 $.post(url, { id: productId }, function (msg) {
2     if (msg!=="OK") {
3         alert("访问人数太多，请稍后再试！");
4     }
5});
```

8.1.3 Ajax 原理和 XMLHttpRequest 对象

Ajax 的原理简单来说通过 XMLHttpRequest 对象来向服务器发异步请求，从服务器获得数据，然后用 javascript 来操作 DOM 而更新页面。这其中最关键一步就是从服务器获得请求数据。要清楚这个过程和原理，我们必须对 XMLHttpRequest 有所了解。

XMLHttpRequest 是 ajax 的核心机制，它是在 IE5 中首先引入的，是一种支持异步请求的技术。简单的说，也就是 javascript 可以及时向服务器提出请求和处理响应，而不阻塞用户。达到无刷新的效果。所以我们先从 XMLHttpRequest 讲起，来看看它的工作原理。首先，我们先来看看 XMLHttpRequest 这个对象的属性。它的属性有：

onreadystatechange 每次状态改变所触发事件的事件处理程序。

responseText 从服务器进程返回数据的字符串形式。

responseXML 从服务器进程返回的 DOM 兼容的文档数据对象。

status 从服务器返回的数字代码，比如常见的 404（未找到）和 200（已就绪）

statusText 伴随状态码的字符串信息

readyState 对象状态值

0 (未初始化) 对象已建立，但是尚未初始化（尚未调用 open 方法）

1 (初始化) 对象已建立，尚未调用 send 方法

2 (发送数据) send 方法已调用，但是当前的状态及 http 头未知

3 (数据传送中) 已接收部分数据，因为响应及 http 头不全，这时通过 responseBody 和 responseText 获取部分数据会出现错误，

4 (完成) 数据接收完毕，此时可以通过 responseXml 和 responseText 获取完整的回应数据

但是,由于各浏览器之间存在差异,所以创建一个 XMLHttpRequest 对象可能需要不同的方法。这个差异主要体现在 IE 和其它浏览器之间。下面是一个比较标准的创建 XMLHttpRequest 对象的方法。

```

1  function CreateXmlHttp()
2  {
3      //非IE浏览器创建XmlHttpRequest对象
4      if(window.XMLHttpRequest)
5      {
6          xmlhttp=new XMLHttpRequest();
7      }
8      //IE浏览器创建XmlHttpRequest对象
9      if(window.ActiveXObject)
10     {
11         try
12         {
13             xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
14         }
15         catch(e)
16         {
17             try{
18                 xmlhttp=new ActiveXObject("msxml2.XMLHTTP");
19             }
20             catch(ex){
21             }
22         }
23     }
24 }
```

8.2 jQery

jQery 的基本设计思想和主要用法,就是“选择某个网页元素,然后对其进行某种操作”。这是它区别于其他 Javascript 库的根本特点。使用 jQery 的第一步,往往就是将一个选择表达式,放进构造函数 `jQuery()` (简写为 `$`),然后得到被选中的元素²。jQery 具有如下优势:

1. 轻量级
2. 强大的选择器
3. 出色的 DOM 操作封装
4. 可靠的事件处理机制
5. 完善的 Ajax
6. 不污染顶级变量

²http://www.ruanyifeng.com/blog/2011/07/jquery_fundamentals.html

7. 出色的浏览器兼容性
8. 链式操作方式
9. 隐式迭代，大幅减少代码量，减少循环结构
10. 丰富的插件支持
11. 行为层与结构层分离，是 jQuery 开发 HTML 开发各司其职
12. 完善的文档
13. 开源

8.2.1 jQuery 事件绑定

jQuery 事件绑定的 3 种写法：

```

1 $("#apply").click(function(){});
2 $("#apply").bind("click",function(){});
3 $("body").on("click","#apply",function(){});
```

第一种写法只是 bind 方式的一种简写，bind 的第一个参数代表的含义是：事件类型（注意不需要加 on），function 中的代码就是你要执行的逻辑代码。bind 还允许你绑定多个事件，事件名字之间用空格隔开。

```
1 $("a").bind("click mouseover",function(){});
```

不管你用的是（click/bind/delegate）之中那个方法，最终都是 jQuery 底层都是调用 on 方法来完成最终的事件绑定。因此从某种角度来讲除了在书写的方便程度及习惯上挑选，不如直接都采用 on 方法来的痛快和直接。采用最简单写法容易产生过多绑定事件，比较消耗内存。.bind()，.live()，.delegate() 都是通过.on() 来实现的，.unbind()，.die()，.undelegate()，也是一样的都是通过.off() 来实现的。on 方法的语法为：

```

1 // jQuery 1.7+
2 $("ancestor").on( "click", "selector" [, data ], handler );
```

第一个参数是事件名称，第二个参数是子选择器，从 jQuery 1.7 开始，on() 函数提供了绑定事件处理程序所需的所有功能，用于统一取代以前的 bind()、delegate()、live() 等事件函数。

8.2.2 \$(function () {}) 的作用

\$(function () {}) 其实是 \$(document).ready(){} 函数的缩写形式。在 \$(document).ready(){} 函数内的所有代码都将在 DOM 加载完毕后，页面全部内容（包括图片等）加载完毕前被执行。在 jQuery 文件里面有定义 \$=jQuery；用 \$(" #id") 和用 jQuery("#id") 是一样的。只是 \$ 在其它的 js 框架里面也被用过，可能出现重复。

```
1 var Hello = jQuery.noConflict();
```

这样以后就可以用 Hello 做前缀了

```
1 Hello("txt1");
```

以下三种方式相同：

```
1 $(document).ready(function() {
2     'Handler for .ready() called'
3});
```

这个方法接收一个 function 类型的参数 ready(handler), 方法的作用是：Specify a function to execute when the DOM is fully loaded. 即当 DOM 加载完毕的时候，执行这个指定的方法。因为只有 document 的状态 ready 之后，对 page 的操作才是安全的。\$(document).ready() 仅在 DOM 准备好的时候执行一次。与之相比，load 事件会等到页面渲染完成执行，即等到所有的资源（比如图片）都完全加载完成的时候。\$(window).load(function())³ 会等整个页面，不仅仅是 DOM，还包括图像和 iframes 都准备好之后，再执行。而 ready() 是在 DOM 准备好之后就执行了，即 DOM 树建立完成的时候。所以通常 ready() 是一个更好的时机。

```
1 $(function() {
2     'Handler for .ready() called'
3});
```

```
1 $().ready(handler)
```

8.2.3 CheckBox

获取页面中选中的 CheckBox 的 ID 值。

```
1 var str = "";
2 $("input[name='wjdc']:checked").each(function () {
3     str += $(this).attr("id") + ",";
4 });
5 alert(str);
```

this 是 JavaScript 语言的一个关键字，它代表函数运行时，自动生成的一个内部对象。input 标签下名称为 wjdc 且类型为 CheckBox 的 ID 值。设置 checkbox 选中状态。

```
1 $("#item_2-3").attr("checked", false);
```

³jQuery 的写法，JavaScript 原生的写法如：window.onload=function(){code goes on here...}。详情可参见：http://www.30t.com/index-htm-LangType-cn-BasePage-702-article_id-6480.htm

其中 #item_2-3 为控件 id，在赋值控件名称时需要考虑 jQuery 特殊字符，因为 jQuery 选择器是用正则表达式实现的，所以正则表达式的特殊符号一般都要处理，文档里面描述：使用如下任何的元字符

```
1 !"#$%&'()*+,./:;<=>?@[\]^|
```

作为名称的文本部分时，它必须被两个反斜杠转义。

8.2.4 jQuery 中 prop 与 attr 区别

在 jQuery 1.6 之前，只有 attr() 函数可用，该函数不仅承担了 attribute 的设置和获取工作，还同时承担了 property 的设置和获取工作。例如：在 jQuery 1.6 之前，attr() 也可以设置或获取 tagName、className、nodeName、nodeType 等 DOM 元素的 property。

直到 jQuery 1.6 新增 prop() 函数，并用来承担 property 的设置或获取工作之后，attr() 才只用来负责 attribute 的设置和获取工作。

此外，对于表单元素的“checked”、“selected”、“disabled”等属性，在 jQuery 1.6 之前，attr() 获取这些属性的返回值为 Boolean 类型：如果被选中（或禁用）就返回 true，否则返回 false。

但是从 1.6 开始，使用 attr() 获取这些属性的返回值为 String 类型，如果被选中（或禁用）就返回“checked”、“selected”或“disabled”，否则（即元素节点没有该属性）返回 undefined。并且，在某些版本中，这些属性值表示文档加载时的初始状态值，即使之后更改了这些元素的选中（或禁用）状态，对应的属性值也不会发生改变。

因为 jQuery 认为：attribute 的“checked”、“selected”、“disabled”就是表示该属性初始状态的值，property 的 checked、selected、disabled 才表示该属性实时状态的值（值为 true 或 false）。

因此，在 jQuery 1.6 及以后版本中，请使用 prop() 函数来设置或获取 checked、selected、disabled 等属性。对于其它能够用 prop() 实现的操作，也尽量使用 prop() 函数。

8.2.5 NETWORK_ERR: XMLHttpRequest Exception 101

Request aborted because it was cached or previously requested。将 aAjax 请求设置为异步即可。

```
1 $.ajax({
2     type: "get",
3     data: { "userId": userId, "wxmediaId": res.serverId, "activityId": activityId },
4     dataType: "json",
5     cache: false,
6     async: true,
7     url: "/QiuJiFangJiaoHui/UpLoadImg?sjm=" + timestamp,
8     success: function (data) {
9         alert("上传成功");
10        //设置上传状态，避免反复上传图片（如用户点击后退按钮返回到此页面）
11        $("#IsUploaded").val("1");
```

```
12 //跳转到特殊抽奖页面
13 var webpath = $("#webpath").val();
14 var url = webpath + "QiuJiFangJiaoHui/lotterySpecial?activityId=" + activityId
15 ;;
16 window.location = url;
17 },
18 error: function (XMLHttpRequest, textStatus, errorThrown) {
19     alert("errorThrown:" + errorThrown);
20 }
```

8.2.6 Ajax 失效的

ajax 请求是同步还是异步造成的问题。有时候我们会遇到这种情况，ajax 请求方法，里面配置和传值等等都是正确的，但是就是请求不到想要的数据，到最后甚至怀疑是不是开发工具的问题，这时候你就应该观察一下，ajax 请求是异步还是同步。例如，你用 post 请求传值到另一个页面后台，但是页面一加载你的 ajax 就已经执行过了，传值接收是在后台才完成的，这时候就请求不到数据，所以可以考虑把 ajax 请求改为同步试试，Ajax 默认为异步。

By default, all requests are sent asynchronous (e.g. this is set to true by default). If you need synchronous requests, set this option to false. Note that synchronous requests may temporarily lock the browser, disabling any actions while the request is active.

Chapter 9

HTML

9.0.1 规范

- 用两个空格来代替制表符 (tab) – 这是唯一能保证在所有环境下获得一致展现的方法
- 嵌套元素应当缩进一次（即两个空格）
- 对于属性的定义，确保全部使用双引号，绝不要使用单引号
- 不要在自闭合 (self-closing) 元素的尾部添加斜线 – HTML5 规范中明确说明这是可选的
- 不要省略可选的结束标签 (closing tag) (例如, 或 </body>)
- 建议为 html 根元素指定 lang 属性，从而为文档设置正确的语言。这将有助于语音合成工具确定其所应该采用的发音，有助于翻译工具确定其翻译时所应遵守的规则等等。

```
1 <html lang="zh-CN">
2   <!-- ... -->
3 </html>
```

- 根据 HTML5 规范，在引入 CSS 和 JavaScript 文件时一般不需要指定 type 属性，因为 text/css 和 text/javascript 分别是它们的默认值
- http://codeguide.bootcss.com/
- /是 WebSite 根目录， /是 ASP.NET Application 根目录...

9.0.2 HTML5

<!DOCTYPE> 声明必须位于 HTML5 文档中的第一行，也就是位于 <html> 标签之前。该标签告知浏览器文档所使用的 HTML 规范。doctype 声明不属于 HTML 标签；tag; 它是一条指令，告诉浏览器编写页面所用的标记的版本。在所有 HTML 文档中规定 doctype 是非常重要的，这样浏览器就能了解预期的文档类型。HTML 4.01 中的 doctype 需要对 DTD 进行

引用，因为 HTML 4.01 基于 SGML。而 HTML 5 不基于 SGML，因此不需要对 DTD 进行引用，但是需要 doctype 来规范浏览器的行为（让浏览器按照它们应该的方式来运行。）。没有定义 doctype 才会开启怪异模式（Quirks Mode），也就是说你只需要定义 <!doctype html> 就可以让浏览器在严格模式（标准模式）下渲染页面，而不需要指定某个类型 dtd。让我们来回顾一下，所有的浏览器都需要两种模式：怪异模式和严格模式（Strict Mode）或者叫标准模式（Standards Mode）。IE 6 for Windows/mac, Mozilla,Safari 和 Opera 都实现了这两种模式，但是 IE 6 以下版本永远定在了怪异模式。关于两种模式，你需要知道以下几点：

- 在标准化之前写的页面是没有 doctype 的，因此没有 doctype 的页面是在怪异模式下渲染的。
- 反过来说，如果 web 开发人员加入的 doctype，说明他知道他所要做的事情，大部分的 doctype 会开启严格模式（标准模式），页面也会按照标准来渲染。
- 任何新的或者未知的 doctype 都会开启严格模式（标准模式）。每个浏览器都有自己的方式来激活怪异模式。你可以看看这个清单：<http://hsivonen.iki.fi/doctype/>

9.1 a 标签

9.1.1 href="#"

通常情况下应该避免使用 href="#"，它会导致点击任何地方的时候页面自动移动到顶部。 You should (almost) never have href="#" . It is a link to an undefined anchor (which will be the top of the page). People who use it normally do so because they want to dangle JavaScript off it. # 包含了一个位置信息，默认的锚是 #top 也就是网页的上端，所以出现了点击链接时网页自动定位到顶端的现象，为了防止将页面定位到顶部，可以换一种写法：

```

1 <a href="javascript:void(0);" target="_blank"></a>
2 <a href="###"></a>
3 <a href="javascript:void(null)"></a>
```

void 是一个操作符，这个操作符指定要计算一个表达式但是不返回值。如果在 void 中写入 0(void(0))，则什么也不执行，从而也就形成了一个空链接。target 为 blank 表示点击这个链接时跳转到一个空的页面。

9.1.2 Meta

META (Metadata) 元素通常用于指定网页的描述，关键词，的文件的最后修改，作者，和其他元数据。元数据可以被使用浏览器（如何显示内容或重新加载页面），搜索引擎（关键词），或其他 Web 服务调用。

```

1 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
2 <meta charset="utf-8" />
```

charset 属性是 HTML5 中的新属性，且替换了：<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">。仍然允许使用 http-equiv 属性来规定字符集，但是使用新方法可以减少代码量。http-equiv 为 HTTP 响应的标题头 Http Response Header)。

```
1 <meta name="参数" content="具体的参数值"/>
2 <meta http-equiv="参数" content="参数变量值"/>
```


Chapter 10

CSS

10.1 规范

文件规范

1、文件均归档至约定的目录中。具体要求通过豆瓣的 CSS 规范进行讲解：所有的 CSS 分为两大类：通用类和业务类。通用的 CSS 文件，放在如下目录中：

- 基本样式库 /css/core
- 通用 UI 元素样式库 /css/lib
- JS 组件相关样式库 /css/ui

业务类的 CSS 是指和具体产品相关的文件，放在如下目录中：

- 读书 /css/book/
- 电影 /css/movie/
- 音乐 /css/music/
- 社区 /css/sns/
- 小站 /css/site/
- 同城 /css/location/
- 电台 /css/radio/

外联 CSS 文件适用于全站级和产品级通用的大文件。内联 CSS 文件适用于在一个或几个页面共用的 CSS。另外一对具体的 CSS 进行文档化的整理。如：

- util-01 reset /css/core/reset.css

- util-02 通用模块容器 /css/core/mod.css
- ui-01. 喜欢按钮 /css/core/fav_btn.css
- ui-02. 视频/相册列表项 /css/core/media_item.css
- ui-03. 评星 /css/core/rating.css
- ui-04. 通用按钮 /css/core/common_button.css
- ui-05. 分页 /css/core/pagination.css
- ui-06. 推荐按钮 /css/core/rec_btn.css
- ui-07. 老版对话框 /css/core/old_dialog.css
- ui-08. 老版 Tab /css/core/old_tab.css
- ui-09. 老版成员列表 /css/core/old_userlist.css
- ui-10. 老版信息区 /css/core/notify.css
- ui-11. 社区用户导航 /css/core/profile_nav.css
- ui-12. 当前大社区导航 /css/core/site_nav.css
- ui-13. 加载中 /css/lib/loading.css

2、文件引入可通过外联或内联方式引入。外联方式：（类型声明 type=”text/css” 可以省略）内联方式：（类型声明 type=”text/css” 可以省略）link 和 style 标签都应该放入 head 中，原则上，不允许在 html 上直接写样式。避免在 CSS 中使用 @import，嵌套不要超过一层。

3、文件名、文件编码及文件大小文件名必须由小写字母、数字、中划线组成文件必须用 UTF-8 编码，使用 UTF-8（非 BOM），在 HTML 中指定 UTF-8 编码，在 CSS 中则不需要特别指定因为默认就是 UTF-8。单个 CSS 文件避免过大（建议少于 300 行）

命名规范

- 规则命名中，一律采用小写加中划线的方式，不允许使用大写字母或 _
- id 用于标识模块或页面的某一个父容器区域，名称必须唯一，不要随意新建 id

图片命名规范

图片的名称分为头尾两部分，用下划线隔开，头部分表示此图片的大类性质例如：广告、标志、菜单、按钮等等。放置在页面顶部的广告、装饰图案等长方形的图片取名：banner 标志性的图片取名为：logo 在页面上位置不固定并且带有链接的小图片我们取名为 button 在页面上某一个位置连续出现，性质相同的链接栏目的图片我们取名：menu 装饰用的照片我们取名：pic 不带链接表示标题的图片我们取名：title 范例：banner_sohu.gif banner_sina.gif menu_aboutus.gif menu_job.gif title_news.gif logo_police.gif logo_national.gif pic_people.jpg 鼠标感应效果图片命名规范为”图片名 +_on/off”。例如：menu1_on.gif menu1_off.gif

10.1.1 选择器优先级

一般而言，选择器越特殊，它的优先级越高。也就是选择器指向的越准确，它的优先级就越高。通常我们用 1 表示标签名选择器的优先级，用 10 表示类选择器的优先级，用 100 标示 ID 选择器的优先级，内联样式表的权值为 1000。一般不建议使用内联样式表，也完全违背了内容和显示分离或结构与表现分离的思想，DIV+CSS 的优点也不能再有任何体现。良好结构的 HTML 页面内几乎没有表现属性的标签。代码非常干净简洁。例如，原先的代码，现在可以只在 HTML 中写，所有控制表现的东西都写到 CSS 中去，在结构化的 HTML 中，table 就是表格，而不是其他什么（更不能被用来布局和定位）。

10.1.2 标签

Listing 10.1: CSS 代码标签示例

```
1 .content{  
2     line-height: 35px;  
3     font-size: 25px;  
4     text-align: center;  
5     font-family: Georgia,Serif❶;  
6     display: block❷;  
7     width: 600px;  
8     height: 400px;  
9     overflow: scroll❸;  
10    text-decoration: underline;  
11    border: 5px solid #e2d218❹;  
12    clear: both❺;  
13 }
```

- ❶ font-family 可以把多个字体名称作为一个“回退”系统来保存。如果浏览器不支持第一个字体，则会尝试下一个。也就是说，font-family 属性的值是用于某个元素的字体族名称或/及类族名称的一个优先表。浏览器会使用它可识别的第一个值。
- ❷ 这个属性用于定义建立布局时元素生成的显示框类型。对于 HTML 等文档类型，如果使用 display 不谨慎会很危险，因为可能违反 HTML 中已经定义的显示层次结构。对于 XML，由于 XML 没有内置的这种层次结构，所有 display 是绝对必要的。值为 block 时此元素将显示为块级元素，此元素前后会带有换行符。
- ❸ overflow 属性规定当内容溢出元素框时发生的事情，如果值为 scroll，不论是否需要，用户代理都会提供一种滚动机制。因此，有可能即使元素框中可以放下所有内容也会出现滚动条。
- ❹ border-width border-style border-color
- ❺ clear 属性规定元素的哪一侧不允许其他浮动元素，both 在左右两侧均不允许浮动元素。

使用 max-width 替代 width 可以使浏览器更好地处理小窗口的情况。这点在移动设备上显得尤为重要。“响应式设计 (Responsive Design)” 是一种让网站针对不同的浏览器和设备“响应”不同显示效果的策略，这样可以让网站在任何情况下显示的很棒！

10.1.3 水平居中

把 margin 设为 auto 就是把要居中的元素的 margin-left 和 margin-right 都设为 auto，如下代码所示：

```
1 margin:0 auto;
```

此方法只能进行水平的居中，且对浮动元素或绝对定位元素无效。行级元素设置其父元素的 text-align center，块级元素设置其本身的 left 和 right margins 为 auto 即可。

使用 text-align:center 只能对图片，按钮，文字等行内元素 (display 为 inline 或 inline-block 等) 进行水平居中。但要说明的是在 IE6、7 这两个奇葩的浏览器中，它是能对任何元素进行水平居中的。

使用 line-height 让单行的文字垂直居中 把文字的 line-height 设为文字父容器的高度，适用于只有一行文字的情况。

10.1.4 宽高自适应

在 IE7+ 及 chrome、firefox 等浏览器中，高度自适应可以利用绝对定位来解决。但一个元素是绝对定位时，如果没有给它设定高度或宽度，则它的高度和宽度是由它的 top、right、bottom、left 属性决定的，但这一法则在 IE6 中并不适用，因此在 IE6 中还得另辟蹊径。在 IE6 中给 html 设定 padding，并不会撑大 html 元素的尺寸，这正是我们要利用的地方。

```
1 <style>
2   .headbanner {
3     position: absolute;
4     bottom: 0px;
5     top: 0px;①
6     left :0;
7     right:0;②
8     text-align: center;
9     max-width: 720px;
10    }
11
12   .headbanner img{
13     width: 100%;
14     height: 100%;③
15   }
16 </style>
```

```
17 <body>
18   <div class="headbanner">
19     
20   </div>
21 </body>
```

- ❶ bottom 和 top 设置高度自适应
- ❷ left 和 right 设置宽度自适应
- ❸ 高度的计算方式完全不一样。事实上，浏览器根本就不计算内容的高度，除非内容超出了视窗范围（导致滚动条出现）。或者你给整个页面设置一个绝对高度。否则，浏览器就会简单的让内容往下堆砌，页面的高度根本就无需考虑。想让一个元素的百分比高度起作用，你需要给这个元素的至少一个父元素的高度设定一个有效值。百分比是指其相对父块高度而定义的高度，也就是按照离它最近且有定义高度的父层的高度来定义高度。

10.1.5 竖直居中

先要设置 div 元素的祖先元素 html 和 body 的高度为 100%（因为他们默认是为 0 的），并且清除默认样式，即把 margin 和 padding 设置为 0(如果不清除默认样式的话，浏览器就会出现滚动条)。竖着居中的 CSS 代码如下所示：

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>index</title>
6     <style>
7       html,body {
8         width: 100%;
9         height: 100%;
10        margin: 0;
11        padding: 0;
12      }
13      .content {
14        width: 300px;
15        height: 300px;
16        background: orange;
17        margin: 0 auto; /*水平居中*/
18        position: relative; /*脱离文档流*/
19        top: 50%; /*偏移*/
20        margin-top: -150px;
21      }
22     </style>
23   </head>
24   <body>
25     <div class="content"></div>
```

```
26  </body>
27  </html>
```

10.1.6 CSS3 Media Queries

10.1.7 属性

display

一个常用的 display 值是 none。一些特殊元素的默认 display 值是它，例如 script。display:none 通常被 JavaScript 用来在不删除元素的情况下隐藏或显示元素。它和 visibility 属性不一样。把 display 设置成 none 不会保留元素本该显示的空间，但是 visibility: hidden; 还会保留。

z-index

用来确定定位元素在垂直于显示屏方向（以下称为 Z 轴）上的层叠顺序。要想给元素设置 z-index 样式，必须先让它变成定位元素，再通俗一点说，就是要给元素设置一个 position: relative(或者 absolute 或者 fixed) 样式。不要给想控制“上、下”的元素设置 z-index，而是对他们的父容器设置 z-index 样式。这一点详细解释和原因在《精通 html 和 css 设计模式》一书中较为详细的解释。

每个 box 都归属于一个 stacking context，它是元素在 z 轴方向上定位的参考。根元素形成 root stacking context，其他 stacking context 由定位元素设置 z-index 为非 auto 时产生。如 #div1{position:relative;z-index:0;} 即可使 id=div1 的元素产生 stacking context。stacking context 和 containing block 并没有必然联系。在一个 stacking context 中的每个 box，都有一个 stack level（即层叠级别，以下统一用 stack level），它决定着在同一 stacking context 中每个 box 在 z 轴上的显示顺序。同一 stacking context 中，stack level 值大的显示在上，stack level 值小的显示在下，同一 stack level 的遵循后来居上的原则（back-to-front）。不同 stacking context 中，元素显示顺序以父级的 stacking context 的 stack level 来决定显示的先后情况。于自身 stack level 无关。注意 stack level 和 z-index 并不是统一概念。

rel

rel 是 relationship 的英文缩写，rel 属性通常出现在 a,link 标签中，在引用 css 时，rel 不能省略，只有 rel 属性的“stylesheet”值得到了所有浏览器的支持，其他值只得到了部分地支持。

10.1.8 title

W3C 组织建议把所有网页上的对像都放在一个盒 (box) 中，设计师可以通过创建定义来控制这个盒的属性，这些对像包括段落、列表、标题、图片以及层。盒模型主要定义四个区域：内

容 (content)、边框距 (padding)、边界 (border) 和边距 (margin)。对于初学者，经常会搞不清楚 margin, background-color, background-image, padding, content, border 之间的层次、关系和相互影响。这里提供一张盒模型的 3D 示意图，希望便于你的理解和记忆。

10.1.9 css !important

css !important 作用是提高指定 CSS 样式规则的应用优先权。

10.2 AngularJS

AngularJS 是比较新的技术，版本 1.0 是在 2012 年发布的。AngularJS 是由 Google 的员工 Miško Hevery 从 2009 年开始着手开发。这是一个非常好的构想，该项目目前已由 Google 正式支持，有一个全职的开发团队继续开发和维护这个库。AngularJS 是一个 JavaScript 框架。它是一个以 JavaScript 编写的库。AngularJS is an MVC framework for building web applications. The core features include HTML enhanced with custom component and data-binding capabilities, dependency injection and strong focus on simplicity, testability, maintainability and boiler-plate reduction. 百度开源静态链接：<http://apps.bdimg.com/libs/angular.js/1.2.16/angular.min.js>。

10.3 线程

10.3.1 托管线程

10.4 扩展方法

Chapter 11

数据库

11.1 SQLite

11.1.1 SQLite 连接

The System.Data.Common namespace provides classes for creating DbProviderFactory instances to work with specific data sources. When you create a DbProviderFactory instance and pass it information about the data provider, the DbProviderFactory can determine the correct, strongly typed connection object to return based on the information it has been provided.

```
1 public SqliteOperateHelper(DbProviderFactory factory, string connectionString)
2 {
3     _dbConnection = factory.CreateConnection();
4     if (_dbConnection == null) return;
5     _dbConnection.ConnectionString = connectionString;
6     var cnnstring = factory.CreateConnectionStringBuilder();
7     if (cnnstring == null) return;
8     cnnstring.ConnectionString = connectionString;
9     _dbConnection.Open();
10 }
```

11.2 数据恢复

误删除了微信用户表中的数据。查看数据库的备份历史。

```
1 SELECT database_name,recovery_model,name,backup_start_date
2 FROM msdb.dbo.backupset
3 order by backup_start_date desc
```

Step 1 将微信用户数据 Pull 到测试库中。删除表中重复的记录。

```

1  /*删除tb_wxUser表中重复的记录（根据OpenID匹配）*/
2  delete from tb_wxUser
3  where openid in
4  (
5      select openid
6      from tb_wxUser
7      group by openid
8      having count(openid) > 1
9  )
10 and wxUserId not in
11 (
12     select top 1 wxUserId
13     from tb_wxUser
14     where openid =
15     (
16         select top 1 openid
17         from tb_wxUser
18         group by openid
19         having count(openid)>1
20     )
21     order by wxUserId desc
22 )

```

数据库的恢复模式设置为完整。

11.3 数据库还原

11.3.1 切换到单用户还原

在还原数据库时，有时会提示无法获取数据库的独占访问权，是因为其他用户在还原的时刻连接数据库，此时可将数据库设置为单用户模式。

```

1  --将数据库切换到单用户模式
2  alter database [zouwo-weixin] set single_user;
3
4  --将数据库切换到多用户模式
5  alter database [zouwo-weixin] set multi_user;

```

11.3.2 活动监视器

也可以打开 SQL Server 的活动监视器终止当前连接的用户再进行还原，此种方式比较暴力，不推荐此种方式。

11.3.3 数据泵 (Data Pump)

SQL Server 跨服务器不同数据库之间复制表的数据，需要用到链接服务器。

A linked server configuration enables SQL Server to execute commands against OLE DB data sources on remote servers. Linked servers offer the following advantages:

- Remote server access.
- The ability to issue distributed queries, updates, commands, and transactions on heterogeneous data sources across the enterprise.
- The ability to address diverse data sources similarly.

创建链接服务器可以通过 2 种方式。创建链接服务器在服务器安全对象下的链接服务器，右键创建链接服务器，链接服务器填写服务器 IP，非默认端口时填写端口。安全性选项卡中选择使用此上下文建立链接，输入登录用户名密码。

```

1  --添加远程链接服务器
2  exec sp_addlinkedserver
3      @server='LinkServer', --远程服务器别名
4      @srvproduct='',
5      @provider='SQLOLEDB',
6      @datasrc='139.28.81.128,5002'
7  go
8
9  --添加远程登录
10 exec sp_addlinkedsrvlogin 'LinkServer','false',null,'数据库登录名','数据库登录密码'
11 go
12
13 --以后不再使用时删除链接服务器
14 exec sp_dropserver 'LinkServer','droplogins'
15
16 --显示可用的服务器
17 exec sp_helpserver

```

sp_dropserver 存储过程为不再使用时删除链接服务器，往服务器上导入数据如下语句所示。

```

1  --将表 1 结构复制到表 2
2  select *
3  into [renren-weixin].[dbo].[tb_wxUser1]
4  from [renren-weixin].[dbo].[tb_wxUser]
5  where 1=0
6
7  --把本地表导入远程表，远程表必须已经存在，且远程表不应设置标识列或主键
8  insert openquery ([129.21.80.128,5002], ' SELECT * FROM [renren-weixin].[dbo].[tb_wxUser1] ' )

```

```

9  select * from [zouwo-weixin].[dbo].tb_wxUser
10
11  --将远程表导入本地表
12  select * into tb_awardsCopy from openrowset('SQLOLEDB ','129.21.80.128,5002';'
13      'renren-admin';'renren-admin!@#','select * from [renren-system].[dbo].['
14      'tb_awards]')
15
16  --用表 2 的记录更新表 1 (通过 openid 匹配, openid 唯一)
17  update tb_wxUser
18  set tb_wxUser.subscribe_time=tb_wxUser1.subscribe_time
19  from tb_wxUser1
20  where tb_wxUser.openid=tb_wxUser1.openid
21  and tb_wxUser.accountId=1
22
23  --根据最后一次关注时间更新微信用户表数据的添加时间
24  update tb_wxUser
25  set addDate =dateadd(s,cast(tb_wxUser.subscribe_time as bigint),'1970-01-01
26      00:00:00:000')
27  where addDate >'2015-09-17 15:09:09.000'

```

在将远程表导入本地表时，数据库应该不存在相同表名的表。将本地的 tb_wxUser 表数据导入到远程服务器的 tb_wxUser1 表中。链接服务器如图11.1所示。

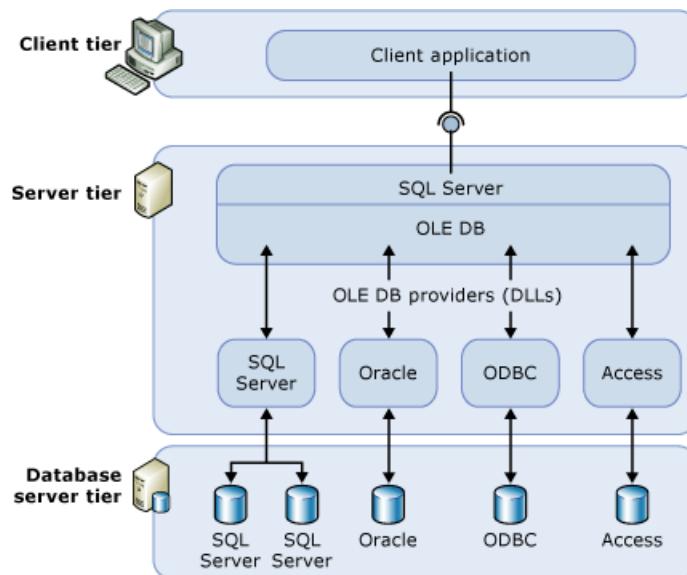


图 11.1: Link Servers

微信消息接口中的订阅时间（Subscribe Time）表示距离 1970 年的秒数。

11.3.4 常用 SQL

```

1  --修改列的字段类型长度
2  alter table tb_question alter column "question" nvarchar(50) null
3

```

```
4  --修改列名
5  exec sp_rename 'tb_question.[questionnaireType]', 'questionType', 'column'
6
7  --添加列
8
9  --数据表是否存在
10 if exists (
11 select *
12 from sysobjects
13 where id = object_id(N'tb_surveyAnalysisDetail'))
14 begin
15     drop table dbo.tb_surveyAnalysisDetail
16 end
17
18 --数据列是否存在（当列名是整型时，需要手动转换为字符类型，在添加四个双引号与列名匹配）
19 if not exists
20 (
21     select *
22     from syscolumns
23     where id=object_id('tb_surveyAnalysisDetail')
24     and name=''''+Convert(nvarchar(256),@questionId)+''''
25 )
26
27 --当运行是才知道列名时，动态指定列名
28 select @updateSql='update tb_surveyAnalysisDetail set '+@columnName+'='''+
    @answerContent+''' where userId=@userId'
29 exec sp_executesql @updateSql
30             ,N'@columnName nvarchar(256),@userId int'
31             ,@columnName=@columnName
32             ,@userId=@userid
33
34 --查询端口号
35 exec sys.sp_readerrorlog 0, 1, 'listening'
36
37 --将一个表（表结构 + 数据）复制到另一个表，跨库不跨 IP
38 select * into [renren-tour].[dbo].tour_seller
39 from [RR-CCN].[dbo].discount_seller
40
41 --查询 PV、UV
42 select count(distinct userId) as 'UV'①,
43 count(*) as PV
44 from discount_visitorLog
45 where siteType=3
46 and visitTime between '2015-11-10 00:00:00.000' and '2015-11-10 23:59:59.000'
47
48 --查看某一个数据库（zouwo-system）当前的连接数
49 select *
50 from [master].[dbo].[sysprocesses] where [dbid] in
51 (
52     select [dbid]
53     from [master].[dbo].[sysdatabases]
```

```

54     where name='zouwo-system'
55   )
56
57 --使数据库脱机 (还原数据库时)
58 alter database [database] set offline with rollback immediate

```

- ① 剔除掉重复的用户并统计出结果集的行数可用 count(distinct 列名) 的方式

Value of error log file you want to read: 0 = current, 1 = Archive #1, 2 = Archive #2, etc...
Log file type: 1 or NULL = error log, 2 = SQL Agent log Search string 1: String one you want to search for Search string 2: String two you want to search for to further refine the results

11.3.5 函数 (Function)

分割字符串

```

1  ALTER FUNCTION [dbo].[SplitString]
2  (
3      @Input NVARCHAR(MAX), --input string to be separated
4      @Separator NVARCHAR(MAX)=',', --a string that delimit the substrings in the
                                     input string
5      @RemoveEmptyEntries BIT=1 --the return value does not include array elements that
                                     contain an empty string
6  )
7  RETURNS @TABLE TABLE
8  (
9      [Id] INT IDENTITY(1,1),
10     [VALUE] NVARCHAR(MAX)
11  )
12 AS
13 BEGIN
14     DECLARE @Index INT, @Entry NVARCHAR(MAX)
15     SET @Index = CHARINDEX(@Separator,@Input)
16     WHILE (@Index>0)
17         BEGIN
18             SET @Entry=LTRIM(RTRIM(SUBSTRING(@Input, 1, @Index-1)))
19             IF (@RemoveEmptyEntries=0) OR (@RemoveEmptyEntries=1 AND @Entry<>'')
20                 BEGIN
21                     INSERT INTO @TABLE([VALUE]) VALUES(@Entry)
22                 END
23             SET @Input = SUBSTRING(@Input, @Index+DATALENGTH(@Separator)/2,
24                                     LEN(@Input))
25             SET @Index = CHARINDEX(@Separator, @Input)
26         END
27         SET @Entry=LTRIM(RTRIM(@Input))
28
29         IF (@RemoveEmptyEntries=0) OR (@RemoveEmptyEntries=1 AND @Entry<>'')
30             BEGIN

```

```

30      INSERT INTO @TABLE([VALUE]) VALUES(@Entry)
31      END
32      RETURN
33  END

```

11.3.6 存储过程 (Procedure)

11.4 mysql

11.4.1 常用查询

```

1 mysql --host=localhost --user=readmine_admin --password=123
2 SET PASSWORD FOR 'root'@'localhost' = PASSWORD('newpass');
3
4 #To see which user you are, and whose permissions you have
5 select user(), current_user();

```

11.5 定时任务

工作中经常忘记提交代码，可通过 Windows 自带的 at 命令设置定时任务，比如每天下午的 6 点自动进行代码提交。取消计划任务时，只需要用 at 命令查询出任务的 ID，根据 ID 取消即可，如果不带参数 ID，那么就是删除所有已知的任务。

```

1 #删除指定定时任务
2 at id /delete
3
4 #删除所有定时任务
5 at /delete

```

at 命令虽然可以执行定时任务，一直未尝试成功，可用 schtasks 命令替代。schtasks 命令每天运行的脚本如下：

```

1 #每天晚上的 8: 00 定时关机
2 at 20:00 /every:M,T,W,Th,F,S,Su shutdown -f -s -t 60
3
4 #23:44:00 定时运行脚本提交代码
5 schtasks /create /tn "Code Commit" /tr G:\CommitCode.bat /sc daily /st 23:44:00 /ed
   2100/12/31
6
7 #每隔一分钟运行同步脚本，注意脚本的路径文件夹名称最好不要有空格，可能导致任务无法启动
8 schtasks /create /sc minute /mo 1 /tn "Sync log" /tr "D:\AutoSync.bat"
9
10 #定时提醒提交代码

```

```

11 schtasks /create /tn CodeCommit /sc daily /st 09:12:00 /tr "mshta vbscript:msgbox(\"记得
    提交代码! \",64,\"定时提醒\")(window.close)"

12
13 #查询 schtasks 建立的任务列表时需更改 cmd 的语言为英文（输入命令: chcp 437）,
14 执行查询计划任务命令（命令: schtasks /query）后再改回简体中文（输入命令: chcp 936）

15
16 schtasks /Query

17
18 chcp 437 ①
19 schtasks /query /TN "auto update SVN" ②
20 schtasks /delete /TN "auto update SVN" ③

```

- ① 显示或设置活动代码页编号为英文 (change code page)
- ② 查询出作业名称为 “auto update SVN” 的作业， TN 代表 Task Name
- ③ 移除作业名称为 “auto update SVN” 的作业

脚本的含义为每天晚上的 23:44 运行批处理脚本 CommitCode.bat，直到 2100 年 12 月 31 日为止，任务的名称为 Code Commit。

11.6 刷新页面

```

1 history.go(0)

2 location.reload()

3 location=location

4 location.assign(location)

5 document.execCommand( 'Refresh ' )

6 window.navigate(location)

7 location.replace(location)

8 document.URL=location.href

```

Chapter 12

安全 (Security)

物理路径泄露，CGI 源代码泄露，目录遍历，执行任意命令，缓冲区溢出，拒绝服务，SQL 注入，条件竞争和跨站脚本执行漏洞。首先需要说明的是，入侵者的来源有两种，一种是内部人员利用自己的工作机会和权限来获取不应该获取的权限而进行的攻击。另一种是外部人员入侵，包括远程入侵、网络节点接入入侵等。进行网络攻击是一件系统性很强的工作，其主要工作流程是：收集情报，远程攻击，远程登录，取得普通用户的权限，取得超级用户的权限，留下后门，清除日志。主要内容包括目标分析，文档获取，破解密码，日志清除等技术，如图所示。



图 12.1: 常见 Web 攻击

12.1 Encrypt Algorithms

Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射，pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，而不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

HASH 主要用于信息安全领域中加密算法，它把一些不同长度的信息转化成杂乱的 128 位的编码，这些编码值叫做 HASH 值。也可以说，hash 就是找到一种数据内容和数据存放地址之间的映射关系。常见的 Hash 算法有：MD4、MD5、SHA-1、HAVAL 等。HAVAL is a cryptographic hash function. Unlike MD5, but like most modern cryptographic hash functions, HAVAL can produce hashes of different lengths. HAVAL can produce hashes in lengths of 128

bits, 160 bits, 192 bits, 224 bits, and 256 bits. HAVAL also allows users to specify the number of rounds (3, 4, or 5) to be used to generate the hash. HAVAL 目前还没有 C# 实现。

12.1.1 网站登录安全 (Website Login Security)

博客园登录的用户名密码采用了 JSEncrypt 库进行 RSA 加密，公钥通过 OpenSSL 生成后通过网页分发到客户端，用户名和密码经过 JSEncrypt 加密后发送到服务器。通过 OpenSSL 生成公钥和私钥对，生成公钥的命令如下：

```
1 openssl genrsa -out rsa_1024_priv.pem 1024
```

接下来根据公钥生成私钥，使用如下命令：

```
1 openssl rsa -pubout -in rsa_1024_priv.pem -out rsa_1024_pub.pem
```

私钥保存在服务器端，经过公钥加密后的数据通过私钥进行解密，在与数据库的用户名密码进行比对。注意在后端需要将 OpenSSL 生成的私钥转换为 XML 格式，再进行解密。Javascript 本身是经过明文加密，通过手段理论上可进行破解（研究中），还有就是采用安全控件（淘宝），控件的作用就是用二进制代码来隐藏加密的算法，防止密码在发送前被截获，不知道算法，也就很难破解原文。在不使用原文的情况下，使用密文来模拟用户的登录也是可以的，实际上单纯的加密也是无法满足要求的，所以就有了盐值，相同的密码多次生成的结果完全不同，非随机盐也会被摸出规律。所以就采用 HTTPS 对通讯进行加密，但是 SSL 也会爆出漏洞，病毒木马窥视内存同样存在，所以优秀的算法尽量不将明文保存在内存中，或者尽量在内存中停留极短时间。还有就是采用硬件加密，当然最安全的就是把电源和网线拔了，把电脑拆成零件锁到保险柜里面。

12.2 RESTful API 认证

所有的 REST API 调用必须运行在使用可信的 CA 签名过的证书的 HTTPS 之上。所有客户端与服务端交互之间必须要验证服务端证书。通过使用由可信机构签名的证书，SSL 可以保护你免受“中间人”攻击。中间人攻击的手段是在客户端和服务端之间插入一个代理进而窃听“加密的”通信。如果你不验证服务端的 SSL 证书，你就无法知道谁在接收你的 REST 查询请求。

所有的 REST API 调用应该通过专门的 API 密钥完成，该密钥由标识成分和共享密钥两部分组成。系统必须允许一个指定客户端拥有多个活动的 API 密钥并能方便地让个别密钥失效。前半部分的重点是发起 REST 请求的系统不应是某个交互用户……REST 认证的是程序而不是人，它支持比人使用的用户名/密码更强大的认证手段。后半部分的意思是，每个 REST 服务器应该支持每个客户端拥有多个 API 密钥。该需求使得孤立潜在危害和当危害发生时解决问题更为简单。当应用被破坏时，你也需要一种完善的方式铺开替换的 API 密钥。

所有的 REST 查询必须通过签名令牌签名的方式进行认证，该过程通过对按小写的字母顺序排序的查询参数使用私钥进行签名。签名应在查询字符串的 URL 编码前完成。换言之，你不

能将共享密钥作为查询串的一部分进行传递，而应使用它进行签名。签名后的查询串看起来应该是这样的：

```
1 GET /object?timestamp=1261496500&apiKey=Qwertystyle="border: 1px solid #ccc; padding: 5px;">2010&signature=abcdef0123456789
```

被签名的串是：

```
1 /object?apikey=Qwertystyle="border: 1px solid #ccc; padding: 5px;">2010&timestamp=1261496500
```

而签名是应用 API 密钥的私钥所得到的 HMAC-SHA256 哈希值。

12.2.1 Basic Authentication

Basic Authentication 需要和 HTTPS 一起使用，Http Basic 是一种比较简单的身份认证方式。在 Http header 中添加键值对 Authorization: Basic xxx (xxx 是 username:password base64 值)。而 Base64 的解码是非常方便的，如果不使用 HTTPS，相等于帐号密码直接暴露在请求中。

12.2.2 Api Key + Security Key + Sign

1. 用户登录返回一个 api_key 和 security_key 注意这里返回的两个 key 可以被劫持，所以为了进一步安全可以考虑下放 key 时采用非对称加密进行下放，服务端采用服务端的私钥进行签名，签名后用客户端的公钥进行加密，客户端接收后用客户端的私钥进行解密，服务端的公钥解密出发送的内容 (key)。
2. 然后客户端将 security_key 存在客户端
3. 当要发送请求之前，通过 function2 加密方法，把如图所示的五个值一起加密，得到一个 sign
4. 发送请求的时候，则将除去 security_key 之外的值，以及 sign 一起发送给服务器端
5. 服务器端首先验证时间戳是否有效，比如是服务器时间戳 5 分钟之前的请求视为无效
6. 然后根据 api_key 得到 security_key
7. 最后验证 sign

12.2.3 Amazon 的 API 鉴权流程

1. [客户端] 在调用 REST API 之前，首先将待发送消息体打包,combine a bunch of unique data together (Web Service 端将要接收的数据)

2. [客户端] 用系统分派的密钥使用哈希 (最好是 HMAC¹-SHA²1 or SHA256) 加密 (第一步的数据)

3. [客户端] 向服务器发送数据 发送生成的 HMAC 码.

发送消息体(属性名和属性值),如果是私有信息需要加密,像是("mode=start&number=4&order=desc"或其他不重要的信息)直接发送即可.

(可选项) 避免重放攻击 (Replay Attacks) 的唯一办法就是加上时间戳。在使用 HMAC(Hash-based Message Authentication Code) 算法时加入时间戳, 这样系统就能依据一定的条件去验证是否有重放的请求并拒绝.

用户身份认证信息例如, 用户 ID, 客户 ID 或是其他能别用户身份的信息。这是公共 API, 大家都能访问的到 (自然也包括了那些居心叵测的访问者) 系统仅仅需要这部分信息来区分发信人而不考虑可靠与否 (当然可以通过 HMAC 来判断可靠性).

4. [服务器端] 接收客户端发来的消息

5. [服务器端] (参看可选项) 检查接收时间和发送时间的间隔是否在允许范围内 (5-15 分) 以避免重放攻击 (Replay Attacks).

提示: 确保待检对象的时区无误 Daylight Savings Time

更新: 最近得到的结论就是直接使用 UTC 时区而无需考虑 DST 的问题 use UTC time.

6. [服务器端] 使用发送请求中用户信息 (比如 API 值) 从数据库检索出对应的私匙.

7. [服务器端] 跟客户端相同, 先将消息体打包然后用刚得到的私匙加密 (生成 HMAC) 消息体.

(参看可选项) 如果你使用了加入时间戳的方式避免重放攻击, 请确保服务端生成的加密信息中拥有和客户端相同的时间戳信息以避免中间人攻击 man-in-the-middle attack.

8. [服务器端] 就像在客户端一样, 使用 HMAC 哈希加密刚才的信息体.

9. [服务器端] 将服务器端刚生成的哈希与客户端的对比。如果一致, 则通讯继续; 否则, 拒绝请求!

HMAC 算法更象是一种加密算法, 它引入了密钥, 其安全性已经不完全依赖于所使用的 HASH 算法, 安全性主要是: 使用的密钥是双方事先约定的, 第三方不可能知道。作为非法截获信息的第三方, 能够得到的信息只有作为“挑战”的随机数和作为“响应”的 HMAC 结果, 无法根据这两个数据推算出密钥。由于不知道密钥, 所以无法仿造出一致的响应。

¹HMAC 是密钥相关的哈希运算消息认证码 (Hash-based Message Authentication Code) ,HMAC 运算利用哈希算法, 以一个密钥和一个消息为输入, 生成一个消息摘要作为输出。

²哈希安全算法 (Secure Hash Algorithm), 确切地说, 这不是一种算法, 而是一组加密哈希函数, 由美国国家标准技术研究所首先提出。无论是你的应用商店, 电子邮件和杀毒软件, 还是浏览器等等, 都使用这种算法来保证你正常下载, 以及是否被“中间人攻击”, 或者“网络钓鱼”。

12.3 数字签名 (Digital Signature)

12.3.1 概念

数字签名基于哈希算法和公钥加密算法，对明文报文先用哈希算法计算摘要，然后用私钥对摘要进行加密，得到的值就是原文的数字签名。数字签名（又称公钥数字签名、电子签章）是一种类似写在纸上的普通的物理签名，但是使用了公钥加密领域的技术实现，用于鉴别数字信息的方法。一套数字签名通常定义两种互补的运算，一个用于签名，另一个用于验证。

12.3.2 工作原理

数字签名的使用一般涉及以下几个步骤，我们通过安全电子邮件为案例进行介绍

- (1) 发件人生成或取得独一无二的加密密码组，包括私钥和公钥。
- (2) 发件人书写电子邮件。
- (3) 发件人用安全的摘要算法获取电子邮件的信息摘要。
- (4) 发件人再使用私钥对信息摘要进行加密，即可得到数字签名。
- (5) 发件人将数字签名附在信息之后。
- (6) 发件人将数字签名和信息（加密或未加密）发送给电子收件人。
- (7) 收件人使用发件人的公共密码（公钥）确认发件人的电子签名，即将发件人的数字签名通过公钥进行解密，得到信息摘要。
- (8) 收件人使用同样安全的摘要算法，获取信息（加密或未加密）的“信息摘要”。
- (9) 收件人比较两个信息摘要。假如两者相同，则收件人可以确信信息在签发后并未作任何改变。
- (10) 收件人从证明机构处获得认证证书（或者是通过信息发件人获得），这一证书用以确认发件人发出信息上的数字签名的真实性。证明机构在数字签名系统中是一个典型的受委托管理证明业务的第三方。该证书包含发件人的公共密码和姓名（以及其他可能的附加信息），由证明机构在其上进行数字签名。

其中，第(1) ~ (6)是数字签名的制作过程，(7) ~ (10)是数字签名的核实过程，整体流程如图12.2所示。

12.3.3 SHA1 算法

安全哈希算法 (Secure Hash Algorithm) 主要适用于数字签名标准 (Digital Signature Standard,DSS) 里面定义的数字签名算法 (Digital Signature Algorithm,DSA)。对于长度小于 264 位的消息，SHA1 会产生一个 160 位的消息摘要。当接收到消息的时候，这个消息摘要可以用来验证数据的完整性。在传输的过程中，数据很可能会发生变化，那么这时候就会产生不同

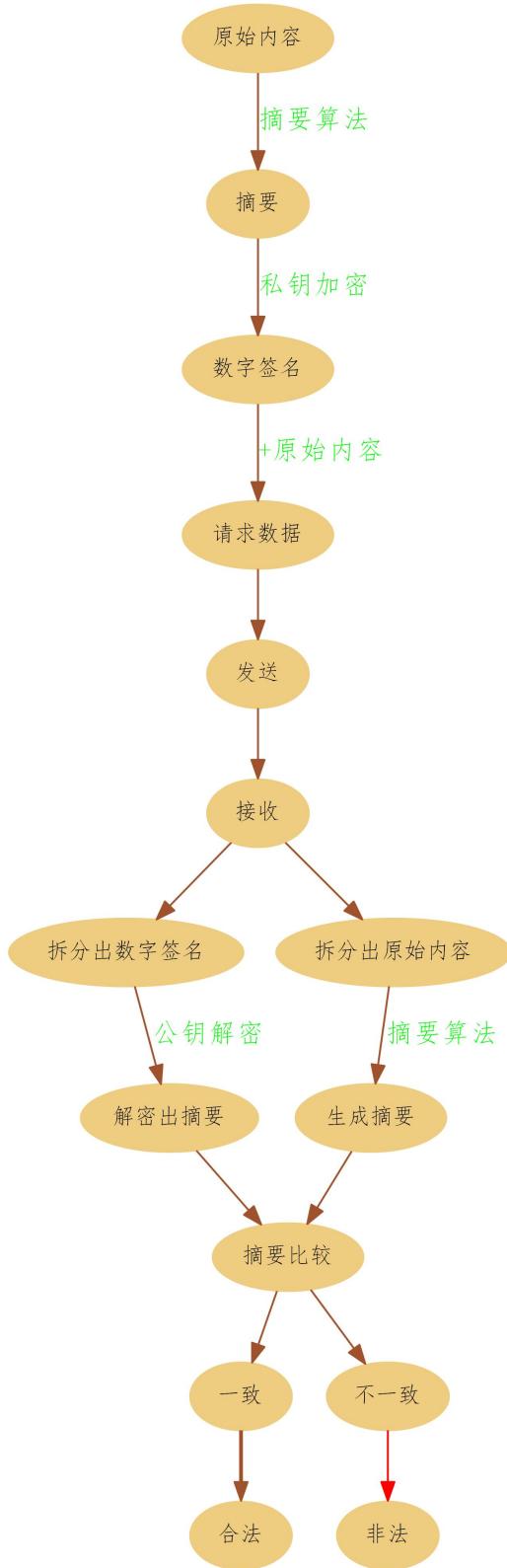


图 12.2: 电子签名流程

的消息摘要。SHA1 有如下特性：不可以从消息摘要中复原信息；两个不同的消息不会产生同样的消息摘要。加密性强的散列一定是不可逆的，这就意味着通过散列结果，无法推出任何部分的原始信息。

12.3.4 生成签名

代码片段12.1展示使用 SHA1 算法生成数字签名，注意生成数字签名尽量不选择 SHA1 算法，此算法被证明是不够安全的。

Listing 12.1: 采用 SHA1 算法生成数字签名

```

1  /// <summary>
2  /// 生成数字签名
3  /// </summary>
4  /// <param name="plaintext"> 原文 </param>
5  /// <param name="privateKey"> 私钥 </param>
6  /// <returns> 签名 </returns>
7  public static string HashAndSignString(string plaintext, string privateKey)
8  {
9      UnicodeEncoding ByteConverter = new UnicodeEncoding();
10     byte[] dataToEncrypt = ByteConverter.GetBytes(plaintext);
11     using (RSACryptoServiceProvider RSAalg = new RSACryptoServiceProvider())
12     {
13         RSAalg.FromXmlString(privateKey);
14         //使用 SHA1 进行摘要算法，生成签名
15         byte[] encryptedData = RSAalg.SignData(dataToEncrypt, new
16             SHA1CryptoServiceProvider());
17         return Convert.ToString(encryptedData);
18     }
}

```

用 ToBase64String 方法可以在不丢失数据的情况下将字节数组转成字符串。在 ToBase64String 方法中，会对字节数组中的连续三字节进行一次编码，编码得的字符串长度为 4 位，而且得出来的 4 位的字符串里面的字符肯定是由大小写字母、数字 (0~9)、+、/ 组成，例如有一个字节数组 212,36,25,23,45,65，ToBase64String 方法会将这个数组分成 2 个数组，分别为 212,36,25 和 23,45,65，212,36,25 计算出来的字符串是 “1CQZ”，而 23,45,65 是 “Fy1B”，如果是 212,36,25,23，则先分成两个数组，212,36,25 和 23，212,36,25 已经计算过了，但 23 不足三字节，怎么办？23 会转换成 “Fw==”，所以 212,36,25 和 23,45,65，212,36,25 转换出来的字符串是 “1CQZFy1B”，212,36,25,23 是 “1CQZFw==”。

12.3.5 验证签名

12.4 Cookie 窃取 (Cookie Hijacking)

12.4.1 XSS 漏洞

XSS 又称 CSS，全称 Cross SiteScript，跨站脚本攻击，是 Web 程序中常见的漏洞，XSS 属于被动式且用于客户端的攻击方式，所以容易被忽略其危害性。其原理是攻击者向有 XSS 漏洞的网站中输入（传入）恶意的 HTML 代码，当其它用户浏览该网站时，这段 HTML 代码会自动执行，从而达到攻击的目的。如，盗取用户 Cookie、破坏页面结构、重定向到其它网站等。

如何预防呢？原则：不相信客户输入的数据注意：攻击代码不一定在 `<script></script>` 中

- 将重要的 cookie 标记为 http only，这样的话 Javascript 中的 `document.cookie` 语句就不能获取到 cookie 了
- 只允许用户输入我们期望的数据。例如：年龄的 textbox 中，只允许用户输入数字
- 而数字之外的字符都过滤掉。对数据进行 Html Encode 处理过滤或移除特殊的 Html 标签，例如：`<script>`, `<iframe>`, `< for <`, `> for >`, `" for` 过滤 JavaScript 事件的标签。例如”`onclick=`”, ”`onfocus`” 等等

12.4.2 跨站请求伪造—CSRF

```

```

CSRF 主要是黑客将伪造的请求 URL 放到一个图片或者其他静态资源里，这种成本极低，且传播性和形象力非常大。举例：Qzone 的签名的修改地址是：`http://qzone.qq.com/cgi-bin/modify?nick=123`

12.4.3 Google Chrome Cookie 读取

Chrome 浏览器版本 33 以上对 Cookies 进行了加密，用 SQLite Developer 打开 Chrome 的 Cookies 文件就会发现，原来的 value 字段已经为空，取而代之的是加密的 `encrypted_value`。Chrome 浏览器已保存的密码都保存在一个 `sqlite3` 数据库文件中，和 Cookies 数据库在同一个文件夹。C# 中使用 `UnprotectData` 函数解密数据库中的密码字段，即可还原密码。

Listing 12.2: 解密 Cookie

```

1 byte[] sAdditionalEntropy = { };
2 var passwordValueArray = (byte[])reader[i];
3 //var decryptArray = DataProtection.UnprotectData(passwordValueArray);
4 var decryptArray = ProtectedData.Unprotect(passwordValueArray, sAdditionalEntropy,
    DataProtectionScope.LocalMachine);

```

```
5 var decryptResult = Encoding.Default.GetString(decryptArray);  
6 row[i] = decryptResult;
```

基于 ProtectedData 实现的加密解密字符串，可以跨 Windows 用户使用，但是不能跨计算机使用。所以在一台计算机上面加密的信息，不能在另一台计算机上面解密。ProtectedData 在 System.Security.dll 的命名空间 System.Security.Cryptography 下，This method can be used to encrypt data such as passwords, keys, or connection strings. The optionalEntropy parameter enables you to add data to increase the complexity of the encryption; specify null for no additional complexity. 另外如果网站选择了“记住密码”选项，Google Chrome 在再次访问此网站时，会在右上角显示一把钥匙图标，点击钥匙图标会进入 Google Chrome 的 Cookie 密码浏览界面。在这里也可以查看所有登录网站的密码，不过需要输入 Windows 的登录密码。也可以访问一下的链接进入 Cookie 密码管理界面：

```
1 chrome://settings/passwords
```

12.5 Session 劫持 (Session Hijacking)

咖啡馆的免费 WiFi，就是一个很理想的劫持环境，因为两个原因：

1. 这种 WiFi 通常不会加密，所以很容易监控所有流量
2. WiFi 通常使用 NAT 进行外网和内网的地址转换，所有内网客户端都共享一个外网地址。
这意味着，被劫持的 Session，看上去很像来自原来的登录者

12.6 HTTPS

12.6.1 TLS 历史

1994 年，NetScape 公司设计了 SSL 协议 (Secure Sockets Layer) 的 1.0 版，但是未发布。1995 年，NetScape 公司发布 SSL 2.0 版，很快发现有严重漏洞。1996 年，SSL 3.0 版问世，得到大规模应用。1999 年，互联网标准化组织 ISOC 接替 NetScape 公司，发布了 SSL 的升级版 TLS 1.0 版。2006 年和 2008 年，TLS 进行了两次升级，分别为 TLS 1.1 版和 TLS 1.2 版。最新的变动是 2011 年 TLS 1.2 的修订版。

目前，应用最广泛的是 TLS 1.0，接下来是 SSL 3.0。但是，主流浏览器都已经实现了 TLS 1.2 的支持。TLS 1.0 通常被标示为 SSL 3.1，TLS 1.1 为 SSL 3.2，TLS 1.2 为 SSL 3.3。SSL/TLS 协议的基本思路是采用公钥加密法，也就是说，客户端先向服务器端索要公钥，然后用公钥加密信息，服务器收到密文后，用自己的私钥解密。

12.7 密码哈希加盐 (Salted Password Hashing)

随机盐的优势是，即使对方知道算法，也很难搞，因为对应于每一份盐，都得花时间空间去生成对应的密码表（或者彩虹表），也就是说一个密码表只能破解一个用户的密码，这还是建立在已经得到数据库和加密算法的情况下，成本瞬间高了几个数量级。只有加密 hash 函数（cryptographic hash functions）可以用来进行密码的 hash。这样的函数有 SHA256, SHA512, RipeMD, WHIRLPOOL 等。如下代码片段 12.3 生成一个随机的盐值。

Listing 12.3: 生成随机盐

```

1  /// <summary>
2  /// Creates a salted PBKDF2 hash of the password.
3  /// </summary>
4  /// <param name="password">The password to hash.</param>
5  /// <returns>The hash of the password.</returns>
6  public static string CreateHash(string password)
7  {
8      // Generate a random salt
9      RNGCryptoServiceProvider csprng = new RNGCryptoServiceProvider();❶
10     byte[] salt = new byte[SALT_BYTES];
11     csprng.GetBytes(salt);❷
12
13     // Hash the password and encode the parameters
14     byte[] hash = PBKDF2(password, salt, PBKDF2_ITERATIONS, HASH_BYTES);
15     return "sha1:" + PBKDF2_ITERATIONS + ":" +
16         Convert.ToString(salt) + ":" +
17         Convert.ToString(hash);
18 }

```

❶ RNG 为 Random Number Generator 的缩写，生成随机数还可以用 Random 类，不过还不够随机，Pseudo-random numbers are chosen with equal probability from a finite set of numbers. The chosen numbers are not completely random because a definite mathematical algorithm is used to select them, but they are sufficiently random for practical purposes. The current implementation of the Random class is based on a modified version of Donald E. Knuth's subtractive random number generator algorithm. For more information, see D. E. Knuth. "The Art of Computer Programming, volume 2: Seminumerical Algorithms". Addison-Wesley, Reading, MA, second edition, 1981.

❷ 此处即完成了生成一个随机盐，SALT_BYTES 为随机盐的长度

加盐可以让攻击者无法使用查表和彩虹表（Rainbow Tables）的方式对大量 hash 进行破解。但是依然无法避免对单个 hash 的字典和暴力攻击。高端的显卡（GPU）和一些定制的硬件每秒可以计算数十亿的 hash，所以针对单个 hash 的攻击依然有效。为了避免字典和暴力攻击，我们可以采用一种称为 key 扩展（key stretching）的技术。思路就是让 hash 的过程变得非常缓慢，即使使用高速 GPU 和特定的硬件，字典和暴力破解的速度也慢到没有实用价值。通过减慢

hash 的过程来防御攻击，但是 hash 速度依然可以保证用户使用的时候没有明显的延迟。代码片段12.4使用 PBKDF2 算法由随机盐生成密码的 Hash。

Listing 12.4: 根据随机盐和密码生成 Hash

```

1  /// <summary>
2  /// Computes the PBKDF2-SHA1 hash of a password.
3  /// </summary>
4  /// <param name="password">The password to hash.</param>
5  /// <param name="salt">The salt.</param>
6  /// <param name="iterations">The PBKDF2 iteration count.</param>
7  /// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
8  /// <returns>A hash of the password.</returns>
9  private static byte[] PBKDF2(string password, byte[] salt, int iterations, int outputBytes)
10 {
11     Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(password, salt);
12     pbkdf2.IterationCount = iterations;①
13     return pbkdf2.GetBytes(outputBytes);②
14 }
```

- ① 定义生成 hash 的迭代次数，这个值决定了 hash 的过程具体有多慢，防止高速 GPU 的暴力破解
- ② PBKDF2 为 Password-Based Key Derivation Function 2 的缩写，假如攻击一个密码所需的 Rainbow Table 有 1 千万条，建立所对应的 Rainbow Table 所需要的时间就是 115 天。这个代价足以让大部分的攻击者忘而生畏。美国政府机构已经将这个方法标准化，并且用于一些政府和军方的系统。

12.7.1 Base64 编码

Base64 是网络上最常见的用于传输 8Bit 字节代码的编码方式之一，大家可以查看 RFC2045 ~ RFC2049，上面有 MIME 的详细规范。Base64 编码可用于在 HTTP 环境下传递较长的标识信息。例如，在 Java Persistence 系统 Hibernate 中，就采用了 Base64 来将一个较长的唯一标识符（一般为 128-bit 的 UUID）编码为一个字符串，用作 HTTP 表单和 HTTP GET URL 中的参数。在其他应用程序中，也常常需要把二进制数据编码为适合放在 URL（包括隐藏表单域）中的形式。此时，采用 Base64 编码具有不可读性，即所编码的数据不会被人用肉眼所直接看到。Base64 在 C# 中的编码解码如下代码片段所示。

```

1 //编码
2 byte[] bytes = Encoding.Default.GetBytes("要转换的字符");
3 string str = Convert.ToBase64String(bytes);
4 //解码
5 byte[] outputByte = Convert.FromBase64String(str);
6 string originalStr = Encoding.Default.GetString(outputByte);
```

某些 RESTful 接口的授权信息采用 Base64 编码的方式，将验证信息经过 Base64 加密后存放在 Http 请求头中，在 Fiddler 中查看请求头中的授权信息如图12.3所示。

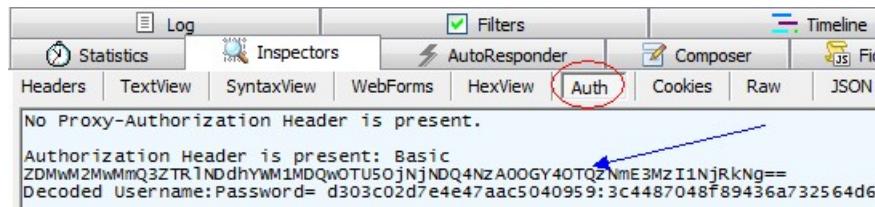


图 12.3: Http 请求头中的 Base64 加密验证信息

12.8 SQL 注入 (SQL Injection)

SQL Injection 为网络安全中漏洞中，发生次数最多的漏洞。主要是在输入 SQL 语法中，插入恶意的数据库查询语法。采用完全参数化的设计，过滤使用者输入的内容。

12.8.1 参数化 SQL 语句

正则表达校验用户输入

参数化 SQL 语句

替换特殊字符

12.8.2 Sqlmap

12.9 Nmap(GNU General Public License)

Nmap(Network Mapper)，网络映射器，是一款开放源代码的网络探测和安全审核的工具。它的设计目标是快速地扫描大型网络，当然用它扫描单个主机也没有问题。Nmap 以新颖的方式使用原始 IP 报文来发现网络上有哪些主机，那些主机提供什么服务 (应用程序名和版本)，那些服务运行在什么操作系统 (包括版本信息)，它们使用什么类型的报文过滤器/防火墙，以及一堆其它功能。虽然 Nmap 通常用于安全审核，许多系统管理员和网络管理员也用它来做一些日常的工作，比如查看整个网络的信息，管理服务升级计划，以及监视主机和服务的运行。Nmap 脚本引擎 (Nmap Script Engine) 是 Nmap 最有力灵活的一个特性。它允许用户撰写和分享一些简单的脚本来一些较大的网络进行扫描任务。基本上这些脚本是用 Lua 编程语言来完成的。通常 Nmap 的脚本引擎可以完成很多事情。

12.9.1 主机发现 (Host Discovery)

主机发现顾名思义就是发现所要扫描的主机是否是正在运行的状态。输入如下命令开始主机扫描

```
1 nmap -F -sT -v nmap.org
2 # 主机发现，生成存活主机列表
```

```

3 nmap -sn -T4 -oG Discovery.gnmap 192.168.24.0/24
4 grep "Status: Up" Discovery.gnmap | cut -f 2 -d ' ' > LiveHosts.txt

```

- -F 扫描 100 个最有可能开放的端口,F 表示快速模式 (Fast Model), 比默认模式扫描更少的端口 (Scan fewer ports than the default scan), 如果需要扫描更多的端口, 指定 p 参数, 如下命令行片段所示:

```
1 nmap -p 0-10000 -A -v <Host>
```

端口号最好分段指定, 否则扫描的速度会非常缓慢, 例如需要扫描 0-65535 端口, 可以先扫描 0-10000 端口, 再扫描 10001-20000, 依次类推。

- -v 获取扫描的信息
- -sT(Scan using TCP) 采用的是 TCP 扫描, 不写也是可以的, 默认采用的就是 TCP 扫描
- -sn 参数表示 ping 扫描, 禁用端口扫描
- -oG 表示输出 Grepable 格式

Nmap 输出的是扫描目标的列表, 以及每个目标的补充信息, 至于哪些信息则依赖于所使用的选项。“所感兴趣的端口表格”是其中的关键。那张表列出端口号, 协议, 服务名称和状态。状态可能是 open(开放的), filtered(被过滤的), closed(关闭的), 或者 unfiltered(未被过滤的)。Open(开放的)意味着目标机器上的应用程序正在该端口监听连接/报文。filtered(被过滤的)意味着防火墙, 过滤器或者其它网络障碍阻止了该端口被访问, Nmap 无法得知它是 open(开放的)还是 closed(关闭的)。closed(关闭的)端口没有应用程序在它上面监听, 但是他们随时可能开放。当端口对 Nmap 的探测做出响应, 但是 Nmap 无法确定它们是关闭还是开放时, 这些端口就被认为是 unfiltered(未被过滤的)如果 Nmap 报告状态组合 open|filtered 和 closed|filtered 时, 那说明 Nmap 无法确定该端口处于两个状态中的哪一个状态。当要求进行版本探测时, 端口表也可以包含软件的版本信息。当要求进行 IP 协议扫描时 (-sO), Nmap 提供关于所支持的 IP 协议而不是正在监听的端口的信息。获取远程主机的系统类型及开放端口:

```
1 nmap -sS -P0 -sV -O <target>
```

12.9.2 端口扫描 (Port Scanning)

端口扫描是 Nmap 最基本最核心的功能, 用于确定目标主机的 TCP/UDP 端口的开放情况。端口扫描简单示例如下代码片段所示:

```

1 #服务扫描
2 nmap -T4 -sV targetip
3
4 #扫描淘宝 IP
5 nmap -p 0-1000 -v 218.201.46.124
6 nmap -T4 -A -v 218.201.46.124

```

通过扫描可以发现一些主机信息，比如淘宝的数字证书用的是 GlobalSign，加密位数是 2048，签名算法是 sha256WithRSAEncryption，操作系统猜测是 Linux 系列的，内核版本可能是 3.X 或者 2.6.X，具体信息如下。

```

1 443/tcp open ssl/http Tengine httpd
2 |_http-server-header: Tengine
3 |_http-title: 501 Not Implemented
4 | ssl-cert: Subject: commonName=*.tmall.com/organizationName=Alibaba (China)
   Technology Co., Ltd./stateOrProvinceName=ZheJiang/countryName=CN
5 | Issuer: commonName=GlobalSign Organization Validation CA - SHA256 - G2/
   organizationName=GlobalSign nv-sa/countryName=BE
6 | Public Key type: rsa
7 | Public Key bits: 2048
8 | Signature Algorithm: sha256WithRSAEncryption
9 | Not valid before: 2015-12-14T10:38:38
10 | Not valid after: 2016-12-14T10:38:38
11 | MD5: 5d67 3ca3 7f0e 2ab7 7b4f 59d5 0700 7223
12 |_SHA-1: d048 e9cf 5487 5030 9f26 e638 7d3f 94ad a2b3 e6fa

```

默认情况下，Nmap 会扫描 1000 个最有可能开放的 TCP 端口。

1) 公认端口 (0 ~ 1023)，又称常用端口，为已经公认定义或为将要公认定义的软件保留的。这些端口紧密绑定一些服务且明确表示了某种服务协议。如 80 端口表示 HTTP 协议。2) 注册端口 (1024 ~ 49151)，又称保留端口，这些端口松散绑定一些服务。3) 动态/私有端口 (49152 ~ 65535)。理论上不应为服务器分配这些端口。按协议类型可以将端口划分为 TCP 和 UDP 端口。1) TCP 端口是指传输控制协议端口，需要在客户端和服务器之间建立连接，提供可靠的数据传输。如 Telnet 服务的 23 端口。2) UDP 端口是指用户数据包协议端口，不需要在客户端和服务器之间建立连接。常见的端口有 DNS 服务的 53 端口。有的服务器为了安全会修改默认端口，比如将 ssh 默认的 22 端口修改为 50000，不过通过 Nmap 扫描也会很容易的发现。详细的扫描指令如下所示：

```

1 nmap -sS -sU -T4 -A -v -PE -PP -PS80,443 -PA3389 -PU40125 -PY -g 53 --
      script "default or (discovery and safe)" 12.26.32.14

```

- -sS 参数 (TCP SYN Scan) 可以利用基本的 SYN 扫描方式探测其端口开放状态
- -sU 参数代表 UDP Scan
- -T<0-5>: Set timing template (higher is faster)
- -A: Enable OS detection, version detection, script scanning, and traceroute
- -v: Increase verbosity level (use -vv or more for greater effect)
- -PE/PP/PM: ICMP echo, timestamp, and netmask request discovery probes
- -PS/PA/PU/PY[portlist]: TCP SYN/ACK, UDP or SCTP discovery to given ports
- -g/-source-port <portnum>: Use given port number

”ppp0” is not an ethernet device 在使用 nmap 扫描时候，提示

¹ Only ethernet devices can be used for raw scans on Windows, and "ppp0" is not an ethernet device. Use the --unprivileged option for this scan. QUITTING!

因为用 Nmap 扫描时，不能用 pppoe 的拨号口，只能用以太网口。根据提示带参数--unprivileged 就可以了，注意扫描的命令不能带-A 参数，否则会要求 root 权限，无法启动扫描。

12.9.3 版本侦测 (Version Detection)

版本侦测在后面添加 sV(Service/Version) 参数即可。

¹ -sV: Probe open ports to determine service/version info

12.10 AppScan

IBM AppScan 该产品是一个领先的 Web 应用安全测试工具，曾以 Watchfire AppScan 的名称享誉业界。Rational AppScan 可自动化 Web 应用的安全漏洞评估工作，能扫描和检测所有常见的 Web 应用安全漏洞，例如 SQL 注入 (SQL-injection)、跨站点脚本攻击 (cross-site scripting)、缓冲区溢出 (buffer overflow) 及最新的 Flash/Flex 应用及 Web 2.0 应用曝露等方面安全漏洞的扫描。

12.10.1 操作系统侦测 (Operating System Detection)

12.11 OpenVPN

12.11.1 OpenVPN 配置 (OpenVPN Configuration)

OpenVPN 的配置可以参考 OpenVPN bible³，Step by Step 可以参考 CodePlayer 上的分享⁴。在服务器上生成所有的配置（包括客户端），将配置拷贝到客户端。

TLS key negotiation failed to occur within 60 seconds (check your network connectivity) One of the most common problems in setting up OpenVPN is that the two OpenVPN daemons on either side of the connection are unable to establish a TCP or UDP connection with each other. This is almost a result of:

³ <https://openvpn.net/index.php/open-source/documentation/howto.html>

⁴ <http://www.365mini.com/page/14.htm>

- A perimeter firewall on the server's network is filtering out incoming OpenVPN packets (by default OpenVPN uses UDP or TCP port number 1194).
- A software firewall running on the OpenVPN server machine itself is filtering incoming connections on port 1194. Be aware that many OSes will block incoming connections by default, unless configured otherwise.
- A NAT gateway on the server's network does not have a port forward rule for TCP/UDP 1194 to the internal address of the OpenVPN server machine.
- The OpenVPN client config does not have the correct server address in its config file. The remote directive in the client config file must point to either the server itself or the public IP address of the server network's gateway.
- Another possible cause is that the windows firewall is blocking access for the openvpn.exe binary. You may need to whitelist (add it to the "Exceptions" list) it for OpenVPN to work.
- 注意所用的是 TCP 连接还是 UDP 连接，如果是采用 UDP 连接，子网上的 NAT 网关应该有一条端口转发规则：forward UDP port 1194 from my public IP address to 192.168.X.X

查看 OpenVPN 的监听端口：

```

1 #netstat 发现计算机上的监听或开放端口
2 netstat -an |find /i "listening"
3
4 netstat -an |find /i "1194"
5
6 #查看一下计算机到底通过哪些端口通信
7 netstat -an |find /i "established"

```

OpenVPN 的配置认证文件的详细解释如下表所示。

File Name	Needed By	Purpose	Secret
ca.crt	server + all clients	Root CA certificate	NO
ca.key	key signing machine only	Root CA key	YES
dhn.pem	Server Only	Diffie Hellman parameters	NO

12.11.2 添加新客户端 (Add New Client)

OpenVPN 多个客户端不能共享一个密钥和证书（严格来说可以共享，但是推荐用单独的证书），共享证书会导致不同客户端连接时获取的 IP 相同，建议在添加新的客户端时需生成新的密钥和证书。切换到服务端的 OpenVPN 的 easy-rsa 目录下，执行如下命令添加新的证书。

```

1 #设置相应的局部环境变量，就是我们在 vars.bat.sample 文件中设置的内容
2 vars
3

```

```

4 # 创建客户端证书
5 build-key client-aoxianghua

```

证书创建成功后，将证书拷贝到客户端 config 目录下，在客户端配置文件中指定证书名称。

```

1 # SSL/TLS parms.
2 # See the server config file for more
3 # description. It's best to use
4 # a separate .crt/.key file pair
5 # for each client. A single ca
6 # file can be used for all clients.
7 ca ca.crt
8 cert client-aoxianghua.crt
9 key client-aoxianghua.key

```

12.12 Windows 自带 VPN

使用 Windows 自带 VPN，省去了安装客户端和服务端的麻烦，非常方便。

12.12.1 VPN 类型

PPTP (Point to Point Tunneling Protocol, 点对点隧道协议)

默认端口号：1723，即 PPTF 协议。该协议是在 PPP 协议的基础上开发的一种新的增强型安全协议，支持多协议虚拟专用网（VPN），可以通过密码身份验证协议（PAP）、可扩展身份验证协议（EAP）等方法增强安全性。可以使远程用户通过拨入 ISP、通过直接连接 Internet 或其他网络安全地访问企业网。点对点隧道协议（PPTP）是一种支持多协议虚拟专用网络的网络技术，它工作在第二层。通过该协议，远程用户能够通过 Microsoft Windows NT 工作站、Windows xp、Windows 2000 和 windows2003、windows7 操作系统以及其它装有点对点协议的系统安全访问公司网络，并能拨号连入本地 ISP，通过 Internet 安全链接到公司网络。PPTP 协议是点对点隧道协议，其将控制包与数据包分开，控制包采用 TCP 控制。PPTP 使用 TCP 协议，适合在没有防火墙限制的网络中使用。

L2TP (Layer 2 Tunneling Protocol, 第二层隧道协议)

默认端口号：1701，L2TP 是一种工业标准的 Internet 隧道协议，功能大致和 PPTP 协议类似，比如同样可以对网络数据流进行加密。不过也有不同之处，比如 PPTP 要求网络为 IP 网络，L2TP 要求面向数据包的点对点连接；PPTP 使用单一隧道，L2TP 使用多隧道；L2TP 提供包头压缩、隧道验证，而 PPTP 不支持。

L2TP 是一个数据链路层协议，基于 UDP。其报文分为数据消息和控制消息两类。数据消息用投递 PPP 帧，该帧作为 L2TP 报文的数据区。L2TP 不保证数据消息的可靠投递，若数据

报文丢失，不予重传，不支持对数据消息的流量控制和拥塞控制。控制消息用以建立、维护和终止控制连接及会话，L2TP 确保其可靠投递，并支持对控制消息的流量控制和拥塞控制。

L2TP 是国际标准隧道协议，它结合了 PPTP 协议以及第二层转发 L2F 协议的优点，能以隧道方式使 PPP 包通过各种网络协议，包括 ATM、SONET 和帧中继。但是 L2TP 没有任何加密措施，更多是和 IPSec 协议结合使用，提供隧道验证。

L2TP 使用 UDP 协议，一般可以穿透防火墙，适合在有防火墙限制、局域网用户，如公司、网吧、学校等场合使用。

PPTP 和 L2TP 二个连接类型在性能上差别不大，如果使用 PPTP 不正常，那就更换为 L2TP。

OpenVPN

默认端口号：1194，OpenVpn 的技术核心是虚拟网卡，其次是 SSL 协议实现。虚拟网卡是使用网络底层编程技术实现的一个驱动软件，安装后在主机上多出现一个网卡，可以像其它网卡一样进行配置。服务程序可以在应用层打开虚拟网卡，如果应用软件（如 IE）向虚拟网卡发送数据，则服务程序可以读取到该数据，如果服务程序写合适的数据到虚拟网卡，应用软件也可以接收得到。虚拟网卡在很多的操作系统下都有相应的实现，这也是 OpenVpn 能够跨平台一个很重要的理由。

OpenVPN 使用 OpenSSL 库加密数据与控制信息：它使用了 OpenSSL 的加密以及验证功能，意味着，它能够使用任何 OpenSSL 支持的算法。它提供了可选的数据包 HMAC 功能以提高连接的安全性。此外，OpenSSL 的硬件加速也能提高它的性能。

OpenVPN 所有的通信都基于一个单一的 IP 端口，默认且推荐使用 UDP 协议通讯，同时 TCP 也被支持。

在选择协议时候，需要注意 2 个加密隧道之间的网络状况，如有高延迟或者丢包较多的情况下，请选择 TCP 协议作为底层协议，UDP 协议由于存在无连接和重传机制，导致要隧道上层的协议进行重传，效率非常低下。

OpenVPN 是一个基于 SSL 加密的纯应用层 VPN 协议，是 SSL VPN 的一种，支持 UDP 与 TCP 两种方式（说明：UDP 和 TCP 是 2 种通讯协议，这里通常 UDP 的效率会比较高，速度也相对较快。所以尽量使用 UDP 连接方式，实在 UDP 没法使用的时候，再使用 TCP 连接方式）。

由于其运行在纯应用层，避免了 PPTP 和 L2TP 在某些 NAT 设备后面不被支持的情况，并且可以绕过一些网络的封锁（通俗点讲，基本上能上网的地方就能用 OpenVPN）。

OpenVPN 客户端软件可以很方便地配合路由表，实现不同线路（如国内和国外）的路由选择，实现一部分 IP 走 VPN，另一部分 IP 走原网络。

易用性：PPTP > L2TP > OpenVPN

速度：PPTP > OpenVPN UDP > L2TP > OpenVPN TCP

安全性: OpenVPN > L2TP > PPTP

稳定性: OpenVPN > L2TP > PPTP

网络适用性: OpenVPN > PPTP > L2TP

电脑上优先使用 PPTP, 无法使用可以尝试 L2TP, 对安全性要求高的优先使用 OpenVPN。
手持设备推荐使用 L2TP。

PPTP: 最常用, 设置最简单, 大多数设备都支持; L2TP: 支持 PPTP 的设备基本都支持此种方式, 设置略复杂, 需要选择 L2TP/IPSec PSK 方式, 且设置预共享密钥 PSK; OpenVPN: 最稳定, 适用于各种网络环境, 但需要安装第三方软件和配置文件, 较复杂。

SSTP: 安全套接字隧道协议

SSTP 可以创建一个在 HTTPS 上传送的 VPN 隧道, 从而消除与基于 PPTP (点对点隧道协议) 或 L2TP (第 2 层隧道协议) VPN 连接有关的诸多问题。因为这些协议有可能受到某些位于客户端与服务器之间的 Web 代理、防火墙和网络地址转换 (NAT) 路由器的阻拦。这种 SSTP 只适用于远程访问, 不能支持站点与站点之间的 VPN 隧道。微软公司希望, 当 IPSec VPN 连接受到防火墙或路由器的阻拦后, SSTP 可以帮助客户减少与 IPSec VPN 有关的问题。此外, SSTP 也不会产生保留的问题, 因为它不会改变最终用户的 VPN 控制权。基于 VPN 隧道的 SSTP 可直接插入当前的微软 VPN 客户端和服务器软件的接口中。

IKEv2(Internet Key Exchange v2)

IKEv2(Internet Key Exchange v2) 是一款新版的安全协议。简单理解的话可以说和大家常见的 PPTP/L2TP 是差不多一类的东西。不过 IKEv2 协议要比 PPTP 和 L2TP 更加优秀, 将会是之后发展的方向之一。二、IKEV2 有什么优势? Quote: IKEv2 连接在用户自身网络状况经常变化的情况下仍旧能够维持加密连接, 而不会出现频繁闪断、断开又重连之类的情况, 能大大提高网络连接的稳定性。IKEv2 能够拥有比 PPTP/L2TP 更加高效的网络通讯效率。IKEv2 是 IKE (v1) 的改良版, 使用了公钥证书和密码等多重认证, 弥补了 v1 时代的安全性上的不足。比 PPTP/L2TP 更是可靠许多。同时 IKEv2 还支持硬件加速, 保证了高效的传输效率。三、IKEV2 的局限: Quote: 优势和局限总是相对的, 新技术面临的问题就是各方面的支持性。现阶段提供 IKEv2 协议连接的 VPN 商家并不多, 或者也可以说很少, 选择余地也很小。对于用户来说 Windows 7 以上包括 RT 系统都对 IKEv2 有原生支持, Android 需要第三方软件才可以支持, iOS 仅支持 IKEv1, 而 Windows Phone 干脆仅支持 IKEv2 协议的 VPN 连接。

IKE(v1) 传输效率大大超于 PPTP 和 L2TP, 但仍然存在某种安全隐患。IKEv2 是 IKE(v1) 的全新改良版, 使用公钥证书和密码等多重认证, 支持硬件加速, 保持了高效的传输效率。如果出门在外, (手机) 网络环境 (IP 地址) 经常变动的情况下, IKEv2 是唯一使用 “MOBIKE” – Mobility and Multihoming 技术来达到加密通讯不中断目的的。所以用户再也无需考虑手动开启或者关闭 VPN, 或者担心因网络环境变化而导致的 VPN 连接中断了。

12.12.2 常见错误

Error 800: 未建立远程链接，因为尝试的 VPN 隧道失败。VPN 服务器可能无法访问。如果该连接尝试使用 L2TP/IPSec 隧道，则 IPSec 协议所需的安全参数可能配置错误。由于 Windows2000/XP/2003 系统缺省情况下启动了 IPSec 功能，因此在发起 VPN 请求时应禁止 IPSec 功能，为什么要如此还无从知晓，此时需要更改注册表，在 cmd 中执行 regedit，找到如下路径

```
1 [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\RasMan\Parameters]
```

新建一个键值，名称 ProhibitIPSec，类型 dword，值 1。或者执行如下脚本：

```
1 Windows Registry Editor Version 5.00
2 [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\RasMan\Parameters]
3 "ProhibitIpSec"=dword:00000001
```

键值修改成功后重新启动计算机即可成功连接 VPN。

Error 809 VPN 类型选择自动。

Error 720: Unable to establish a connection to the remote computer. Might need to change the network settings for this connection 出现此错误原因不明，解决此错误需要在服务端配置 VPN 连接的 IP 范围，Open Network and Sharing Center and click Change Adapter Settings. Find Incoming Connection and right-click on it and click Properties. 具体配置如图12.4所示。

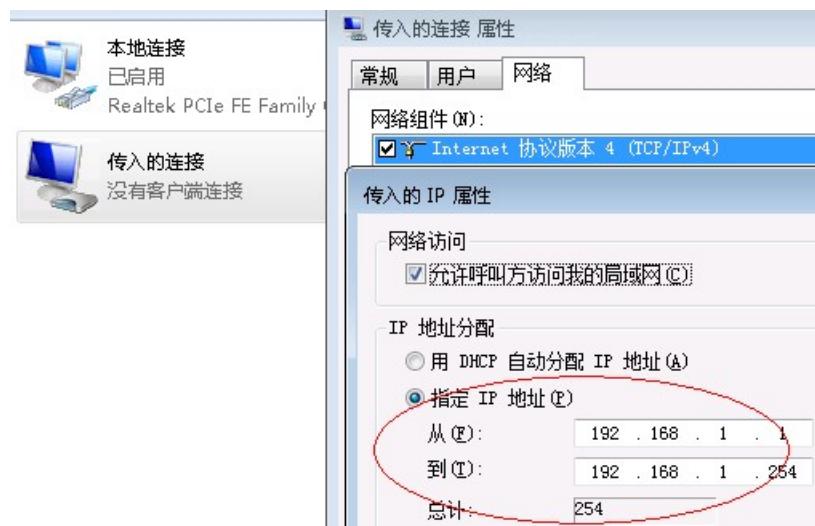


图 12.4: 配置 VPN 客户端连接的 IP 范围

如果 VPN 服务端在局域网中，在传入的 IP 属性中指定 IP 地址段时从 192.168.1.2 开始（图中的 IP 需要修改），因为 192.168.1.1 为路由器的地址，从 192.168.1.1 开始会导致 VPN 服

务器会分配到路由器的默认 IP，导致路由器的部分功能不可用。

12.13 DIRB

DIRB is a Web Content Scanner. It looks for existing (and/or hidden) Web Objects. It basically works by launching a dictionary based attack against a web server and analizing the response.

DIRB comes with a set of preconfigured attack wordlists for easy usage but you can use your custom wordlists. Also DIRB sometimes can be used as a classic CGI scanner, but remember is a content scanner not a vulnerability scanner.

DIRB main purpose is to help in professional web application auditing. Specially in security related testing. It covers some holes not covered by classic web vulnerability scanners. DIRB looks for specific web objects that other generic CGI scanners can't look for. It doesn't search vulnerabilities nor does it look for web contents that can be vulnerables.

12.14 Network

12.14.1 Ubuntu 使用 aircrack 破解 WiFi

破解原理：直接把这段握手包给截取下来，在本地验证好了再发给热点就行。无线路由器的另一种加密方式为 WPA/WPA2 加密，相比较于 WEP 加密，暂时未能从一些公开的方法中找到直接破解 WPA/WPA2 密码的方法，只能先抓获握手包，然后对握手包进行暴力破解，但是结合着一些技巧以及普通用户密码策略比较薄弱，成功的机率也比较高。抓握手包跑字典，这主要看你的字典给不给力了，拼人品的时间到了，和破解 wep 操作一样的，选中信号点 lanch 开始抓包，抓包的时候路由器是一定要有用户使用，目的是攻击导致对方掉线，在自动重连的过程中抓取 WPA 认证的四次握手包。如果一直没找到在线的客户端，就抓不到包的，只能等有人用的时候再试了。Aircrack 是破解 WEP/WPA/WPA2 加密的主流工具之一，Aircrack-ng 套件包含的工具可用于捕获数据包、握手验证。可用来进行暴力破解和字典攻击。在 Ubuntu 14.04 下安装 aircrack-ng：

```
1 apt-get install aircrack-ng
```

力破解和字典攻击。-Aircrack-ng 无线密码破解-Aireplay-ng 流量生成和客户端认证-Airodump-ng 数据包捕获-Airbase-ng 虚假接入点配置

启动无线网卡的监控模式 启动无线网卡的监控模式如下：

```
1 sudo airmon-ng start wlan0
2
3 sudo airodump-ng -a --encrypt WPA mon0
```

无线网卡开始抓包并分析附近有那些热点 (-a), 并且仅显示加密方式为 WPA 或者 WPA2 的热点 (- encrypt WPA) -上图只需要看懂这些参数: BSSID 指一个热点 (PC or Phone), 你可以理解为它的 ID, PWR 指信号强度, 越接近 0 越好, 但是不能等于 0,0 表示断线, 后面将会用到, CH 频道, 这个也会用到, ENC 加密方式, 仅显示了 PSK/PSK2 方式, ESSID 这个就是我们平常看到的热点名称辣

查看无线 AP 在终端中输入:

```
1 sudo airodump-ng mon0
```

(mon0 是启动监控模式后无线网的端口) 查看有哪些采用 wep 加密的 AP 在线, 然后按 ctrl+c 中止, 不要关闭终端。

抓包 打开另一个终端, 输入:

```
1 sudo airodump-ng --channel 6 --bssid AP's MAC -w wep mon0
2
3 sudo airodump-ng --bssid 28:2C:B2:E4:F5:5A -c 13 -w 619 --ignore-negative-one
    mon0
```

(-c 后面跟着的 6 是要破解的 AP 工作频道, -bssid 后面跟着的 AP' sMAC 是要欲破解 AP 的 MAC 地址, -w 后面跟着 wep 的是抓下来的数据包 DATA 保存的文件名, 具体情况根据步骤 2 里面的在线 AP 更改频道和 MAC 地址, DATA 保存的文件名可随便命名)

本机 Mac: 00:25:d3:9a:b6:01

路由 Mac: C0:61:18:9A:76:D2

与 AP 建立虚拟连接 再打开一个新终端, 输入:

```
1 aireplay-ng -2 -F -p 0841 -c ff:ff:ff:ff:ff:ff -b C0:61:18:9A:76:D2 -h 00:25:d3:9a:b6:01
    mon0
```

(-h 后面跟着的 My MAC 是自己的无线网卡的 MAC 地址, 即 ifconfig 命令下 wlan0 对应的 mac 地址)

解密 收集有 15000 个以上的 DATA 之后, 另开一个终端, 切换到 aircrack-ng-1.1 目录, 执行以下命令 sudo aircrack-ng wep*.cap 进行解密 (如果没算出来的话, 继续等, aircrack-ng 会在 DATA 每增加多 15000 个之后就自动再运行, 直到算出密码为至, 注意此处文件的名字要与步骤 3 里面设置的名字一样, 且 * 号是必需的)

Chapter 13

Node.js

13.0.1 Introduce

Node.js 越来越流行，这个基于 Google V8 引擎建立的平台，用于方便地搭建响应速度快、易于扩展的网络应用。Nodejs 标准的 web 开发框架 Express，可以帮助我们迅速建立 web 站点，比起 PHP 的开发效率更高，而且学习曲线更低。非常适合小型网站，个性化网站。

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

13.0.2 Install

Step 1 下载 nodejs，官网：<http://nodejs.org/download/>，我这里下载的是 node-v4.1.1-x64.msi。安装基本是下一步下一步即可完成。当 Node.js 安装完成以后，npm(Node.js Package Management) 也一并安装完毕。

13.0.3 基础的 HTTP 服务器

安装完成后项目的根目录下编写一个简单的服务器端脚本 server.js¹。

```

1 var http = require("http");
2
3 http.createServer(function(request, response) {
4   response.writeHead(200, {"Content-Type": "text/plain"});
5   response.write("Hello World");
6   response.end();
7 }).listen(8888);

```

用 Node 运行这段代码，然后在浏览器端访问链接<http://localhost:8888>即可看到输出的 Hello World。

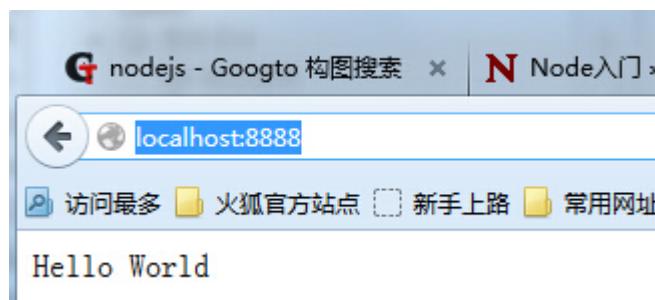


图 13.1: NodeJS Hello World

¹<http://www.nodebeginner.org/index-zh-cn.html>

第一行请求 (require) Node.js 自带的 http 模块，并且把它赋值给 http 变量。接下来我们调用 http 模块提供的函数：createServer。这个函数会返回一个对象，这个对象有一个叫做 listen 的方法，这个方法有一个数值参数，指定这个 HTTP 服务器监听的端口号。

13.0.4 独立出服务端的模块

在 server.js 文件中有一个非常基础的 HTTP 服务器代码，而且通常我们会有一个叫 index.js 的文件去调用应用的其他模块（比如 server.js 中的 HTTP 服务器模块）来引导和启动应用。把 server.js 变成一个真正的 Node.js 模块，使它可以被 index.js 主文件使用。将服务脚本改成如下形式。

```

1 var http = require("http");
2
3 function start() {
4     function onRequest(request, response) {
5         console.log("Request received.");
6         response.writeHead(200, {"Content-Type": "text/plain"});
7         response.write("Hello World");
8         response.end();
9     }
10
11     http.createServer(onRequest).listen(8888);
12     console.log("Server has started.");
13 }
14
15 exports.start = start;

```

现在就可以创建我们的主文件 index.js 并在其中启动我们的 HTTP 了，虽然服务器的代码还在 server.js 中。创建 index.js 文件并写入以下内容：

```

1 var server = require("./server");
2 server.start();

```

Step 2 安装相关环境

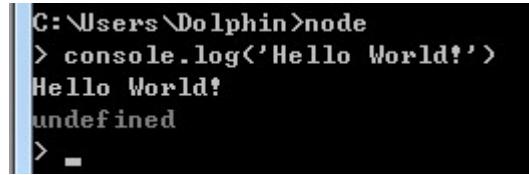
```

1 npm install -g express
2 npm install -g jade
3 npm install -g mysql

```

安装时带上参数 g，表示全局模式，node 的安装分为全局模式和本地模式。一般情况下会以本地模式运行，包会被安装到和你的应用程序代码的本地 node_modules 目录下。在全局模式下，Node 包会被安装到 Node 的安装目录下的 node_modules 下。Express 是 NodeJS 的 Web 开发框架。Jade 是一个高性能的模板引擎，它深受 Haml 影响，它是用 JavaScript 实现的，并且可以供 Node 使用。CoffeeScript 是一套 JavaScript 的转译语言，创建者 Jeremy Ashkenas 戏称它是 JavaScript 的不那么铺张的小兄弟。因为 CoffeeScript 会将类似 Ruby 语法的代码编译成

JavaScript，而且大部分结构都相似，但不同的是 CoffeeScript 拥有更严格的语法。如图所示表示安装完毕。



```
C:\Users\Batman>node
> console.log('Hello World!')
Hello World!
undefined
> =
```

图 13.2: Node JS Install Complete

13.0.5 服务端路由

将 server.js 脚本改成如下的形式。

```
1 var http = require("http");
2 var url=require("url");
3 function start(route){
4     function onRequest(request,response){
5         var pathname=url.parse(request.url).pathname;
6         console.log("Request for " + pathname + " received.");
7         route(pathname);
8         console.log("Request received.");
9         response.writeHead(200, {"Content-Type": "text/plain"});
10        response.write("Hello World");
11        response.end();
12    }
13    http.createServer(onRequest).listen(8888);
14    console.log("Server has started.");
15}
16 exports.start=start;
```

13.0.6 NodeJS RESTful Service

NodeJS 提供 RESTful 接口，此段代码直接定义一个集合资源进行返回。

```
1 var express = require("express") //加载模块
2 var app = express() //实例化之
3 var map = {"list1":{id:1,name:"Pen"}, "list2":{id:2,name:"Pencil"} } //定义一个集合资源
4 app.get("/devices",function(request, response){ //Restful Get方法,查找整个集合资源
5     response.set({"Content-Type":"text/json","Encodeing":"utf8"});
6     response.send(map)
7 })
8
9 app.get("/devices/:id",function(request, response){ //Restful Get方法,查找一个单一资源
10    response.set({"Content-Type":"text/json","Encodeing":"utf8"});
11    response.send(map[request.param("id")])
12    //console.log(request.param("id"))
```

```

13  })
14 app.listen(8888); //监听8888端口

```

将此段代码写入 restfulServer.js 文件中，用 node 命令加载此段代码，用浏览器请求链接 http://localhost:8888/devices 即可获取定义的资源。如果需要获取特定的集合，添加 ID 即可，链接为 http://localhost:8888/devices/ID。

13.0.7 NodeJS connect SQL Server

NodeJS 连接 SQL Server 需要下载依赖包 node-sqlserver，node-sqlserver 是微软官方发布的 SQL Server 的 Node.js 的驱动程序。可允许 Windows 上运行的 Node.js 程序访问 SQL Server 和 Windows Azure SQL 数据库。

```

1 #查看当前项目的本地依赖包
2 npm list

3

4 #查看全局依赖包
5 npm list -g

6

7 #安装 node-gyp
8 npm install -g node-gyp

9

10 #查找与 sqlserver 有关的包
11 npm find sqlserver

12

13 #下载 node-sqlserver 包
14 npm install node-sqlserver -g

```

注意安装 node-sqlserver 时 gyp 工具依赖于 Python，所以还需要安装 Python，且 Python 的版本要求为:>=2.5.0&<3.0.0。

```

1 var sql = require("node-sqlserver");

2

3 var connectionString = "Driver={SQL Server Native Client 11.0};Server={.};Database={";
4     rr-ccn};Trusted_Connection={Yes};uid=sa;PWD=123456;";

5 sql.open(connectionString, function(err, conn) {
6     if(err) {
7         // error handler code
8     }
9     else {
10        // do what you want to do to this database
11        sql.queryRaw(connectionString, "select * from users", function (err, results) {
12            if (err) {
13                console.log(err);
14            }
15            else {
16                for (var i = 0; i < results.rows.length; i++) {

```

```

17         console.log(results.rows[i][0] + results.rows[i][1]);
18     }
19   }
20 }
21 }
22 });

```

<http://geekswithblogs.net/shaunxu/archive/2012/09/17/node.js-adventure—node.js-with-sql-server.aspx>

安装 mssql 包。

Listing 13.1: npm 操作脚本

```

1 #安装 mssql 包
2 npm install mssql -g
3
4 #安装 mssql 包到本地，安装 mssql 包到全局时会提示找不到 mssql 模块
5 npm install mssql --local
6
7 #或者将全局的包和本地的项目链接起来,mssql 为包名
8 npm link mssql
9
10 #强制指定链接为本地链接
11 npm link mssql --local

```

一个连接数据库的例子，index.js 文件的内容如下。

```

1 var server=require("./server");
2 server.start();

```

server.js 的内容如下，此段代码展示最简单的使用 mssql 连接 sql server 数据库。

```

1 var http = require("http");
2 var sql = require("mssql");
3
4 var config = {
5   user: "renren",
6   password: "renren!@#321",
7   server: "129.29.81.129", // You can use "localhost\\instance" to connect to named
8   instance
9   port:"5001",
10  database: "renren-weixin",
11  options: {
12    encrypt: false /* Use this if you're on Windows Azure*/
13  }
14}
15
16 function start(){
17   var connection = new sql.Connection(config, function(err) {

```

```

17     var request = new sql.Request(connection); \\ or: var request = connection.
18     request();
19     request.query("select top 1 * from tb_printMac", function(err, recordset) {
20       console.log(err);
21       console.dir(recordset);
22     });
23     console.log("Server has started.");
24   }
25   exports.start=start;

```

13.0.8 配置文件管理

NodeJS 配置文件可以通过 JavaScript 和 Json 格式的文件进行管理，通过 Json 格式管理，新建 package.json 文件，内容如代码13.2所示。

Listing 13.2: NodeJS json 格式配置

```

1 {
2   "name": "rrmallrestapi",
3   "version": "0.0.1",
4   "dbconnstring":
5   {
6     "renrenweixin":
7     {
8       "user": "renren-admin",
9       "password": "renren-admin!@#321",
10      "server": "129.20.84.118",
11      "port": "5002",
12      "database": "renren-weixin"
13    }
14  }
15 }

```

在 JavaScript 中读取方式如下所示，

```

1 var sysconfig=require("./../../../config/package.json");
2
3 var config = {
4   user: sysconfig.dbconnstring.renrenweixin.user,
5   password: sysconfig.dbconnstring.renrenweixin.password,
6   server: sysconfig.dbconnstring.renrenweixin.server,
7   port:sysconfig.dbconnstring.renrenweixin.port,
8   database: sysconfig.dbconnstring.renrenweixin.database,
9   options: {
10     encrypt: false
11   }
12 }

```

13.0.9 Debug

安装 supervisor。

```

1 #查找可用包
2 npm find packageName
3
4 npm install supervisor -g

```

13.0.10 exports

如果你想你的模块是一个特定的类型就用 Module.exports。如果你想的模块是一个典型的“实例化对象”就用 exports。

13.0.11 处理 POST 请求

13.0.12 Express

Express²是一个基于 Node.js 平台的极简、灵活的 web 应用开发框架，它提供一系列强大的特性，帮助你创建各种 Web 和移动设备应用。

```

1 #安装 express
2 npm install express -g
3
4 #安装 express-generator, 从 express 4.x 版本中将命令工具分出来了, 需要再安装一个命令工具
5 #否则 expresss 命令无法使用
6 npm install express-generator -g

```

进入工程目录，使用 express 命令创建工程³。

```

1
2 D:\workspace\project>express -e nodejs-demo
3
4 cd nodejs-demo && npm install
5
6
7 D:\workspace\project\nodejs-demo>node app.js
8
9 D:\workspace\project\nodejs-demo>supervisor app.js

```

node_modules，存放所有的项目依赖库。(每个项目管理自己的依赖，与 Maven,Gradle 等不同) package.json，项目依赖配置及开发者信息 app.js，程序启动文件 public，静态文件 (css.js,img) routes，路由文件 (MVC 中的 C,controller) Views，页面文件 (Ejs 模板)

²<http://expressjs.com/>

³<http://blog.fens.me/nodejs-express3/>

```
G:\Workspace\project\nodejs-demo>supervisor ./bin/www

Running node-supervisor with
  program './bin/www'
  --watch '.'
  --extensions 'node,js,/bin/www'
  --exec 'node'

Starting child process with 'node ./bin/www'
Watching directory 'G:\Workspace\project\nodejs-demo' for changes.
Press rs for restarting the process.
rs
GET / 200 33.580 ms - 207
GET /stylesheets/style.css 200 10.480 ms - 111
GET /favicon.ico 404 8.913 ms - 1016
GET /favicon.ico 404 5.548 ms - 1016
```

图 13.3: Node JS Install Complete



Express

Welcome to Express

图 13.4: NodeJS Express Install Complete

13.0.13 MEAN

MongoDB

安装完毕后进入 MongoDB 的安装目录，运行如下命令初始化 MongoDB。

```
1 cd C:\Program Files\MongoDB\Server\3.0\bin  
2 mongod --dbpath "d:\DB"
```

dbpath 参数指定 MongoDB 的路径为 d:\DB，运行如下命令启动 MongoDB。

```
1 cd C:\Program Files\MongoDB\Server\3.0\bin  
2 mongo
```

show dbs 显示数据库列表

use dbname 进入 dbname 数据库，大小写敏感，没有这个数据库也不要紧

show collections 显示数据库中的集合，相当于表格

13.1 常用脚本

13.1.1 BAT

```
1 #递归列出当前目录下所有的备份文件, 保存到 txt 文件中  
2 dir *.bak /s>a.txt
```


Chapter 14

异常 (Exception)

在早期的 Win32 API 设计中是通过返回 true/false 来表示一个过程（方法、函数）是否执行成功，在 COM 中是使用 HRESULT 来表示一个过程是否正确执行，然而这种处理异常的方式使开发人员对哪里出错，为什么出错，出什么样的错这些问题很难找到明确的答案，再一点，调用者很容易忽略一个过程执行的结果，如果调用者丢弃了过程执行结果，则代码将“按照期望的状态正常执行”，这是很危险的。后来，在.NET Framework 中，已经不再使用这种简单的以状态码来表示执行结果的处理方式，而是使用抛出异常的方式来告诉调用者。CLR 中的异常是基于 SEH (Structured Exception Handling) 的结构化异常处理机制构建的，在基础的 SEH 机制中是向系统注册出错时的回调函数，当在监视区内出错时，系统会取得当前线程的控制权处理回调函数，处理完毕后，系统释放控制权给当前线程，当前线程继续执行，如果未处理异常的代码段，会导致进程中止。

14.1 Windows SEH

SEH(Structured Exception Handling) 是 Windows 系统提供的功能，跟开发工具无关。值得一提的是，VC 将 SEH 进行了封装 try catch finally, c ++ 中也可以用 c 的封装 __try{}__except(){} 和 __try{}__finally{}. 所以当你建立一个 C++ try 块时，编译器就生成一个 SEH__try 块。SEH 是为 C 语言设计的，但是他们也能够被用于 C++。SEH 异常由 __try{}__except(){} 结构来处理。SEH 是 VC++ 编译器特有的，因此如果你想要编写可移植的代码，就不应当使用 SEH。一个 C++ catch 测试变成一个 SEH 异常过滤器，并且 catch 中的代码变成 SEH__except 块中的代码。实际上，当你写一条 C++ throw 语句时，编译器就生成一个对 Windows 的 Raise Exception 函数的调用。用于 throw 语句的变量传递给 Raise Exception 作为附加的参数。

SEH: 结构化异常处理 VEH: 向量化异常处理 TopLevelEH: 顶层异常处理

1. 交给调试器 (进程必须被调试)
2. **执行 VEH(Vectored Exception Handling)** 向量化异常处理 (VEH) 是结构化异常处理的一个扩展，它在 Windows XP 中被引入。你可以使用 AddVectoredExceptionHandler() 函数添加一个向量化异常处理器，VEH 的缺点是它只能用在 WinXP 及其以后的版本，因此需要在运行时检查 AddVectoredExceptionHandler() 函数是否存在。
3. **执行 SEH(Structured Exception Handling)** 当发生一个 SEH 异常时，你通常会看到一个意图向微软发送错误报告的弹出窗口。你可以使用 RaiseException() 函数自己产生一个 SEH 异常。你可以在你的代码中使用 __try{}__except(Expression){} 结构来捕获 SEH 异常。程序中的 main() 函数被这样的结构保护，因此默认地，所有未被处理的 SEH 异常都会被捕获。
4. TopLevelEH(进程被调试时不会被执行)
5. 交给调试器 (上面的异常处理都说处理不了，就再次交给调试器)
6. 调用异常端口通知 csrss.exe

14.1.1 Don't ever swallow exceptions

The worst thing you can do is catch (Exception) and put an empty code block on it. Never do this.

参照:<http://www.codeproject.com/Articles/9538/Exception-Handling-Best-Practices-in-NET>

14.1.2 Log `Exception.ToString()`; never log only `Exception.Message`

As we're talking about logging, don't forget that you should always log `Exception.ToString()`, and never `Exception.Message`. `Exception.ToString()` will give you a stack trace, the inner exception and the message. Often, this information is priceless and if you only log `Exception.Message`, you'll only have something like "Object reference not set to an instance of an object".

14.1.3 异常列表 (Exception List)

<code>ArgumentException</code>	该类用于处理参数无效的异常
<code>IOException</code>	该类用于处理进行文件输入输出操作时所引发的异常
<code>DivideByZeroException</code>	表示整数十进制运算中试图除以零而引发的异常

14.1.4 Windows Form 中统一异常处理

适当的使用 try 捕获异常，异常信息建议由最上层的框架捕获，将来修改起来相对容易，越少 try...catch 的程序越美。WinFrom 的 Application 对象本身就提供了 ThreadException 时间来捕捉为处理的异常

```

1 static void Main()
2 {
3     //设置应用程序处理异常方式：ThreadException 处理
4     Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);
5     //处理 UI 线程异常
6     Application.ThreadException += new System.Threading.ThreadExceptionEventHandler(
7         Application_ThreadException);
8     //处理非 UI 线程异常
9     AppDomain.CurrentDomain.UnhandledException += new
10        UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);
11        Application.EnableVisualStyles();
12        Application.SetCompatibleTextRenderingDefault(false);
13        Application.Run(new Form1());
14    }
15
16    static void Application_ThreadException(object sender, System.Threading.
17        ThreadExceptionEventArgs e)
18    {

```

```

15     var exceptionMessage = GetExceptionMsg(e.Exception, e.ToString());
16     Log.Logger.Error(exceptionMessage);
17 }
18
19 static void CurrentDomain_UnhandledException(object sender,
20                                         UnhandledEventArgs e)
21 {
22     var exceptionMessage = GetExceptionMsg(e.ExceptionObject as Exception, e.ToString());
23     Log.Logger.Error(exceptionMessage);
24 }
25
26 /// <summary>
27 /// 生成自定义异常消息
28 /// </summary>
29 /// <param name="ex"> 异常对象 </param>
30 /// <param name="backStr"> 备用异常消息: 当 ex 为 null 时有效 </param>
31 /// <returns> 异常字符串文本 </returns>
32 static string GetExceptionMsg(Exception ex,string backStr)
33 {
34     var exceptionContent = new StringBuilder();
35     exceptionContent.AppendLine("*****异常文本*****");
36     exceptionContent.AppendLine("【出现时间】：" + DateTime.Now.ToString(CultureInfo.
37                               InvariantCulture));
38     if (ex != null)
39     {
40         exceptionContent.AppendLine("【异常类型】：" + ex.GetType().Name);
41         exceptionContent.AppendLine("【异常信息】：" + ex.Message);
42         exceptionContent.AppendLine("【堆栈调用】：" + ex.StackTrace);
43     }
44     else
45     {
46         exceptionContent.AppendLine("【未处理异常】：" + backStr);
47     }
48     exceptionContent.AppendLine("*****");
49     return exceptionContent.ToString();
50 }

```

14.1.5 Windows Service Error Handling

This will fire for unhandled exceptions in the given domain no matter what thread they occur on. If your windows service uses multiple AppDomains you'll need to use this value for every domain but most don't.

```

1 //Register error handling event,both are right
2 AppDomain.CurrentDomain.UnhandledException += CurrentDomain_UnhandledException;
3 AppDomain.CurrentDomain.UnhandledException +=new UnhandledExceptionEventHandler
4             (CurrentDomain_UnhandledException);
5 static void CurrentDomain_UnhandledException(object sender,

```

```

6     UnhandledExceptionEventArgs e)
7     {
8         HandleException((Exception)e.ExceptionObject);
9     }
10    static void HandleException(Exception e)
11    {
12        //Handle your Exception here
13    }

```

未捕获的异常，通常就是运行时期的 BUG，于是我们可以在 UnhandledException 的注册事件方法 CurrentDomain_UnhandledException 中将未捕获异常的信息记录在日志中。值得注意的是，UnhandledException 提供的机制并不能阻止应用程序终止，也就是说，CurrentDomain_UnhandledException 方法执行后，应用程序就会被终止。

14.1.6 DllNotFoundException

在.NET 下通过 DLLImport 使用 C++ 编写的 dll 组件，抛出 DllNotFoundException，原因是由于该 dll 依赖于其他 dll，而另其他 dll 不存在。使用 Dependency Walker¹ 工具查看 dll 文件的依赖如图14.1所示：

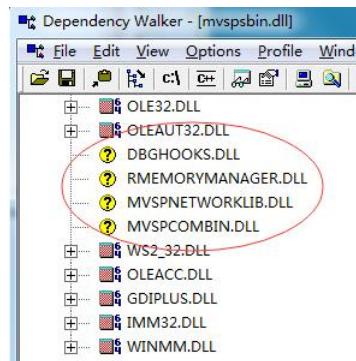


图 14.1: Dll Dependency

14.2 静态文件自动化管理

¹Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module (exe, dll, ocx, sys, etc.) and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions that are exported by that module, and which of those functions are actually being called by other modules. Another view displays the minimum set of required files, along with detailed information about each file including a full path to the file, base address, version numbers, machine type, debug information, and more. Website: <http://dependencywalker.com/>

Chapter 15

网络 (Network)

本地的进程间通信 (IPC) 有很多种方式，但可以总结为下面 4 类：

- 消息传递（管道、FIFO、消息队列）
- 同步（互斥量、条件变量、读写锁、文件和写记录锁、信号量）
- 共享内存（匿名的和具名的）
- 远程过程调用（Solaris 门和 Sun RPC）

15.0.1 RAML

RESTful API Modeling Language (RAML) makes it easy to manage the whole API lifecycle from design to sharing. It's concise - you only write what you need to define - and reusable. It is machine readable API design that is actually human friendly.

15.0.2 RestSharp

现在互联网上的服务接口都是 Restful 的，SOAP 的 Service 已经不是主流。.NET/Mono 下如何消费 Restful Service 呢，再也没有了方便的 Visual Studio 的方便生产代理的工具了，你还在用 `HttpWebRequest` 自己封装吗？Restful Service 还有授权问题，自己写出来的代码是不是很不优雅？通常 Restful Service 返回的数据格式是 XML 或者 Json，还要设置服务的输入参数等等，使用起来很复杂。本文向你推荐一个开源的库 RestSharp 轻松消费 Restful Service。RestSharp 是一个开源的.NET 平台下 REST 和 HTTP API 的客户端库，支持的平台有.NET 3.5/4、Mono、Mono for Android、MonoTouch、Windows Phone 7.1 Mango。他可以简化我们访问 Restful 服务，可以到[这里下载代码](https://github.com/johnsheehan/RestSharp/archives/master)<https://github.com/johnsheehan/RestSharp/archives/master>更简单的使用 NuGet。RestSharp 使用 Json.Net 处理 Json 数据同 Poco 对象的序列化。

```
1 var client = new RestClient { BaseUrl = new Uri(PublicAttribute.WebApiPath) };
```

```

2 var request = new RestRequest { Resource = "/tour/RoadLine/GetRoadLineByPage?tagId=0
3     pageIndex=1&pageSize=5" };
4 request.AddHeader("auth", "value");
5 IRestResponse response = client.Execute(request);

```

15.1 TCP

15.1.1 Ethernet v2

以太网规定，一组电信号构成一个数据包，叫做“帧”(Frame)。每一帧分成两个部分：标头(Head)和数据(Data)。“标头”包含数据包的一些说明项，比如发送者、接受者、数据类型等等；“数据”则是数据包的具体内容。“标头”的长度，固定为18字节。“数据”的长度，最短为46字节，最长为1500字节。因此，整个“帧”最短为64字节，最长为1518字节，如图15.1所示。

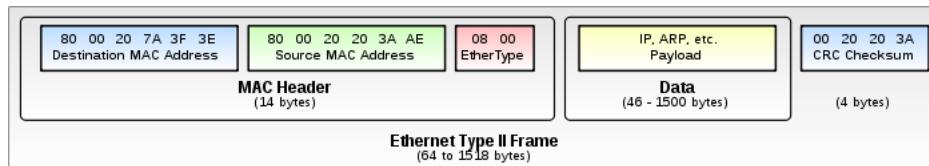


图 15.1: Ethernet 2 消息头

如果数据很长，就必须分割成多个帧进行发送。在本地可以通过进程 PID 来唯一标识一个进程，但是在网络中这是行不通的。其实 TCP/IP 协议族已经帮我们解决了这个问题，网络层的“ip 地址”可以唯一标识网络中的主机，而传输层的“协议 + 端口”可以唯一标识主机中的应用程序（进程）。这样利用三元组（ip 地址，协议，端口）就可以标识网络的进程了，网络中的进程通信就可以利用这个标志与其它进程进行交互。

15.1.2 传输层 (Transport Layer) 的由来

有了 MAC 地址和 IP 地址，我们已经可以在互联网上任意两台主机上建立通信。接下来的问题是，同一台主机上有许多程序都需要用到网络，比如，你一边浏览网页，一边与朋友在线聊天。当一个数据包从互联网上发来的时候，你怎么知道，它是表示网页的内容，还是表示在线聊天的内容？也就是说，我们还需要一个参数，表示这个数据包到底供哪个程序（进程）使用。这个参数就叫做“端口”(port)，它其实是每一个使用网卡的程序的编号。每个数据包都发到主机的特定端口，所以不同的程序就能取到自己所需要的数据。”端口”是0到65535之间的一个整数，正好16个二进制位。0到1023的端口被系统占用，用户只能选用大于1023的端口。不管是浏览网页还是在线聊天，应用程序会随机选用一个端口，然后与服务器的相应端口联系。”传输层”的功能，就是建立”端口到端口”的通信。相比之下，”网络层”的功能是建立”主机到主机”的通信。只要确定主机和端口，我们就能实现程序之间的交流。因此，Unix 系统就把主机 + 端口，叫做”套接字”(socket)。有了它，就可以进行网络应用程序开发了。

15.1.3 什么是 Socket

上面我们已经知道网络中的进程是通过 socket 来通信的，那什么是 socket 呢？socket 起源于 Unix，而 Unix/Linux 基本哲学之一就是“一切皆文件”，都可以用“打开 open -> 读写 write/read -> 关闭 close”模式来操作。我的理解就是 Socket 就是该模式的一个实现，socket 即是一种特殊的文件，一些 socket 函数就是对其进行的操作（读/写 IO、打开、关闭）。socket 一词的起源是在组网领域的首次使用是在 1970 年 2 月 12 日发布的文献 IETF RFC33 中发现的，撰写者为 Stephen Carr、Steve Crocker 和 Vint Cerf。根据美国计算机历史博物馆的记载，Croker 写道：“命名空间的元素都可称为套接字接口。一个套接字接口构成一个连接的一端，而一个连接可完全由一对套接字接口规定。”计算机历史博物馆补充道：“这比 BSD 的套接字接口定义早了大约 12 年。”

15.1.4 三次握手

我们知道 tcp 建立连接要进行“三次握手”，即交换三个分组。大致流程如下：

- 客户端向服务器发送一个 SYN J
- 服务器向客户端响应一个 SYN K，并对 SYN J 进行确认 ACK J+1
- 客户端再向服务器发一个确认 ACK K+1

从图15.2可以看出，当客户端调用 connect 时，触发了连接请求，向服务器发送了 SYN J 包，这时 connect 进入阻塞状态；服务器监听到连接请求，即收到 SYN J 包，调用 accept 函数接收请求向客户端发送 SYN K，ACK J+1，这时 accept 进入阻塞状态；客户端收到服务器的 SYN K，ACK J+1 之后，这时 connect 返回，并对 SYN K 进行确认；服务器收到 ACK K+1 时，accept 返回，至此三次握手完毕，连接建立。

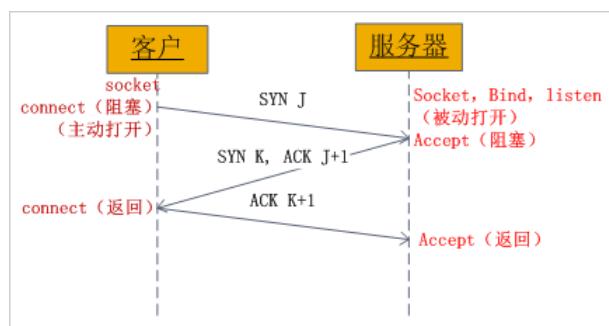


图 15.2: 3 次握手

15.1.5 子网掩码

子网掩码 (subnet mask) 是每个使用互联网的人必须要掌握的基础知识，只有掌握它，才能够真正理解 TCP/IP 协议的设置。子网掩码——屏蔽一个 IP 地址的网络部分的“全 1”比特模式。对于 A 类地址来说，默认的子网掩码是 255.0.0.0；对于 B 类地址来说默认的子网掩码是 255.255.0.0；对于 C 类地址来说默认的子网掩码是 255.255.255.0。

15.1.6 TCP 流量控制

所谓流量控制就是让发送发送速率不要过快，让接收方来得及接收。利用滑动窗口机制就可以实施流量控制。原理这就是运用 TCP 报文段中的窗口大小字段来控制，发送方的发送窗口不可以大于接收方发回的窗口大小。考虑一种特殊的情况，就是接收方若没有缓存足够使用，就会发送零窗口大小的报文，此时发送方将发送窗口设置为 0，停止发送数据。之后接收方有足够的缓存，发送了非零窗口大小的报文，但是这个报文在中途丢失的，那么发送方的发送窗口就一直为零导致死锁。

解决这个问题，TCP 为每一个连接设置一个持续计时器 (persistence timer)。只要 TCP 的一方收到对方的零窗口通知，就启动该计时器，周期性的发送一个零窗口探测报文段。对方就在确认这个报文的时候给出现在窗口大小（注意：TCP 规定，即使设置为零窗口，也必须接收以下几种报文段：零窗口探测报文段、确认报文段和携带紧急数据的报文段）。

15.1.7 通讯协议分析

车辆道路运输车辆卫星定位系统北斗兼容车载终端通讯协议分析 WORD 相当于 C# 中的 short,short 的取值范围为 0 to 65,535，相当于.NET Framework 中的 System.UInt16，对应 2 字节，即 byte[2]。

1	7e(包头,标志位) 0100(消息ID) 002d(消息体属性) 014846362281(终端手机号码)
2	0002(流水号) 002c(省域ID) 012c(市县域ID) 3730313131(制造商ID)
3	42534a2d413642440000-10位-0000000000000000(终端型号)
4	30313436363336(终端ID) 02(车牌颜色) c2b3464c353536(车辆标识) 31ca(校验码) 7e(包尾)

15.1.8 查看网络状态

Netstat 是一款命令行工具，可用于列出系统上所有的网络套接字连接情况，包括 tcp, udp 以及 unix 套接字，另外它还能列出处于监听状态（即等待接入请求）的套接字。如果你想确认系统上的 Web 服务有没有起来，你可以查看 80 端口有没有打开。以上功能使 netstat 成为网管和系统管理员的必备利器。

1	#Windows 查看 20 端口情况
2	netstat -ano find "20"

参数 a 表示显示所有 (all) 连接和侦听端口。参数 n 表示以数字 (number) 形式显示地址和端口号。参数 o 表示显示拥有的与每个连接关联的进程 ID。

15.2 X11

X11 (X Window System core protocol) 是由 MIT 于 1984 年设计出来的开源传输协议，一直发展至今，最新版本是 X11R7.5。它是 X Window System 的基本协议。而 X Window System 系统生来就是为瘦客户服务的，从设计之初，它就被设计成计算和显示分离的架构，即程序的运行可以在一台计算机，而显示又在另外一台计算机。随着 X11 的不断演变发展，出现了各种不同形式的改良版本，其中最著名的就是 NoMachine 公司开发的 NX 协议，NX 协议在 X11 的基础上，加入了缓存机制、压缩传输等，使其性能得到飞跃的提升。

X11 的设计原则是：create Mechanism, not Policy，所以 X 故意没有规范应用程式的使用者界面，例如按钮、选单和视窗的标题栏等等。这些都由视窗管理器 (window managers)、GUI 构件工具包、桌面环境 (desktop environments) 或者应用程序指定的 GUI (如 POS 机) 等等诸如此类的用户软件来提供。这样我们就可以理解，为什么 Linux 系统中会有诸多如 Gnome, KDE 之类桌面系统，同样使用 X 协议，绘制的界面却不尽相同。X11 是 X Window System Protocol, Version 11 (RFC1013)，是 X server 和 X client 之间的通信协议。X server 是 xfree86/xorg 驱动下的显示设备鼠标键盘统称，X client 通过 X11 协议和 xfree86/xorg 实现的 X server 通信，比如，告诉它画一个左上角坐标为 (x,y)，宽为 w，高为 h 的窗口，xfree86 就让显示器把屏幕上的小灯（像素）打亮，然后你就看到了一个窗口。为了方便开发人员编写 X clients，就有了 Xlib 来封装协议；Xlib 不够方便，于是就有了 qt 和 gtk，提供了很多窗口控件 (widgets)。为了方便用户，就出现了 gnome 和 kde 等桌面管理系统。一般来说，linux 用户看到的界面就是其中之一了。gnome 用的是 gtk 库，kde 用的是 qt 库。

X11 本身并不复杂，Server 和 Client 交互的请求一共四种：Requests, Replies, Events, Errors。

1. Request: Client 请求 Server 端返回信息或执行动作。
2. Reply: Server 针对 Request 的返回。不是所有 Request 都有返回。
3. Event: Server 发送的一些界面相关的事件给 Client，例如：键盘、鼠标输入，窗口移动，Resize 等等。
4. Error: 当 Request 请求无效时，Server 发送错误信息给 Client。

15.3 即时通信

15.3.1 title

现客户端跟服务器之前相互发送接受消息的功能

15.4 RPC

15.4.1 RPC(远程过程调用) 是什么

简单的说，RPC 就是从一台机器（客户端）上通过参数传递的方式调用另一台机器（服务器）上的一个函数或方法（可以统称为服务）并得到返回的结果。RPC 会隐藏底层的通讯细节（不需要直接处理 Socket 通讯或 Http 通讯）RPC 是一个请求响应模型。客户端发起请求，服务器返回响应（类似于 Http 的工作方式）RPC 在使用形式上像调用本地函数（或方法）一样去调用远程的函数（或方法）。

15.4.2 远程过程调用发展历程

ONC RPC (开放网络计算的远程过程调用), OSF RPC (开放软件基金会的远程过程调用) CORBA (Common Object Request Broker Architecture 公共对象请求代理体系结构) DCOM (分布式组件对象模型), COM+ Java RMI .NET Remoting XML-RPC, SOAP, Web Service PHPRPC, Hessian, JSON-RPC Microsoft WCF, WebAPI ZeroC Ice, Thrift, GRPC Hprose

早期的 RPC 第一代 RPC (ONC RPC, OSF RPC) 不支持对象的传递。CORBA 太复杂，各种不同实现不兼容，一般程序员也玩不转。DCOM, COM+ 逃不出 Windows 的手掌心。RMI 只能在 Java 里面玩。.NET Remoting 只能在.NET 平台上玩。

XML-RPC, SOAP, WebService 冗余数据太多，处理速度太慢。RPC 风格的 Web Service 跨语言性不佳，而 Document 风格的 Web Service 又太过难用。Web Service 没有解决用户的真实问题，只是把一个问题变成了另一个问题。Web Service 的规范太过复杂，以至于在.NET 和 Java 平台以外没有真正好用的实现，甚至没有可用的实现。跨语言跨平台只是 Web Service 的一个口号，虽然很多人迷信这一点，但事实上它并没有真正实现。

PHPRPC 基于 PHP 内置的序列化格式，在跨语言的类型映射上存在硬伤。通讯上依赖于 HTTP 协议，没有其它底层通讯方式的选择。内置的加密传输既是特点，也是缺点。虽然比基于 XML 的 RPC 速度快，但还不是足够快。

Hessian 二进制的数据格式完全不具有可读性。官方只提供了两个半语言的实现 (Java, ActionScript 和不怎么完美的 Python 实现)，其它语言的第三方实现良莠不齐。支持的语言不够多，对 Web 前端的 JavaScript 完全无视。虽然是动态 RPC，但动态性仍然欠佳。虽然比基于 XML 的 RPC 速度快，但还不是足够快。

JSON-RPC JSON 具有文本可读性，且比 XML 更简洁。JSON 受 JavaScript 语言子集的限制，可表示的数据类型不够多。JSON 格式无法表示数据内的自引用，互引用和循环引用。某些

语言具有多种版本的实现，但在类型影射上没有统一标准，存在兼容性问题。JSON-RPC 虽然有规范，但是却没有统一的实现。在不同语言中的各自实现存在兼容性问题，无法真正互通。

Microsoft WCF, WebAPI 它们是微软对已有技术的一个.NET 平台上的统一封装，是对.NET Remoting、WebService 和基于 JSON 、XML 等数据格式的 REST 风格的服务等技术的一个整合。虽然号称可以在.NET 平台以外来调用它的这些服务，但实际上跟在.NET 平台内调用完全是两码事。它没有提供任何在其他平台的语言中可以使用的任何工具。

ZeroC Ice, Thrift, GRPC 初代 RPC 技术的跨语言面向对象的回归。仍然需要通过中间语言来编写类型和接口定义。仍然需要用代码生成器来将中间语言编写的类型和接口定义翻译成你所使用的编程语言的客户端和服务端的占位程序 (stub)。你必须要基于生成的服务器代码来单独编写服务，而不能将已有代码直接作为服务发布。你必须用生成的客户端代码来调用服务，而没有其它更灵活的方式。如果你的中间代码做了修改，以上所有步骤你都要至少重复一遍。

Hprose 无侵入式设计，不需要单独定义类型，不需要单独编写服务，已有代码可以直接发布为服务。具有丰富的数据类型和完美的跨语言类型映射，支持自引用，互引用和循环引用数据。支持众多传输方式，如 HTTP、TCP、Websocket 等。客户端具有更灵活的调用方式，支持同步调用，异步调用，动态参数，可变参数，引用参数传递，多结果返回（Golang）等语言特征，Hprose 2.0 甚至支持推送。具有良好的可扩展性，可以通过过滤器和中间件实现加密、压缩、缓存、代理等各种功能性扩展。兼容的无差别跨语言调用支持更多的常用语言和平台支持浏览器端的跨域调用没有中间语言，无需学习成本性能卓越，使用简单

Chapter 16

.NET



图 16.1: 微软 Net 生态

16.1 委托 (Delegate)

查看 BeginInvoke 调用的方法，可查找委托事件添加的位置。

16.2 泛型 (Generic) 与集合 (Collection)

16.2.1 List 与深拷贝

在 List 添加对象时，如果对象是引用类型，List 添加的只是引用，引用类型传递的是对象的指针，List 指向的是对象的地址。如果添加的是值类型，值类型传递的是另外一个副本，那么 List 添加的将是一新的值类型。所造成的现象就是：当引用类型对象的值改变时，List 的值也会跟着改变。如下代码片段所示：

Listing 16.1: List 添加对象引用示例

```

1 var productModel = new ProductModel();
2 foreach (var current in originProduct)
3 {
4     productModel = HttpPost.MapModel(productModel, current);
5     distnateProduct.Add(productModel);
6 }

```

添加到 List 中的永远是当前对象的地址，由于对象属于全局变量，所有添加到对象都记录的是同一个内存地址，最终导致保存的是同一个对象且是最末的那个对象，此处需要将创建对象的实例写到循环中，也可以在类中加上 ICloneable 接口，并实现 Clone 方法。

16.3 Microsoft Build Engine(MIT License)

The Microsoft Build Engine is a platform for building applications. This engine, which is also known as MSBuild, provides an XML schema for a project file that controls how the build platform processes and builds software. Visual Studio uses MSBuild, but MSBuild does not depend on Visual Studio. By invoking msbuild.exe on your project or solution file, you can orchestrate(vt. 把…编成管弦乐曲; (美) 精心安排; 把…协调地结合起来) and build products in environments where Visual Studio isn't installed. MSBuild 是一个既熟悉又陌生的名字, Visual Studio 的项目加载和构建均通过 MSBuild 来实现。VS 中右键打开项目菜单, “生成” 对应 MSBuild 的 Build 目标, VS 中的“重新生成”对应 MSBuild 的 Rebuild 目标, 清理对应 MSBuild 的 Clean 目标, “发布” 对应 MSBuild 的 PublishOnly 目标。到这里我想大家都明白 MSBuild 就和 Ant 一样就是一个用于项目构建的任务执行引擎, 只不过它被融入到 VS 中, 降低了入门难度。但融入 VS 中只是方便我们使用而已, 并不代表不用了解学习, 尤其项目规模愈发庞大时, 编写结构良好的 MSBuild Script 来作为项目构建和管理的基石是必不可少。

文件类型	作用
*.sln	项目、解决方案在磁盘上的引用, VS 通过该类文件加载整个项目、解决方案
*.suo	用户界面的自定义配置(包括布局、断电和项目最后编译后而又没有关闭的文件标签等), 下一次打开 VS 时会恢复这些配置
*.csproj.user	保存 VS 的个人配置
*.csproj	XML 格式, 保存项目的依赖项和项目构建步骤、任务等(需要上传到版本库的)

使用如下命令编译项目

```
1 msbuild.exe /p:VisualStudioVersion=12.0 D:\项目\192.168.1.222\Tour\trunk\RR.Web.CCN.
   Tour\RR.Web.CCN.Tour.csproj
```

HintPath 配置节可理解为提示路径, 它给编译器到哪里寻找文件提供提示或者说是线索。 HintPath is an optional attribute of the item. It gives the clue to the build process on where to find the assembly. The SearchPaths property defined in Microsoft.Common.targets define the search order of the assemblies. The user can change the order by editing the property value. Please note that modifying standard targets file is not recommended. The default search order is:

- Files from the current project –indicated by CandidateAssemblyFiles.
- \$(ReferencePath) property that comes from .user/targets file.
- \$(HintPath) indicated by reference item.
- Target framework directory.
- Directories found in registry that uses AssemblyFoldersEx Registration.

- Registered assembly folders, indicated by AssemblyFolders.
- \$(OutputPath) or \$(OutDir)
- GAC

The reference item include attribute as if it were a complete file name. Please note that in resolving file references components in HKCU are always preferred over HKLM current framework target version is preferred over older target version

16.4 Windows dll 装载过程

Windows 系统平台上，你可以将独立的程序模块创建为较小的 DLL(Dynamic Linkable Library) 文件，并可对它们单独编译和测试。在运行时，只有当 EXE 程序确实要调用这些 DLL 模块的情况下，系统才会将它们装载到内存空间中。这种方式不仅减少了 EXE 文件的大小和对内存空间的需求，而且使这些 DLL 模块可以同时被多个应用程序使用。Microsoft Windows 自己就将一些主要的系统功能以 DLL 模块的形式实现。例如 IE 中的一些基本功能就是由 DLL 文件实现的，它可以被其它应用程序调用和集成。一般来说，DLL 是一种磁盘文件（通常带有 DLL 扩展名，是标准 win32 可执行文件 - “PE” 格式），它由全局数据、服务函数和资源组成，在运行时被系统加载到进程的虚拟空间中，成为调用进程的一部分，进程中所有线程都可以调用其中的函数。如果与其它 DLL 之间没有冲突，该文件通常映射到进程虚拟空间的同一地址上。DLL 模块中包含各种导出函数，用于向外界提供服务。Windows 在加载 DLL 模块时将进程函数调用与 DLL 文件的导出函数相匹配。

在 Win32 环境中，每个进程都复制了自己的读/写全局变量。如果想要与其它进程共享内存，必须使用内存映射文件或者声明一个共享数据段。DLL 模块需要的堆栈内存都是从运行进程的堆栈中分配出来的。DLL 文件中包含一个导出函数表（存在于 PE 的.edata 节中）。这些导出函数由它们的符号名和称为标识号的整数与外界联系起来。函数表中还包含了 DLL 中函数的地址。当应用程序加载 DLL 模块时时，它并不知道调用函数的实际地址，但它知道函数的符号名和标识号。动态链接过程在加载的 DLL 模块时动态建立一个函数调用与函数地址的对应表。如果重新编译和重建 DLL 文件，并不需要修改应用程序，除非你改变了导出函数的符号名和参数序列。简单的 DLL 文件只为应用程序提供导出函数，比较复杂的 DLL 文件除了提供导出函数以外，还调用其它 DLL 文件中的函数。

每个 DLL 都有一个入口函数¹(DLLMain)，系统在特定环境下会调用 DLLMain。在下面的事件发生时会调用 dll 入口函数：1. 进程装载 DLL。2. 进程卸载 DLL。3.DLL 在被装载之后创建了新线程。4. DLL 在被装载之后一个线程被终止了。应用程序导入函数与 DLL 文件中的导出函数进行链接有两种方式：隐式链接和显式链接。

¹ 此种说法不完全正确，类库没有入口函数，main 入口函数并不是必须的，执行的过程类似于：用户执行编译器输出的应用程序 (PE 文件)，操作系统载入 PE 文件，以及其他 DLL(.NET 动态连接库)。操作系统装载器根据前面 PE 文件中的可执行文件头跳转到程序的入口点。显然，操作系统并不能执行中间语言，该入口点也被设计为跳转到 mscoree.dll (.NET 平台的核心支持 DLL) 的 __CorExeMain() 函数入口。CorExeMain() 函数开始执行 PE 文件中的中间语言代码。这里的执行的意思是通用语言运行时按照调用的对象方法为单位，用即时编译器将中间语言编译成本地机二进制代码，执行并根据需要存于机器缓存。

隐式链接 (load-time dynamic linking) 是指在应用程序中不需指明 DLL 文件的实际存储路径，程序员不需关心 DLL 文件的实际装载（由编译器自动完成地址分配）。采用隐式链接方式，程序员在建立一个 DLL 文件时，链接程序会自动生成一个与之对应的 LIB 导入文件。该文件包含了每一个 DLL 导出函数的符号名和可选的标识号，但是并不含有实际的代码。LIB 文件作为 DLL 的替代文件被编译到应用程序项目中。当程序员通过静态链接方式编译生成应用程序时，应用程序中的调用函数与 LIB 文件中导出符号相匹配，这些符号或标识号进入到生成的 EXE 文件中。LIB 文件中也包含了对应的 DLL 文件名（但不是完全的路径名），链接程序将其存储在 EXE 文件内部。当应用程序运行过程中需要加载 DLL 文件时，Windows 根据这些信息发现并加载 DLL，然后通过符号名或标识号实现对 DLL 函数的动态链接。我们使用的大部分系统 Dll 就是通过这样的方式链接的。若找不到需要的 Dll 则会给出一个 Dll 缺少的错误消息。

显式链接 (run-time dynamic linking) 与此相反。用户程序在编译的时候并没有指明需要哪些 Dll，而是在运行起来之后调用 Win32 的 LoadLibrary() 函数，去装载 Dll。若没有找到 Dll 则这个函数就会返回一个错误。在用 LoadLibrary() 函数装载 Dll 之后，应用程序还需要用 GetProcAddress() 函数去获得 Dll 输出函数的地址。显式链接方式对于集成化的开发语言比较适合。有了显式链接，程序员就不必再使用导入文件，而是直接调用 Win32 的 LoadLibrary() 函数，并指定 DLL 的路径作为参数。还要说明一点的就是 Known Dlls 就是保证在通过 LoadLibrary() 去装载系统 Dll 的时候，只从特定的系统目录去装载，防止装载错。装载的时候会去看注册表下是否有一样的注册表键名。如果是装载 windows/system32/ 目录下的对应的 Dll。

Dll 的搜索顺序，在 Windows 上有个注册表键值决定了 Dll 的搜索顺序：HKLM/System/CurrentControlSet/SessionManager/SafeDllSearchMode。在 Vista,Server 2003,XP sp2 中这个值为 1，在 xp,2000 sp4 中为 0。1 值时的搜索顺序为：

1. 可执行文件所在目录
2. 系统目录 windows/system32/
3. 16 位系统目录
4. Windows 目录
5. 当前进程目录
6. 环境变量 PATH 中的目录

0 值时的搜索顺序为：

1. 可执行文件所在目录
2. 当前进程目录
3. 系统目录 windows/system32/
4. 16 位系统目录
5. Windows 目录

6. 环境变量 PATH 中的目录

没有做强名称签名的程序集，对于这种情况，CLR 查找和加载程序集的方式如下

程序的根目录根目录下面，与被引用程序集同名的子目录根目录下面被明确定义为私有目录的子目录同时，这种情况下，如果有定义 codebase，则 codebase 的优先级最高，而且如果 codebase 指定的路径找不到，则直接报告错误，不再查找其他目录

有做强名称签名的程序集，对于这种情况，CLR 查找和加载程序集的方式如下

全局程序集缓存如果有定义 codebase，则以 codebase 定义为准，如果 codebase 指定的路径找不到，则直接报告错误程序的根目录根目录下面，与被引用程序集同名的子目录根目录下面被明确定义为私有目录的子目录

16.5 PE 文件

一个 PE 文件至少需要两个 Section，一个是存放代码，一个存放数据。NT 上的 PE 文件基本上有 9 个预定义的 Section。分别是：.text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, 和.debug。一些 PE 文件中只需要其中的一部分 Section. 以下是通常的分类：

1. 执行代码 Section，通常命名为：.text (MS) or CODE (Borland)
2. 数据 Section, 通常命名为：.data, .rdata, 或.bss(MS) 或 DATA(Borland)
3. 资源 Section, 通常命名为：.edata
4. 输入数据 Section, 通常命名为：.idata
5. 调试信息 Section, 通常命名为：.debug

这些只是命名方式，便于识别。通常与系统并无直接关系。通常，一个 PE 文件在磁盘上的映像跟内存中的基本一致。但并不是完全的拷贝。Windows 加载器会决定加载哪些部分，哪些部分不需要加载。而且由于磁盘对齐与内存对齐的不一致，加载到内存的 PE 文件与磁盘上的 PE 文件各个部分的分布都会有差异。PE 文件的结构如图16.2所示。

最开头的是部分是 DOS 部首，DOS 部首由两部分组成：DOS 的 MZ 文件标志和 DOS stub(DOS 存根程序)。之所以设置 DOS 部首是微软为了兼容原有的 DOS 系统下的程序而设立的。紧接着的是真正的 PE 文件头。值得注意的是 PE 文件头中的 IMAGE_OPTIONAL_HEADER32 是一个非常重要的结构，PE 文件中的导入表、导出表、资源、重定位表等数据的位置和长度都保存在这个结构里。

16.5.1 .reloc

代码里 call 的地址是一个模块内的地址，而且是一个 VA，那么如果模块基址发生了变化，这个地址岂不是就无效了？这个问题如何解决？

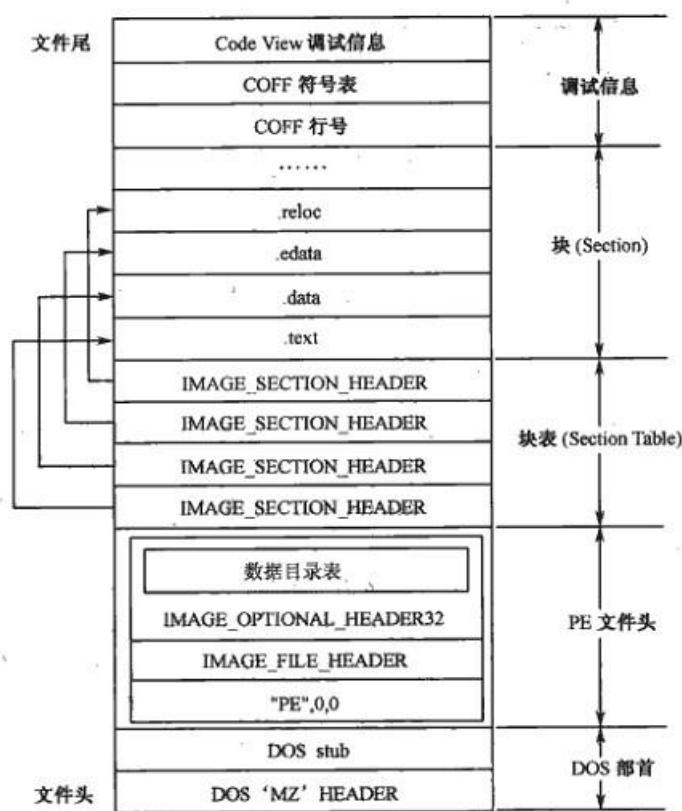


图 16.2: PE 文件结构

答案是：Windows 使用重定位机制保证以上代码无论模块加载到哪个基址都能正确被调用。听起来很神奇，是怎么做到的呢？其实原理并不很复杂，这个过程分三步：

1. 编译的时候由编译器识别出哪些项使用了模块内的直接 VA，比如 push 一个全局变量、函数地址，这些指令的操作数在模块加载的时候就需要被重定位。
2. 链接器生成 PE 文件的时候将编译器识别的重定位的项纪录在一张表里，这张表就是重定位表，保存在 DataDirectory 中，序号是 IMAGE_DIRECTORY_ENTRY_BASERELOC。
3. PE 文件加载时，PE 加载器分析重定位表，将其中每一项按照现在的模块基址进行重定位。

16.5.2 Dos stub

DOS 头，如果电脑安装有 Windows SDK，则在路径：

```
1 C:\Program Files\Microsoft SDKs\Windows\v7.1\Include
```

下的文件 WinNT.h 中可以找到对于 PE 文件中对于 DOS 头的结构定义。结构体对于头文件的定义如下：

```
1 typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
2     WORD    e_magic;           // Magic number
3     WORD    e_cblp;           // Bytes on last page of file
4     WORD    e_cp;             // Pages in file
5     WORD    e_crlc;           // Relocations
6     WORD    e_cparhdr;        // Size of header in paragraphs
7     WORD    e_minalloc;       // Minimum extra paragraphs needed
8     WORD    e_maxalloc;       // Maximum extra paragraphs needed
9     WORD    e_ss;              // Initial (relative) SS value
10    WORD    e_sp;              // Initial SP value
11    WORD    e_csum;            // Checksum
12    WORD    e_ip;              // Initial IP value
13    WORD    e_cs;              // Initial (relative) CS value
14    WORD    e_lfarlc;          // File address of relocation table
15    WORD    e_ovno;            // Overlay number
16    WORD    e_res[4];          // Reserved words
17    WORD    e_oemid;           // OEM identifier (for e_oeminfo)
18    WORD    e_oeminfo;          // OEM information; e_oemid specific
19    WORD    e_res2[10];         // Reserved words
20    LONG    e_lfanew;           // File address of new exe header
21 } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

由 100 个左右的字节所组成，用来输出类似“这个程序不能在 DOS 下运行！”这样的错误信息；DOS 头的作用是兼容 MS-DOS 操作系统中的可执行文件，对于 32 位 PE 文件来说，DOS 所起的作用就是显示一行文字，提示用户：我需要在 32 位 Windows 上才可以运行。MS-DOC MZ Header 和 MS-DOS Stub 是为了兼容 DOS 系统存在的，目的是使这个 exe 在 DOS 下执行时弹出一个提示“This program cannot be run in DOS mode”。

16.5.3 NT Header

顺着 DOS 头中的 e_lfanew，我们很容易可以找到 NT 头，这个才是 32 位 PE 文件中最有用的头，定义如下：

```

1 typedef struct _IMAGE_NT_HEADERS {
2     DWORD Signature;
3     IMAGE_FILE_HEADER FileHeader;
4     IMAGE_OPTIONAL_HEADER32 OptionalHeader;
5 } IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;

```

NT 头与 File Header、Signature、Optional Header 的关系如图16.3所示：



图 16.3: PE 文件的结构 (NT Header)

PE Signature DWORD 类型，PE 文件签名，用来表示这是个 PE 文件，用 ASCII 码表示；

File Header 包含 PE 文件最基本信息，如：文件创建日期，文件类型，Section 的数量，Optional Header 的大小等等。详细可以参考 Winnt.h 里的结构 _IMAGE_FILE_HEADER。通过 dumpbin 工具可以看到，打开 Visual Studio 命令调试窗口，输入命令：

```

1 dumpbin -all "E:\OneDrive\Source Code\GitHubFosc\Fosc\Fosc.Dolphin.UI\Fosc.
Dolphin.UI\bin\x64\Debug\Fosc.Dolphin.UI.exe" >>C:\dump.txt

```

结果如图16.4所示。从这里可以看到：CPU 类型为 14c，是 Intel I386、I486 或者 I586；

section 的数量为 2; 链接器产生这个文件的日期; COFF 符号表的文件偏移量, 为 0; COFF 符号表的符号数目, 为 0; Optional Header 的大小。

```
Microsoft (R) COFF/PE Dumper Version 12.00.21005.1
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file E:\OneDrive\Source Code\Dolphin\GitHubFosc\Fo:
PE signature found
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
    14C machine (x86)
    3 number of sections
    56D1B66D time date stamp Sat Feb 27 22:45:01 2016
        0 file pointer to symbol table
        0 number of symbols
        E0 size of optional header
    102 characteristics
        Executable
        32 bit word machine
```

图 16.4: PE 文件 File Header

COFF –通用对象文件格式 (Common Object File Format), 是一种很流行的对像文件格式 (注意: 这里不说它是“目标”文件, 是为了和编译器产生的目标文件 (*.o/* .obj) 相区别, 因为这种格式不只用于目标文件, 库文件、可执行文件也经常是这种格式)。大家可能会经常使用 VC 吧? 它所产生的目标文件 (*.obj) 就是这种格式。其它的编译器, 如 GCC (GNU Compiler Collection)、ICL (Intel C/C++ Compiler)、VectorC, 也使用这种格式的目标文件。不仅仅是 C/C++, 很多其它语言也使用这种格式的对像文件。

3、PE loader 接着会找到 Entry Point, 例如本例中图16.5所示。

```
OPTIONAL HEADER VALUES
    10B magic # (PE32)
    11.00 linker version
    6000 size of code
    800 size of initialized data
    0 size of uninitialized data
    7FAE entry point (00407FAE)
    2000 base of code
    8000 base of data
    400000 image base (00400000 to 0040BFFF)
    2000 section alignment
    200 file alignment
```

图 16.5: PE 文件 Entry Point

这个 PE 文件的入口点地址为 00407FAE, 然后通过这个地址来查找.text section 的原始数据表, 由图 6 所示, 0040251E 这个地址开始的 6 个字节的内容为 【FF 25 00 20 40 00】，这个内容就是由编译器写入 PE 文件的.text section 的重要信息, FF 在 x86 汇编语言与机器码对照表中代表无条件转移指令 Jmp, 这条指令的作用是无条件跳转到 00402000 地址处, 从图 3 可以看到 image base 是 00400000, 2000 是 import address table 的 RVA 地址, 由图 7 可以看到, 此时程序会跳转到 00402000 这个地址所引用的 mscoree.dll 的 _CorExeMain(_CorExeMain 为 mscoree.dll 的入口方法) 方法, 所有的托管应用都会通过上述过程找到并执行 _CorExeMain 方

法；

Optional Header PE Optional Header 则包含了文件的版本号以及重要的基址和 AddressOfEntryPoint (RVA-Relative Virtual Address)，这是程序执行的入口地址，双击 exe 后就从这里开始执行。对 C# 程序来说，这里指向的是.NET 的核心库 MsCorEE.dll 的 _CorExeMain() 函数。当然这是针对 XP 系统的，XP 以后的系统，OS Loader 已经可以判断出这个 PE 是否包含 CLR 头来决定是否运行 MsCorEE.dll 的 _CorExeMain() 函数。

16.5.4 Section

Section 有很多，包括代码节，数据节等，C# 程序会把 CLR 头，元数据，IL 放在这里面。

16.6 CLR

CLR 是托管程序运行的环境，就像 Windows 是普通的 PE 程序的运行环境一样。在 Windows 中，整个 CLR 系统的实现基本其实就是几个关键的 DLL，比如 mscoree.dll、mscorjit.dll，它们共同的特点就是前缀均为 msco。在 win32 中，可执行文件在开始运行时，有操作系统载入到内存中，然后运行文件中的.text 代码，结束时有操作系统负责卸载。而在.NET 下，这个过程却不大一样。用 PE 结构查看工具（这里使用 PEiD）载入一个实例文件，观察导入表，可以发现整个表只引入了一个 mscoree.dll 中的一个方法：_CorExeMain。而这个方法，正是该可执行文件在 win32 意义上的入口点。托管 Assembly 本身只包含 CLR 可识别的 MetaData(元资料)，不包含机器指令。托管 Assembly 都与 mscoree.dll 绑定。mscoree.dll 在 system32 目录下，全称是 Microsoft Core Execution Engine。它的功能是选择合适的 CLR Execution Engine 来加载。事实上，类型安全 (Type Checker)、垃圾回收 (Garbage Collector)、异常处理 (Exception Manager)、向下兼容 (COM Marshaler) 等很多 C# 中的特性都是由 CLR 来提供的。

16.6.1 PE Loader 载入 PE 文件

当你双击一个.exe 文件时，Windows 操作系统提供的 PE Loader 会将该 exe 文件载入内存，exe 文件就是一种 PE 文件，PE(Portable Execute) 文件是微软 Windows 操作系统上的程序文件，EXE、DLL、OCX、SYS 文件以及 COM 组件都是 PE 文件；PE loader 通过查找 CLR 头发现该目录不为空，则自动将 mscoree.dll 载入进程地址空间中，mscoree.dll 一定是唯一的，且总是处于系统目录的 system32 下，例如我的机器为 C:/WINDOWS/system32 目录下。.net 2.0 的 mscoree.dll 的大小只有 256k 左右，这个 dll 被叫做 shim，它的作用是连接 PE 文件和 CLR 之间的一个桥梁。PE 文件的结构如图16.6所示：

当 mscoree.dll 加载后，它根据托管代码的 MetaData 和 app.config，选择恰当版本的引擎加载。同时 mscoree 还负责判断应该用何种 GC Flavor。GC Flavor 包括 Workstation GC 和 Server GC。在 CLR1 中，Workstation GC 对应到 mscorewks.dll，而 Server GC 对应到 mscoresvr.dll 文

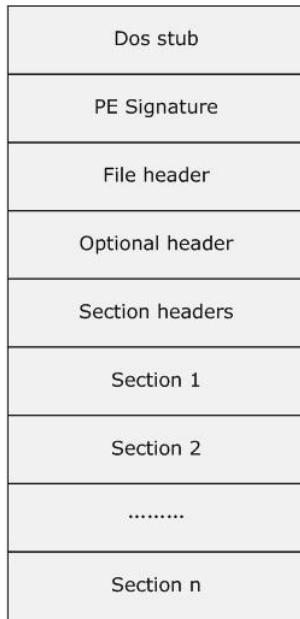


图 16.6: PE 文件的结构

件. 在 CLR2 中虽然保留了 mscoresvr.dll 文件, 但是 mscorewks.dll 已经包含了两种 GC Flavor 的实现, 只需要加载 mscorewks 就可以了.

CLR 加载后, 先初始化 CLR 需要的各种功能, 比如必要的全局变量, 引擎需要的模块 (ClassLoader, assembly Loader, JitEngine, Copntext 等), 启动 Finalizer thread 和 GC thread, 创建 System AppDomain 和 Shared AppDomain, 创建 RCDebugger Thread, 加载 CLR 基础类 (比如 mscorelib.dll, system.dll)

当 CLR 引擎初始化完成后, CLR 会找到当前 exe 的元数据, 然后找到 Main 函数, 编译 Main 函数, 执行 Main 函数.

16.6.2 启动 CLR 服务

_CorExeMain 方法会帮助程序找到并载入适当的 CLR 版本, 在.net 2.0 以后实现 CLR 的程序集为 mscorewks.dll 或 mscoresvr.dll, 例如, 在我的机器上 mscorewks.dll 的位置是:

1 C:\Windows\Microsoft.NET\Framework64\v2.0.50727

mscorewks.dll 是 dotNet 的核心文件, 尤其是在 net2.0 中, 以前分散的功能都集中到了这个 dll 中。NET 1.1 中, 还有一个文件 mscoresvr.dll 和 mscorewks.dll 是同等地位的。它们分别对应于 Windows Service 程序以及 desktop 程序。在 NET 2.0 中, 它们都统一到了 mscorewks.dll 中。同时在 NET 2.0 中 mscoresn.dll 的功能也合并到了 mscorewks.dll 中。它就是 DotNet 运行库的核心。DotNet 的执行引擎 (Execute Engine), 内部对象的实现都在这个 dll 里面。

在我们用 reflector 查看 dotnet 类库源代码时经常会遇到一些函数看不到源代码, 只是标记成内部实现。这些函数基本上实际实现的代码就在这个 dll 里面, 是 native 实现的。如反射功

能的相关对象以及实现就是这里面。

5、启动 CLR 服务，开始初始化工作，这个初始化工作包括：

- 分配一块内存空间，建立托管堆及其它必要的堆，由 GC 监控整个托管堆
- 创建线程池
- 创建应用程序域 (AppDomain)

利用 sos.dll 可以查看 CLR 创建了哪些 AppDomain。由图16.7可见，CLR 创建了 System Domain、Shared Domain 和 Domain1，这个 Domain1 是默认 Appdomain。

```

0:000> .load sos
0:000> !dumpdomain
-----
System Domain      000007fef105210
LowFrequencyHeap: 000007fef105790
HighFrequencyHeap: 000007fef105818
StubHeap:          000007fef1058a0
Stage:             OPEN
Name:              None

Shared Domain      000007fef104c30
LowFrequencyHeap: 000007fef105790
HighFrequencyHeap: 000007fef105818
StubHeap:          000007fef1058a0
Stage:             OPEN
Name:              None
Assembly:          000000000084ccb0 [C:\win
ClassLoader:        0000000000875340
Module Name:       000007fef7bd1000           C:\windows\Micro
                                         ...
-----
```

```

Domain 1           0000000000217da0
LowFrequencyHeap: 0000000000218558
HighFrequencyHeap: 00000000002185e0
StubHeap:          0000000000218668
Stage:             OPEN
Name:              Fosc.Dolphin.UI.exe
Assembly:          000000000084ccb0 [C:\win
ClassLoader:        0000000000875340
SecurityDescriptor: 000000000084d2b0
-----
```

图 16.7: Dump CLR AppDomain

6、接下来就会向默认 AppDomain 中载入 mscorelib.dll，由图16.8可见。

任何托管代码，CLR 在创建好默认 AppDomain 后，第一个载入的组件一定是 mscorelib.dll，根据命名，我的理解是 Microsoft Core Library（微软核心库）。实际上这个组件定义了 System.Object、所有基元类型：如 System.Int32 等，如图16.9所示。

利用 sos.dll(SOS Debugging Extension) 可以看到有哪些类被载入，依据 Domain 1 里的 Module 地址，在 WinDbg 窗口敲入命令!dumpmodule -mt 000007fef7111000，其中 000007fef7111000 为 Mudule 的地址，在!dumpdomain 输出的 Assembly 中。结果比较长，只列出部分。

7、产生主线程后可能会触发一些 mscorelib.dll 里的类型并加载入内存，接着，当你的 PE 文件：hello.exe 被载入后，默认 Appdomain 的名字被改为你的 PE 文件的名字，载入过程完成后的结果可见图 8。

8、包含在 mscorewks.dll 中的 _CorExeMain2 方法接管主线程，_CorExeMain2 的代码如下：代码中 GetEntryPoint 将通过 MetaData 表提供的接口查找包含.entrypoint 的类型，接着返回入口方法（在 C# 中这个入口方法一定是 Main 方法）的一个 MethodDesc 类型的实例，获取 MethodDesc 类型实例的这个过程我认为是：CLR 通过读取元数据表，定位入口方法所属的类型，建立这个类型的 EECCLASS(EECLASS 结构中包含重要信息有：指向当前类型父类的指

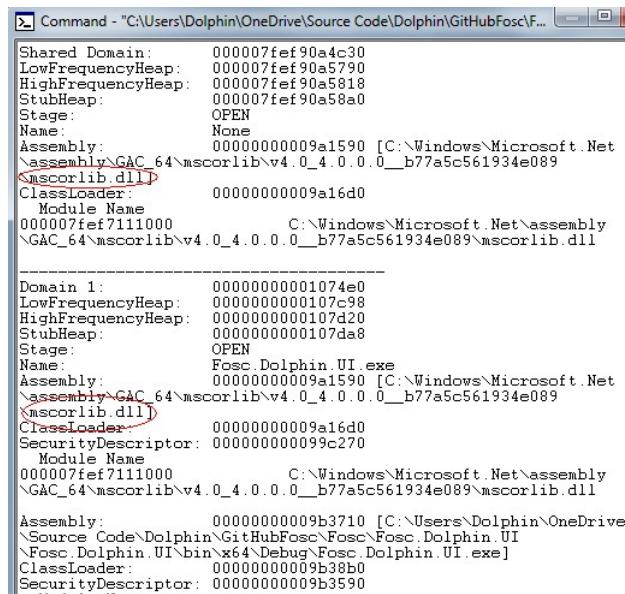


图 16.8: Load Compenet mscorelib

```

0:000> !dumpmodule -mt 000007fef7111000
Name: C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0_4.0.0.0_b77a5c561934e089
Attributes: PEFile
Assembly: 00000000009a1590
LoaderHeap: 00000000000000000000
TypeDefToMethodTableMap: 000007fef851c83c
TypeRefToMethodTableMap: 000007fef740e4e8
MethodDefToDescMap: 000007fef851ea54
FieldDefToDescMap: 000007fef740e548
MemberRefToDescMap: 0000000000000000
FileReferencesMap: 000007fef71288e0
AssemblyReferencesMap: 000007fef7128910
MetaData start address: 000007fef798192c (2755936 bytes)

Types defined in this module

```

MT	TypeDef	Name
000007fef77f13e8	0x02000002	System.Object
000007fef7803340	0x02000003	System.Runtime.Serialization.ISerializable
000007fef77fff10	0x02000004	System.Runtime.InteropServices._Exception
000007fef77f1038	0x02000005	System.Exception
000007fef77c7cb8	0x02000006	System.Exception+__RestrictedErrorObject
000007fef77f3390	0x02000007	System.ValueType
000007fef7800a68	0x02000008	System.IComparable
000007fef77f61b8	0x02000009	System.IFormattable
000007fef77fd48	0x0200000a	System.IConvertible
000007fef77f33d8	0x0200000b	System.Enum

图 16.9: CLR Dump Module

针、指向方法表的指针、实例字段和静态字段等) 和这个类型所包含方法的 Method Table(方法表由一个个 Method Descriptor 组成, 具体到内存中就是指向若干 MethodDesc 类型实例的地址), 通过 EEClass::FindMethod 方法找到并返回入口方法的 MethodDesc 类型实例。最后通过 ClassLoader::RunMain 来执行入口方法。

16.6.3 进入 Main 方法

9、进入 Main 方法, 进而执行后续程序。最后, 从上述分析也可以看出, .NET 的几个核心组件的被调用顺序大致是: mscoree.dll -> mscorwks.dll(mscorsvr.dll) -> msclib.dll -> msclijit.dll。一般来说调试.NET 程序使用 VS2005 就可以了, 但是要想得到更详细的信息, 如内存情况等就需要借助其他工具了, 个人觉得 sos.dll 和 Windbg² 是很好的工具。而如果你装的是 VS2005 Team Version, 那么自带 sos.dll。

关于 CLR 加载过程的详细内容, 大家可以通过微软的 Shared Source Common Language Infrastructure(SSCLI)³, 来了解关于 CLR 的一些内部机理。相信会对理解 CLR 有所帮助, 另外就是由蔡学镛写的 <http://www.microsoft.com/taiwan/msdn/columns/DoNet/loader.htm>, 文章挺早, 但很经典, 大家可以看看。

16.7 Metadata

IL 和 Metadata 是.NET PE (Portable Execute) 档案内的两大重点。Metadata 叙述静态的结构, IL 叙述动态的程式 [3]。Metadata 就像是一个资料库, 包含许多 table, 且 table 之间可能会互相参照 (reference), 也就是类似关联式资料库的 Foreign Key。目前, .NET PE 档的 Metadata 中可以使用的 table 有 44 种。.NET 使用 Metadata 的主要目的是实现组件的高度复用性, Metadata 非常占用空间, 一般占到整个 EXE/DLL 总大小的 50% 到 70%, 使用 ildasm.exe 查看 Metadata 占用空间比例, 打开 dll 文件, 在 View->Statistics 下, 如图16.10所示。

在.NET 平台中 Metadata 的存在绝对不仅仅只是支持反射。实际上, 支持反射只是 Metadata 一个很小的用武之地, Metadata 是整个.NET 平台非常核心的基础支撑设施, 它在.NET 平台中有着广泛的应用, 是.NET 平台的灵魂⁴。

.NET 创建之前, Windows 平台的软件技术经历了以下几个阶段: DDE、OLE、ActiveX/-COM。这些技术所努力解决的核心目标是: 软件组件的复用性——这是软件开发的核心命题, 是软件平台厂商竞逐的焦点。“软件组件的复用性”有以下几个含义:

1. 强调二进制级的复用 (黑盒复用), 而不是源代码级的复用 (白盒复用)。例如: 我想在我的软件中集成 PDF 阅读器的能力, 我不需要找 Adobe 要 PDF 阅读器的源代码, 而是去找 Adobe 要一个支持 PDF 阅读的二进制组件——然后通过接口的方式来使用它。2. 强调多语言

²Windbg 可以在 <http://www.microsoft.com/whdc/devtools/debugging/default.mspx> 下载

³下 载 地 址: <http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en>

⁴详细请参考: <http://m.blog.csdn.net/blog/u013885963/24008399>

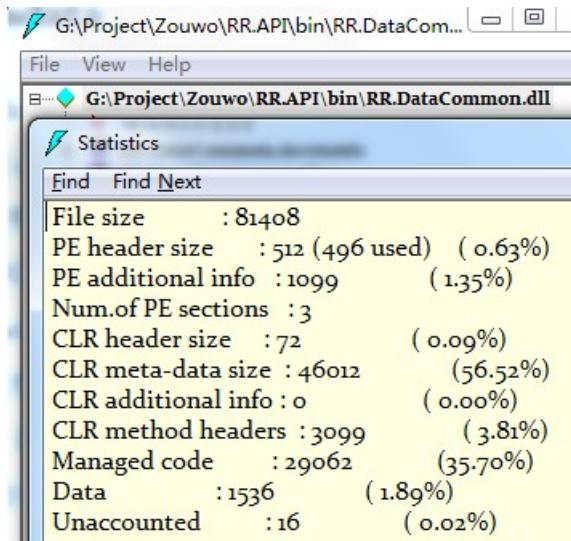


图 16.10: 使用 ildasm 查看 Metadata 大小

创建的组件之间的复用，而不是单一语言创建的组件之间的复用。例如：我在 C++ 中写一个 Email 收发组件，可以在 VB 中直接使用。

DDE、OLE、ActiveX/COM 无一不是围绕这个目标。COM 是这些技术发展进程中的一个高峰。但是 COM 技术本身在发展过程中暴露出了很多问题。Don Box 在《Essential .NET》一书的第一章 “The CLR as a Better COM” 对 COM 的问题有非常深刻的解剖。

首先，要达致“组件复用”这一目标，必须有合同来规范组件与执行环境、组件与组件之间的各种约定。COM 使用的是 IDL 或 TLB 作为组件合同，但它们有几个根深蒂固的缺点：

1. IDL/TLB 规范的是物理语义（例如 v-table 偏移，栈帧，参数，数据结构，对齐等内存细节），且使用的是 C++ 语言的一个子集约定。
2. IDL/TLB 规范描述与组件代码本身分离。
3. IDL/TLB 规范较为混乱，没有达成业界统一的标准（第三方难以扩展开发）

16.8 JIT(MIT Licence)

MSIL 是将.NET 代码转化为机器语言的一个中间过程。它是一种介于高级语言和基于 Intel 的汇编语言的伪汇编语言。当用户编译一个.NET 程序时，编译器将源代码翻译成 Microsoft 中间语言 (MSIL)，它是一组可以有效地转换为本机代码且独立于 CPU 的指令。当执行这些指令时，实时 (JIT) 编译器将它们转化为 CPU 特定的代码。即时编译 (Just In-Time compile)，这是.NET 运行可执行程序的基本方式，也就是在需要运行的时候，才将对应的 IL 代码编译为本机指令。传入 JIT 的是 IL 代码，输出的是本机代码，所以部分加密软件通过挂钩 JIT 来进行 IL 加密，同时又保证程序正常运行。JIT(Just In-Time compile) 编译针对的对象总是方法，不论是入口方法还是其他方法的 JIT 过程都类似上述过程，Metadata 这里的作用不言而喻，可以说没有 Metadata 的支持就无法进行 Jit，Metadata 在 JIT 编译期间的作用至少有三个：

- 1、JIT 编译器通过查找 Metadata 来找到入口方法；

2、JIT 编译器通过查找 Metadata 来定位待编译方法并利用其 RVA(Relative Virtual Address) 找到存储于 PE 文件中的 IL 代码在内存中的实际地址；

3、JIT 编译器在找到 IL 代码并准备编译为本地 CPU 指令前所进行的 IL 代码验证同样会用到 Metadata，例如，验证方法的合法性需要去核实方法参数数量是正确的、传给方法的每个参数是否都有正确的类型、方法返回值是否正确等等。同解释执行的代码相比，JIT 的执行效率要高很多。

16.8.1 类型构造器 (Class Constructor)

类型构造器也称为静态构造器，类构造器，或类型初始化器，实例构造器用来设置一个类型某个实例的初始化状态，类型构造器用来设置一个类型的初始化状态。默认情况下，类型没有定义类型构造器。当 JIT 编译器编译一个方法时，它会检查在代码里面是否引入了其他类型。如果引入的的其他类型定义了类型构造器，则 JIT 会检测是否已经在 AppDomain 里面执行过。如果没有执行，则发起对类型构造器的调用，否则不调用。在编译之后，线程会开始执行并最终获取调用构造器的代码。实际上有可能会是多个线程执行同一个方法，CLR 想要确保一个类型构造器在一个 AppDomain 里面只执行一次，当一个类型构造器被调用时，调用的线程会获取一个互斥的线程同步锁，这时如果有其他的线程在调用，则会阻塞。当第一个线程执行完后离开，其他的线程被唤醒并发现构造器的代码执行过了，所以不会继续去执行了，从构造器方法返回。CLR 通过这种方式来确保构造器仅仅被执行一次。如果在声明静态字段时同时对其赋值，它在编译后会被搬到cctor 中，并且是放在前面，然后才到显式定义的静态构造方法体中的代码，也就是说 count 在这里会被赋值两次，第一次 50，第二次 100。在反编译 IL 查看时，.cctor() 指定就是构造器，而.ctor() 就是类的构造方法。

16.9 FCL(MIT Licence)

.net Framework Class Library(FCL),the .NET Framework class library is a library of classes, interfaces, and value types that provides access to system functionality and is designed to be the foundation on which .NET Framework applications, components, and controls are built.

16.10 GC(Garbage Collection)

16.10.1 为什么要 GC

有了 Microsoft.Net clr 中的垃圾回收机制程序员不需要再关注什么时候释放内存，释放内存这件事儿完全由 GC 做了，对程序员来说是透明的。尽管如此，作为一个.Net 程序员很有必要理解垃圾回收是如何工作的。这篇文章我们就来看下.Net 是如何分配和管理托管内存的，之后再一步一步描述垃圾回收器工作的算法机制。

为程序设计一个适当的内存管理策略是困难的也是乏味的，这个工作还会影响你专注于解

决程序本身要解决的问题。有没有一种内置的方法可以帮助开发人员解决内存管理的问题呢？当然有了，在.NET 中就是 GC，垃圾回收。

让我们想一下，每一个程序都要使用内存资源：例如屏幕显示，网络连接，数据库资源等等。实际上，在一个面向对象环境中，每一种类型都需要占用一点内存资源来存放他的数据，对象需要按照如下的步骤使用内存：1. 为类型分配内存空间 2. 初始化内存，将内存设置为可用状态 3. 存取对象的成员 4. 销毁对象，使内存变成清空状态 5. 释放内存

这种貌似简单的内存使用模式导致过很多的程序问题，有时候程序员可能会忘记释放不再使用的对象，有时候又会试图访问已经释放的对象。这两种 bug 通常都有一定的隐藏性，不容易发现，他们不像逻辑错误，发现了就可以修改掉。他们可能会在程序运行一段时间之后内存泄漏导致意外的崩溃。事实上，有很多工具可以帮助开发人员检测内存问题，比如：任务管理器，System Monitor AcitvieX Control，以及 Rational 的 Purify。

而 GC 可以完全不需要开发人员去关注什么时候释放内存。然而，垃圾回收器并不是可以管理内存中的所有资源。有些资源垃圾回收器不知道该如何回收他们，这部分资源就需要开发人员自己写代码实现回收。在.NET Framework 中，开发人员通常会把清理这类资源的代码写到 Close、Dispose 或者 Finalize 方法中，稍后我们会看下 Finalize 方法，这个方法垃圾回收器会自动调用。

不过，有很多对象是不需要自己实现释放资源的代码的，比如：Rectangle，清空它只需要清空它的 left, right, width, height 字段就可以了，这垃圾回收器完全可以做。下面让我们来看下内存是如何分配给对象使用的。

16.10.2 对象分配过程

16.10.3 GC 回收原理

为程序设计一个适当的内存管理策略是困难的也是乏味的，这个工作还会影响你专注于解决程序本身要解决的问题。有没有一种内置的方法可以帮助开发人员解决内存管理的问题呢？当然有了，在.NET 中就是 GC，垃圾回收 (Garbage Collection)。GC 会把所有托管堆内的对象分为 3 代，0 代、1 代、和 2 代。越小的代拥有着越多被释放的机会，CLR 的基本算法是：每执行 N 次 0 代的回收，才会执行 1 次 1 代的回收，而每执行 N 次 1 代的回收才会执行一次 2 代的回收。当某个对象在 GC 执行时被发现仍然在使用，它将被移动到下一代上，新分配的对象实例属于第 0 代。根据.NET 的垃圾回收机制，0 代 1 代 2 代的初始分配空间分别为 256K、2MB 和 10MB。避免保留已经不再使用的对象，把对象的引用设置为 null。当没有对象引用指向堆中某个对象实例时，这个对象就将被视为不再使用。第一个就是很多人用.NET 写程序，会谈到托管这个概念。那么.NET 所指的资源托管到底是什么意思，是相对于所有资源，还是只限于某一方面资源？很多人对此不是很了解，其实.NET 所指的托管只是针对内存这一个方面，并不是对于所有的资源；因此对于 Stream，数据库的连接，GDI+ 的相关对象，还有 Com 对象等等，这些资源并不是受到.NET 管理而统称为非托管资源。而对于内存的释放和回收，系统提供了 GC-Garbage Collector，而至于其他资源则需要手动进行释放。

那么第二个概念就是什么是垃圾，通过我以前的文章，会了解到.NET 类型分为两大类，一

个就是值类型，另一个就是引用类型。前者是分配在栈上，并不需要 GC 回收；后者是分配在堆上，因此它的内存释放和回收需要通过 GC 来完成。GC 的全称为“Garbage Collector”，顾名思义就是垃圾回收器，那么只有被称为垃圾的对象才能被 GC 回收。也就是说，一个引用类型对象所占用的内存需要被 GC 回收，需要先成为垃圾。那么 .Net 如何判定一个引用类型对象是垃圾呢，.Net 的判断很简单，只要判定此对象或者其包含的子对象没有任何引用是有效的，那么系统就认为它是垃圾。

明确了这两个基本概念，接下来说说 GC 的运作方式以及其的功能。内存的释放和回收需要伴随着程序的运行，因此系统为 GC 安排了独立的线程。那么 GC 的工作大致是，查询内存中对象是否成为垃圾，然后对垃圾进行释放和回收。那么对于 GC 对于内存回收采取了一定的优先算法进行轮循回收内存资源。其次，对于内存中的垃圾分为两种，一种是需要调用对象的析构函数，另一种是不需要调用的。GC 对于前者的回收需要通过两步完成，第一步是调用对象的析构函数，第二步是回收内存，但是要注意这两步不是在 GC 一次轮循完成，即需要两次轮循；相对于后者，则只是回收内存而已。那么对于程序资源来说，我们应该做些什么，以及如何去做，才能使程序效率最高，同时占用资源能尽快的释放。前面也说了，资源分为两种，托管的内存资源，这是不需要我们操心的，系统已经为我们进行管理了；那么对于非托管的资源，这里再重申一下，就是 Stream，数据库的连接，GDI+ 的相关对象，还有 Com 对象等等这些资源，需要我们手动去释放。

如何去释放，应该把这些操作放到哪里比较好呢。.Net 提供了三种方法，也是最常见的三种，大致如下：1. 析构函数；2. 继承 IDisposable 接口，实现 Dispose 方法；3. 提供 Close 方法。

经过前面的介绍，可以知道析构函数只能被 GC 来调用的，那么无法确定它什么时候被调用，因此用它作为资源的释放并不是很合理，因为资源释放不及时；但是为了防止资源泄漏，毕竟它会被 GC 调用，因此析构函数可以作为一个补救方法。而 Close 与 Dispose 这两种方法的区别在于，调用完了对象的 Close 方法后，此对象有可能被重新进行使用；而 Dispose 方法来说，此对象所占有的资源需要被标记为无用了，也就是此对象被销毁了，不能再被使用。例如，常见 SqlConnection 这个类，当调用完 Close 方法后，可以通过 Open 重新打开数据库连接，当彻底不用这个对象了就可以调用 Dispose 方法来标记此对象无用，等待 GC 回收。明白了这两种方法的意思后，大家在往自己的类中添加的接口时候，不要歪曲了这两者意思。

GC 为了提高回收的效率使用了 Generation 的概念，原理是这样的，第一次回收之前创建的对象属于 Generation 0，之后，每次回收时这个 Generation 的号码就会向后挪一，也就是说，第二次回收时原来的 Generation 0 变成了 Generation 1，而在第一次回收后和第二次回收前创建的对象将属于 Generation 0。GC 会先试着在属于 Generation 0 的对象中回收，因为这些是最新的，所以最有可能会被回收，比如一些函数中的局部变量在退出函数时就没有引用了（可被回收）。如果在 Generation 0 中回收了足够的内存，那么 GC 就不会再接着回收了。

C# 的垃圾回收采用了 reference tracking 的算法，大概的意思是说在执行垃圾回收时，所有的对象都默认认为是可以被回收的，然后遍历所有的 roots（指向 reference type 的对象，包括类成员变量，静态变量，函数参数，函数局部变量），把这个 root 指向的对象标记成不能被回收的。

16.10.4 GC 触发时机

在讨论什么时候回收大对象之前先来看下普通的垃圾回收操作什么时机执行吧。垃圾回收在下列情况下发生：

1. 申请的空间超过 0 代内存大小或者大对象堆的阀值，多数的托管堆垃圾回收在这种情况下发生
2. 在程序代码中调用 GC.Collect 方法时；如果在调用 GC.Collect 方法是传入 GC.MaxGeneration 参数时，会执行所有代对象的垃圾回收，包括大对象堆的垃圾回收
3. 操作系统内存不足时，当应用程序收到操作系统发出的高内存通知时
4. 如果垃圾回收算法认为做二代回收是有收效时会触发二代垃圾回收
5. 每一代对象堆的都有一个所占空间大小阀值的属性，当你分配对象到某一代，你增长了内存总量接近了该代的阀值，或者分配对象导致这一代的堆大小超过了堆阀值，就会发生一次垃圾回收。因此当你分配小对象或者大对象时，会对应消耗 0 代堆或者大对象堆的阀值。当垃圾回收器将对象代数提升到 1 代或者 2 代时，会消耗 1、2 代的阀值。在程序运行中这些阀值是动态变化的。

16.10.5 LOH 堆

CLR 垃圾回收器根据所占空间大小划分对象。大对象和小对象的处理方式有很大区别。比如内存碎片整理——在内存中移动大对象的成本是昂贵的。在.NET 1.0 和 2.0 中，如果一个对象的大小超过 $85000\text{byte} = 10625\text{Byte} \approx 10\text{KB}$ ，就认为这是一个大对象。这个数字是根据性能优化的经验得到的。当一个对象申请内存大小达到这个阀值，它就会被分配到大对象堆上。http://www.cnblogs.com/yukaizhao/archive/2011/11/21/dot_net_gc_large_object_heap.html

16.11 IL(Intermediate Language)

MSIL 是将.NET 程序编译成机器语言的一种过程。编译成的代码不专用于任何一种操作系统，它是一种介于高级语言和基于 Intel 的汇编语言。查看 IL 代码可以通过工具 IL Disassembler(IL DASM) 进行，图标含义如图16.11所示：

每当编译程序，编译器将源代码翻译成 MSIL，它是一组可以有效地转换为本机代码且独立于 CPU 指令。当执行这些指令时 JIT 将它转换为 CPU 的特定代码。由于 MSIL 支持多种 JIT，所以同一 MSIL 代码可以被编译运行在不同的结构上。MSIL 包括用于加载、存储和初始化对象以及对对象调用方法的指令，还有控制流内存直接访问、异常逻辑运算。再交由 JIT 编译器转换为特定 CPU 的代码。当编译器产生 MSIL 时，它也产生元数据。MSIL 和元数据包括在一个可移植可执行的 PE 文件中。执行期间 MSIL 会被编译成本地代码，这有点像解释性编译语言，但却与解释性编译有很大不同，就在于会重用先前编译的本地代码，在每一次执行代码

符号	含义
▶	更多信息
▣	命名空间
■	类
▢	接口
▢	值类
▢	枚举
▢	方法
▢	静态方法
▢	字段
▢	静态字段
▢	事件
▲	属性
▶	清单或类信息项

图 16.11: 使用 IL DASM 查看 IL

均会查看是否已经编译过，若没有编译成本地代码，则执行一次编译，若编译过，则重用先前的本地代码。部分 CIL 操作码如下（参考这里：<http://www.cnblogs.com/zery/p/3368460.html>）：

操作码	作用
add, sub, mul, div, rem	用于在两个值上进行二进制操作
add, or, not, xor	用于两个数加减乘除求模
ceq, cgt, clt	用不同的方法比较两个在栈上的值，ceq：是否相等；cgt：是否大于；clt：是否小于
box, unbox	在引用类型和值类型之间转换
ret	退出方法和返回一个值
beq, bgt, ble, blt, switch	控制方法中的条件分支，beg：如果相等就中止到代码标签；bgt：如果大于就中止到代码标签；ble：如果小于等于就中止到代码标签；blt：如果小于就中止到代码标签；所有的分支控制操作码都需要给出一个 CIL 代码标签作为条件为真的跳转目的
br.s	(无条件) 中止到代码标签
call	调用一个成员
nearer, newobj	在内存中创建一个新的数组或新的对象类型

主要的入栈 CIL 操作码 (ld 加载)

操作码	作用
ldarg (及多个变化形式)	加载方法的参数到栈中。除了泛型 ldarg(需要一个索引作为参数), 还有后其他很多的变化形式 eg. 有个数字后缀的 ldarg 操作码来指定需要加载的参数。同时还有很多 ldarg 的变化形式允许加载指定的数据类型 (ldarg_i4, 加载 int32) 和值 (ldarg_i4_5 加载一个值为 5 的 int32)
ldc (及多个变化形式)	加载一个常数到栈中 (load const)
ldfld (及多个变化形式)	加载一个对象实例的成员到栈中
ldloc (及多个变化形式)	加载一个本地变量到栈中
ldobj	获得一个堆对象的所有数据, 并将它们放置到栈中
ldstr	加载一个字符串数据到栈中

主要的弹出栈操作码 (st 存储)

操作码作用

pop 删除当前栈顶的值, 但是并不影响存储的值

starg 存储栈顶的值到给出方法的参数, 根据索引确定这个参数

stloc (及多个变化形式) 弹出当前栈顶的值并存储在一个本地变量列表中, 根据所以确定这个参数

stobj 从栈中复制一个特定的类型到指定的内存地址

stfld 用从栈中获得的值替换对象成员的值

16.11.1 简单的 IL 示例

```

1 .class private auto ansi beforefieldinit ConsoleApplication1.Program
2 extends [mscorlib]System.Object
3 {
4     // Methods
5     .method private hidebysig static
6         void Main (
7             string[] args
8         ) cil managed
9     {
10         // Method begins at RVA 0x2050
11         // Code size 4 (0x4)
12         .maxstack 1
13         .entrypoint
14         .locals init (
15             [0] int32 i
16         )
17
18         IL_0000: nop
19         IL_0001: ldc.i4.1

```

```

20     IL_0002: stloc.0
21     IL_0003: ret
22 } // end of method Program::Main
23
24 .method public hidebysig specialname rtspecialname
25 instance void .ctor () cil managed
26 {
27     // Method begins at RVA 0x2060
28     // Code size 7 (0x7)
29     .maxstack 8
30
31     IL_0000: ldarg.0
32     IL_0001: call instance void [mscorlib]System.Object::ctor()
33     IL_0006: ret
34 } // end of method Program::ctor
35
36 } // end of class ConsoleApplication1.Program

```

对于同样的 IL，JIT 会把它编译为不同的 CPU 架构（如 x86/IA64 等等）生成不同的机器码。.entrypoint //该指令代表该函数程序的入口函数。每一个托管应用程序都有且只有一个入口函数，CLR 加载程序时，首先从.entrypoint 函数开始执行。.maxstack 2 //执行构造函数时，评估堆栈可容纳数据项的最大个数。评估堆栈是保存方法中所需要变量的值的一个内存区域，该区域在方法执行结束时会被清空，或者存储一个返回值。//hidebysig 指令表示如果当前类为父类，用该指令标记的方法将不会被子类继承。

16.11.2 属性 (Attribute)

本质上，属性和 Get/Set 实现方法没有区别，C# 编译器在编译时，会把属性转换成一对 GetXXX 和 SetXXX 方法，属性的设计是代替暴露的内部成员，任何定义公开的内部成员，都应该设计成 private 或者 protected，并且配上一个属性，属性相对于公共变量，提高了程序的可扩展性。

16.11.3 编辑 DLL(Edit Dynamic Linkable Library)

使用 ILDasm.exe(IL Disassembler) 打开 DLL。转储 DLL 文件，如图16.12所示。

用记事本打开转储后的文件(中间语言文件，文件后缀是 il)，编辑后保存，使用 ilasm.exe 工具编译成 dll，ilasm.exe 的路径为：C:/Windows/Microsoft.NET/Framework64/v4.0.30319，ILAsm (IL Assembler) generates a portable executable (PE) file from Common Intermediate Language (CIL) code. 新生成的 dll 即是修改后的 dll，可以使用 ILSpy 等工具打开新生成的 dll 验证修改是否已经生效，编译命令如图16.13所示。

命令为：

```
1 ilasm.exe /dll /output=E:\dll\A.dll /Resource=E:\dll\b.res E:\dll\b.il
```

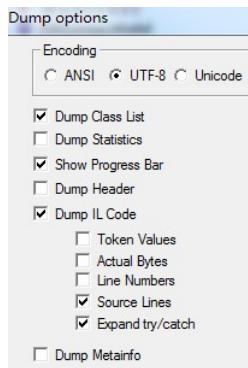


图 16.12: Dump dll

```

s CLASS 23      Props: 20;
Class 24      Props: 9;
Class 25      Props: 9;
r Class 26      Props: 6;
Class 27      Props: 29;
Resolving local member refs: 0 -> 0 defs, 0 refs, 0 unresolved
Writing PE file
Operation completed successfully

C:\Windows\Microsoft.NET\Framework64\v4.0.30319>ilasm.exe /dll /output=G:\a.dll
/Resource=G:\DBSystem.Model.res G:\DBSystem.Model.il

```

图 16.13: 使用 ilasm(MSIL Assembler) 编译 IL

dll 参数表示编译成 dll 文件，如果不指定参数，默认编译成 exe 文件。除了用微软提供的 dumpbin 工具编辑 IL 之外，也可以使用 dnSpy⁵工具编辑 dll。使用 dnSpy 工具打开 dll 之后，在面板中点击右键选择“Edit IL Instruction(编辑 IL 指令)”即可修改 IL 代码，修改完成后点击 OK，导出 dll 即可。

16.12 调试 (Debug)

如果使用的是 Visual Studio 2013，则 Visual Studio 中的 Windows 调试器支持 SOS.dll，但 Visual Studio 调试器的“即时”窗口不支持 SOS.dll。Visual Studio 在调试的过程中可以修改代码，可以拖动调试断点。调试器可以采用三种模式来调试被调试程序（在下文中，如果没有特别说明的话，简称程序）：

直接调试模式 即直接从调试器里面启动程序，就如同我们在 Visual Studio 里面按下 F5 就可以调试程序那样。

附加 (attach) 模式 即你可以在程序已经启动的情况下，把你的调试器附加到程序上，进行调试。这种模式通常在调试服务 (Service) 程序非常有用，例如你的 ASP.NET 网站可能会存在这样一种 bug，在网站运行的过程当中，有一个异常不知道从那个角落里面抛了出来，这时你可以使用附加模式来调试你的网站。

⁵源码托管地址：<https://github.com/0xd4d/dnSpy>

验尸 (post-mortem) 调试 这种调试模式允许你调试在客户机器上出错的程序。也就是说，当你的程序发布给客户以后，客户在使用程序的过程中，可能会碰到一些很难重现的错误 (bug)。这时操作系统可以将出错的程序的内存保存到一个文件里面，然后你可以在自己的开发机上调试这个文件，找到程序错误的原因。

16.12.1 调试器工作原理

在 Windows 平台上调试应用程序首选 Windows 调试工具箱，该工具箱包含了一套专门用来针对 Windows 进行很多复杂场景调试所需要的工具和组件。需要注意的是此工具箱是针对于非托管.NET 平台用的，意思就是说此工具箱的所有工具和组件默认是不能够进行.NET 应用程序调试的，只能用来对原生 Windows 程序进行调试。

那么.NET 平台也并不是有自己一套专用的调试工具箱，毕竟.NET 还是属于 Windows 平台的，所以很大部分的运行时原理还是基于 Windows 的，要想在原生的调试器中对.NET 这个具有虚拟运行时程序进行调试就需要专门的翻译器才能够执行。SOS.DLL、SOSEX.DLL 这两个就是用来对.NET 程序在 Windows 调试工具中起到翻译作用的调试器扩展。简单讲就是，这两个组件是.NET 项目组专门开发出来用来对.NET 应用程序进行方便调试用的，当然不用这两个扩展也能调试.NET 程序，只不过就会很困难，会被很多细节束缚住。有了这个调试扩展之后，我们就可以让原生 Windows 调试器正确的翻译出.NET 相关概念。

16.12.2 符号文件 (Symbol File)

其实调试器所做的一系列的魔术，都是和符号文件分不开的。微软的 Visual Studio 的符号文件以.pdb 为后缀名，当你在 Visual Studio 当中编译好解决方案后，可以在与编译出来的程序的相同文件夹里找到其对应的符号文件，默认情况下，文件名和程序名相同。

符号文件其实是编译器生成的，因此你也可以使用 C# 的编译器 csc 为你的程序生成符号文件。Csc 程序的/debug 开关可以用来控制符号文件的生成，如下表所示：

编译选项	说明
/debug:pdbonly	使用这个开关，生成的符号文件允许你在直接调试模式当中获得源代码级别的调试支持，但是在附加模式当中，调试器只会显示汇编代码。
/debug:full	这个开关生成的符号文件，在三种调试模式当中，都支持源代码级别的调试。因此我推荐读者在编译程序的时候，总是打开这个开关。

具体的设置在 Visual Studio 项目属性的生成选项卡高级生成设置的属性中。

16.12.3 符号文件服务器

由于符号文件包含了二进制程序和源代码文件之间的对应关系，因此每次程序升级以后，你可能会同时有多个版本的程序在不同的客户机上运行。而在调试的时候，手工寻找正确版本的

符号文件肯定是一个非常费力的事情。因此就有了符号文件服务器软件的出生，符号文件服务器的工作就是，当你在调试程序的时候，调试器会自动和符号文件服务器交互，获取正确版本的符号文件。在后面的章节里面，会介绍如何创建一个符号文件服务器，以及如何使用符号文件服务器。

16.12.4 常用调试技巧

编辑然后继续运行 (Edit and Continue) 在运行一个很复杂的程序和插件时，发现一个错误，但是不想浪费时间去重编译重启程序。只要在这个位置修改这个 bug，然后继续调试。Visual studio 会修改这个程序，使得你可以继续调试而不需要重启程序。开启很简单，打开“工具”→“调试”→“编辑并继续”→勾选启用“编辑并继续”即可。值得注意的是“编辑然后继续运行”这个功能有几个限制。一，它不能在 64 位代码上使用。如果想使用这个功能，到项目设置里的编译选项，选择 x86 作为目标平台。不要担心，这目标平台在 reslease 配置是和 debug 是分离的，也就是说依然是 Any CPU 的设置。二，“编辑然后继续运行”这个功能仅适用于一个函数内部改变。如果你想要改变这个函数的声明或者增加新的方法，你只能选择重启程序，或者不做任何改变继续。如果修改的方法中包含 lambda 表达式，则意味着修改了编译器自动生成的委托类型，这样会导致编译器停止运行。在调试时修改源代码提示：“启用非托管调试时不允许更改”，检查项目设置→Web→ 调试器选择项中是否勾选了“本机代码选项”，如果已经勾选，则将之去掉。

运行到光标处 (Run to Cursor–Ctrl+F10) 有的时候我们打的断点可能与实际需要关心的代码行有较远的距离，当我们需要运行到实际关心的代码行时，如果单击 F10/F11 就不是最好的选择，此时只需要按 Ctrl+F10 即可。

断点标签 (Breakpoint Label) 这是 VS2010 提供的新特征 (feature)。用于更好的管理断点。它使得我们能够更好的分组和过滤断点。这像是对断点的归类。如果我们添加了与某一功能相关不同的不同类型的断点，我们可以根据需要使能 (enable)、取消 (disable)、过滤 (filter) 这些断点。断点标签在你调试大量代码，多个工程等情况下非常有用。

设置下一语句 (Set Next Statement) 这是一个非常有趣的特性。设置下一语句允许你在调试的时候改变程序的执行路径。如果你的程序在某一行处暂停而且你想改变执行路径，跳到指定行，在这一行上右击，在右击菜单中选择“设置下一语句”。这样程序就会转到哪一行执行而不执行先前的代码。这在如下情况中非常有用：当你发现代码中某些行可能会导致程序的中断 (break) 而你不想让程序在那个时候中断。快捷键是 Ctrl + Shift + F10，将光标移动到需要跳转到的行上，按下快捷键即可。也可以直接拖动左侧的黄色箭头到指定行，也可以达到同样的效果。

显示下一语句 (Show Next Statement[Ctrl+*]) 这一行用黄色箭头标记。这行是程序继续执行时下一条将执行的语句。例如此时已经切换到另外的代码视图检视了部分源码，但是不记

得当前断点运行到哪里了，可以使用显示下一语句的功能回到当前调试的位置，当然也可以直接按 F10 回到调试位置，除了这个用处，实在想不出显示下一语句有什么用途。

Pinned DataTips Visual Studio 2010 also includes some nice new “DataTip pinning” features that enable you to better see and track variable and expression values when in the debugger.

Simply hover over a variable or expression within the debugger to expose its DataTip (which is a tooltip that displays its value) – and then click the new “pin” button on it to make the DataTip always visible:

```

var controllerName = string.Empty;
var actionName = string.Empty;
try
{
    var requestUrlObj = filterContext.HttpContext.Request.
        if (requestUrlObj {http://localhost:4961/})
    var userId = LoginService.GetUserId();
    controllerName = (string) filterContext.RouteData.Values["controller"];
    actionName = (string) filterContext.RouteData.Values["action"];
}

```

图 16.14: DataTips Pin

启用非托管代码调试 Visual Studio 作为一种强大的开发平台，已经提供了非常多的调试手段。但这些调试手段相对来说还是停留在表面上，无非是设置断点、变量查看以及调用堆栈列表等。某些时候我们希望了解更多的东西，尤其是那些被隐藏到背后和运行期的东西，诸如对象运行状态、内存分布等等，这些相对底层的知识可以让我们更好地理解.NET CLR/JIT 的一些行为。当然，并不是所有人都需要了解这些知识，毕竟汇编和高级调试器使用起来还是非常麻烦的。SOS.dll 是 Microsoft 提供的一种调试扩展，全称是 Son of Strike，可用来调试托管代码。SOS.dll 拥有非常强大的功能，包括 Cracker 常用的内存脱壳等。目的并不是研究如何破解，而是如何使用 SOS.dll 来协助我们学习.NET CLR/JIT 的一些知识。启用非托管代码调试后，不能使用 Visual Studio 的编辑并继续功能，这点需要注意。

快速定位异常 如果你在代码外层显式的加上了 Try-Catch 异常捕获的时候，默认情况下，Visual Studio 会直接跳到异常处理代码块，而不是出现异常的代码行。在调试状态下运行时，Visual Studio 会将代码停在 catch 这一行，而不是出现错误代码的那一行。如果代码简单那倒无所谓，但是假想我们的代码是经过层层的函数调用，最外层却加了这么个 Try-Catch，那么异常函数调用内抛出了异常，我们也很难定位到异常出错的代码（StackTrace 也只能定位哪一个函数调用出错了）。

在 CLR 异常的 Thrown 列打上勾，那么以后遇到 CLR 的异常就不再是定位到用户处理代码了，而是直接停在抛出异常的代码上。这样可以大大方便我们调试程序的 Bug。

监视变量 在调试时有时不方便查看一个语句的变量计算结果，那么可以选择一个这个表达式，右键添加监视，即可看到这个表达式的值。如图16.16所示：

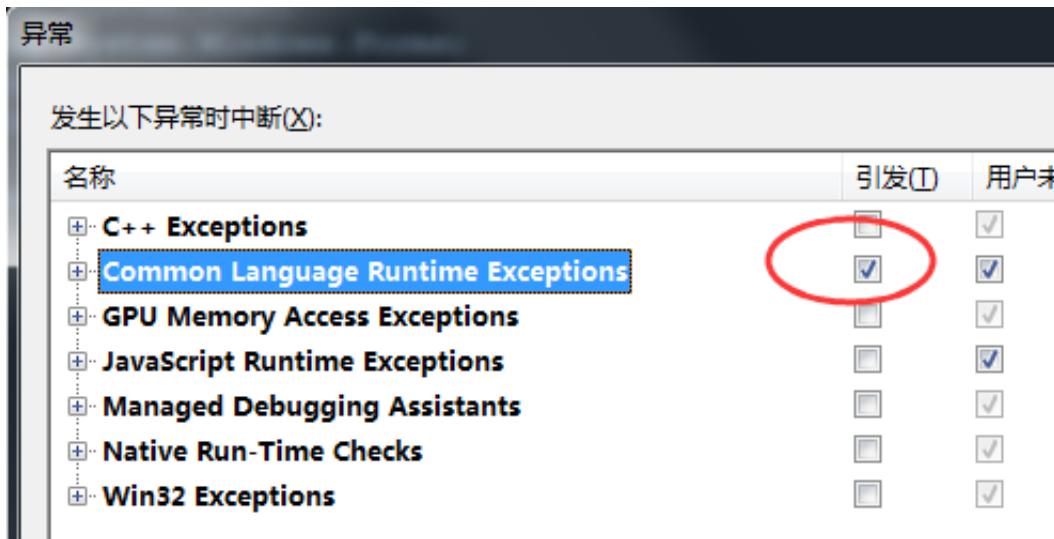


图 16.15: 调试异常设置

监视 1		
名称	值	类型
(int)DriverTaskState.DriverTaskComplete	当前上下文中不存在名称“Dir	
EnumMessageType.RequestVehicleLindRelation	RequestVehicleLindRelation	MVSP.Enum.EnumMessageType
(int)EnumMessageType.RequestVehicleLindRelation	53249	int
EnumMessageType.RequestVehicleLindRelation	RequestVehicleLindRelation	MVSP.Enum.EnumMessageType

图 16.16: 监视表达式的值

查看调试实时调用栈 有时一个方法引用 (Usage) 太多，不知道运行时哪里调用此方法，那么可以在 IntelliTrace 中在运行时查看调用栈，打开设置在调试->IntelliTrace-> 打开 IntelliTrace 设置，打开此设置之后会降低性能，如图16.17所示。

16.12.5 不能设置断点的检查步骤

当 Visual Studio 提示：“当前不会命中断点，还没有为该文档加载任何符号”时，如果确定是符号文件没有加载正确，单击“调试” –> “窗口” –> “模块”列出程序加载的所有 DLL 文件，注意“模块”窗口只有在 Visual Studio 开启调试之后才出现，模块窗口还会列出每一个 DLL 文件的符号文件加载信息，找到不能设置断点的 dll，右键单击，选择“加载符号”选项单击即可，此时会自动为未加载符号的 dll 加载符号，加载完毕之后即可进入调试，还可查看每个文件的加载路径，从加载路径可以看出调试时 IIS 所加载的实际的 dll 路径为：

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET Files\root\
fa41332c\c10b2499\assembly\dl3\5e331c01\f4a72a2a_807ad101\DBTour.Model.dll
```

而不是应用程序的 bin 目录，如图16.18所示：

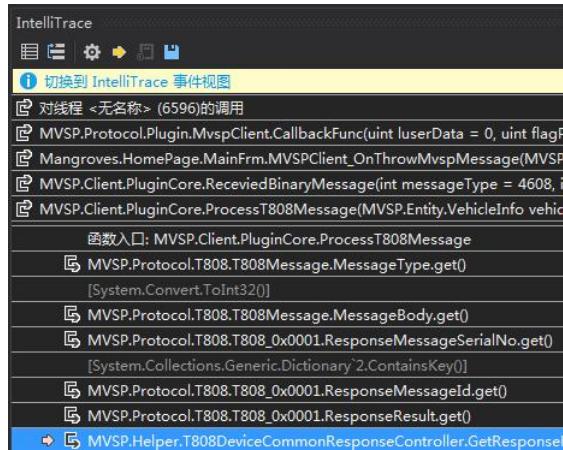


图 16.17: 查看实时调用 Stack

模块				
名称	路径	已优化	用户代码	符号状态
mscorlib.dll	C:\windows\Microsoft.Ne...	是	N/A	已加载符号。
iisexpress.exe	C:\Program Files (x86)\IIS...	N/A	N/A	已加载符号。
System.Web.dll	C:\windows\Microsoft.Ne...	是	N/A	已加载符号。
ntdll.dll	C:\Windows\SysWOW64\...	N/A	N/A	已加载符号。
System.dll	C:\windows\Microsoft.Ne...	是	N/A	已加载符号。
kernel32.dll	C:\Windows\SysWOW64\...	N/A	N/A	已加载符号。
System.Core.dll	C:\windows\Microsoft.Ne...	是	N/A	已加载符号。
KernelBase.dll	C:\Windows\SysWOW64\...	N/A	N/A	已加载符号。

图 16.18: 查看调试时加载的模块。

16.12.6 定位内存偏高、内存泄漏 (Memory Leak)

内存泄漏是指内存空间上产生了不再被实际使用却又不能被分配的内存，内存泄漏的意义很广泛，例如程序无意义的保持对象内存、内存碎片、不彻底的内存释放等。内存泄漏将导致主机的内存随程序的运行而逐渐减少，所有的内存泄漏都是需要努力避免。.NET 的托管堆可能出现内存泄漏现象，大对象的频繁分配和释放、不恰当的保留根引用和错误的 Finalize 方法都可能导致内存泄漏。服务程序最怕的性能问题之一就是内存，当内存很高的情况下我们能够通过对 dump 文件进行查看，看哪些对象导致内存一直高。当内存一直高的情况下就会容易导致内存溢出异常，甚至是 GC 频繁的执行，当 GC 一执行就会导致服务并发下降，因为它要挂起所有的线程（这里指的是服务器模式的.NET CLR，相对应的还有工作站模式的.NET CLR）。

在不知道对象类型的情况下比较简单的方式就是使用：!dumpheap -stat 命令，该命令的意思是统计当前堆的信息，在这里就可以一眼找到哪个对象占用多少内存。

16.12.7 定位线程问题 (CPU 过高, 线程死锁)

CPU 过高也是线上比较棘手的问题之一，查看 CPU 过高的步骤一般分为两步，查看线程的执行时间，然后切换到线程上下文，执行!ClrStack -a，看当前线程在哪里工作，到底做什么操作。输入命令查看调试线程，如下。

```
0:000> !runaway
User Mode Time
 Thread      Time
 0:136c      0 days 0:00:00.000
```

图 16.19: 查看调试程序运行的线程

16.12.8 PDB 文件

大部分的开发人员应该都知道 PDB 文件是用来帮助软件的调试的。但是他究竟是如何工作的呢，我们可能并不熟悉。PDB 文件的存储和内容是什么？debugger 如何找到 binay 相应的 PDB 文件？debugger 如何找到与 binay 对应的源代码文件？

在开始前，我们先定义 2 个术语：private build，用来表示在开发人员自己机器上生成的 build；public build，表示在公用的 build 机器上生成的 build。private build 相对来说比较简单，因为 PDB 和 binay 在相同的地方，通常地我们遇到的问题都是关于 public build。

所有的开发人员需要知道的最重要的事情是“PDB 文件跟源代码同样的重要”，没有 PDB 文件，你甚至不能 debugging。对于 public build，需要 symbol server 存储所有的 PDB，然后当用户报告错误的时候，debugger 才可以自动地找到 binay 相应的 PDB 文件，visual studio 和 WinDbg 都知道如何访问 symbol server。在将 PDB 和 binay 存储到 symbol server 前，还需要对 PDB 进行 source indexing，source indexing 的作用是将 PDB 和 source 关联起来。

接下来的部分假设有已经设置好了 symbol server 和 source server indexing。TFS2010 中可以很简单地完成对一个新的 build 的 source indexing 和 symbol server copying。

PDB 是如何工作的呢？当你加载一个模块到进程的地址空间的时候，debugger 用 2 中信息来找到相应的 PDB 文件。第一个毫无疑问就是文件的名字，如果加载 zzz.dll,debugger 则查找 zzz.pdb 文件。在文件名字相同的情况下 debugger 还通过嵌入到 PDB 和 binary 的 GUID 来确保 PDB 和 binary 的真正的匹配。所以即使没有任何的代码修改，昨天的 binary 和今天的 PDB 是不能匹配的。可以使用 dumpbin.exe 来查看 binary 的 GUID 以及 PDB 文件的路径，查看如下代码片断所示。

```
1 dumpbin.exe /headers "C:\Fosc.Dolphin.UI.exe"
```

输出的内容中有 Debug Directory，会展示当前 Debug GUID 和 PDB 文件的路径。在 Visual Studio 中的 modules 窗口的 symbol file 列可以查看 PDB 的 load 顺序。第一个搜索的路径是 binary 所在的路径，如果不在 binary 所在的路径，则查找 binary 中 hardcoded 记录的 build 目录，例如 obj\debug*.pdb, 如果以上两个路径都没有找到 PDB，则根据 symbol server 的设置，在本地的 symbol server 的 cache 中查找，如果在本地的 symbol server 的 cache 中没有对应的 PDB，则最后才到远程的 symbol server 中查找。通过上面的查找顺序我们可以看出为什么 public build 和 private build 的 PDB 查找不会冲突。

16.12.9 调试第三方 Dll

在安装了 Resharper 后，在动态链接库右键点击，选择 View In Assembly Explorer，选择生成 PDB 文件。

16.13 Visual Studio

16.13.1 Shared Source Common Language Infrastructure 2.0

The Shared Source CLI is a compressed archive of the source code to a working implementation of the ECMA CLI and the ECMA C# language specification. This implementation builds and runs on Windows XP.

16.13.2 Visual Studio 快捷键

键	动作
Ctrl+Alt+L	显示 Solution
Ctrl+Shift+B	生成项目
Shift + Alt+ C	添加新类
Alt + Shift + 箭头键 ($\leftarrow, \uparrow, \downarrow, \rightarrow$)	选择代码的自定义部分
Ctrl + H	显示替换对话框
Ctrl + I	渐进搜索
Ctrl + F3	搜索当前选择的文本

16.13.3 Visual Studio 如何查找源文件

VS 查找源文件的流程是这样的：

- 1、根据可执行模块中保存的 pdb 文件信息 (guid, pdb 文件名, age) 查找加载 pdb 文件。
- 2、根据 pdb 中的源文件路径查找源文件。如果源文件在路径中不存在，则在“图 1”中的“Browse to find source”可以使用，点击后会弹出文件打开对话框。如果 pdb 是 public pdb 文件，则没有包含源文件信息，这时“图 1”中的“Browse to find source”变成灰色，不可使用。
.NET PDB 只包含了 2 部分信息：* 源代码文件名字和行数；* 和局部变量的名字；* 所有的其他的数据都已经包含在了.NET Metadata 中了；

16.13.4 重置 Visual Studio

开始菜单 -> 所有程序->Visual Studio 2012 文件夹 -> Visual Studio Tools -> Developer Command Prompt for VS2012

输入 DOS 命令: CD Common7/IDE 进入到该工具下的子文件夹中

输入: devenv.exe /resetsettings , 重置 Visual Studio

16.13.5 Visual Studio 显示行号

16.13.6 低版本打开高版本工程

在 Visual Studio 中打开高版本 Visual Studio 新建的工程。

16.13.7 使用后期生成事件

在做插件开发的过程中需要将插件生成的 dll 拷贝到项目的主目录，每次生成都拷贝相当麻烦，使用生成后事件在插件项目成功生成后自动将插件 dll 拷贝到主项目的 Plugin 文件夹中。命令为：

```

1 #$(ProjectDir) 的目录示例为当前项目的根目录
2 #如 D:/OneDrive/SourceCode/ZWLBS/Client/Source/2.0/MVSP.Client
3 xcopy /r /y $(ProjectDir)bin\Debug\MVSP.Client.dll $(ProjectDir)..\\Mangroves\\plugins

```

r 参数表示覆盖只读文件，y 参数表示取消提示以确认要覆盖现有目标文件。在项目属性->生成事件-> 编辑后期生成事件的宏选项中，可以看到宏脚本常用的变量符，脚本中可能用到的环境变量如图16.20所示：

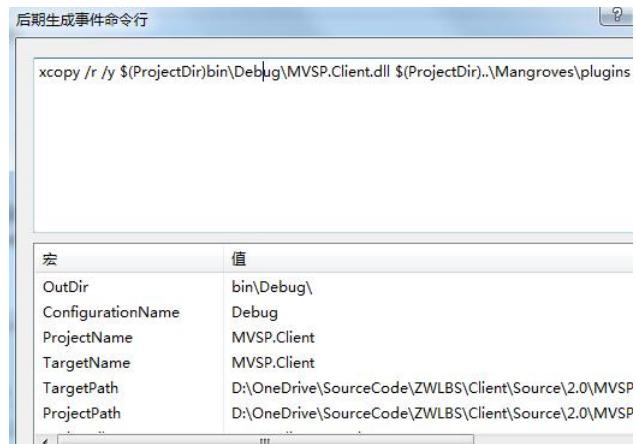


图 16.20: Visual Studio 中查看宏变量

16.13.8 解决方案文件夹

正在处理包含许多项目的解决方案，则可以使用解决方案文件夹将相关的项目组织成组，然后对这些项目组执行操作。有关解决方案文件夹的信息（包括它们包含的项目和解决方案项）存储在解决方案文件 (.sln) 中，解决方案文件夹是解决方案资源管理器中的一种组织工具；不会创建相应的 Windows 文件夹。建议采用在解决方案中组织项目时所用的方式在磁盘上组织项目。添加解决方案文件夹后的项目如图16.21所示。

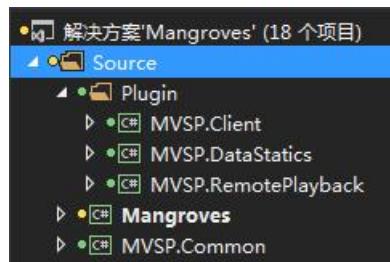


图 16.21: 解决方案文件夹

16.13.9 插件

- Indent Guides 给每个代码块增加垂直对齐虚线

16.13.10 Visual Studio 加载外部工具

在 Visual Studio 中可以加载外部工具，加载外部工具如图16.22所示，加载之后在 Visual Studio 中的菜单 Tools 中可以直接打开工具，非常方便。

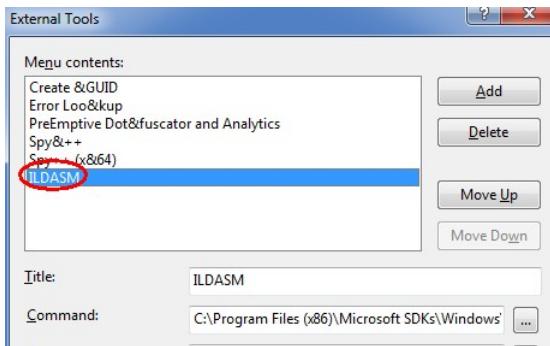


图 16.22: 解决方案文件夹

16.13.11 Configure Visual Studio 2013 for debugging .NET framework

调试系统 dll 有助于理解程序更加深层次的处理逻辑，在开发基于网络的应用程序时，调试系统 dll 有助于理解从浏览器构造请求到服务器端经过处理后，返回结果到客户端，可以理解服务器的处理流程，对于 Microsoft 以往一直闭源的策略，调试系统 dll 尤其显得更加重要。

- Disable just my code
- Disable step over properties and operators
- Disable require source files to exactly match the original version
- Enable .NET framework source stepping
- Enable source server support

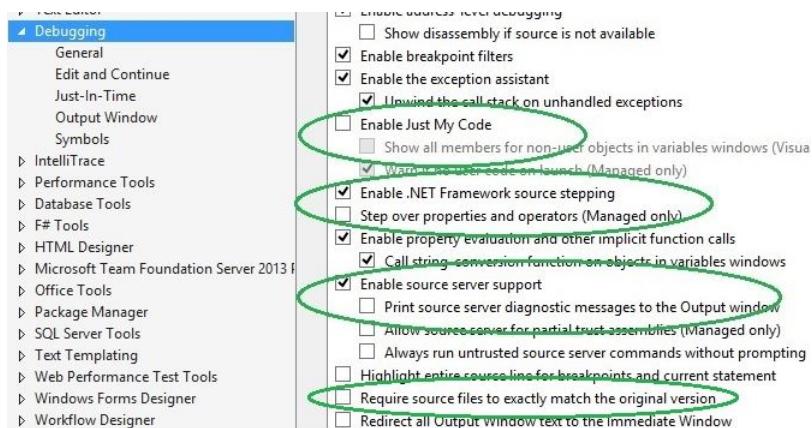


图 16.23: Enable .NET Framework Source Code Debugging

具体配置参考<http://referencesource.microsoft.com/setup.html>。利用 Resharper 生成系统 dll 的调试文件（PDB: Program Database）如图16.24所示。

调试 System.Web.dll 如图16.25所示。

在.NET Framework 4.5 中调试源代码需要下载源代码文件，下载链接为<http://referencesource.microsoft.com/>，下载源码后添加源码安装路径到 Visual Studio 的符号文件加载路径中即可。

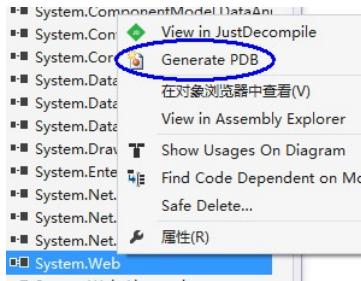


图 16.24: Generate Program Debugging File

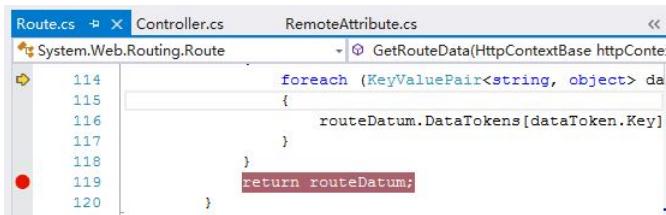


图 16.25: Debugging System.Web.dll

16.13.12 Visual Studio 查看 Call Stack

调用公共方法的上层函数较多，有时在调试时需要查看当前运行时是哪个函数调取的当前方法，可以查看调用栈 (Call Stack)，在 Visual Studio 中查看调用栈在调试 (Debug)→ 窗口 (Window)→ 调用堆栈 (Call Stack) 中，注意需要启动调试后才能查看，如图16.26所示。

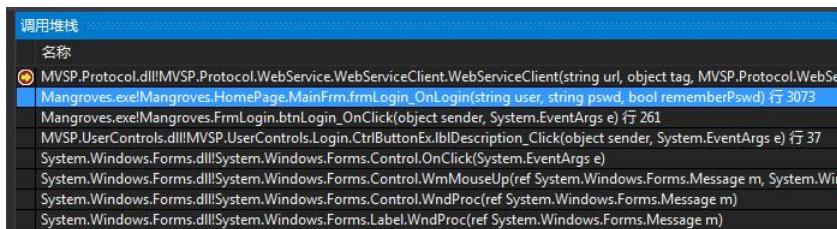


图 16.26: 查看调用堆栈

16.14 发布 (Publish)

开始部署网站时往往将开发好的项目完全拷贝到 IIS 中进行部署，虽然完全可以正常运行，但是包含了许多开发过程中的许多无用文件，经过发布后的项目去掉了许多开发过程中的文件，发布目录相对于直接拷贝显得更加干净整洁，同时发布项目也让开发者更深入的明白哪些文件是程序运行过程中真正需要并实际使用的，哪些文件是开发过程文件。编写好 ASP.NET MVC 网站应用程序后，在 Visual Studio 中发布项目时，常常会遇到如图16.27所示的错误。

对于“该项目中不存在目标”*GatherAllFilesToPublish*“(The target ”*GatherAllFilesToPublish*” does not exist in the project)” 的错误，在项目文件 (*.csproj) 中添加如下语句 (Target 配

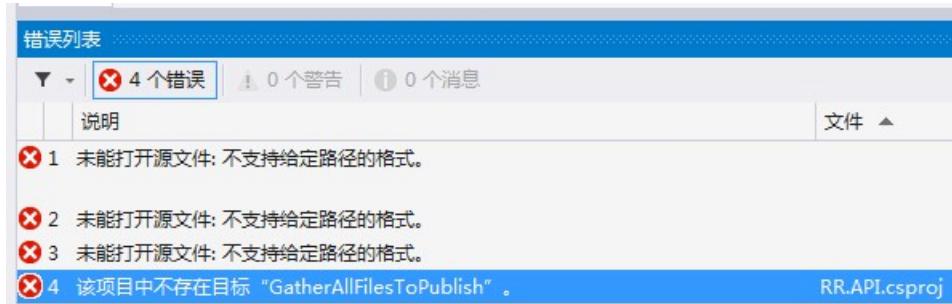


图 16.27: 发布程序时错误

置节与 ProjectExtensions 配置节之间) 可解决此问题⁶，环境为 Visual Studio 2013。

```
1 <Target Name="GatherAllFilesToPublish"></Target>
```

对于“未能打开源文件：不支持给定路径的格式 (*Could not open Source file: The given path's format is not supported*)”，造成此问题的原因是发布的项目以直接引用项目的方式引用了其他项目，故在发布时有此错误提示，具体原因无从知晓，为了避免此问题可在发布项目时直接引用其他项目编译完成的 dll，而非直接引用项目，可解决此问题。有时未直接引用项目也会引发此错误提示，可临时将项目文件排除项目（右键-从项目中排除）再进行发布，一步一步可找出原因，发布项目的过程为先将文件拷贝到临时发布目录 obj/Release/Package/PackageTmp 下，再将临时发布目录的文件拷贝到发布目录。例如项目 RR.Web.Activity 项目引用了项目 RR.DataCommon，在项目发布时将 RR.DataCommon 编译完成拷贝编译后的 dll 到 lib 文件夹中，或者直接引用 Debug 目录下的 dll 文件，RR.Web.Activity 直接引用 lib 文件夹中的 RR.DataCommon.dll 而非直接引用项目。如果对需要发布的文件比较明确，还有可以直接在文件或者文件夹上右键弹出发布单个文件或文件夹而不是发布整个项目，也可以避免此错误造成的发布失败问题。发布后的文件有时也包含 *.pdb 文件，*.csproj 文件，在 csproj 文件中编辑将不需要的文件删除即可。在发布过程中，如果目标目录不存在，则程序会自动创建目标发布目录。

提交到源码库的 csproj 文件引用 (ItemGroup 的 Content 配置节) 中要确保没有包含 bin 及 obj 目录下的文件 如果将 bin 目录下的文件包含在项目中，那么发布的时候会到 bin 目录下去 Copy 发布文件，而在 svn 服务器上是不存在 bin 目录的 (bin 目录在本地的编译环境中)，会导致发布失败。右键点击选者“从项目中排除”，在重新生成时 MSBuild 会修改 csproj 下 ItemGroup 对应的 Content 配置节，自动编译时会动态生成 bin、obj 对应的文件。When you deploy, it doesn't care what files are in your BIN folder. It cares about what files your CSPROJ project file specifies as part of the project deployment. 发布的时候对于某些资源文件需要选择复制到输出目录，这样发布之后会包含在最终的发布文件夹里面，否则在发布后的目录里会缺少某些必要文件，导致程序运行失败。

⁶参考链 接: <http://stackoverflow.com/questions/10989051/why-do-i-get-the-error-the-target-gatherallfilestopublish-does-not-exist>

16.14.1 Visual Studio 中采用 FTP 发布

采用 FTP 发布站点配置如图16.28所示，站点路径填写根目录，用户名密码为 FTP 的用户名和密码，对程序作出修改后直接在解决方案管理器点击右键发布文件即可。



图 16.28: 发布站点

发布的进程为 Visual Studio 根据 csproj 文件中 ItemGroup 配置的文件列表，拷贝相应的文件到发布目录中，有时发布后的目录部署时总是提示缺少这样那样的文件，可检查开发目录 bin 下是否有此文件，如果有此文件，一定要包括此文件在项目中（文件在 Visual Studio 目录树上不能是灰色），才会在 ItemGroup 中有文件的记录，发布时才会拷贝相应的文件到目标目录中。

16.14.2 Browser Link

多浏览器开发时，当修改一个地方需要同时在多个浏览器中查看修改效果时，使用此功能。

data-* 是 html5 的属性

16.14.3 Shadow Copy

运行程序时提示错误：未能从程序集 DBSystem.BLL.dll 中加载类型 tb_abc。使用代码查看当前项目所引用的程序集以及程序集的全路径。

```

1 AppDomain domain = AppDomain.CurrentDomain;
2 Assembly[] assemblyArr = domain.GetAssemblies();
3 foreach (Assembly assembly in assemblyArr)

```

```

4 {
5   logger.Info(assembly.FullName); //当前使用的程序集的名称
6   logger.Info(assembly.Location); //当前使用的程序集的全路径
7 }

```

根据输出的路径找到 DBSystem.BLL.dll 的路径为：

```

1 DBSystem.BLL, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
2 C:\ProgramData\Red Gate\.NET Reflector\DevPath\DBSystem.BLL.dll

```

可以看出是引用另一个路径下的 dll 而非 bin 目录下的 dll。用反编译工具查看 dll 的确没有类 tb_abc，将此目录下的 dll 更新后问题解决。是因为.NET 下的 Shadow Copy 技术，Shadow Copy 技术允许你在程序运行时编辑和重新编译程序集⁷。

Shadow copying enables assemblies that are used in an application domain to be updated without unloading the application domain. This is particularly useful for applications that must be available continuously, such as ASP.NET sites.

The common language runtime locks an assembly file when the assembly is loaded, so the file cannot be updated until the assembly is unloaded. The only way to unload an assembly from an application domain is by unloading the application domain, so under normal circumstances, an assembly cannot be updated on disk until all the application domains that are using it have been unloaded.

When an application domain is configured to shadow copy files, assemblies from the application path are copied to another location and loaded from that location. The copy is locked, but the original assembly file is unlocked and can be updated.

16.14.4 技巧 (Little Tricks)

当在 Visual Studio 中需要用到批量替换替换某一个项目的代码时，为了不影响其他项目的代码，可在项目上右键单击，选择“限定为此范围”，如图16.29所示。

替换完毕后点击解决方案下的后退按钮，即可返回。有时 Visual Studio 项目树上会出现带黄色叹号的文件，如图16.30所示。

那是因为此文件在项目文件 (*.csproj) 中已经添加，但是不存在此物理文件的缘故，一般是在别处添加了文件而没有提交此物理文件到 SVN，提交此文件即可，也可以从项目文件将其移除。

⁷NUnit normally uses .Net shadow-copying in order to allow you to edit and recompile assemblies while it is running. Uncheck this box to disable shadow-copy only if you have a particular problem that requires it. 来自 NUnit 文档: <http://www.nunit.org/index.php?p=optionsDialog&r=2.4.8>

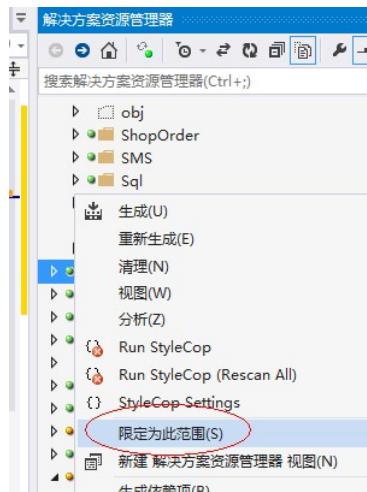


图 16.29: 限定项目范围

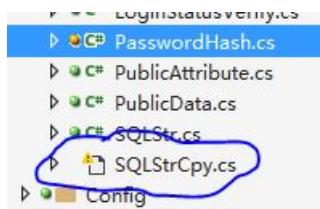


图 16.30: 项目树缺失文件

16.14.5 Visual Studio 生成操作

内容 (Content) - 不编译该文件，但将其包含在“内容” (Content) 输出组中。编译 (Compile) - 将该文件编译到生成输出中。此设置用于代码文件。嵌入资源 (Embedded Resource) - 将该文件作为 DLL 或可执行文件嵌入主项目生成输出中。链接资源 (Linked Resources) 作为文件存储在项目中；在编译期间，从这些文件中取得资源数据，并将其添加到应用程序的清单中。应用程序的资源文件 (.resx) 只存储指向磁盘上的文件的相对路径或链接。对于嵌入资源，资源数据直接以二进制数据的文本表示形式存储在.resx 文件中。在任何一种情况下，资源数据都将编译到可执行文件中。如果必须在多个项目之间共享应用程序资源 (.resx) 文件，则嵌入的资源是最佳选择。例如，如果您有一个包含公司徽标、商标信息等类似内容的通用资源文件，则应使用嵌入的资源，这样您只需复制.resx 文件，而不用复制关联的资源数据文件。

16.14.6 字符串比较 (String Compare)

The .NET Framework provides extensive support for developing localized and globalized applications, and makes it easy to apply the conventions of either the current culture or a specific culture when performing common operations such as sorting and displaying strings. But sorting or comparing strings is not always a culture-sensitive operation. For example, strings that are used internally by an application typically should be handled identically across all cultures. When culturally independent string data, such as XML tags, HTML tags, user names, file paths, and

the names of system objects, are interpreted as if they were culture-sensitive, application code can be subject to subtle bugs, poor performance, and, in some cases, security issues.

- Use comparisons with StringComparison.OrdinalIgnoreCase or StringComparison.OrdinalIgnoreCase for better performance. 在进行调用 String.Compare(string1, string2, StringComparison.OrdinalIgnoreCase) 的时候是进行非语言 (non-linguistic) 上的比较, API 运行时将会对两个字符串进行 byte 级别的比较, 因此这种是比较严格和准确的, 并且在性能上也很好, 一般通过 StringComparison.OrdinalIgnoreCase 来进行比较比使用 String.Compare(string1, string2) 来比较要快 10 倍左右. StringComparison.OrdinalIgnoreCase 就是忽略大小写的比较, 同样是 byte 级别的比较. 性能稍弱于 StringComparison.OrdinalIgnoreCase.
- Use overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type StringComparison.
- Use StringComparison.OrdinalIgnoreCase or StringComparison.OrdinalIgnoreCase for comparisons as your safe default for culture-agnostic string matching.

16.14.7 .NET 索引器

16.15 正则表达式 (Regular Expression)

16.15.1 元字符 (Metacharacter)

\d	匹配数字
^	匹配字符串的开始

16.15.2 常用正则表达式

Javascript 中验证格式如: 09:00-10:00 的字符串。

```

1  if (!obj.goDuringTime.match(/^\d{2}:\d{2}-\d{2}:\d{2}/)) {
2      alert("活动时间段不正确哟，请检查是否有中文字符");
3      return;
4 }
```

两个特殊的符号'` 和'\$'。他们的作用是分别指出一个字符串的开始和结束。表` 表示打头的字符要匹配紧跟` 后` 面的规则

\$ 表示打头的字符要匹配紧靠 \$ 前面的规则

[] 中的内容是可选字符集

[0-9] 表示要求字符范围在 0-9 之间

1,20 表示数字字符串长度合法为 1 到 20，即为 [0-9] 中的字符出现次数的范围是 1 到 20 次。/和 \\$/成对使用应该是表示要求整个字符串完全匹配定义的规则，而不是只匹配字符串中的一个子串。

16.16 配置管理 (Configuration Management)

16.16.1 配置读取

在 Web 开发中可针对配置文件定义一个公共类，专负责配置文件的读取和存储，避免在分散的类中定义配置读取，如下代码片段所示。

```

1 public class PublicAttribute
2 {
3     /// <summary>
4     /// 静态引用地址
5     /// </summary>
6     public static string staticPath = ConfigurationManager.AppSettings["staticPath"];
7 }
```

C# 中没有传统意义上的全局变量，但是可以使用公共静态变量定义一个全局可见的变量。公共静态变量必须属于某个类型，对静态变量的访问也需要通过该类型实现。全局变量在 ASP.NET 中可以使用 Application 对象，或者使用应用程序配置文件存储可配置数据。从 Application 这个单词上大致可以看出 Application 状态是整个应用程序全局的。在 ASP 时代我们通常会在 Application 中存储一些公共数据，而 ASP.NET 中 Application 的基本意义没有变：在服务器内存中存储数量较少又独立于用户请求的数据。由于它的访问速度非常快而且只要应用程序不停止，数据一直存在，我们通常在 Application_Start 的时候去初始化一些数据，在以后的访问中可以迅速访问和检索。在 ASP.NET 2.0 中，Application 已经变得不是非常重要了。因为 Application 的自我管理功能非常薄弱，它没有类似 Session 的超时机制。也就是说，Application 中的数据只有通过手动删除或者修改才能释放内存，只要应用程序不停止，Application 中的内容就不会消失。可以使用 Cache 实现类似 Application 的功能，同时 Cache 又有丰富而强大的自我管理机制。

16.16.2 针对不同发布环境调整配置 (Self-adaption Configuration)

有时候需要在不同的环境中应用不同的配置，比如在生产环境中，接口的配置地址要修改成生产环境的地址，静态服务器地址需要修改成正式地址、数据库连接串也需要修改等等，单纯的依靠手工改动非常容易发生遗漏等等意外情况，那么可以根据不同的生产环境预先定义不同的配置，在发布时选择不同的发布环境，不同的发布环境应用不同的配置，有效避免了手滑造成配置错误、遗漏等等现象。打开 Visual Studio 的配置文件管理器，如图16.31所示。

新建了 2 个配置选项，一个是服务器的生产环境配置 (Server Production Environment)，

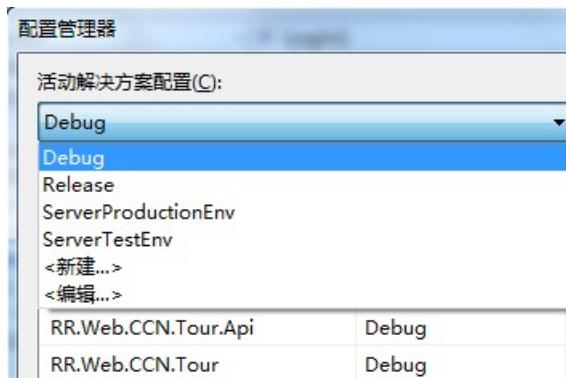


图 16.31: Visual Studio 添加配置选项

一个是服务器的测试环境配置 (Server Test Environment), 2 个环境所用的 Web.config 需要各自定义。在配置管理器添加完毕配置转换后，可在项目的 Web.config 中点击右键“添加配置转换”，如图16.32所示。



图 16.32: Visual Studio 添加配置转换

在新建的生产环境配置文件 Web.Production.config 中添加如代码16.2所示的配置。

Listing 16.2: 配置文件转换

```

1<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
2<appSettings>
3    <!-- WCF正式接口路径 -->
4    <add key="WCFPath" value="http://129.21.11.122:88" xdt:Transform="SetAttributes"
5        xdt:Locator="Match(key)"/>①
6    <!--新版静态文件地址-->
7    <add key="newStaticServer" value="http://192.168.35.17:8083" xdt:Transform="
8        SetAttributes" xdt:Locator="Match(key)"/>
9</appSettings>
10<connectionstring>
11    <add name="MyDB"
12        connectionString="Data Source=TestSQLServer;Initial Catalog=MyTestDB;
13        Integrated Security=True"
14        xdt:Transform="Replace" xdt:Locator="Match(name)"/>②
15</connectionstring>
16<system.web>
17    <compilation xdt:Transform="RemoveAttributes(debug)" />
18    <customErrors mode="RemoteOnly" defaultRedirect="/error/index" xdt:Transform="
19        Replace"/>③

```

```

16  </system.web>
17  </configuration>

```

- ① 设置 (SetAttributes) 生产环境所调用的接口的值，根据 key 进行匹配 (Match(key))
- ② 生产环境中只允许在主机上查看错误信息，避免将错误信息暴露给客户端，此处替换 customErrors 中 mode 的值，不管在调试环境中的值如何设定，发布后统一设定为 RemoteOnly

转换文件是一个 XML 文件，该文件指定在部署 Web.config 文件时应如何更改该文件。转换操作通过使用在 XML-Document-Transform 命名空间（映射到 xdt 前缀）中定义的 XML 特性来指定。XML-Document-Transform 命名空间定义两个特性：Locator 和 Transform。Locator 特性指定要以某种方式更改的 Web.config 元素或一组元素。Transform 特性指定要对 Locator 特性所查找的元素执行哪些操作。

16.17 Inversion of Control

IoC 意味着将你设计好的类交给系统去控制，而不是在你的类内部控制，这称为控制反转，就是被调用类的实例由原先的调用类控制创建、销毁现在转变成由容器管理。在项目中以数据表生成的类名都需要一一修改，有没有一种方式可以做到非常大胆的修改类而不影响原有代码？或者原有的代码只需要稍加调整即可。在使用对象时使用反射可以做到，可以根据类名生成类的实例，而类名是可以定义在配置文件中的，当类名修改之后，只需要更改配置文件的类名即可。但反射会造成编译时无法保证类型安全性以及性能问题。控制反转 (Inversion of Control,IoC)，简单地说就是应用本身并不负责依赖对象的创建和维护，而将此任务交给一个外部容器，这样控制权就由应用转移到了这个外部 IoC 容器，控制权就实现了所谓的反转。比如在 A 类型中需要使用 B 类型的实例，而 B 实例的创建并不由 A 来负责，而是通过外部容器来创建。使用依赖注入有以下好处：

- 将被依赖类的创建代码从依赖类中移出，不用显式的写 new
- 可以单独维护被依赖类的创建过程
- 方便该类的被共享
- 如果该类初始化时，所需属性很多，使用配置，远比硬代码编写简单
- 有多层依赖时，依赖关系的移出，事实上简化了依赖关系的查看和维护

16.17.1 Spring.NET

Spring.NET is an open source application framework that makes building enterprise .NET applications easier. Providing components based on proven design patterns that can be integrated into all tiers of your application architecture, Spring helps increase development productivity and improve application quality and performance.

Spring 解决了的最核心的问题就是把对象之间的依赖关系转为用配置文件来管理，也就是 Spring 的依赖注入机制。这个注入机制是在 IoC 容器中进行管理的。从 Spring.NET 的官方网站 (<http://www.springframework.net/index.html>) 上下载项目文件，选择合适的版本（这里是 1.3.1）和文件 (.NET Framework 4.0) 引入到项目中，主要引入 Spring.Core.dll、Common.Logging.dll 和 Spring.Data.dll 文件。新建一个 IPersonDao 接口，定义一个 Save 方法，如代码片段所示。

```

1  namespace RR.DataCommon.Dao.Interface
2  {
3      public interface IPersonDao
4      {
5          void Save();
6      }
7  }
```

定义一个类 PersonDao，实现定义的 IPersonDao 接口，如代码片段所示。

```

1  namespace RR.DataCommon.DaoImpl
2  {
3      public class PersonDao:IPersonDao
4      {
5          public void Save()
6          {
7              logger.Info("I am saving...");
8          }
9      }
10 }
```

配置文件中增加相应的配置，如代码片段所示。

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <configuration>
3      <configSections>
4          <sectionGroup name="spring">
5              <section name="context" type="Spring.Context.Support.ContextHandler, Spring.
6                  Core" />
7              <section name="objects" type="Spring.Context.Support.DefaultSectionHandler,
8                  Spring.Core" />
9          </sectionGroup>
10     </configSections>
11     <spring>
12         <context>
13             <resource uri="config://spring/objects" />①
14         </context>
15         <objects xmlns="http://www.springframework.net">
16             <description>一个简单的控制反转例子</description>
17             <object id="PersonDao" type="Dao.PersonDao, Dao" />②
18         </objects>
19     </spring>
```

```
18 | </configuration>
```

- ① 指定配置的需要容器管理的类的位置在配置文件 spring 下的 objects 节点下
- ② type 第一个参数是类在 dll 的全路径（也就是需要包含命名空间），第二个参数是 dll 文件的名称

调取对象如下代码片段所示。

```
1 IApplicationContext ctx = ContextRegistry.GetContext();
2 IPersonDao dao = ctx.GetObject("PersonDao") as IPersonDao;
3 if (dao != null)
4 {
5     dao.Save();
6     logger.Info("我是IoC方法");
7 }
8 else
9 {
10    logger.Error("Object is null");
11 }
```

使用 IoC 主要是为了避免程序中出现类的依赖关系，如 ClassA a=new ClassA() 这样的写法，程序规模较大时，ClassA 发生改变，修改起来比较麻烦，我的理解 IoC 把变化的点封装到了一个 XML 文件中，当类发生变化时，只需要修改 XML 文件即可，将软件中可能变化的点放置在一个地方。再结合 dynamic 类型，应该可以做到完全不必写出显示的类名。在 Spring.NET 框架中也可以通过如下的写法获取到对象实例。

```
1 IApplicationContext ctx = ContextRegistry.GetContext();
2 dynamic dao = ctx.GetObject("PersonDao");
3 if (dao != null)
4 {
5     dao.Save();
6     logger.Info("我是IoC方法");
7 }
8 else
9 {
10    logger.Error("Object is null");
11 }
```

这样也可以不用定义接口实现的模式也可以自由的获取对象的实例⁸。

16.17.2 Spring.NET 配置独立的 Object.xml

Spring.NET 配置独立的 Object.xml 如下代码片段所示。

⁸参考自刘冬的博客：http://www.cnblogs.com/GoodHelper/archive/2009/10/25/Spring_NET_IoC.html

```

1 string[] xmlFiles = new string[] {"file:///G:/Objects.xml"};
2 IApplicationContext ctx = new XmlApplicationContext(xmlFiles);
3 dynamic dao = ctx.GetObject("PersonDao");
4 if (dao != null)
5 {
6     dao.Save();
7     logger.Info("我是IoC方法");
8 }
9 else
10 {
11     logger.Error("Object is null");
12 }

```

也可以在 Web.config 中指定独立的对象容器地址（配置对象的 XML 文件），如下代码片段所示。

```

<spring>
    <context>
        <resource uri="file:///D:\\项目\\Tour\\trunk\\Config\\Objects.xml" />①
        <resource uri="file:///~\\Config\\Objects.xml" />②
    </context>
</spring>

```

- ①** 在配置文件中以绝对路径的方式指定 Objects.xml 文件的位置
- ②** 在配置文件中以相对路径的方式指定 Objects.xml 文件的位置，在 NUnit 单元测试中，相对路径为 NUnit 项目文件的存放路径

在编辑配置文件时如果需要 Visual Studio 智能提示功能，可将 Spring.NET\doc\schema 目录下的 spring-objects-1.3.xsd 文件拷贝到 Visual Studio 的 Xml\Schemas 目录中。

16.17.3 Castle Windsor(Apache License Version 2.0)

Castle Windsor⁹ is a best of breed, mature Inversion of Control container available for .NET and Silverlight.

We are a bunch of programmers who share a collective frustration with the status quo of the development tools and frameworks for .net. We have jobs, lives and hobbies, and we value being productive. We also value delivering software that is maintainable and long lasting.

That was the motivation for this open source project.

Castle Project is an umbrella for a handful of open source projects for .net. We aspire to simplify the development of enterprise and/or web applications.

⁹官方地址: <http://www.castleproject.org/>

Castle helps you get more done with less code and in less time (but not in the traditional code generator mess that some preach). Castle Windsor 安装命令如下：

```
1 Install – Package Castle.Windsor
```

16.17.4 Autofac

Autofac is an addictive Inversion of Control container for .NET 4.5, Silverlight 5, Windows Store apps, and Windows Phone 8 apps.

16.17.5 命令行开发.NET

一是开发者或许想深入地了解一下.NET 应用的编译过程，而不希望被图形用户界面蒙蔽了双眼。

二是开发者或许出于某些原因，例如平台或工作环境的原因而无法安装 VS。

三是开发者只打算开发一个非常简单的应用，为了一个 5KB 大小的应用去下载一个 5GB 大小的 IDE 好像有些太过奢侈。

```
1 using System;
2 using System.Collections.Generic;
3
4 namespace HelloWorld
{
5
6     public class Program
7     {
8
9         static void Main (string[] args)
10        {
11            List<string> list = new List<string>();
12            list.Add("J");
13            list.Add("X");
14            list.Add("Q");
15            string str = string.Join(", ", list);
16            Console.WriteLine(str);
17        }
18    }
}
```

使用如下命令编译程序：

```
1 csc.exe /out:a.exe a.cs
```

有时只想要测试某一小段代码的运行结果，采用命令行的方式会更加简单直接，避免了 Visual Studio 的缓慢。

16.18 AOP(Aspect-Oriented Programming)

AOP(Aspect-Oriented Programming) 面向切面编程，看着是跟 OOP（面向对象编程）挺相近的，但实际上又有什么区别呢？OOP 具有封装，继承，多态等东西来定义从上到下这种层次关系，但要想实现从左到右的关系的话就开始有点水土不服了，例如用户的权限控制，操作日志等，这些与我们要实现的核心功能不大有关系的东西散布在我们代码的周边，显示十分不好看。于是我们引入了 AOP 的模式。我们通常在实现一个页面逻辑的时候，通常伴随着操作日志，安全监测，事务处理等几个逻辑，在实现逻辑的时候都要写一大堆这种代码。而 AOP 就是将这些与主体业务无关，但又有为业务提供服务的逻辑代码封装起来，降低模块之间的耦合度。如图所示中的圆柱体好比如我们的业务流程，AOP 代表的是那个横向的操作，俗称切面编程。或许上面的这些理论有点头疼，对于 AOP 我的大体理解是：将那些与业务核心不大相关的杂七杂八的东西独立开，每次实现了业务核心之前或之后，调用一下对应的方法。AOP 这种切面编程能干很多事情，例如验证登陆，权限，性能检测，错误信息记录等等，AOP 的目的就是将这些东西分离开来，让开发人员专注与核心关注点。

16.18.1 AOP 验证登陆

16.18.2 AOP 验证权限

16.18.3 AOP 记录日志

16.18.4 AOP 事务处理

16.18.5 iMacros 辅助网页调试

工作中做到重复比较多少一项就是修改了页面源码后刷新页面查看修改后是否达到预期效果，每次切换窗口点击刷新，等待页面出现最终效果（修改后 IIS 一般会重启，IIS 首次访问时相当缓慢的，IIS 需要拷贝文件，启动进程），相当繁琐，尝试过自动刷新插件，效果不甚理想，而 iMacros 刚好可以解决这种简单重复的劳动。iMacros is an extension for the Mozilla Firefox web browsers which adds record and replay functionality similar to that found in web testing and form filler software. The macros can be combined and controlled via Javascript. The extension was developed by iOpus. The current stable release of iMacros is version 0.9.0.2, released on July 7, 2007. As of July 14, 2007 iMacros is one of the TOP 10 most popular Firefox extensions in the Bookmark, Web Data, Alerts, and Widgets and Social and Sharing categories.

16.19 异步 (Async)

16.19.1 使用 IAsyncResult 对象的异步操作

异步委托提供以异步方式调用同步方法的能力。当同步调用一个委托时，“Invoke”方法直接对当前线程调用目标方法。如果编译器支持异步委托，则它将生成“Invoke”方法以及“BeginInvoke”和“EndInvoke”方法。如果调用“BeginInvoke”方法，则公共语言运行库(CLR)将对请求进行排队并立即返回到调用方。将对来自线程池的线程调用该目标方法。提交请求的原始线程自由地继续与目标方法并行执行，该目标方法是对线程池线程运行的。如果在对“BeginInvoke”方法的调用中指定了回调方法，则当目标方法返回时将调用该回调方法。在回调方法中，“EndInvoke”方法获取返回值和所有输入/输出参数。如果在调用“BeginInvoke”时未指定任何回调方法，则可以从调用“BeginInvoke”的线程中调用“EndInvoke”。

当我们直接调用委托时，实际上是调用了 Invoke() 方法，它会中断调用它的客户端，然后在客户端线程上执行所有订阅者的方法（客户端无法继续执行后面代码），最后将控制权返回客户端。注意到 BeginInvoke()、EndInvoke() 方法，在.NET 中，异步执行的方法通常都会配对出现，并且以 Begin 和 End 作为方法的开头（最常见的可能就是 Stream 类的 BeginRead() 和 EndRead() 方法了）。它们用于方法的异步执行，即是在调用 BeginInvoke() 之后，客户端从线程池中抓取一个闲置线程，然后交由这个线程去执行订阅者的方法，而客户端线程则可以继续执行下面的代码。

16.19.2 Task

.NET 4.0 推出了新一代的多线程模型 Task。async/await 特性是与 Task 紧密相关的，所以在了解 async/await 前必须充分了解 Task 的使用。这里将以一个简单的 Demo 来看一下 Task 的使用，同时与 Thread 的创建方式做一下对比。

16.19.3 async/await 特性

16.20 多线程

16.20.1 lock

lock 关键字可以用来确保代码块完成运行，而不会被其他线程中断。它可以把一段代码定义为互斥段 (CriticalSection)，互斥段在一个时刻内只允许一个线程进入执行，而其他线程必须等待。这是通过在代码块运行期间为给定对象获取互斥锁来实现的。

16.20.2 线程简介

线程是程序中的一个执行流，每个线程都有自己的专有寄存器（栈指针、程序计数器等），但代码区是共享的，即不同的线程可以执行同样的函数。

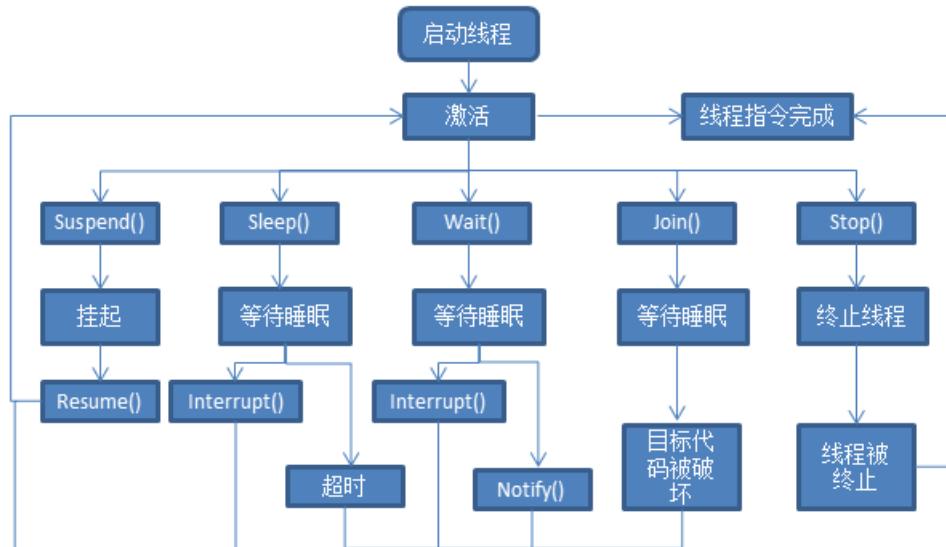


图 16.33: 线程的生命周期

多线程的好处：

可以提高 CPU 的利用率。在多线程程序中，一个线程必须等待的时候，CPU 可以运行其它的线程而不是等待，这样就大大提高了程序的效率。

多线程的不利方面：

线程也是程序，所以线程需要占用内存，线程越多占用内存也越多；多线程需要协调和管理，所以需要 CPU 时间跟踪线程；线程之间对共享资源的访问会相互影响，必须解决竞用共享资源的问题；线程太多会导致控制太复杂，最终可能造成很多 Bug；

16.20.3 Timer

System.Threading.Timer 在线程池启动一个后台任务。System.Windows.Forms.Timer 告诉 windows 把一个计时器和调用它的线程（UI 线程）关联起来，通过往 UI 线程的消息队列里放一个 WM_TIMER 的消息来实现，所以它的 callback 一定是在 UI 线程调用的，不存在多线程调用的问题。System.Windows.Threading.DispatcherTimer 用在 WPF 和 Silverlight 中，对应于 System.Windows.Forms.Timer。使用 System.Windows.Forms.Timer 如下代码片段所示：

```

1 private readonly System.Windows.Forms.Timer _aTimer = new System.Windows.Forms.Timer();
2
3 private void SetTimer()
4 {

```

```

5   _aTimer.Interval = 5000;
6   _aTimer.Enabled = true;
7   _aTimer.Tick += delegate(object o, EventArgs args)
8   {
9     Log.Logger.Info("The Elapsed event was raised at {0:HH:mm:ss.fff}" + args);
10    GetFrontEndData();
11  };
12  _aTimer.Start();
13 }

```

Windows.UI.Xaml.DispatcherTimer 用在 windows store app 中, 对应于 System.Windows.Forms.Timer。System.Timers.Timer 包装了一下 System.Threading.Timer, 使它有了和 System.Windows.Forms.Timer 类似的接口, 而且也能在 visual studio 的 toolbox designer 里找到。它也是在线程池中执行, 但是如果你是在 visual studio 的 designer 中使用它, visual studio 会自动把它所在的 control 设为这个 timer 的 SynchronizingObject, 这样就会保证 callback 会在 UI 线程调用了。Jeffrey Richter 不建议使用它, 建议直接用 System.Threading.Timer。

16.20.4 线程间操作无效: 从不是创建控件的线程访问它。

从.NET Framework 2.0 开始, 为了安全, 不允许跨线程操作控件。在 C# 中只有 UI 线程才能操作 UI 控件, 如果一个工作线程操作 UI 控件, 就会抛这个异常。通常的解决办法很简单, 就是使用 InvokeRequired。如果返回 True, 说明不在 UI 线程, 需要调用 Invoke 或者 BeginInvoke 来扔给 UI 线程操作。在操作 Winform 窗体时, 需要保证是当前窗体的线程操作, 而不是重新实例化另一个窗体来操作当前窗体, 如果是这样的话, 会报出未创建句柄的错误。获取指定窗体的实例代码如下:

```

1 var instanceForm=Application.OpenForms["FormName"];

```

Application.OpenForms gets a collection of open forms owned by the application. The OpenForms property represents a read-only collection of forms owned by the application. This collection can be searched by index position or by the Name of the Form. 如果 Handle 还没有创建好, 那么即使在后台工作线程, InvokeRequired 也返回 False。This means that InvokeRequired can return false if Invoke is not required (the call occurs on the same thread), or if the control was created on a different thread but the control's handle has not yet been created. You can protect against this case by also checking the value of IsHandleCreated when InvokeRequired returns false on a background thread. 这也解释了“在创建窗口句柄之前, 不能在控件上调用 Invoke 或 BeginInvoke” 错误。解决此种问题的方法是调用窗体的方法时传入此窗体的实例。

使用 delegate 和 Invoke 来从其他线程中调用控件 委托是安全封装方法的类型, 类似于 C 和 C++ 中的函数指针。具体的代码如下所示:

```

1 private void button2_Click(object sender, EventArgs e)
2 {

```

```

3   Thread thread1 = new Thread(new ParameterizedThreadStart(UpdateLabel2));
4   thread1.Start("更新Label");
5 }
6
7 private void UpdateLabel2(object str)
8 {
9   if (label2.InvokeRequired)
10  {
11    // 当一个控件的 InvokeRequired 属性值为真时，说明有一个创建它以外的线程想访问它
12    Action<string> actionDelegate = (x) => { this.label2.Text = x.ToString(); };
13    // 或者
14    // Action<string> actionDelegate = delegate(string txt) this.label2.Text = txt; ;
15    this.label2.Invoke(actionDelegate, str);
16  }
17  else
18  {
19    this.label2.Text = str.ToString();
20  }
21 }

```

第一步先声明一个拥有委托类型的方法，作为代理操作控件的代码：

```

1 /// <summary>
2 /// 在线程中操作窗体的控件
3 /// </summary>
4 /// <param name="action"></param>
5 private void OpeMainFormControl(Action action)
6 {
7   if (InvokeRequired)
8   {
9     this.Invoke(action); // 返回主线程（创建控件的线程）
10  }
11  else
12  {
13    action();
14  }
15 }

```

在窗体中调用控件：

```

1 var t = new Thread(delegate()
2 {
3   // 线程中操作控件，委托给 invoker
4   OpeMainFormControl(delegate
5   {
6     LoadLineIcon("dddd,0");
7   });
8 });
9 t.Start(); // 启动线程

```

也可直接调用委托操作窗体控件。

```

1 public void LoadData(object taskList)
2 {
3     OpeMainFormControl(delegate()
4     {
5         var driverActionInfos = taskList as List<DriverActionInfo>;
6         if (driverActionInfos == null) return;
7         foreach (var singleTask in driverActionInfos)
8         {
9             var taskIndex = driverActionInfos.IndexOf(singleTask);
10            //Create the new row first and get the index of the new row
11            var rowIndex = this.dataGridViewEx1.Rows.Add();
12            //Obtain a reference to the newly created DataGridViewRow
13            var dataGridRow = this.dataGridViewEx1.Rows[rowIndex];
14            //Now this won't fail since the row and columns exist
15            dataGridRow.Cells["vehicleLicense"].Value = singleTask.Carlicense;
16        }
17    });
18 }
```

16.20.5 Lambda 表达式和委托 (delegate)

名字来自微积分数学中的 λ ，其涵义是声明为了表达一个函数具体需要什么。更确切的说，它描述了一个数学逻辑系统，通过变量结合和替换来表达计算。所以，基本上我们有 0-n 个输入参数和一个返回值。Lambda 表达式在.NET 3.5 Framework 中引入，所谓 Lambda 表达式就是 delegate 和匿名方法的简写形式。这种表达式可以取代 delegate，作为方法指针来使用。在 C# 2.0 及 C# 1.x 中，需要使用 delegate 来定义方法指针。从 C# 2.0 开始支持匿名方法，开发人员可以通过匿名方法用内联代码形式取代 delegate。从本质上讲，Lamdba 表达式经过 C# 编译器编译后，仍然会变成 delegate 的形式，也就是说 Lamdba 表达式只是在语法层次上的改进，并不是 IL 提供的新的指令。如下面的两行代码是等价的：

```

1 public delegate bool Filter(int num); // 定义委托
2 Filter filter = i => { return (i & 1) == 0; };
3 Filter filter = delegate(int i) { return ((i & 1) == 0); };
```

如果遇到又需要返回值，又需要参数的时候，就可以考虑用异步。

匿名委托 (Anonymous Delegate) 通过使用匿名方法，由于不必创建单独的方法，因此减少了实例化委托所需的编码系统开销。例如，如果创建方法所需的系统开销是不必要的，则指定代码块（而不是委托）可能非常有用。启动新线程即是一个很好的示例。无需为委托创建更多方法，线程类即可创建一个线程并且包含该线程执行的代码。

```

1 void StartThread()
2 {
```

```
3 System.Threading.Thread t = new System.Threading.Thread (delegate()
4 {
5     System.Console.Write("new Thread is running.");
6 });
7 t.Start();
8 }
```

16.20.6 System.ObjectDisposedException

在使用 Invoke 操作窗体时，向窗体中的 Panel 控件动态添加其他控件时，引发此异常

```
1 AccessibilityObject = "(dotNetBarTabControl2.Controls.Owner).AccessibilityObject" 引发了
2 "System.ObjectDisposedException" 类型的异常
3 其他信息: 无法访问已释放的对象。
```

在调用操作 Panel 的方法中将主窗体的实例传入，使用传入的实例调用方法操作 Panel 的控件即可。

16.20.7 线程退出码 (Exit Code)

Use a non-zero number to indicate an error. In your application, you can define your own error codes in an enumeration, and return the appropriate error code based on the scenario. For example, return a value of 1 to indicate that the required file is not present and a value of 2 to indicate that the file is in the wrong format. For a list of exit codes used by the Windows operating system, see System Error Codes in the Windows documentation. This function returns immediately. If the specified thread has not terminated and the function succeeds, the status returned is STILL_ACTIVE. If the thread has terminated and the function succeeds, the status returned is one of the following values: The exit value specified in the ExitThread or TerminateThread function. The return value from the thread function. The exit value of the thread's process. Important The GetExitCodeThread function returns a valid error code defined by the application only after the thread terminates. Therefore, an application should not use STILL_ACTIVE (259) as an error code. If a thread returns STILL_ACTIVE (259) as an error code, applications that test for this value could interpret it to mean that the thread is still running and continue to test for the completion of the thread after the thread has terminated, which could put the application into an infinite loop.

16.21 Windows 32 接口编程

16.21.1 自动注册 COM 控件

将控件放在安装目录下，程序启动时在未注册控件时自动注册 ocx 控件，DllImport 会按照顺序自动去寻找的地方：1、exe 所在目录 2、System32 目录 3、环境变量目录所以只需要你把引用的 DLL 拷贝到这三个目录下就可以不用写路径。有时在通过 DLLImport 注册控件时提示：无法加载 DLL “..Library/videocx3.ocx”：找不到指定的模块。是因为控件 videocx3.ocx 依赖于其他控件，而其他控件在此路径下不存在导致。使用 Dependency Walker 查看 ocx 控件的依赖，注意查看 ocx 控件依赖时最好将之拷贝到单独的目录下，这样才能真正看到 ocx 控件缺少的依赖项，如图16.34所示：

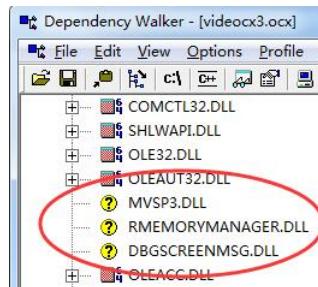


图 16.34: 查看 ocx 控件的依赖

注册 ocx 控件的代码为：

```

1 [DllImport("videocx3.ocx")]
2 private static extern int DllRegisterServer();
3
4 //在函数中调用注册方法
5 public static void IsDllRegisteredSuccess(string[] classId)
6 {
7     foreach (var currentClassId in classId)
8     {
9         var registerKey = Registry.ClassesRoot.OpenSubKey("CLSID\\" + currentClassId);
10        if (registerKey != null) continue;
11        var i = DllRegisterServer();
12        if (i < 0)
13        {
14            Log.Logger.Error("Control registered failed");
15        }
16    }
17}

```

16.22 插件 (Plug-In)

16.22.1 定义

插件是一种遵循统一的预定义接口规范编写出来的程序，应用程序在运行时通过接口规范对插件进行调用，以扩展应用程序的功能。插件在英文中通常称为 plug-in、plugin 或者 plug in。插件最典型的例子是 Microsoft 的 ActiveX 控件和 COM (Component Object Model，部件对象模型) 实际上 ActiveX 控件不过是一个更高继承层次的 COM 而已。此外还有 Photoshop 的滤镜 (Filter) 也是一种比较常见的插件。

关于 ActiveX 和 COM 在 Microsoft 的 .Net Framework 推出的之前（大约是 2003 年之前吧），ActiveX 和 COM 可是炙手可热的技术啊！在那个年代，一个顶尖的 VC++ 高手的标志是什么？是会 COM 编程！不知道 IUnknown 接口和 QueryInterface 函数，你怎么可能通过 Microsoft 的 MCSD 认证考试？现在当然不同了，我曾经见过不少断言 COM 和 ActiveX 已经消亡或终将消亡的文章。但是不管怎么说，个人认为，ActiveX 和 COM 代表了插件技术的最高境界，通过对 ActiveX 和 COM 的研究，我们可以对插件有更深刻的认识。关于 ActiveX 控件和 COM 技术的详细介绍，有兴趣的朋友不妨去“百度一下”，相信能够获得很多相关信息的。

插件技术过时了吗？COM 技术的逐渐淡出，使不少程序员产生了困惑：插件技术已经过时了吗？NO！至少我不这样认为！毕竟，没有了插件技术，我们还有什么更好的方法为应用程序提供运行时的功能扩展呢？COM 的没落自然有其原因，例如编程实在是太复杂而难以掌握，还有就是在这个病毒和木马肆虐的年代，其安全性也令人堪忧。但至少我们可以看到，插件技术的成功应用还是有的：比如 PhotoShop 的滤镜，比如各大主流工控软件的功能扩展。对于插件的理解，我们应该注意以下几点：一、插件是遵循统一的预定义接口规范编写的。

16.22.2 插件的优缺点

二、应用程序是在运行时调用插件以实现功能扩展的插件最吸引人的地方当然就是其所实现“运行时 (run-time)”功能扩展。这意味着软件开发者可以通过公布插件的预定义接口规范，从而允许第三方的软件开发者通过开发插件对软件的功能进行扩展，而无需对整个程序代码进行重新编译。运行时 (run-time) 是相对于编译时 (assembly-time) 而言的。一般来说，软件开发者对软件功能更新时，是在源代码级别进行更新，然后对整个程序或部分动态链接库 (DLL) 进行重新编译，进而发布应用程序的新版本，这就是编译时 (assembly-time) 的软件更新。三、插件技术的优缺点运行时的软件功能扩展其优点是显而易见的：1、对软件的开发者而言，只需对主程序和某些常用插件进行更新和维护，然后通过公布插件接口吸引第三方的软件开发者对主程序的功能进行扩展，这是一种“我为人人，人人为我”的双赢策略；2、对最终用户而言，可以通过有选择地购买第三方提供的插件实现自己所需要的功能，从而实现最佳性价比组合，以节省不必要的开支。但是，运行时的软件功能扩展也有其弊端：1、为实现运行时的软件扩展，程序开发者必须编写更多、更复杂的代码，从而会对程序的执行效率产生一定的影响。关于这一点，我会在第二讲中详细论述；2、由于插件是在运行时加载的，因此第三方插件可能对用户造成危害。这种危害通常可以分为两类：(1) 由于插件开发者的技术水平原因导致的插件 BUG，

这种 BUG 可能导致内存泄露、死机、数据丢失等等故障，从而影响到用户对软件的使用；（2）插件开发者恶意开发类似于病毒和木马的插件，窃取或毁坏用户数据，使用户遭受不必要的损失；为了避免此类缺点，软件开发者可能需要付出额外的代价，如需要对第三方插件进行检验和认证，或者干脆不对外提供插件开发接口，仅由自己提供插件。

16.22.3 插件的实现

反射 (reflection) 机制可以用来加载插件，接口和抽象类可以用于和插件通信。

16.23 .NET Core

16.24 常见问题

不是有效的 Win32 应用程序 查看程序的架构是否一样，是统一的 x86 还是 x64。一般情况下都是由于架构不一致导致的此种问题。

没有注册类 (异常来自 HRESULT:0x80040154 (REGDB_E_CLASSNOTREG)) 将项目改为 x86 平台。

Chapter 17

基础引擎 (Fundation Engine)

消息引擎、实体引擎、服务引擎、工作流引擎、规则引擎

编程语言、编译器、解释器、数据库与操作系统、Web 服务器、网络开发框架。

17.1 HTTP 服务

```
1 import socket
2 HOST, PORT = '', 8888
3
4 listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5 listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
6 listen_socket.bind((HOST, PORT))
7 listen_socket.listen(1)
8 print 'Serving HTTP on port %s ...' % PORT
9 while True:
10     client_connection, client_address = listen_socket.accept()
11     request = client_connection.recv(1024)
12     print request
13
14     http_response = """\
15         HTTP/1.1 200 OK
16
17         Hello, World!
18
19
20     client_connection.sendall(http_response)
21     client_connection.close()
```

17.2 解释器和编译器 (Compiler)

首先编译器进行语法分析，也就是要把那些字符串分离出来。然后进行语义分析，就是把各个由语法分析分析出的语法单元的意义搞清楚。最后生成的是目标文件，也称为 obj 文件。再经过链接器的链接就可以生成最后的 EXE 文件了。有些时候需要把多个文件产生的目标文件进行链接，产生最后的代码。这一过程称为交叉链接。

17.3 数据库 (Database)

二叉查找树的结构不适合数据库，因为它的查找效率与层数相关。越处在下层的数据，就需要越多次比较。极端情况下， n 个数据需要 n 次比较才能找到目标值。对于数据库来说，每进入一层，就要从硬盘读取一次数据，这非常致命，因为硬盘的读取时间远远大于数据处理时间，数据库读取硬盘的次数越少越好。

B 树是对二叉查找树的改进。它的设计思想是，将相关数据尽量集中在一起，以便一次读取多个数据，减少硬盘操作次数。Oracle 数据存取上限 8EB，SQL Server 数据库大小 524,272TB（当然可以允许多个数据库）。MySQL 的单表限大小为 4GB，当时的 MySQL 的存储引擎还是 ISAM 存储引擎。但是，当出现 MyISAM 存储引擎之后，也就是从 MySQL 3.23 开始，MySQL 单表最大限制就已经扩大到了 64PB 了（官方文档显示）。也就是说，从目前的技术环境来看，MySQL 数据库的 MyISAM 存储引擎单表大小限制已经不是由 MySQL 数据库本身来决定，而是由所在主机的 OS 上面的文件系统来决定了。

17.4 操作系统 (Operation System)

Chapter 18

Android

18.0.1 环境搭建

18.1 常见问题

18.1.1 The import android.support cannot be resolved

解决办法是增加所缺的 jar 包。步骤如下：

- 1、在 Eclipse 中，右击当前工程，选择 Properties
- 2、选择 Java Build Path
- 3、选择 Libraries tab，点击右边面板的 Add External JARs 按钮
- 4、选择 android-support-v4.jar 文件，在你的 andriod 的 sdk 目录下：

`\android-sdks\extras\android\support\v4\android-support-v4.jar`

18.2 Android SDK

18.2.1 离线安装

国内下载 Android SDK 很是不方便，可以通过 360 管家下载 Android SDK 进行安装，不过本人不用 360 的产品，可以翻墙从 Google 下载，非常麻烦。

18.3 PhoneGap

根据 Forrester 针对开发人员的调查研究显示，在 2014 年，31% 的开发人员编写的都是原生应用程序；27% 的编写的是基于 Web 的应用程序；22% 的编写的是混合应用程序（Apache Cordova 或 Adobe 的 PhoneGap(基于 Cordova)）；12% 的使用的是“跨平台”的方式，在该方式中，他们在诸如 Xamarin 的一款平台编写代码，而其不是基于 HTML 5 的。Facemire 认为企业只将一种方法作为公司的规范会是一个错误。他说：“如果您企业只构建原生应用程序，移动 Web 网站，或混合应用程序都是非常短视的。这应该与您企业发展的需求相匹配。一家企业可能希望为他们的消费者构建原生的应用程序，但同时在企业内部使用移动 Web 网络，如公司名录。”

18.3.1 PhoneGap 安装

JDK XuYY，这是 JDK 发布的版本格式。其中 X 代表大版本号，每个大版本有自己的新特性，如 JDK7 和 JDK8 的特性是不一样的。u 是 update 的首字母，所以 YY 代表的是小版本号，一般是对当前大版本的一些细节更新，如 bug 修复，打补丁等。

```
1 #安装 PhoneGap
2 npm install -g phonegap
3
4 #查看安装的 PhoneGap 版本
5 npm install -g phonegap
```

设置 Android 路径，变量名：ANDROID_SDK_HOME，变量值是 android-SDK-windows 的路径。

18.3.2 Cordova 安装

狗日的 PhoneGap 又搞成 Cordova 了，乱搞，苦逼程序员。安装 Cordova CLI (cordova command-line interface)，输入如下命令：

```
1 #安装 Cordova
2 npm install -g cordova
3
4 #创建一个新的 Cordova 项目
5 cordova create hello com.example.hello Helloworld
6
7 #为该项目添加 Android 平台支持
8 cordova platform add android
9
10 #运行 Cordova
11 cordova run android
12 #运行，指定模拟器
13 cordova run --target=name_of_your_emulator android
14
```

```
15 #build 应用  
16 cordova build  
17  
18 #启动 Android 模拟器  
19 emulator.exe avd Dolphin  
20  
21 #Cordova 调试器  
22 npm install -g ripple-emulator
```

cordova create 项目时，第一个参数为创建项目的路径，第二个参数 (com.example.hello) 为 ID，第三个参数 (Helloworld) 为项目名称。build 应用成功以后会输出一个 apk 安装文件，将生成的 apk 拷贝到手机上面运行即可。

18.4 React Native for Android

18.4.1 环境搭建

安装

```
1 npm install -g react-native-cli  
2  
3 #初始化项目  
4 react-native init MyProject  
5  
6 #To run your app on Android, 在你的模拟器设备上面安装生成的应用  
7 react-native run-android
```


Chapter 19

Windows SDK

大致说来 windows 编程有两种方法：1.windows c 方式（SDK），2.c++ 方式：即对 SDK 函数进行包装，如 VC 的 MFC,BCB 的 OWL 等，如果要深入下去，还是要熟悉 SDK。

两种方法有哪些区别呢：SDK 编程就是直接调用 windows 的 API 进行编程，但是有上千个 API 组成（win95 的 API 有两千多个），这种数目太大了，对于编程显然不利。而 MFC 把这些 API 封闭起来，共有一百多个类组成。一般只需 20 多个 windows 类和另外 20 多个通用的非 windows 类就可“干活”了，这一改变无疑是很有好处的。尽管 MFC 如此方便，但是要学 VC，直接去学 MFC 却是不明智的选择。只有在熟悉了 MFC 的运行机制的情况下，才有可能深入下去。那些如多少天精通什么什么的书籍其实讲的全是些如怎么使用 VC 这种工具的话题，学来学去学会了怎么会使用 VC 这种工具，而不能深入 MFC 编程。象 VB 这类工具就更令人感觉到太闷了，不过各有各的好处。

MFC 虽然提高了程序员编程的效率，但是也失去了 SDK 编程的灵活性在 Windows 平台上，最常见最流行的编程就是 MFC 编程了，在网上可以搜索出大把的 MFC 编程相关的文章，今天我们来讨论另外一种 windows 下的编程模式，即 Windows SDK 编程。这种编程具有更加灵活和强大的控制，能实现一些 MFC 不易实现甚至难以实现的功能。

其实 Windows 的三大模块就是以 DLL 的形式提供的（Kernel32.dll, User32.dll, GDI32.dll），里面就含有了 API 函数的执行代码。为了使用 DLL 中的 API 函数，我们必须要有 API 函数的声明 (.H) 和其导入库 (.LIB)，函数的原型声明不难理解，那么导入库又是做什么用的呢？我们暂时先这样理解：导入库是为了在 DLL 中找到 API 的入口点而使用的。所以，为了使用 API 函数，我们就要有跟 API 所对应的.H 和.LIB 文件，而 SDK 正是提供了一整套开发 Windows 应用程序所需的相关文件、范例和工具的“工具包”。到此为止，我们才真正的解释清楚了 SDK 的含义。由于 SDK 包含了使用 API 的必需资料，所以人们也常把仅使用 API 来编写 Windows 应用程序的开发方式叫做“SDK 编程”。而 API 和 SDK 是开发 Windows 应用程序所必需的东西，所以其它编程框架和类库都是建立在它们之上的，比如 VCL 和 MFC，虽然他们比起“SDK 编程”来有着更高的抽象度，但这丝毫不妨碍它们在需要的时候随时直接调用 API 函数。

Chapter 20

SOA

20.1 SOA 简介

SOA(Service-Oriented Architecture)，即面向服务的架构，这是最近一两年出现在各种技术期刊上最多的词汇了。现在有很多架构设计师和设计开发人员简单的把 SOA 和 Web Services 技术等同起来，认为 SOA 就是 Web Service 的一种实现。本质上来说，SOA 体现的是一种新的系统架构，SOA 的出现，将为整个企业级软件架构设计带来巨大的影响。

基于以上图示，SOA 具有以下五个特征：

- **可重用**一个服务创建后能用于多个应用和业务流程。
- **松耦合**服务请求者到服务提供者的绑定与服务之间应该是松耦合的。因此，服务请求者不需要知道服务提供者实现的技术细节，例如程序语言、底层平台等等。
- **明确定义的接口**服务交互必须是明确定义的。Web 服务描述语言 (Web Services Description Language, WSDL) 是用于描述服务请求者所要求的绑定到服务提供者的细节。WSDL 不包括服务实现的任何技术细节。服务请求者不知道也不关心服务究竟是由哪种程序设计语言编写的。
- **无状态的服务设计**服务应该是独立的、自包含的请求，在实现时它不需要获取从一个请求到另一个请求的信息或状态。服务不应该依赖于其他服务的上下文和状态。当产生依赖时，它们可以定义成通用业务流程、函数和数据模型。
- **基于开放标准**当前 SOA 的实现形式是 Web 服务，基于的是公开的 W3C 及其他公认标准。采用第一代 Web 服务定义的 SOAP、WSDL 和 UDDI 以及第二代 Web 服务定义的 WS-* 来实现 SOA。

20.2 微服务 (Microservices)

20.2.1 概念

微服务是个新概念，但它没有一个明确的定义，各家对微服务的描述不尽相同，本人更倾向于用一些架构原理来描述它，因为架构原理是相对抽象和稳定的，而具体实现可以千差万别。微服务原理和软件工程，面向对象设计中的基本原理相通，体现如下：

单一职责 (Single Responsibility) 一个服务应当承担尽可能单一的职责，服务应基于有界的上下文 (bounded context, 通常是边界清晰的业务领域) 构建，服务理想应当只有一个变更的理由 (类似 Robert C. Martin 讲的：A class should have only one reason to change)，当一个服务承担过多职责，就会产生各种耦合性问题，需要进一步拆分使其尽可能职责单一化。

关注分离 (Separation of Concerns)，跨横切面逻辑，例如日志分析、监控、限流、安全等等，尽可能与具体的业务逻辑相互分离，让开发人员能专注于业务逻辑的开发，减轻他们的思考负担，这个也是有界上下文 (bounded context) 的一个体现。

模块化 (Modularity) 和分而治之 (Divide & Conquer)，这个是解决复杂性问题的一般性方法，将大问题（如单块架构）大而化小（模块化和微服务化），然后分而治之。

微服务架构同时还是一个组织原理的体现，这个原理就是康威定律 (Conway's Law)，Melvin Conway 在 1968 年指出：“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure”，翻译成中文就是：设计系统的组织，其产生的设计和架构等价于组织间的沟通结构。Dan North 对此还补充说：“Those systems then constrain the options for organization change”，简言之，这些系统在建成之后反过来还会约束和限制组织的改变。

20.2.2 用户体验适配层 (Backend For Frontend)

随着无线技术的发展和各种智能设备的兴起，互联网应用已经从单一 Web 浏览器时代演进到以 API 驱动的无线优先 (Mobile First) 和面向全渠道体验 (omni-channel experience oriented) 的时代。

Chapter 21

Java

21.1 JAX-WS(Java™ API for XML Web Services)

通过使用 Java™ API for XML Web Services (JAX-WS) 技术设计和开发 Web 服务，可以带来很多好处，能简化 Web 服务的开发和部署，并能加速 Web 服务的开发。

21.1.1 Google 调试 WebSocket

在 Google 浏览器中查看 Socket。

```
1 chrome://net-internals/#sockets
```

当你访问 localhost:8080 时，也就是访问 root 下面的文件。所以展示不出主页时，确认下 root 文件夹内容。

```
Usage: java javassist.tools.web.Webserver <port number>
```

21.2 MyEclipse

21.2.1 常用操作

21.3 Maven

Maven is a software tool that helps you manage Java projects, and automate application builds. IntelliJ IDEA fully integrates with Maven version 2.2 and later versions, allowing you to create or import Maven modules, download artifacts and perform the goals of the build lifecycle and plugins.

人们离不开 Maven，其实是离不开它这两个功能 1. 包管理 2. 项目打包。Maven 是个独立的工具，安装之后可以在命令行界面使用。IDE 需要安装 Maven 插件，关联上已经安装的 Maven 之后，才能在 IDE 中调用 Maven 的功能。其中 Eclipse 的 Maven 插件为 m2eclipse。IntelliJ 则默认自带了 Maven 插件，直接做相应的关联配置即可。

21.4 Apache MINA(Multipurpose Infrastructure for Network Applications)

mina 2.0 在合理的负载下 1000 个并发，最高能达到 5076.81rps(Requests Per Second)。Apache MINA(Multipurpose Infrastructure for Network Applications) 是 Apache 组织一个较新的项目，它为开发高性能和高可用性的网络应用程序提供了非常便利的框架。当前发行的 MINA 版本支持基于 Java NIO 技术的 TCP/UDP 应用程序开发、串口通讯程序(只在最新的预览版中提供)，MINA 所支持的功能也在进一步的扩展中。新项目里的 web service 需要高并发服务能力，tomcat 目前最多支持到 200 个并发。Apache MINA is a network application framework which helps users develop high performance and high scalability network applications easily. It provides an abstract event-driven asynchronous API over various transports such as TCP/IP and UDP/IP via Java NIO.

目前主流网站都是由开源软件构建的。使用 Nginx 做为 Web 服务器，Tomcat/Resin 做 App 容器，Memcached 做通用 Cache，MySQL 做数据库，使用 Linux 操作系统。网站系统刚上线初期，用户数并不多，所有的模块都整合一个系统中，所有业务由一个应用提供，此时采取将全部的逻辑都放在一个应用的方式利于系统的维护和管理。但是，随着网站用户的不断增加，系统的访问压力越来越大，为了满足越来越多用户的需求，原有的系统需要增加新的功能进来，随着系统功能模块的增多，系统就会变得越来越难以维护和扩展，同时系统伸缩性和可用性也会受到影响。例如一个网站初期只有用户服务功能，随着网站发展，可能会需要用户信息中心、充值支付中心、商户服务中心等越来越多的子系统，如果把这些子系统都整合在原有的系统中，整个网站将会变得非常复杂，并且难以维护。另外，由于所有子系统都整合在一起，只要有一个模块出问题，那么所有的功能都会受影响，造成非常严重的后果。所以系统发展遇到的瓶颈就是随着系统的发展，如果所有模块都整合在一起，系统的可伸缩性和扩展性将受到影响。遇到以上瓶颈该如何解决呢？明智的办法就是系统拆分，将系统根据一定的标准，比如业务相关性等拆分为不同的子系统，不同的子系统负责不同的业务功。拆分完成后，每个子系统单独进行扩展和维护，不会影响其他子系统，从而大大提高整个网站系统的扩展性和可维护性，同时系统的水平伸缩性也大大提升了。对于压力比较大的子系统可以再进行扩展而不影响其他子系统，如果某个子系统出现问题也不会影响其他服务。从而增强了整个网站系统的健壮性，更有利于保障核心业务。因此一个大型的互联网应用，肯定是要经过系统拆分的，因为只有进行拆分，系统的扩展性、维护性、伸缩性、可用性才会变得更好。但是拆分也会给系统带来问题，就是子系统之间如何通信。本文介绍 MINA2 就是用来充当消息中间件解决各子系统之间的通信问题

21.5 Apache Kafka

使用 kafka 的理由：1. 分布式，高吞吐量，速度快（kafka 是直接通过磁盘存储，线性读写，速度快：避免了数据在 JVM 内存和系统内存之间的复制，减少耗性能的对象创建和垃圾回收）2. 同时支持实时和离线两种解决方案（相信很多项目都有类似的需求，这也是 LinkedIn 的官方架构，我们是一部分数据通过 storm 做实时计算处理，一部分到 hadoop 做离线分析）。3. open source (open source 谁不喜欢呢) 4. 源码由 scala 编写，可以运行在 JVM 上（笔者对 scala 很有好感，函数式语言一直都挺帅的，spark 也是由 scala 写的，看来以后有空得刷刷 scala）

21.5.1 Apache Kafka 安装

1、关闭 SELINUX

```

1 vi /etc/selinux/config
2
3 #SELINUX=enforcing #注释掉
4
5 #SELINUXTYPE=targeted #注释掉
6
7 SELINUX=disabled #增加
8
9 :wq! #保存退出
10
11 setenforce 0 #使配置立即生效

```

21.5.2 安装 JDK

kafka 运行需要 JDK 支持

下载 JDK，链接为：<http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>。

```

1 #启动 MySQL 守护进程
2 service start mysqld
3
4
5 #卸载 MySQL

```

mysql.server 脚本通过调用 mysqld_safe 启动服务器，该脚本可以通过参数 start 和 stop 指定启动还是关闭。mysql.server 脚本在 MySQL 安装目录下的 share/mysql 目录中，如果是采用源码安装的 MySQL，则可以在 support-files 目录里找到。

```

1 find <指定目录> <指定条件> <指定动作>
2
3 #搜索 home 目录(含子目录) 中所有文件名以 a 开头的文件并显示它们的详细信息

```

```
4 find /home -type f -mmin -10
5
6 #查找名为 support-files 的文件夹
7 find / -type d -name support-files
8
9 #查找名为 my 的文件
10 find . -name 'my'
11 #从根目录开始查找名为 lan.conf 的文件
12 find / -name "lan.conf"
```

21.6 Apache Tomcat

21.6.1 常识 (Common Sense)

Apache Tomcat 部署的项目存在于 webapps 目录下，按照项目名称进行组织，启动项目后可根据项目名称进行访问，例如部署了一个新的项目名字叫 DeviceGPS，下有一个文件 Index.jsp，那么相应的访问 URL 路径为：

```
1 http://192.168.24.79:8081/DeviceGPS/index.jsp
```

Chapter 22

Python

Python 的拟人化表示如图22.1所示。



图 22.1: Python 拟人化理解

Python，是一种面向对象、直译式的计算机程序语言，具有近二十年的发展历史。它包含了一组功能完备的标准库，能够轻松完成很多常见的任务。它的语法简单，与其它大多数程序设计语言使用大括号不一样，它使用缩进来定义语句块。

与 Scheme、Ruby、Perl、Tcl 等动态语言一样，Python 具备垃圾回收功能，能够自动管理内存使用。它经常被当作脚本语言用于处理系统管理任务和网络程序编写，然而它也非常适合完成各种高级任务。Python 虚拟机本身几乎可以在所有的作业系统中运行。使用一些诸如 py2exe、PyPy、PyInstaller 之类的工具可以将 Python 源代码转换成可以脱离 Python 解释器

运行的程序。

Python 的官方解释器是 CPython，该解释器用 C 语言编写，是一个由社区驱动的自由软件，目前由 Python 软件基金会管理。

Python 支持命令式程序设计、面向对象程序设计、函数式编程、面向侧面的程序设计、泛型编程多种编程范式。Python 的设计哲学是“优雅”、“明确”、“简单”。Python 开发者的哲学是“用一种方法，最好是只有一种方法来做一件事”，也因此它和拥有明显个人风格的其他语言很不一样。在设计 Python 语言时，如果面临多种选择，Python 开发者一般会拒绝花俏的语法，而选择明确没有或者很少有歧义的语法。这些准则被称为“Python 格言”。

22.1 框架

22.1.1 Django

pip is a package manager for Python. It makes installing and uninstalling Python packages (such as Django!) very easy. For the rest of the installation, we'll use pip to install Python packages from the command line.

安装的过程中可能会出现“ImportError: No module named setuptools”的错误提示，这是新手很常遇见的错误提示。不用担心，这是因为 Windows 环境下 Python 默认是没有安装 setuptools 这个模块的，这也是一个第三方模块。下载地址为 <http://pypi.python.org/pypi/setuptools>。如果是 Windows 环境的话，下载 exe 形式的安装程序就可以了（傻瓜式安装，非常快）。安装了 setuptools 之后，再运行“python setup.py install”就可以方便地安装各种第三方模块了。

22.1.2 paramiko

paramiko 是用 python 语言写的一个模块，遵循 SSH2 协议，支持以加密和认证的方式，进行远程服务器的连接。由于使用的是 python 这样的能够跨平台运行的语言，所以所有 python 支持的平台，如 Linux, Solaris, BSD, MacOS X, Windows 等，paramiko 都可以支持，因此，如果需要使用 SSH 从一个平台连接到另外一个平台，进行一系列的操作时，paramiko 是最佳工具之一。从远程服务器上使用 ssh 下载文件时，不需要对远程服务器进行配置，特别是在应对多台服务器时。

22.1.3 paramiko 安装

安装 paramiko 有两个先决条件，python 和另外一个名为 PyCrypto 的模块。pycrypto 是用 Python 写的加密工具包。This is a collection of both secure hash functions (such as SHA256 and RIPEMD160), and various encryption algorithms (AES, DES, RSA, ElGamal, etc.).

安装 easy_install 工具 安装完毕后记得将路径 C:/Python27/Scripts 添加到环境变量中。

安装 PyCrypto 模块 运行 python.exe, 在提示符下输入: Import Crypto, 如果没有出现错误则证明安装成功。

安装 paramiko 使用如下命令安装 paramiko:

```
1 #使用 easy_install 安装  
2 easy_install paramiko  
3  
4 #使用 pip 安装  
5 pip install paramiko
```

安装时提示: Microsoft Visual C++ 9.0 is required (Unable to find vcvarsall.bat). 按照提示安装相应的 C++ 版本 (Microsoft Visual C++ Compiler for Python 2.7) 即可。安装 paramiko 时提示: 安装时提示: ImportError: No module named cryptography.hazmat.backends。通过如下命令安装 pyOpenSSL 可以解决:

```
1 easy_install pyOpenSSL==0.13  
2  
3 pip install pyOpenSSL==0.13
```

安装 pyOpenSSL 可能出现错误, 可尝试按照命令行提示升级 pip(python -m pip install -upgrade pip)。

22.2 工具

22.2.1 编码风格

不要在行尾加分号, 也不要将两条命令放在同一行.

22.2.2 Python Tools for Visual Studio

22.2.3 PyCharm

显示行号:File->Settings->Editor->General->Appearance->Show line numbers。

22.3 初级任务

22.3.1 Python 进制转换

```
1 #十进制转换二进制  
2 >>> bin(10)
```

```

3 '0b1010'
4
5 #十进制转换十六进制
6 >>> hex(10)
7 '0xa'
8
9 #二进制转换十进制
10 >>> int('1010',2)
11 10
12
13 #十六进制转换十进制
14 >>> int('0xa',16)
15 10
16
17 #十六进制转换二进制
18 >>> bin(0xa)
19 '0b1010'
20
21 #二进制转换十六进制
22 >>> hex(0b1010)
23 '0xa'

```

22.3.2 使用 Python 下载文件

使用 Python 的 paramiko 模块下载文件，可以不用在服务器上配置，多台服务器上下载文件时优势明显。

```

1 #!/usr/bin/python
2
3 import sys
4 import paramiko
5 import ssh
6
7 def download():
8     print("scan")
9     sshClient=ssh.SSHClient()
10    sshClient.set_missing_host_key_policy(ssh.AutoAddPolicy())
11    sshClient.connect("10.6.12.14",port = 21,username = "root",password = "123456")
12    stdin,stdout,stderr = sshClient.exec_command("ls -alh")
13    stdout.read()
14    sftp = sshClient.open_sftp()
15    sftp.get('/root/package/scripts/log/log.log','G:\\a.log')
16
17 if __name__=="__main__":
18     print("main")
19     download()

```

22.3.3 Python 引入目录下文件

python 中，每个 py 文件被称之为模块，每个具有 `__init__.py` 文件的目录被称为包。只要模块或者包所在的目录在 `sys.path` 中，就可以使用 `import` 模块或 `import` 包来使用。如果你要使用的模块（py 文件）和当前模块在同一目录，只要 `import` 相应的文件名就好。要包含目录下的文件时需要在目录下声明一个 `__init__.py` 文件，即使这个文件是空的也可以。当然这个文件也可以初始一些数据。

```
1 #!/usr/bin/python
2
3 import Ssh.SshDown
4
5 def sshdownload():
6     Ssh.SshDown.downfile()
```

其中 `Ssh` 为当前的工作目录。也可以用此种方式导入路径：

```
1 #!/usr/bin/python
2
3 sys.path.append("D:\\\\OneDrive\\\\Document\\\\doc\\\\DolphinDev\\\\DolphinDevManual\\\\Scripts
4 \\Python\\\\Source\\\\Python\\\\UI\\\\Widgets")
5 import protocolParser
```

22.3.4 python xml 解析

22.3.5 Python 记录日志

使用 python 的标准日志模块。python 社区将日志做成了一个标准模块。它非常简单易用且十分灵活。定义一个日志的公共记录模块。

```
1 #coding=utf-8
2 import logging
3
4 logging.basicConfig(level=logging.INFO)
5 logger = logging.getLogger(__name__)
6
7 #Create a file handler
8 handler = logging.FileHandler('output.log')
9 handler.setLevel(logging.INFO)
10
11 #Create a logging format
12 formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
13 handler.setFormatter(formatter)
14
15 # add the handlers to the logger
16 logger.addHandler(handler)
17
```

```
18 def loginfo():
19     logger.info('info')
20     return
```

在任意模块导入即可试用公共模块记录日志了，非常方便。

```
1 #coding=utf-8
2
3 import commonlogger
4
5 commonlogger.loginfo()
```

22.3.6 试用配置文件定义日志

可以在 python 代码中配置你的日志系统，但是这样并不够灵活。最好的方法是使用一个配置文件来配置。在 Python2.7 及之后的版本中，你可以从字典中加载 logging 配置。这也就意味着你可以从 JSON 或者 YAML 文件中加载日志的配置。尽管你还能用原来.ini 文件来配置，但是它既很难读也很难写。设置日志记录的话，在程序启动的时候调用 log 模块的 setup_logging 方法就行了。它默认读取 logging.json 或是 logging.yaml。你可以设置 LOG_CFG 环境变量来指定从某个路径中加载日志配置。

22.4 常见问题

22.4.1 TabError: inconsistent use of tabs and spaces in indentation

Python 的代码块通过缩进对齐表达代码逻辑，而不是使用大括号（从此告别神圣的大括号战争了），Python 支持制表符缩进和空格缩进，但 Python 社区推荐使用四空格缩进。也可以使用制表符缩进，但切不可以混用两种缩进符号。

Chapter 23

Linux

23.1 shell

23.1.1 风格

Shell 的风格推荐¹。

缩进两个空格，没有制表符

请将; do , ; then 和 while , for , if 放在同一行

所有的错误信息都应该被导向 STDERR 推荐使用类似如下函数，将错误信息和其他状态信息一起打印出来。

```
1 err() {
2     echo "[$(date +'%Y-%m-%dT%H:%M:%S%z')]: $@"
3 }
4
5 if ! do_something; then
6     err "Unable to do_something"
7     exit "${E_DID NOTHING}"
8 fi
```

23.1.2 常用 Shell 脚本

判断文件夹是否存在 判断文件夹是否存在如下脚本所示：

```
1 #!/bin/sh
```

¹<http://zh-google-styleguide.readthedocs.io/en/latest/google-shell-styleguide/formatting/>

```

2 if [ ! -d "/root/package/my" ] ;then
3 mkdir /root/package/my
4 fi

```

注意条件判断语句的中括号两边都需要留有空格。

使用 tcpdump 抓取包 如下代码使用 tcpdump 抓取指定的包，并按照时间存放到不同的目录下。

```

1 #!/bin/bash
2 currentDate=`date '+%Y-%m-%d'`
3 currentTime=`date +'Y%m%d%H%M%S'`
4 packageStoreDir=/root/package/location/$currentDate
5 if [ ! -d "$packageStoreDir" ] ; then
6 mkdir "$packageStoreDir"
7 fi
8 #tcpdump -i eth1 port 6973 and host 12.26.32.14 -w packageStoreDir/currentTime.cap
9 tcpdump -i eth1 port 6973 and host 12.26.32.14 -w $packageStoreDir/$currentTime.cap

```

要让进程在后台运行，一般情况下，我们在命令后面加上 & 即可，实际上，这样是将命令放入到一个作业队列中。但是如上方到后台执行的进程，其父进程还是当前终端 shell 的进程，而一旦父进程退出，则会发送 hangup 信号给所有子进程，子进程收到 hangup 以后也会退出。如果我们要在退出 shell 的时候继续运行进程，则需要使用 nohup 忽略 hangup 信号，或者 setsid 将父进程设为 init 进程（进程号为 1）。

```
1 nohup ./capture.sh &
```

可以使用 jobs 命令查看后台的进程。

批量终止进程 批量终止指定的进程如下命令所示：

```

1 #!/bin/bash
2 for processId in $(ps -ef|grep "tcpdump"|grep -v grep|cut -c 10-15); do
3 if [ $processId != 1 ] ;then
4 kill -9 $processId
5 echo "kill $processId success" >> ./log/log.log
6 fi
7 done

```

此脚本用于结束计算机中所有的 tcpdump 进程，cut -c 10-15 用于过滤出进程的 ID 集合。

判断进程是否存在 有时在启动一个进程时，需要判断启动的进程是否已经存在从而进行不同的操作，判断进程是否存在如下代码片段所示。

```
1 #!/bin/bash
```

```
2 process_exists=false
3 for processId in $(ps -ef|grep "tcpdump"|grep -v grep|cut -c 10-15); do
4   if [ $processId != 1 ] ;then
5     process_exists=true
6   echo "tcpdump process already exists,will not start capture process" >> ./log/
7           capture.log
8   fi
9 done
10 if [ "$process_exists" != true ] ;then
11   nohup /root/package/scripts/capture.sh &
12   fi
```

shell 变量名字是大小写敏感的。给变量赋值时，等号周围不能有任何空白符。如此处的:process_exists=true。

23.1.3 shell 日志处理

23.2 基础

Linux 下安装应用程序时它的默认安装路径。默认目录一般建议为/usr 所有的软件都仍这里。

23.2.1 常用命令

输入”cat /etc/issue”，显示的是发行版本信息。解压 tar.gz 文件：

```
1 #查看当前登录用户的组内成员，/etc/group 文件包含所有组
2 yum install texlive-scheme-full
3
4 #改变文件夹所属用户和组
5 #v 表示输出详细信息
6 #R 表示递归 (recursive)
7 chmod -vR dolphin:dolphin DolphinDevMannual
8
9 tar -zxvf 压缩文件名.tar.gz
10
11 #查看 Linux 发行版本
12 cat /proc/version
13
14 #查看进程
15 ps -aux
16
17 #开机自动启动 Apach 和 MySQL
18 chkconfig httpd on
19 chkconfig mysqld on
```

x 代表从文件中 extract, f 是文件 (file) 的意思, z 参数代表 gzip, filter the archive through gzip, v 代表 verbose, 即输出详细的信息。

从服务器拷贝文件 scp 是有 Security 的文件 copy, 基于 ssh 登录。

```

1 scp -r -P 50000 root@servername:/root/package/scripts ./
2
3 #将本地文件夹上传到服务器
4 scp -r /tmp/local_dir username@servername:remote_dir

```

P 参数指定端口为自定义端口, 此处为 50000, 服务名称可以是 IP 地址。

使用命令输出作为参数

```

1 cd |xargs whereis texlive
2
3 #杀掉进程名称带 dolphin 的进程
4 ps -ef|grep dolphin|grep -v grep|cut -c 9-15|xargs kill -9

```

23.2.2 日志查看

```

1 #查看位置上报信息
2 tail -f Device.log |grep "7E 02 00"
3
4 #过滤设备的报警信息
5 tail -f MVSP_DeviceServer.log |grep "7E 02 00 00 . . . . . [^0] [^0]
6 [^0] [^0] [^0] [^0] [^0] [^0]"
7 #查看设备上报的 GPS 信息

```

23.2.3 网络

自动获取 IP 虚拟机使用桥接模式, 相当于连接到物理机的网络里, 物理机网络有 DHCP 服务器自动分配 IP 地址。

```

1 #自动获取 ip 地址命令
2 dhclient
3 #查询系统里网卡信息, ip 地址、MAC 地址
4 ifconfig

```

手动设置 ip 地址 如果虚拟机不能自动获取 IP, 只能手动配置, 配置方法如下: 输入命令

```

1 #编辑网卡的配置文件
2 vi /etc/sysconfig/network-scripts/ifcfg-eth0

```

输入上述命令后回车，打开配置文件，使用方向键移动光标到最后一行，按字母键“O”，进入编辑模式，输入以下内容：IPADDR=192.168.4.10 NETMASK=255.255.255.0 GATEWAY=192.168.4.1 另外光标移动到“ONBOOT=no”这一行，更改为 ONBOOT=yes “BOOTPROTO=dhcp”，更改为 BOOTPROTO=none，配置完毕之后如下代码片段所示。

```

1 DEVICE=eth0
2 HWADDR=00:0C:29:A5:78:61
3 TYPE=Ethernet
4 UUID=282eb4cc-6657-4a54-98e3-29605f5998b1
5 ONBOOT=yes
6 NM_CONTROLLED=yes
7 BOOTPROTO=none
8 IPADDR=192.168.24.85
9 NETMASK=255.255.255.0
10 GATEWAY=192.168.24.255

```

完成后，按一下键盘左上角 ESC 键，输入:wq 在屏幕的左下方可以看到，输入回车保存配置文件。之后需要重启一下网络服务，命令为

```
1 service network restart
```

网络重启后，eth0 的 ip 就生效了，使用如下命令查看

```
1 ifconfig eth0
```

接下来检测配置的 IP 是否可以 ping 通，在物理机使用快捷键 WINDOWS+R 打开运行框，输入命令 cmd，输入 ping 192.168.24.85 进行检测，ping 通说明 IP 配置正确。

iptables(防火墙) 配置 iptables 是一个强大的 Linux 防火墙，使用频率极高。

```

1 #CentOS iptables
2 nano /etc/sysconfig/iptables
3
4 #查看配置情况，可显示相应的端口
5 iptables -L -n
6 #添加 input 记录
7 /sbin/iptables -I INPUT -p tcp --dport 8080 -j ACCEPT
8 #添加 output 记录
9 iptables -A OUTPUT -p tcp -sport 22 -j ACCEPT
10 #记得保存
11 /etc/init.d/iptables save
12 #在 CentOS 上执行如下命令，需要加上路径
13 /etc/rc.d/init.d/iptables save

```

一些软件的默认端口：

ftp 用到端口是 20 21

ssh 端口是 22
telnet 端口是 23
rsync 端口是 873
svn 端口 3690
pop3 端口 110
smtp 端口 25
dns 端口 53
mysql 端口 3306
nfs 端口 111

23.2.4 tcpdump

使用 tcpdump 时需要看看目前服务器一共多少网卡，抓取所有经过 eth1，目的或源地址是 12.26.32.14 的网络数据：

```
1 tcpdump -i eth1 host 12.26.32.14 -vv
```

抓取所有经过 eth1，目的地址是 12.26.32.14，且端口是 6973 的网络数据：

```
1 tcpdump -i eth1 dst host 12.26.32.14 and port 6973 -vv
```

抓取所有经过 eth1，数据包类型是 TCP，目的地址是 12.26.32.14，且端口是 6973 的网络数据：

```
1 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 -vv
```

抓取所有经过 eth1，数据包类型是 TCP，目的地址是 12.26.32.14，端口是 6973，且数据包长度大于 140 字节的网络数据：

```
1 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>140' -vv
```

抓取所有经过 eth1，数据包类型是 TCP，目的地址是 12.26.32.14，端口是 6973，且数据包长度大于 140 字节的网络数据：

```
1 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>140' -vv -X
```

-X 选项告诉 tcpdump 命令，需要把协议头和包内容都原原本本的显示出来（tcpdump 会以 16 进制和 ASCII 的形式显示），这在进行协议分析时是绝对的利器。抓取所有经过 eth1，数据包类型是 TCP，目的地址是 12.26.32.14，端口是 6973，数据包长度大于 140 字节，抓取 100 个数据包的网络数据：

```
1 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>140' -vv -X -c
   100
```

抓取所有经过 eth1，数据包类型是 TCP，目的地址是 12.26.32.14，端口是 6973，数据包长度大于 140 字节，且 IP 包 40 位的后两位数据是 7e02 的网络数据（位置信息上报），注意过滤时需要添加 0x 前缀：

```
1 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>140 and ip[40,2]=0
   x7e02' -vv -X -c 100
```

抓取所有经过 eth1，数据包类型是 TCP，目的地址是 12.26.32.14，端口是 6973，数据包长度大于 140 字节，且 IP 包 40 位的后两位数据是 7e02 的网络数据（位置信息上报），注意过滤时需要添加 0x 前缀：

```
1 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>140 and ip[40,2]=0
   x7e02' -vv -X -c 100
```

抓取所有经过 eth1，数据包类型是 TCP，目的地址是 12.26.32.14，端口是 6973，数据包长度大于 140 字节，且 IP 包 40 位的后两位数据是 7e02 的网络数据（位置信息上报），查看终端报警情况：

```
1 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>0 and ip[41:2]=0
   x0200 and ip[53:4]!=0x00000000' -vv -X -c 100
```

抓取所有经过 eth1，数据包类型是 TCP，目的地址是 12.26.32.14，端口是 6973，数据包长度大于 140 字节，且 IP 包 40 位的后两位数据是 7e02 的网络数据（位置信息上报），报警终端电话号码不是 0x41310718，查看终端报警情况：

```
1 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>0 and ip[41:2]=0
   x0200 and ip[53:4]!=0x00000000 and ip[46:4]!=0x41310718' -vv -X -c 100
2 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>100 and ip[41:2]=0
   x0200 and ip[53:4]==0x200000 and ip[46:4]!=0x41310718' -vv -X -c 10000
3
4 #抓取所有进出路线报警
5 tcpdump -i eth1 tcp and dst host 12.26.32.14 and port 6973 and 'ip[2:2]>0 and ip[41:2]=0
   x0200 and ip[53:4]==0x00100000' -vv -X -c 100
```

终端电话号码的后 7 位为设备 ID。

tcpdump 包写入文件

当 tcpdump 指定-w 参数时，可将抓取的包写入文件，默认是 1,000,000 bytes。Savefiles after the first save-file will have the name specified with the -w flag, with a number after it, starting at 1 and continuing upward. The units of file_size are millions of bytes (1,000,000 bytes, not

1,048,576 bytes). 所以持续的捕获消息会生成许多差不多大小的包。

tcpdump 详细日期

有时 dump 下的包需要查看报文捕获的详细时间，可以在 tcpdump 后添加-tttt 参数，如下代码片段所示。

```
1 /usr/sbin/tcpdump -i eth1 port 6973 and host 12.26.32.14 -tttt -s0 -w $packageStoreDir/
$currentTime.cap
```

此种方式捕获的数据包导入 Wireshark 即可看到详细的捕获时间。

tcpdump 显示所有数据

有时候捕获的包太长无法完全展示，添加-s0 参数即可：

```
1 tcpdump -i eth1 tcp and host 10.27.13.14 and port 6973 -vv -X -c 1000
```

23.2.5 vi

操作码	作用
vi + filename	打开一个文件，将光标置于最后一行首
^f	下一页 (forward)
^b	上一页 (before)
^d	往下半个屏幕 (down)
^u	往上半个屏幕 (up)
^n	输入斜线查询后，按 n 查找下一个 (next) 匹配
^N	输入斜线查询后，按 N 查找上一个匹配
u	撤销 (undo) 上一步的操作
Ctrl+r	恢复 (redo) 上一步被撤销的操作

多行复制与粘贴

- 1) 单行复制

在命令模式下，将光标移动到将要复制的行处，按 “yy” 进行复制；

- 2) 多行复制

在命令模式下，将光标移动到将要复制的首行处，按 “nyy” 复制 n 行；其中 n 为 1、2、3

.....

- 2、粘贴

在命令模式下，将光标移动到将要粘贴的行处，按 “p” 进行粘贴

23.2.6 Vim

操作码	作用
gg	光标跳到第一行
\$	光标移动到一行的结尾
[N]yy	复制一行或者 N 行
G	光标跳到最后一行
[n]gg	光标跳到指定行
Ctrl+G/:file/:ls	查看当前打开的文件名称

magic vim 中有个 magic 的设定。设定方法为：

```

1 #设置 magic
2 :set magic
3
4 #取消 magic
5 :set nomagic
6
7 #查看帮助
8 :h magic

```

vim 毕竟是个编辑器，正则表达式中包含的大量元字符如果原封不动地引用（像 perl 那样），势必会给不懂正则表达式的人造成麻烦，比如 /foo(1) 命令，大多数人都用它来查找 foo(1) 这个字符串，但如果按照正则表达式来解释，被查找的对象就成了 foo1 了。

于是，vim 就规定，正则表达式的元字符必须用反斜杠进行转义才行，如上面的例子，如果确实要用正则表达式，就应当写成 /foo\(\1\)\)。但是，像 * 这种极其常用的元字符，都加上反斜杠就太麻烦了。而且，众口难调，有些人喜欢用正则表达式，有些人不喜欢用……

为了解决这个问题，vim 设置了 magic 这个东西。简单地说，magic 就是设置哪些元字符要加反斜杠哪些不用加的。简单来说：magic (\m)：除了 \$. * ^ 外其他元字符都要加反斜杠。nomagic (\M)：除了 \$ ^ 外其他元字符都要加反斜杠。这个设置也可以在正则表达式中通过 \m \M 开关临时切换。后”面的正则表达式会按照 magic 处理，\M 后面的正则表达式按照 nomagic 处理，而忽略实际的 magic 设置。

```

1 /type\[22\

```

23.2.7 grep

grep(缩写来自 Globally search a Regular Expression and Print) 是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。过滤多个关键字时（任意一个关键字满足条件）使用如下语句：

```

1 #过滤包含 vid[10] 或 vid[11] 的行

```

```

2 #加-E 参数表示使用 egrep, 扩展 grep
3 grep -E 'vid\[10\]|vid\[11\]'
4
5 #过滤不包含 vid[10] 的行
6 grep -v 'vid\[10\]'

```

-v 参数表示反向选择，即显示出没有‘搜寻字符串’内容的那一行。添加-color 参数，可以将找到的关键词部分加上颜色的显示。

23.2.8 anacron

23.2.9 cron 执行定时任务

Ubuntu 下定时运行任务使用 cron 表达式，切换到/etc 目录，使用 vi crontab 命令打开配置文件，配置文件如下所示。

```

1 /etc/crontab: system-wide crontab
2 Unlike any other crontab you don't have to run the 'crontab'
3 command to install the new version when you edit this file
4 and files in /etc/cron.d. These files also have username fields,
5 that none of the other crontabs do.
6
7 SHELL=/bin/sh
8 PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

```

打开 Cron 日志记录：

修改 rsyslog 文件，将/etc/rsyslog.d/50-default.conf 文件中的 #cron.* 前的 # 删掉；重启 rsyslog 服务 service rsyslog restart；重启 cron 服务 service cron restart；

more /var/log/cron.log；查看 Cron 日志文件。日志文件将通过邮件服务器发送，如果没有邮件服务器的话，会提示：No MTA installed, discarding output。可以通过以下命令将日志写到指定的文件中：

```

1 * * * * * dolphin /home/dolphin/cron.sh >>$HOME/cronlog.log 2>&1

```

crontab 常用指令如下：

```

1 #启动 crontab
2 /etc/init.d/cron start
3
4 #重新启动 crontab
5 /etc/init.d/cron restart
6
7 #查看 cron 运行状态
8 ps -ef|grep cron
9

```

```

10 #列出 crontab 任务
11 crontab -l
12
13 #每天早上 8 点执行 capture.sh 脚本
14 0 8 * * * root /root/package/scripts/start_capture.sh >> /root/package/capture.log

```

23.2.10 CentOS 安裝 PHP 5.4

RHEL / CentOS 4 預設的 PHP 版本是 5.3, 以下會介紹在 CentOS 6 透過 Yum 安裝 PHP 5.4 的方法:

1. 安裝 SCL repo

```
1 yum install centos-release-SCL
```

2. 安裝 php 5.4

```
1 yum install php54
```

Step 3. 在系統使用 PHP 5.4 開 \square /etc/profile \square 加入以下一行:

```
1 source /opt/rh/php54/enable
```

\square 執行以下指令:

```
1 source /etc/profile
```

完成後可以用 php -v 檢查是否升級到 PHP 5.4.

23.3 Angile

23.3.1 Scrum

产品订单 (product backlog) 是整个专案的概要文档。产品订单包括所有所需特性的粗略的描述。产品订单是关于将要生产什么样的产品。产品订单是开放的，每个人都可以编辑。产品订单包括粗略的估算，通常以天为单位。估算将帮助产品负责人衡量时程表和优先级（例如，如果“增加拼写检查”特性的估计需要花 3 天或 3 个月，将影响产品负责人对该特性的渴望）。

冲刺订单 (sprint backlog) 是大大细化了的文档，包含团队如何实现下一个冲刺的需求的信息。任务被分解为以小时为单位，没有任务可以超过 16 个小时。如果一个任务超过 16 个小时，那么它就应该被进一步分解。冲刺订单上的任务不会被分派，而是由团队成员签名认领他们喜爱的任务。

23.3.2 Redmine 慢

是因为在新建问题和更新问题时会发送邮件，而且是同步操作的，就是说需要等到邮件发送成功后才会返回。redmine 的配置文件存放位置：/var/www/redmine/config/configuration.yml，可以通过如下命令查找：

```
1 find / -name configuration.yml
```

找到 delivery_method: :smtp，将 smtp 改为 async_smtp，保存退出，重启 Apache 服务即可。

23.4 Code Review

23.4.1 Tools

23.4.2 Install Review Board

Part II

公共组件

Premature optimization is the
root of all evil!

Donald Knuth

尽可能地使用开源，并且如果有能力的话也可以把自己的成果分享给大家。整个社会的智慧结晶肯定比一些大公司自管自闭门造车要好。

最显而易见的原因就是使用开源软件，就像是站在巨人的肩膀上，能让你更快更有效地构建软件。处于行业领先地位的公司必须先发制人，抓住已经过锤炼的生态系统，并在此基础之上添加自己的创新。如果凡事都慢人一步，都不是最优的，那你终将会被时代的浪潮甩在后面。

不需要再发明一次轮子

采用开源解决方案意味着使用标准化的解决方案。这种标准化的使用和工作模式，可以促使执行一系列标准化的组织实践，而这将惠及其他公司的许多工程师。这种标准化的方式可产生更优化的组织，直指目标，避免更多的时间浪费。换言之，开源带来的标准化组织实践，有助于避免不必要的试验。

标准化实践的效率

Chapter 24

日志消息组件

24.1 Apache Kafka

Apache Kafka is an open-source message broker project developed by the Apache Software Foundation written in Scala. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. The design is heavily influenced by transaction logs.

24.2 log4net

日志的等级： OFF > FATAL > ERROR > WARN > INFO > DEBUG > ALL

24.2.1 创建日志表

当需要将日志写入数据库时，需要先在数据库中创建日志表 Log。其中 MachineID 为机器 ID，为自定义字段，非 log4net 定义的标准字段，作为每台机器写入日志的区分。

```

1  CREATE TABLE [dbo].[Log](
2      [Id] [int] IDENTITY(1,1) NOT NULL,
3      [Date] [datetime] NULL,
4      [Thread] [varchar](255) NULL,
5      [Level] [varchar](50) NULL,
6      [Logger] [varchar](255) NULL,
7      [Message] [varchar](4000) NULL,
8      [Exception] [varchar](2000) NULL,
9      [location] [varchar](512) NULL,
10     [MachineID] [varchar](64) NULL)
```

24.2.2 log4net 配置 (RollingFileAppender)

在程序的 app.config (如果是 Web 应用程序那么就在 web.config) 中添加如下内容，

```

1  <configSections>
2      <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler
3          , log4net"/>
4  </configSections>
```

添加 log4net.config 配置文件，如下是将日志输出到文件配置。

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3      <configSections>
4          <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,
5              log4net"/>
6      </configSections>
7      <log4net>
```

```

7   <!-- 日志输出为文件格式的设定 -->
8   <appender name="RollingFileAppender" type="log4net.Appender.RollingFileAppender
      ">
9     <file value="log\log_" />
10    <appendToFile value="true" />
11    <rollingStyle value="Date" />
12    <datePattern value="yyyyMMdd'.txt'" />
13    <staticLogFileName value="false" />
14    <param name="lockingModel" type="log4net.Appender.FileAppender+MinimalLock"
          />
15    <layout type="log4net.Layout.PatternLayout">
16      <ConversionPattern value="[%date] [%thread] [%level] [%logger] [%ndc] - %
          message%newline" />
17    </layout>
18  </appender>
19
20  <root>
21    <level value="ALL" />
22    <appender-ref ref="rollingFile" />
23  </root>
24
25  <logger name="*">
26    <level value="ALL" />
27    <appender-ref ref="RollingFileAppender" />
28  </logger>
29 </log4net>
30 </configuration>

```

24.2.3 log4net.config(ADONetAppender)

增加 log4net¹的配置文件 log4net.config，将日志写到 SQL Server，具体内容如下：

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <configuration>
3    <configSections>
4      <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,
          log4net" />
5    </configSections>
6    <common>
7      <logging>
8        <factoryAdapter type="Common.Logging.Log4Net.Log4NetLoggerFactoryAdapter, Common
          .Logging.Log4net">
9          <arg key="configType" value="INLINE" />
10         </factoryAdapter>
11      </logging>
12    </common>

```

¹The Apache log4net library is a tool to help the programmer output log statements to a variety of output targets. log4net is a port of the excellent Apache log4j™ framework to the Microsoft® .NET runtime. 更多信息请参阅：<http://logging.apache.org/log4net/>

```
13 <!--日志记录组建配置-->
14 <log4net>
15 <!-- Console部分log输出格式的设定 -->
16 <appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
17   <layout type="log4net.Layout.PatternLayout">
18     <conversionPattern value="%date [%thread] %-5level %logger %ndc - %message%
19       newline" />
20   </layout>
21 </appender>
22 <!-- 日志文件部分log输出格式的设定 -->
23 <appender name="RollingLogFileAppender" type="log4net.Appender.
24   RollingFileAppender">
25   <file value="log\Log" />
26   <appendToFile value="true" />
27   <rollingStyle value="Date" />
28   <datePattern value="yyyyMMdd'.txt'" />
29   <!--设置无限备份=-1 , 最大备份数为1000-->
30   <param name="MaxSizeRollBackups" value="10"/>
31   <!--每个文件的大小-->
32   <param name="MaximumFileSize" value="50MB"/>
33   <staticLogFileName value="false" />
34   <layout type="log4net.Layout.PatternLayout">
35     <ConversionPattern value="%date [%thread] %-5level %logger [%ndc] - %message%
36       newline" />
37   </layout>
38 </appender>
39 <appender name="ADONetAppender" type="log4net.Appender.AdoNetAppender">
40   <bufferSize value="0"/>
41   <connectionType value="System.Data.SqlClient.SqlConnection, System.Data, Version
42     =1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
43   <connectionString value="Data Source=;Initial Catalog=zouwo-weixin;User ID=
44     Password=/>
45   <commandText value="INSERT INTO Log ([Date], [Thread], [Level], [Logger], [Message]
46     ], [Exception]) VALUES (@log_date, @thread, @log_level, @logger, @message,
47     @exception)"/>
48   <parameter>
49     <parameterName value="@log_date"/>
50     <dbType value="DateTime"/>
51     <layout type="log4net.Layout.RawTimeStampLayout"/>
52   </parameter>
53   <parameter>
54     <parameterName value="@thread"/>
55     <dbType value="String"/>
56     <size value="255"/>
57     <layout type="log4net.Layout.PatternLayout">
58       <conversionPattern value="%thread"/>
59     </layout>
60   </parameter>
61   <parameter>
62     <parameterName value="@log_level"/>
63     <dbType value="String"/>
```

```
57 <size value="50"/>
58 <layout type="log4net.Layout.PatternLayout">
59   <conversionPattern value="%level"/>
60 </layout>
61 </parameter>
62 <parameter>
63   <parameterName value="@logger"/>
64   <dbType value="String"/>
65   <size value="255"/>
66   <layout type="log4net.Layout.PatternLayout">
67     <conversionPattern value="%logger"/>
68   </layout>
69 </parameter>
70 <parameter>
71   <parameterName value="@message"/>
72   <dbType value="String"/>
73   <size value="8000"/>
74   <layout type="log4net.Layout.PatternLayout">
75     <conversionPattern value="%message"/>
76   </layout>
77 </parameter>
78 <parameter>
79   <parameterName value="@exception"/>
80   <dbType value="String"/>
81   <size value="8000"/>
82   <layout type="log4net.Layout.ExceptionLayout"/>
83 </parameter>
84 <parameter>
85   <parameterName value="@MachineID" />
86   <dbType value="String" />
87   <size value="64" />
88   <layout type="log4net.Layout.PatternLayout" >
89     <param name="ConversionPattern" value="%property{MachineID}"/>
90   </layout>
91 </parameter>
92 </appender>
93 <!-- Setup the root category, add the appenders and set the default level -->
94 <root>
95   <level value="ALL" />
96   <appender-ref ref="ConsoleAppender" />
97   <appender-ref ref="RollingLogFileAppender" />
98   <appender-ref ref="ADONetAppender" />
99 </root>
100 </log4net>
101 </configuration>
```

24.2.4 初始化配置文件

如果遇到初始化配置时间太长的情况，检查是否添加了 ADONetAppender，当数据库无法成功连接时会造成初始化日志时间非常长。初始化配置文件的一种方式为在启动项目的 AssemblyInfo.cs 文件中添加如下语句：

```
1 [assembly: log4net.Config.XmlConfigurator(Watch = true, ConfigFile = "./config/log4net.config")]
```

指定 log4net 配置文件的存放位置在根目录的 config 文件夹下，此种配置方式适合 Client/Server 架构和 Browser/Server 架构的项目，在 Windows Service 中也可以采用此种写法指定 log4net 配置文件的存放位置。初始化配置文件的另一种方式为在系统启动时初始化日志配置，其中 log4net.config 文件存放在 config 文件夹下，此种方式适合基于 Client/Server 架构的项目。

```
1 /*
2      Initial log4net in Windows Form
3 */
4 var assemblyFilePath = Assembly.GetExecutingAssembly().Location;
5 var assemblyDirPath = Path.GetDirectoryName(assemblyFilePath);
6 var configFilePath = assemblyDirPath + @"\config\log4net.config";
7
8 /*
9 * Always check if the 'configFilePath' is the correct 'log4net.config' file path.
10 */
11 log4net.Config.XmlConfigurator.ConfigureAndWatch(new FileInfo(configFilePath));
12
13 /*
14     Initail log4net in MVC
15 */
16 log4net.Config.XmlConfigurator.Configure(new System.IO.FileInfo(Server.MapPath("~/
17         config/log4net.config")));
18
19 /*
20     Initial log4net in ASP.NET Web API
21     Write this code in Global.asax
22     Global.asax文件（也叫做 ASP.NET 应用程序文件）是一个可选的文件，该文件包含响应 ASP
23     .NET 或 HTTP 模块引发的应用程序级别事件的代码
24 */
25 /*在新版的Log4net中，可以通过此语句简单配置即可*/
26 BasicConfigurator.Configure();
```

在配置完成后，需要至少初始化一次，这样日志才能够成功输出。在 MVC 中配置 log4net 需在 Global.asax 中的 Application_Start 方法中添加类似配置初始化 log4net 配置文件。在类中可以通过

```
1 private static readonly log4net.ILog logger = log4net.LogManager.GetLogger("FrmTest");
```

语句获取一个 logger 对象。此种方式在每个类中写成类属性时都需要更改类名，比较麻烦。也可通过如下语句利用反射来初始化日志实例：

```
1 private static readonly log4net.ILog logger = log4net.LogManager.GetLogger(MethodBase.  
GetCurrentMethod().DeclaringType);
```

注意引用 `using System.Reflection` 命名空间。此种写法可以成为类的通用属性，引用时可避免手动更改类名。`DeclaringType` 获取声明此方法的类（Gets the class that declares this member）。避免在每个类中定义 logger 实例，可将 logger 在基类中实例化，其他类继承基类即可。使用的过程中发现初次初始化 log4net 时非常缓慢，可采用异步的方式进行初始化，如下代码片断所示：

```
1 Task.Run(() => AsyncLog4Net());
```

`AsyncLog4Net` 方法为初始化 log4net 的相关代码，不过是放入线程池运行。另外如果在使用异步方式记录日志时，将记录日志的语句放入线程池中，而不是线程中，详细的原因如下代码片断所示。

```
1 //应该避免的写法  
2 for (int n = 1; n <= 5; n++)  
3 {  
4     Task.Run(() => logger.DebugFormat("Log entry {0}", n));  
5 }  
6  
7 /*  
8 输出的结果：  
9 DEBUG Log entry 6  
10 DEBUG Log entry 6  
11 DEBUG Log entry 6  
12 DEBUG Log entry 6  
13 DEBUG Log entry 6  
14 */  
15  
16 //推荐写法  
17 public void Debug(string message)  
18 {  
19     ThreadPool.QueueUserWorkItem(task => logger.Debug(message));  
20 }
```

24.2.5 log 文件名累加

在 log4net 的日志文件生成过程中会出现文件名累加的情况，如图24.1所示，估计是 IIS 多线程机制导致写入冲突引起的，当多个线程同时修改一个文件时导致新建 log 文件。

解决方法为在 log4net 配置文件的 Appender 配置节里添加：



图 24.1: 日志文件名累加

```
1 <param name="lockingModel" type="log4net.Appender.FileAppender+MinimalLock" />
```

lockingModel 属性为文件锁类型, RollingFileAppender 本身不是线程安全的, 如果在程序中没有进行线程安全的限制, 可以在这里进行配置, 确保写入时的安全。文件锁定的模式, 官方文档上他有三个可选值 “FileAppender.ExclusiveLock, FileAppender.MinimalLock and FileAppender.InterProcessLock”, 默认是第一个值, 排他锁定, 一次只能有一个进程访问文件, close 后另外一个进程才可以访问; 第二个是最小锁定模式, 允许多个进程可以同时写入一个文件。

24.2.6 log4net 自定义字段

在 log4net 配置文件中添加自定义参数配置节:

```
1 <parameter>
2   <parameterName value="@MachineID" />
3   <dbType value="String" />
4   <size value="64" />
5   <layout type="log4net.Layout.PatternLayout" >
6     <param name="ConversionPattern" value="%MachineID"/>
7   </layout>
8 </parameter>
```

编写自定义 PatternLayout 类:

```
1 public MyLayout()
2 {
3   AddConverter("MachineID", typeof(MachineIDPatternConverter));
4 }
```

编写自定义 PatternLayoutConverter 类:

```
1 protected override void Convert(TextWriter writer, LoggingEvent loggingEvent)
2 {
3   LogMessage logMessage = loggingEvent.MessageObject as LogMessage;
4   //将 MachineID 作为日志信息输出
5   writer.Write(logMessage.MachineID);
6 }
```

编写自定义日志输出类:

```

1  public class LogMessage
2  {
3      #region 机器ID
4      private string _machineID;
5
6      public string MachineID
7      {
8          get { return _machineID; }
9          set { _machineID = value; }
10     }
11     #endregion
12 }
```

目前方法编写出来只能实现输出方法的名称，还有待继续研究。log4net 中添加自定义字段还有更为简便的方式²。

1. 在配置文件中添加自定义列的配置。

```

1 <parameter>
2     <parameterName value="@MachineID" />
3     <dbType value="String" />
4     <size value="64" />
5     <layout type="log4net.Layout.PatternLayout" >
6         <param name="ConversionPattern" value="%property{MachineID}"/>
7     </layout>
8 </parameter>
```

2. 在启动方法中添加如下语句即可，其中 macId 即为自定义字段需要保存的值。

```

1 //log4net自定义字段
2 log4net.GlobalContext.Properties["MachineID"] = macId;
```

24.2.7 log4net 日志写入界面

将日志写入窗体的控件里，构造函数:

```

1 this.Closing += new CancelEventHandler(Form1_Closing);
2 logger = new log4net.Appender.MemoryAppender();
3 log4net.Config.BasicConfigurator.Configure(logger);
4 logWatcher = new Thread(new ThreadStart(LogWatcher));
5 logWatcher.Start();
```

添加如下属性字段:

²参考: <http://stackoverflow.com/questions/12139486/log4net-how-to-add-a-custom-field-to-my-logging>

```

1 private bool logWatching = true;
2 private log4net.Appender.MemoryAppender logger;
3 private Thread logWatcher;

```

LogWatcher 方法如下：

```

1 private void LogWatcher()
2 {
3     while (logWatching)
4     {
5         LoggingEvent[] events = logger.GetEvents();
6         if (events != null && events.Length > 0)
7         {
8             logger.Clear();
9             foreach (LoggingEvent ev in events)
10            {
11                string line = ev.LoggerName + ":" + ev.RenderedMessage + "\r\n";
12                AppendLog(line);
13            }
14        }
15        Thread.Sleep(500);
16    }
17}

```

新的 LogAppend 如下：

```

1 delegate void delOneStr(string log);
2 void AppendLog(string _log)
3 {
4     if (txtLog.InvokeRequired)
5     {
6         delOneStr dd = new delOneStr(AppendLog);
7         txtLog.Invoke(dd, new object[] { _log });
8     }
9     else
10    {
11        StringBuilder builder;
12        if (txtLog.Lines.Length > 99)
13        {
14            builder = new StringBuilder(txtLog.Text);
15            builder.Remove(0, txtLog.Text.IndexOf('\r', 3000) + 2);
16            builder.Append(_log);
17            txtLog.Clear();
18            txtLog.AppendText(builder.ToString());
19        }
20        else
21        {
22            txtLog.AppendText(_log);
23        }
24    }

```

```
25 }
```

窗体的关闭如下：

```
1 void Form1_Closing(object sender, CancelEventArgs e)
2 {
3     logWatching = false;
4     logWatcher.Join();
5 }
```

也可借助 log4net 的 TraceAppender 先写到 System.Diagnostics.Trace 中，配置如下所示：

```
1 <appender name="TraceAppender" type="log4net.Appender.TraceAppender">
2     <layout type="log4net.Layout.PatternLayout">
3         <conversionPattern value="%date [%thread] %-5level %logger [%property{NDC}] -
4             %message%newline" />
5     </layout>
6 </appender>
```

再用定制的 TraceListener 将监听到的，写到控件 CustomTraceListener : TraceListener

```
_customListener = new CustomTraceListener(this); System.Diagnostics.Trace.Listeners.Add(_customListener);
```

24.2.8 Trace 和 Debug

使用如下代码片段所示：

```
1 Trace.TraceError("这是一个Error级别的日志");
2 Trace.TraceWarning("这是一个Warning级别的日志");
3 Trace.TraceInformation("这是一个Info级别的日志");
4 Trace.WriteLine("这是一个普通日志");
5 Trace.Flush()//立即输出
```

在 Release 模式下 (没有定义 DEBUG 常量时)，该方法不会被编译的 (不是不执行，而是根本不会编译到程序中去)。Debug.XXX() 方法仅在 Debug 模式下运行，这个可以为我们省下很多事

24.2.9 多个文件记录日志

一般是在配置文件中增加 Logger 节点，用 name 属性进行对应。在源代码中创建日志记录器 (Logger) 的时候，使用 GetLogger 方法和日志记录器名称即可获得对应的日志记录器实例。

24.2.10 统一记录日志

在应用程序中，可通过定义日志的公共静态实例来进行日志记录，在代码中使用公共静态实例而不是 log4net，带来的好处是 log 记录的变化比较方便的控制，只需要修改静态变量的 log 实例即可，一处修改，处处修改。

```

1 public static class PublicAttribute<T>
2 {
3     /// <summary>
4     /// 日志实例
5     /// </summary>
6     public static readonly ILog Logger = LogManager.GetLogger(typeof(T));
7 }
```

公共类中传入泛型参数，带来的好处是日志显示的类是记录日志实际的类，而不是定义的公共类 PublicAttribute，不方便的地方是调用记录时需要传入类名。

24.2.11 记录建议 (Logging Advisor)

- 不要记录后又重新向外抛出 记录日志后重新抛出异常是一个异常反模式 (anti-pattern)。
- 日志信息应该用英语 英语意味着你的 log 是用 ASCII 编码的。这非常重要，因为你不会真正知道 log 信息会发生什么，或是它被归档前经过何种软件层和介质。如果你的信息里面使用了特殊字符集，乃至 UTF-8，它可能并不会被正确地显示 (render)，更糟的是，它可能在传输过程中被损坏，变得不可读。不过这还有个问题，log 用户输入时，可能有各种字符集或者编码。

24.2.12 远程 Tail 日志

log.io rTail

24.2.13 常见问题

log4net 在本地可以正常写日志，但是部署到 IIS 后不输出日志，一般是由于权限问题，IIS 6.0 默认运行在 NETWORK SERVICE 账户下，更改网站或者 log 目录权限即可。

24.3 Opserver

Opserver 是 Stack Exchange 的一个开源监控系统，基于 Net、MVC 开发，所以 Net 程序员可以轻松基于它二次开发。

24.3.1 title

24.4 ELMAH

ELMAH 是 Error Logging Modules and Handlers for ASP.NET 的缩写。ELMAH 可以让你记录下你的网站发生的任何一个错误，在将来，你可以重新检查这些错误。你可以从 ELMAH 项目的官方网站免费下载 ELMAH: <http://code.google.com/p/elmah/>。

ELMAH 既支持 ASP.NET Web Forms 又支持 ASP.NET MVC。你可以对 ELMAH 进行配置来存储各种不同的错误（XML 文件，事件日志，Access 数据库，SQL 数据库，Oracle 数据库，或者计算机 RAM。）你还可以让 ELMAH 在错误发生的时候，把错误信息 email 给你。

在默认情况下，在一个已经安装 ELMAH 的网站中，你可以通过请求的 elmah.axd 页面的方式来访问 ELMAH。这是“Superexpert.com”网站的 elmah 页面的外观（这个页面是密码保护的，因为在一个错误信息中，可能会泄露出一些应该保密的信息。）

24.5 NLog

24.6 log.io

Real-time log monitoring in your browser.

24.7 ELK

24.7.1 logstash 简介

Logstash 是一款轻量级的日志搜集处理框架，可以方便的把分散的、多样化的日志搜集起来，并进行自定义的处理，然后传输到指定的位置，比如某个服务器或者文件。Elasticsearch + Logstash + Kibana (ELK) 是一套开源的日志管理方案，分析网站的访问情况时我们一般会借助 Google/百度/CNZZ 等方式嵌入 JS 做数据统计，但是当网站访问异常或者被攻击时我们需要在后台分析如 Nginx 的具体日志，而 Nginx 日志分割/GoAccess/Awstats 都是相对简单的单节点解决方案，针对分布式集群或者数据量级较大时会显得心有余而力不足，而 ELK 的出现可以使我们从容面对新的挑战。Logstash：负责日志的收集，处理和储存 Elasticsearch：负责日志检索和分析 Kibana：负责日志的可视化。关于安装文档，网络上有很多，可以参考，不可以全信，而且三件套各自的版本很多，差别也不一样，需要版本匹配上才能使用。

elasticsearch 安装 elasticsearch 安装过程如下：

1 # 下载安装包

```
2 wget https://download.elastic.co/elasticsearch/release/org/elasticsearch/distribution/tar/
      elasticsearch/2.3.3/elasticsearch-2.3.3.tar.gz
3
4 #解压
5 tar -xzvf elasticsearch-2.3.3.tar.gz
6
7 #放到安装目录下
8 mv elasticsearch-2.3.3 /usr/local
9
10 #创建软链接
11 ln -s /usr/local/elasticsearch-2.3.3 /usr/local/elasticsearch
12
13 #测试 elasticsearch
14 ./elasticsearch -e 'input { stdin { } } output { stdout { } }'
15
16 #创建配置文件目录
17 mkdir -p /usr/local/logstash/etc
18
19 #创建配置文件
20 vim /usr/local/logstash/etc/hello_search.conf
21
22 #启动
23 /usr/local/logstash/bin/logstash -f /usr/local/logstash/etc/hello_search.conf
24
25 #运行 logstash agent
26 ./logstash agent
```

logstash 安装 logstash 安装过程如下：

```
1 #下载安装包
2 wget https://download.elastic.co/logstash/logstash-2.3.3.tar.gz
3
4 #解压
5 tar -xzvf logstash-2.3.3.tar.gz
6
7 #放到安装目录下
8 mv logstash-2.3.3 /usr/local
9
10 #创建软链接
11 ln -s /usr/local/logstash-2.3.3 /usr/local/logstash
12
13 #测试 logstash
14 ./logstash -e 'input { stdin { } } output { stdout { } }'
15
16 #创建配置文件目录
17 mkdir -p /usr/local/logstash/etc
18
19 #创建配置文件
20 vim /usr/local/logstash/etc/hello_search.conf
21
```

```
22 #测试配置文件  
23 ./logstash --configtest -f ..etc/hello_search.conf  
24  
25 #启动  
26 /usr/local/logstash/bin/logstash -f /usr/local/logstash/etc/hello_search.conf  
27  
28 #运行 logstash agent  
29 ./logstash agent
```


Chapter 25

集成组件

25.1 SVN

有时需要修改 VisualSVN Server 的 Commit 链接地址，初始安装时是服务器名称，将之修改为 IP 地址更容易记忆，在管理面板左侧的根目录下，配置授权选项 (Configure authentication options...), 在 Network 选项卡中，修改 Server Name 的值为 IP 即可。SVN 的缺点是没有良好的 Code Review 工具，速度比较慢 (因为有许多远程操作比如 Commit、Update 等等)。

25.1.1 SVN Revert

`svn revert —Undo all local edits.`

25.1.2 svn cleanup failed—previous operation has not finished; run cleanup if it was interrupted

1. Install sqlite (32 bit binary for windows) from here
2. `sqlite .svn/wc.db "select * from work_queue"`

```
D:\项目\hotmark\.svn>sqlite3 wc.db "select * from work_queue"
1586 :(dir-remove RRMall.AutoUpdater/bin 1 1)
```

图 25.1: 查看 SVN 操作队列

The SELECT should show you your offending folder/file as part of the work queue. What you need to do is delete this item from the work queue.

3. `sqlite .svn/wc.db "delete from work_queue"`

```
D:\项目\hotmark\.svn>sqlite3 wc.db "delete from work_queue"
D:\项目\hotmark\.svn>
```

图 25.2: 清空 SVN 操作队列

That's it. Now, you can run `cleanup1` again –and it should work. Or you can proceed directly to the task you were doing before being prompted to run cleanup (adding a new file etc.)

Also, `svn.exe` (a command line tool) is part of the Tortoise installer –but is unchecked for some reason. Just run the installer again, choose ‘modify’ and select the ‘command line tools’

¹SVN 更新或签出时，由于一些操作中断更新（磁盘空间不够等），可能会造成本地文件被锁定的情况，使用 `clean up` 来清除锁定。

25.1.3 SVN 文件重命名冲突

当给 SVN 中的文件重命名时，由于 Windows 文件命名大小写不敏感，而 SVN 大小写敏感，会造成命名冲突的问题。重命名后，在 svn 的菜单选择 Add (as replacement) ... 选项即可，会弹出 Adjust Case 界面，如图25.3所示。

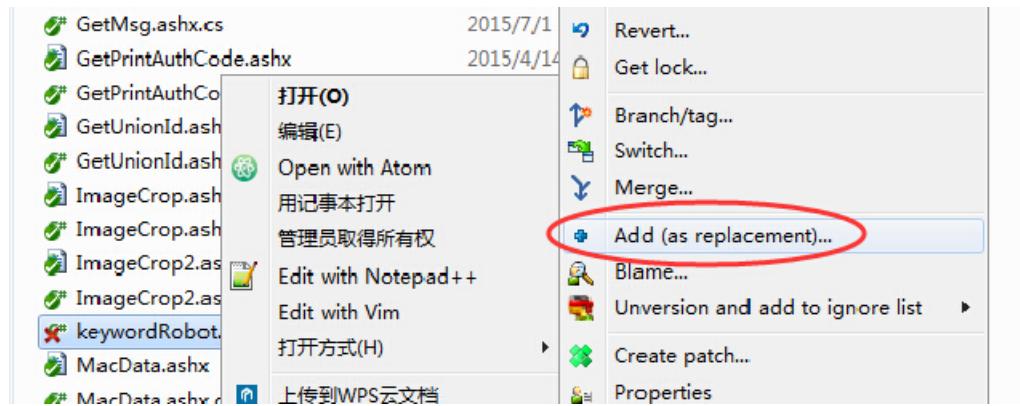


图 25.3: SVN 文件名冲突解决

25.1.4 tags、trunk、branches

一般开发目录分别有 trunk,tags,branches，有时项目发布了一个新的版本或者 Milestone (视情况而定)，那么此时就需要对其进行 tag 标记。trunk 为项目的主干，branches 为项目的分支开发目录，tags 为项目的存档目录，不允许修改。在 branch 时需要手动创建 branches 文件夹，branches 下面的子文件夹由 svn 创建，子文件夹无需手动创建。在标记出 tags 时，tags 文件夹需要手动创建，tags 下的子文件夹不需要手动创建。在创建分支时，所有的项目都需要是最新状态，不能处于冲突或者是修改状态。

25.1.5 switch、relocate

switch 用于在同一个版本库内不同分支之间的切换 relocate 用于版本库访问地址变更时，重新定位版本库，Relocate the working copy to point to a different repository root URL。当源码库的 IP 改变或者端口变化时，可使用 relocate 将库重新定位到新的 URL，svn relocate 命令如下代码片段所示，也可以在图形界面中进行操作。

```
1  svn switch --relocate https://192.168.1.222:888/svn/Zouwo https://183.230.47.195:888/svn/
   Zouwo
```

第一个 URL 为源 URL，第二个 URL 为目标 URL。比如，由于 SVN 服务器更换到另一台主机上，这是 SVN 服务器的地址改变了，那么各客户端就无法连接 SVN 服务器了，这时各客户端就需要执行 relocate，将本地工作区的连接到新的服务器上去。

而如果同一个版本库内，如果有多个分支，比如你现在正在 trunk 上开发，但需要切换到

某个分支上开发，那么你可以用 switch 来进行这个切换操作，这时 SVN 会比较 trunk 和这个分支之间的差异，将差异部分传送到你的本地工作区，而不用将整个分支传送给，从而避免巨量数据的传输。switch 操作之后，你所进行的 update、commit 操作都变成了针对那个分支，当你在分支上的工作完成后，还可以再次 switch 回 trunk。

switch 还有另外一些用途，比如希望让分支中的某个文件夹保持和 trunk 同步，因为有人正在 trunk 的这个文件夹中进行开发，在分支中想用到开发的最新成果，那么就可以在分支的这个文件夹上设置 switch 到 trunk，这时 update 整个分支的话，就会把 trunk 上的这个文件夹取下来了。但是，当然你如果修改了这个文件夹的内容，commit 后也是提交到了主干而不是提交到了分支。

25.1.6 SVN 锁

如果使用锁，每个用户编辑前先 get-lock，然后编辑，提交。这个过程中，其他人打开的时候开到的是 read-only 的文件，无法保持，这样就保证了单一性。这就是“锁定-编辑-解锁”模型，对不容易合并的非二进制文件很有好处。

25.1.7 SVN 版本库镜像

25.1.8 备份库

执行初始化 svnsync init file:///之前新建立的 Repository 源代码地址 URL

执行同步 svnsync sync file:///之前新建立的 Repository

1. 为两个 repository 建一个同样的用户，如 admin/admin
2. 在 backup server 上，在 hooks 文件夹中将 pre_revprop_change.tmpl 改为 pre_revprop_change.bat 并将内容清空然后输入 exit 0，执行 svnsync init http://localhost/svn/backup http://localhost/svn/src –username admin –password admin

5. 在 src server 上，将 post-commit.tmpl 改为 post-commit.bat，内容改为如下：svnsync sync –non-interactive http://localhost/svn/backup –username admin –password admin

25.1.9 SVN 自动更新

团队中有忘记更新即提交代码而覆盖代码的现象出现（这种现象应该是不会出现的，当提交代码时，如果团队里其他人对提交的文件做了改动，那么 SVN 会自动提示必须先更新，然后再提交，所以未更新即提交而导致先前的代码覆盖的情况逻辑上是不会出现的），为了降低此种现象发生的概率，写了批处理脚本更新项目的源码，通过 Windows 的定时任务定期执行更新操作，毕竟总是无法同时想到非常多的注意事项，代码片段实现自动更新 SVN。

```

1 @echo off
2 =====
3 @echo author 罗敏贵

```

```

4 @echo modifier 蒋小强
5 @echo blog:http://luomingui.cnblogs.com;jiangxiaoqiang@gmail.com
6 @echo update:2012-08-27
7 @echo modifyDate:2015-11-18
8 =====
9 rem 指定SVN的安装目录
10 set svn_home=C:\Program Files\TortoiseSVN\bin
11 rem SVN工作目录
12 set svn_work=D:\项目\Tour
13 rem SVN日志目录
14 set setup_path=E:\我的坚果云\bat
15
16 if exist %svn_work% goto :UPDATE
17
18 :UPDATE
19 @echo 正在更新目录 %svn_work%
20 if exist "%setup_path%\autoUpdate.log" (echo update: %date% %time% >> "%setup_path%\\autoUpdate.log") else echo create: %date% %time% >"%setup_path%\autoUpdate.log"
21 "%svn_home%\TortoiseProc.exe /command:update /path:\"%svn_work%\" /notempfile /
22 closeonend:1
23 @echo 更新完成退出
24 pause
25 exit

```

如下代码片段是建立 Windows 定时任务执行批处理脚本。

```

1 @echo off
2 =====
3 @echo author 罗敏贵
4 @echo modifier 蒋小强
5 @echo blog:http://luomingui.cnblogs.com;jiangtingqiang@gmail.com
6 @echo email:luomingui@hailin.com
7 @echo update:2012-08-27
8 @echo modifyDate:2015-11-18
9 =====
10 @echo 添加任务
11 rem SCHTASKS /Create /SC MINUTE /ST 08:30:00 /TN "auto update SVN" /TR "%~
12     sdpAutosvn.bat" /F
13 SCHTASKS /Create /SC MINUTE /mo 5 /ST 08:30:00 /TN "auto update SVN" /TR "%~
14     sdpAutosvn.bat" /F①
15 pause

```

- ① 建立定时任务，任务的名称为“auto update SVN”，名称通过 TN 参数（Task Name）指定，任务运行指定的 autosvn.bat 批处理脚本，通过 TR 参数指定（Task Run），任务每隔 5 分钟运行一次，即每 5 分钟更新一次项目代码，时间的长短可根据实际应用环境进行调整

在 Visual Studio 安装了 SVN 插件之后，可直接点击工具栏上的按钮进行提交与更新，相

对来说更加便捷，如图25.4所示。



图 25.4: SVN 提交与更新

25.1.10 设置忽略列表 (Ignore List Setting)

在多数项目中你总会有文件和目录不需要进行版本控制。这可能包括一些由编译器生成的文件，如 *.obj, *.lst, *.sln, *.proj，或许是一个用于存放可执行程序的输出文件夹，如 bin 和 obj 目录。只要你提交修改，TortoiseSVN 就会在提交对话框的文件列表中显示出未版本控制文件。当然你可以关闭这个显示，不过你可能会忘记添加新的源文件。

最好的避免类似问题的方法是添加参考文件到该项目的忽略列表，如图25.5所示。

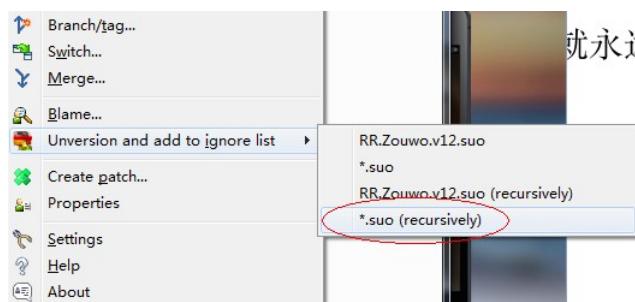


图 25.5: SVN 添加文件到忽略列表

这样他们就永远不会出现在提交对话框中，而真正的未版本控制文件则仍然列出，`recursively` 代表递归进行忽略。

25.1.11 SVN 提交规则 (SVN Commit Rule)

在开发的过程中，*.sln/*.suo/*.log/bin/obj 等文件不需要提交到版本库中，可以在服务器端通过 pre-commit 钩子 (Hooks) 进行阻止。

```

1 @echo off
2 setlocal
3 set REPOS=%1
4 set REV=%2
5 set svnlook="C:\Program Files\VisualSVN Server\bin\svnlook.exe"
6 rem check that logmessage contains at least 10 characters
7 rem %svnlook% log "%REPOS%" -t "%REV%" | findstr ..... >nul
8 rem if %errorlevel% gtr 0 (goto err) else exit 0
9
10 rem 过滤文件类型

```

```

11 %svnlook% changed -t "%REV%" "%REPOS%" | findstr "suo$"
12 if %errorlevel% EQU 0 (goto SuffixError) else exit 0
13
14 %svnlook% changed -t "%REV%" "%REPOS%" | findstr "sln$"
15 if %errorlevel% EQU 0 (goto SuffixError) else exit 0
16
17 :err
18 echo. 1>&2
19 echo Your commit has been blocked because you didn't give any log message 1>&2
20 echo Please write a log message describing the purpose of your changes and 1>&2
21 echo then try committing again. --- Thank you 1>&2
22 exit 1
23
24 :SuffixError
25 echo Don't commit forbidden file,like:obj\pdb\exp\ilk.....,thank you 1>&2
26 exit 1

```

%1 就是命令行上的第一个参数，%2 是第二个，依次类推。例如你的批处理文件名为 ABC.BAT，其中有 set a=%1 这样的语句，则当你发出命令: ABC.BAT HOW ARE YOU 后，批处理文件在执行时那一句就变成 set a=HOW 同时%2 就等于 ARE%3 就等于 YOU。

findstr findstr 可以在任何 ASCII 文件或文件中精确查找所要查找的文本。然而，有时要匹配的信息只有一部分或要查找更宽广的信息范围。在这种情况下，findstr 具有使用正则表达式搜索各种文本的强大功能。正则表达式是用于指定文本类型的符号，与精确的字符串相反。标记使用文字字符和元字符。每个在常规的表达式语法中没有特殊意义的字符都是文字字符，与出现的该字符匹配。例如，字母和数字是文字符号。元字符是在正则表达式语法中具有特殊意义（操作符或分隔符）的符号，\$ 表示行位置：行的结尾。

EQU - 等于 NEQ - 不等于 LSS - 小于 LEQ - 小于或等于 GTR - 大于 GEQ - 大于或等于

1>&2 重定向符号主要有: >, », <, >&, <& 和 |, 1 代表正常输出（即所谓的“标准输出”-stdout），2 代表错误输出（即所谓的“标准错误输出”-stderr），它是所谓“标准输入”的标识，数字代号是 0。

25.1.12 SVN 多个.svn 隐藏目录

在 1.7.X 版本之前，每个目录下面都有个.svn 的隐藏文件夹，1.7.X 之后只是在检出的根目录下有一个.svn，这就是 Subversion 1.7 的新特性。

25.1.13 diff 和 patch

diff 和 patch 是一对工具，在数学上来说，diff 是对两个集合的差运算，patch 是对两个集合的和运算。diff 比较两个文件或文件集合的差异，并记录下来，生成一个 diff 文件，这也是我们常说的 patch 文件，即补丁文件。

patch 能将 diff 文件运用于原来的两个集合之一，从而得到另一个集合。举个例子来说文件 A 和文件 B, 经过 diff 之后生成了补丁文件 C, 那么着个过程相当于 $A - B = C$, 那么 patch 的过程就是 $B+C = A$ 或 $A-C = B$ 。因此我们只要能得到 A, B, C 三个文件中的任何两个，就能用 diff 和 patch 这对工具生成另外一个文件。这就是 diff 和 patch 的妙处。

25.2 Git

25.2.1 基本使用

如下代码片段是 Git 的常见用法。

```
1 #查看本地所有分支
2 git branch
3
4 #查看所有分支
5 git branch -a
6
7 #查看远程所有分支
8 git branch -r
9
10 # 在当前目录新建一个 Git 代码库
11 $ git init
12
13 # 添加指定目录到暂存区，包括子目录
14 $ git add [dir]
15
16 # 添加所有修改文件
17 $ git add .
18
19 #移除文件，-r 参数代表递归移除
20 $ git rm -r --cache Fosc.Dolphin.UI/Fosc.Dolphin.UI/obj/
21
22 #浏览器中列出 git add 命令的帮助内容
23 $ git add --help
24
25 #命令行中列出 git add 命令的帮助内容
26 $ git add -h
27
28 #Git 提交
29 $ git commit -m "Init"
```

Windows 下 Git Bash 切换目录比较特殊，在 Git Bash 下切换目录到 Git Repository 如下代码片段所示。

```
1 #切换到 E: 盘, 注意与 DOS 下写法的区别
2 $ cd /e
3
```

```

4 #切换到 Repository 目录
5 $ cd Temp/doc/DolphinDev/DolphinDevManual
6
7 #列出当前 Repository 的状态
8 $ git status

```

配置 使用 Git 之前的基本配置操作

```

1 # 显示当前的 Git 配置
2 $ git config --list
3
4 # 编辑 Git 配置文件
5 $ git config -e [--global]
6
7 # 设置提交代码时的用户信息
8 $ git config [--global] user.name "[name]"
9 $ git config [--global] user.email "[email address]"

```

代码提交 使用 Git 进行代码的提交操作

```

1 # 提交暂存区到仓库区
2 $ git commit -m [message]

```

在将代码推送到 GitHub 之前，需要设置 SSH，查看当前已经有的 SSH Key 如下代码片段所示：

```

1 # Lists the files in your .ssh directory, if they exist
2 $ ls -al ~/.ssh

```

如果没有 SSH Key 执行如下命令创建：

```

1 #生成密钥
2 ssh-keygen -t rsa -C "jiangtingqiang@gmail.com"
3
4 #验证是否成功配置 Github 公钥
5 ssh -T git@github.com

```

在执行命令时如找不到 ssh-keygen 命令时，需要将 ssh-keygen 的路径添加到系统的环境变量中。在 github.com 的网站上到 ssh 密钥管理页，添加新公钥并取名，内容粘贴本地生成的公钥。在 push 时可能出现错误：fatal: could not read Username for ‘https://github.com’ : No such file or directory。运行命令设置 git 的远程 url 即可解决问题，在 Url 中添加用户名，注意用户名是大小写敏感的：

```

1 git remote set-url origin git@github.com:jiangxiaoqiang/Fosc.git

```

生成了密钥之后记得将密钥添加到 ssh-agent 中，add your SSH key to the ssh-agent。

```
1 ssh-add ~/.ssh/id_rsa
```

若执行命令出现如下错误 Could not open a connection to your authentication agent，则先执行如下命令即可：

```
1 ssh-agent bash
```

在 ssh bash 中添加 RSA key 之后退出即可。要保持新生成的密匙文件的名字同“ssh_config”中“IdentityFile”字段的值一致即可。在生成密钥的时候，请在“/.ssh/”目录下操作。或者生成后把文件移动到“/.ssh/”目录下。

git pull git pull 命令的作用是，取回远程主机某个分支的更新，再与本地的指定分支合并。

```
1 git pull <远程主机名> <远程分支名>:<本地分支名>
```

为了便于管理，Git 要求每个远程主机都必须指定一个主机名。git remote 命令就用于管理主机名。不带选项的时候，git remote 命令列出所有远程主机。使用-v 选项，可以参看远程主机的网址。

```
1 git remote -v
```

加上-a 参数可以查看远程分支，远程分支会用红色表示出来（如果开了颜色支持的话）：

```
1 git branch -a
```

git push git push 命令用于将本地分支的更新，推送到远程主机。它的格式与 git pull 命令相仿。

```
1 git push <远程主机名> <本地分支名>:<远程分支名>
```

25.2.2 推送到 GitHub

每次启动 Git 后切换到工作目录是非常浪费时间的，可右键 Git Bash 在属性页中设置初始启动目录。在 GitHub 上添加第一个 Git 仓库，配置项目名称和相关信息。

```
1 #第一次推送的时候要添加远程的代码库到配置
2 git remote add origin master https://github.com/jiangxiaoqiang/DolphinDevManual.git
3
4 #将远程代码库克隆到本地
5 git clone https://github.com/jiangxiaoqiang/DolphinDevManual.git
6
```

```
7 #初始化
8 git init
9
10 #fetch 主干分支
11 git fetch origin master
12
13 #pull 分支
14 git pull https://github.com/jiangxiaoqiang/XiaoqiangResume.git
15
16 #将本地文件添加到 git 仓库中
17 git add [file1]
18
19 #推送到 GitHub
20 git push origin master
```

注意每次提交都需要执行 git add 命令，git add 与 svn 中的 add 有明显的区别，修改过的文件也要执行 git add 后才能够提交。git fetch 从远程仓储导入 commit 到你的本地仓储。这些 fetch 到的 commit 是做为一个远程分支存储在你本地的。这样你可以在集成这些 commit 到你的项目前先看看都有些什么修改。当你想看看其他人都做了些什么工作的时候你可以使用 fetch。因为 fetch 到的内容是做为一个 remote 分支的形式展现出来的，所以不会对你的本地开发有什么影响。因此 fetch 是在你想 merge 远程的 commit 前，先来查看查看这些 commit 的一个安全的办法。不强迫你 merge。

25.2.3 Git 配置

每次 git push 时都需要输入用户名和密码，需要将密码输入保存在本地。Git 支持如下方式的凭据保存：

默认所有都不缓存 每一次连接都会询问你的用户名和密码。

cache “cache” 模式会将凭证存放在内存中一段时间。密码永远不会被存储在磁盘中，并且在 15 分钟后从内存中清除。

store “store” 模式会将凭证用明文的形式存放在磁盘中，并且永不过期。这意味着除非你修改了你在 Git 服务器上的密码，否则你永远不需要再次输入你的凭证信息。这种方式的缺点是你的密码是用明文的方式存放在你的 home 目录下。

osxkeychain 如果你使用的是 Mac，Git 还有一种“osxkeychain”模式，它会将凭证缓存到你系统用户的钥匙串中。这种方式将凭证存放在磁盘中，并且永不过期，但是是被加密的，这种加密方式与存放 HTTPS 凭证以及 Safari 的自动填写是相同的。如果你使用的是 Windows，你可以安装一个叫做“winstore”的辅助工具。这和上面说的“osxkeychain”十分类似，但是是使用 Windows Credential Store 来控制敏感信息。

25.2.4 添加忽略列表 (Add Ignore List)

有的文件并不需要进行版本管理，比如日志文件和中间生成的辅助文件临时文件等等，此时就需要将不需要管理的文件放在 Git 的忽略列表中。在 Git 中添加忽略列表首先在 Repository 根目录下创建.gitignore 文件。

```
1 $ touch .gitignore
```

编辑文件添加需要忽略的文件或者忽略规则。

```
1 vi .gitignore
```

添加了忽略文件后查看是否还报需要忽略的文件 Untrace，如果没有报告，则证明添加到忽略列表的文件生效了。

```
1 git status
```

25.2.5 常见问题

debug1: No more authentication methods to try.Permission denied (publickey) 运行命令

```
1 ssh -vT git@github.com
```

出现以上错误，解决方法是将 Public Key 拷贝到 GitHub 的设置（SSH Key Settings）中。

25.3 StyleCop.exe

StyleCop analyzes C# source code to enforce a set of style and consistency rules. It can be run from inside of Visual Studio or integrated into an MSBuild project. StyleCop has also been integrated into many third-party development tools.

代码规范检查工具。

- 系统引用需要置于其他引用之前
- 变量命名需要以小写开头
- 引用需要以字母从前往后进行排序
- 私有方法需要置于公共方法之后
- 属性方法前面必须要有注释

25.4 CruiseControl.Net

CruiseControl²: 简称 CC , 持续集成工具, 主要提供了基于版本管理工具(如 CVS、VSS、SVN)感知变化或每天定时的持续集成, 并提供持续集成报告、Email、Jabber 等等方式通知相关负责人, 其要求是需要进行日构建的项目已编写好全自动的项目编译脚本(可基于 Maven 或 Ant)。

CruiseControl.NET-1.8.5.0-Setup.exe 为服务器端, 安装时可以选择生成 Windows Service 以便开启, 建议测试时不用 Windows Service, 直接用 CruiseControl.NET 的运行文件, 这样配置有问题时可以直接在上面看到错误原因。安装完成后将 webdashboard 目录部署为 IIS 网站。

CruiseControl.NET-CCTray-1.8.5.0-Setup.exe 为客户端, 在配置好了 ccnet.config³以后, 可以 setting 中增加需要管理的 project。

25.4.1 自动获取源代码

持续集成的单位是以项目为单位, 在 ccnet.config 文件里添加 Project。

```

1 <project name="HotMark" description="RRMall advertise by print machine." queue="Q1"
2   >
3   <webURL>http://127.0.0.1/ccnet</webURL>
4   <workingDirectory>D:\Dailybuild</workingDirectory>
5   <!--集成的日志, 需要确保文件夹路径存在-->
6   <artifactDirectory>D:\Dailybuild</artifactDirectory>
7   <!--源码修改后延迟多少秒执行集成-->
8   <modificationDelaySeconds>10</modificationDelaySeconds>
9   <category>Advertisement</category>
10  <!-- specify a state folder to prevent CCNet from saving it in Program Files\CruiseControl.
11    NET\server
12    programs may not standard write their data in it on windows Vista and up)
13  -->
14  <state type="state" directory="D:\Dailybuild" />
15
16  <triggers>
17    <!-- check the source control every X time for changes, and run the tasks if changes are
18      found -->
19    <intervalTrigger
20      name="continuous"
21      seconds="60"
22      buildCondition="IfModificationExists"
23      initialSeconds="5"/>
24  </triggers>
25
26  </project>
```

²官方主页: <http://www.cruisecontrolnet.org/>

³ccnet.config 为 CruiseControl.NET 的配置文件, 在安装路径 server 文件夹下

triggers 配置节里说明持续集成的触发条件, seconds 表示间隔 60 秒触发一次持续集成。配置项目的源代地址, 包括本地工作地址和源代码管理服务地址, 对于使用 SVN 的源码管理器, 持续集成插件会自动检出源码进行编译, 向 Project 配置节下添加源代码控制相关配置:

```

1 <!--自动获取源码-->
2 <sourcecontrol type="svn">
3   <!--该项目的SVN路径-->
4   <trunkUrl>https://129.11.21.112/svn/hotmark</trunkUrl>
5   <!--配置的这个目录将作为CruiseControl.NET的工作目录,CruiseControl.NET会将代码从
       SVN中check out到这个目录中.-->
6   <workingDirectory>D:\Dailybuild\Source</workingDirectory>
7   <!--机器上的SVN的可执行文件路径-->
8   <executable>C:\Program Files (x86)\VisualSVN\bin\svn.exe</executable>
9   <!--SVN的用户名与密码-->
10  <username>jiangxiaoqiang</username>
11  <autoGetSource>true</autoGetSource>
12  <password>jiangxiaoqiang</password>
13 </sourcecontrol>
```

CruiseControl 定时检测源代码的变化, 当检测到变化时, 自动更新持续集成工作目录的源码库, 进行构建等后续操作。

25.4.2 自动 Build

实现自动 Build, 向 Project 节点下增加 tasks 节点, 如下:

```

1 <msbuild>
2   <executable>C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe</
      executable>
3   <workingDirectory>D:\Dailybuild\Source</workingDirectory>
4   <projectFile>HotMark.sln</projectFile>
5   <buildArgs>/p:OutputPath=D:\Dailybuild\BuildOutput</buildArgs>
6   <targets>Build</targets>
7   <timeout>9000</timeout>
8   <!--记录编译日志-->
9   <logger>C:\Program Files (x86)\CruiseControl.NET\server\ThoughtWorks.
      CruiseControl.MSBuild.dll</logger>
10  </msbuild>
```

<msbuild> 的父节点是 task 节点。

25.4.3 自动化回归测试

```

1 <nunit>
2   <path>C:\Program Files (x86)\NUnit 2.6.4\bin\nunit-console.exe</path>
3   <outputfile>D:\Dailybuild\buildlogs\NunitLog\nunit-results.xml</outputfile>
4   <assemblies>
```

```

5   <assembly>D:\Dailybuild\BuildOutput\RRMall.WxPrint.Dal.dll</assembly>
6   </assemblies>
7 </nunit>

```

<nunit> 的父节点是 task 节点。在单元测试中，各个程序集使用的各自的配置文件，

25.4.4 自动部署

其中 sourceDir 为测试通过的程序的地址

```

1 <!--发布到站点-->
2 <buildpublisher>
3   <sourceDir>D:\Dailybuild\BuildOutput</sourceDir>
4   <publishDir>D:\publish</publishDir>
5   <useLabelSubDirectory>false</useLabelSubDirectory>
6 </buildpublisher>

```

<buildpublisher> 的父节点是 task 节点。sourceDir 是编译成功后的文件的地址，publishDir 是需要发布到的地址。

25.4.5 邮件提醒

当构建失败或者成功时，可以将构建状态通过邮件通知组内成员：

```

1 <publishers>
2   <!--记录日志-->
3   <xmlogger />
4
5   <!--保留多少日志文件-->
6   <artifactcleanup cleanUpMethod="KeepLastXBuilds"
7     cleanUpValue="50" />
8
9   <!--发送邮件-->
10  <email mailport="25" includeDetails="true" mailhostUsername="
11    jiangxiaoqiang@renrenmall.com" mailhostPassword="123456" useSSL="FALSE">
12    <from>jiangxiaoqiang@renrenmall.com</from>
13    <mailhost>smtp.renrenmall.com</mailhost>
14    <users>
15      <user name="cheng" group="developers" address="xiefei@renrenmall.com" />
16    </users>
17
18    <groups>
19      <group name="developers">
20        <notifications>
21          <notificationType>Failed</notificationType>
22          <notificationType>Fixed</notificationType>
23        </notifications>
24      </group>

```

```

24 </groups>
25
26 <!--失败的邮件标题-->
27 <subjectSettings>
28   <subject buildResult="StillBroken" value="Build is still broken for {
29     CCNetProject}" />
30 </subjectSettings>
31   <attachments>
32     <file>D:\CCNET2\CCNET2\TestResults\mstest-results.xml</file>
33   </attachments>
34 </email>
</publishers>

```

<buildpublisher> 的父节点是 task 节点。

25.4.6 Dashboard Configuration

初次使用时需要配置 Dashboard 的密码, 在配置文件 dashboard.config 中, 路径为 CruiseControl.NET/webdashboard, 修改如下配置节添加 dashboard 密码:

```

1 <administrationPlugin password="123456"/>

```

25.4.7 常见问题处理

未找到导入的项目 在自动化编译的过程中出现错误:

```

1 “D:\ZouwoDailybuild\Source\RR.Zouwo.sln” (Build 目标) (1) ->
2 “D:\ZouwoDailybuild\Source\RR.AdminMange\RR.AdminMange.csproj” (默认目标) (6) ->
3 D:\ZouwoDailybuild\Source\RR.AdminMange\RR.AdminMange.csproj(564,3): error
    MSB4019: 未找到导入的项目 “C:\Program Files (x86)\MSBuild\Microsoft\
    VisualStudio\v11.0\WebApplications\Microsoft.WebApplication.targets” 。请确认 <
    Import> 声明中的路径正确, 且磁盘上存在该文件。

```

由于本项目 Visual Studio 使用的版本是 2012 版, 将 csproj 文件用导入的项目路径更改即可。原路径:

```

1 <PropertyGroup>
2   <VisualStudioVersion Condition="$(VisualStudioVersion) == ''>10.0</
      VisualStudioVersion>
3   <VSToolsPath Condition="$(VSToolsPath) == ''>$(MSBuildExtensionsPath32) \
      Microsoft\VisualStudio\v$(VisualStudioVersion)</VSToolsPath>
4 </PropertyGroup>
5 <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
6 <Import Project="$(VSToolsPath)\WebApplications\Microsoft.WebApplication.targets" \
      Condition="$(VSToolsPath) != '' />
7 <Import Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v10.0 \
      WebApplications\Microsoft.WebApplication.targets" Condition="false" />

```

```

8   <Target Name="MvcBuildViews" AfterTargets="AfterBuild" Condition=" '$(MvcBuildViews)'=='true' " >
9     <AspNetCompiler VirtualPath="temp" PhysicalPath="$(WebProjectOutputDir)" />
10  </Target>

```

修改为：

```

1  <PropertyGroup>
2    <VisualStudioVersion Condition=" '$(VisualStudioVersion)' == ''>12.0</
3      VisualStudioVersion>
4    <VSToolsPath Condition=" '$(VSToolsPath)' == ''>$(MSBuildExtensionsPath32)＼
5      Microsoft\VisualStudio\v$(VisualStudioVersion)</VSToolsPath>
6  </PropertyGroup>
7  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
8  <Import Project="$(VSToolsPath)\WebApplications\Microsoft.WebApplication.targets"
9    Condition=" '$(Solutions.VSVersion)' == '12.0' " />
10 <Import Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v12.0\
11   WebApplications\Microsoft.WebApplication.targets" Condition=" '$(Solutions.
12   VSVersion)' == '12.0' " />
13  <Target Name="MvcBuildViews" AfterTargets="AfterBuild" Condition=" '$(MvcBuildViews)'=='true' " >
14    <AspNetCompiler VirtualPath="temp" PhysicalPath="$(WebProjectOutputDir)" />
15  </Target>

```

25.5 Jenkins

Jenkins is an open source continuous integration tool written in Java. The project was forked from Hudson after a dispute(辩论, 争论) with Oracle.

Jenkins provides continuous integration services for software development. It is a server-based system running in a servlet container such as Apache Tomcat. It supports SCM tools including AccuRev, CVS, Subversion, Git, Mercurial, Perforce, Clearcase and RTC, and can execute Apache Ant and Apache Maven based projects as well as arbitrary shell scripts and Windows batch commands. The primary developer of Jenkins is Kohsuke Kawaguchi. Released under the MIT License, Jenkins is free software.

25.5.1 插件安装 (Install Plug-in)

在 Jenkins->Plugin Manager 中可以进行插件的安装，需要编译.NET 项目需要安装 MSBuild 插件，在 Available Tab 页中安装 MSBuild Plugin，在使用 MSBuild 插件时需要在系统设置中制定 MSBuild 的路径。

25.5.2 创建 Job

一般添加节点时指定 jenkins 的主目录，安装好的 jenkins 目录为 C:/Program Files (x86)/Jenkins，构建的项目一般就在这个目录下的 workspace 目录下，以构建的项目名为目录名。在高级配置选项中可以配置“使用自定义的工作空间”。

点击“Back to Dashboard”，回到 Dashboard 上来，然后点击“New Job”链接。你会看到一组 job 类型，选择“Build a free-style software project”，给它命名为“HelloCI-RunUnitTests”，点击 OK。

下一步是 job 配置页面。这一页有很多配置项，而且大多数都带有详细的描述信息，点击右侧的帮助图标就可以看到。我们现在只配置两部分，一是代码库所在位置，二是如何用 MSBuild 构建项目。

Source Code Management 找到“Source Code Management”，选择 Subversion。在 Repository URL 中输入 Subversion 版本库的 URL，在 Local module directory 中定义路径，此处定义的路径默认在工作目录下，例如项目为 Tour，路径定义为./trunk，那么最终检出的代码的路径为：

```
1 C:\Program Files (x86)\Jenkins\jobs\Tour\workspace\trunk
```

Build 在 Windows 平台上的构建主要是使用 MSBuild，安装好.NET Framework 后 MSBuild 的路径为：

```
1 C:\Windows\Microsoft.NET\Framework\v4.0.30319
```

接下来到“Build”这部分。点击“Add build step”按钮后，下拉框中就会出现一系列的 step 类型以供选择，其中便包括“Build a Visual Studio project or solution using MSBuild”，如果你没看到这个选项，就说明 MSBuild 插件没有正确安装。MSBuild Build File 是项目文件或者工程文件的名称。在安装了 Jenkins 的 MSBuild 插件之后，点击“Build a Visual Studio project or solution using MSBuild”，在“MSBuild Build File”输入框中输入构建脚本的名字：HelloCI.msbuild。我们想让 Jenkins 执行“RunUnitTests”这个 Target，如果你没有把 DefaultTargets 属性设成 RunUnitTests 的话，可以在“Command Line Arguments”中输入“/t:RunUnitTests”，其中/t 是/target 的简写。然后就是 MSBuild 的命令行参数了，示例的命令行参数如下代码片段所示。

```
1 /t:ResolveReferences;Compile /t:_CopyWebApplication /p:VisualStudioVersion=12.0 /p:  
Configuration=Release /property:TargetFrameworkVersion=v4.0 /p:  
WebProjectOutputDir=E:\Web\RR.Web.CCN.Tour\1.0.1 /p:OutputPath=E:\  
JenkinsPublish\Bin
```

- /t:Rebuild 表示每次都重建，不使用增量编译
- /property:Configuration=Release 表示编译 Release 版本

- /property:TargetFrameworkVersion=v4.0 表示编译的目标是.NET 4.0
- /p:WebProjectOutputDir 表示编译的目标文件的输出路径
- /p:OutputPath 表示编译文件的临时目标路径

保存后，点击左侧 Build Now 开始测试一次编译。编译默认是要输出 PDB 文件，如果需要 MSBuild 不输出 PDB 文件，则在后面添加参数即可，如下代码片段所示：

```
1 MSBuild.exe YourProject.csproj /p:DebugSymbols=false /p:DebugType=None
```

指定项目文件，制定 MSBuild Build File，指定项目文件的格式如下所示：

```
1 D:\项目\192.168.1.222\Tour\trunk\RR.Web.CCN.Tour\RR.Web.CCN.Tour.csproj
```

点击“Save”或“Apply”保存之后，这个 job 一旦被触发，就可以 pull 代码下来，编译项目，执行单元测试。我们先来手工触发一次，看看配置是否正确。先回到 Dashboard，这时可以在屏幕中央看到我们的 job。点击 job 名字，然后在左侧的链接中找到“Build Now”链接，点击它，Jenkins 就会开始执行。在这组链接的下方有一个“Build History”列表，它显示的是这个 job 的所有构建历史，当第一次构建开始运行的时候，你会在列表中看到一个进度条，同时还有一个小圆球显示构建状态。圆球闪烁表示构建正在进行中，它停止闪烁的时候一般会是红色或蓝色，红色表示构建失败，蓝色表示成功。

如果这个 job 能够访问 Mercurial 版本库，找到了 HelloCI.msbuild 脚本，“RunUnitTest”执行成功，这个圆球应该会变蓝。这时候你也就顺利完成了第一个 Jenkins 构建。如果构建失败，请点击“Build History”对应的编号查看详细信息，然后点击“Console Output”，就可以看到 Jenkins 所执行的每一个命令和对应结果，从中可以分析出构建失败的原因。

```
1 The imported project "C:\Program Files (x86)\MSBuild\Microsoft\VisualStudio\v12.0\WebApplications\Microsoft.WebApplication.targets" was not found. Confirm that the path in the <Import> declaration is correct, and that the file exists on disk.
```

如果机器上安装有 Visual Studio 可以通过添加参数指定 Visual Studio 的版本来解决，如果未安装 Visual Studio 可以删除项目文件中的 Import 项。

error MSB1008: Only one project can be specified 是由于项目文件路径中有空格导致，将 *.csproj 文件的路径添加上双引号即可，如下代码片段所示。

```
1 msbuild.exe /p:VisualStudioVersion=12.0 "C:\Program Files(x86)\Jenkins\jobs\Tour\workspace\trunk\trunk\RR.Web.CCN.Tour\RR.Web.CCN.Tour.csproj"
```

MSB3030:Could not copy the file because it was not found msbuild.exe 在编译项目时，会将引用的项目文件拷贝到 bin 目录下，出现此错误首先查看引用文件的路径，在项目文件 *.csproj 中的 ItemGroup 的 Reference 配置节中，查看所有的引用文件在相应目录下是

否能够找到。如果用 Jenkins 始终无法成功构建，则可以用 msbuild.exe 手动尝试编译项目，确保使用 msbuild.exe 编译项目是成功的，用 msbuild.exe 编译项目的完整命令为：

```

1 #直接手动编译项目指定的参数
2 msbuild.exe /t:_CopyWebApplication /p:VisualStudioVersion=12.0 /p:Configuration=Release
   /p:outDir="E:\JenkinsPublish\Bin" "C:\Program Files (x86)\Jenkins\jobs\Tour
   \workspace\trunk\trunk\RR.Web.CCN.Tour\RR.Web.CCN.Tour.csproj"
3
4 #在 Jenkins 中指定的参数
5 /t:ResolveReferences;Compile /t:_CopyWebApplication /p:VisualStudioVersion=12.0 /p:
   Configuration=Release /p:WebProjectOutputDir=E:\Web\RR.Web.CCN.Tour\1.0.1
   /p:OutputPath=E:\JenkinsPublish\Bin

```

设置构建规则，编辑项目的构建触发器 (Build Trigger)，选择 SCM(Source Code Management)，设置如下：

```

1 #(每 5 分钟执行一次源码变动检查)
2 */5 * * * *
3 #To allow periodically scheduled tasks to produce even load on the system
4 #the symbol H (for “hash” ) should be used wherever possible.
5 #(每 5 分钟执行一次源码变动检查,Jenkin 推荐写法，与上一种写法等价)
6 H/5 * * * *

```

Jenkins 会每隔 5 分钟 Pull 源码，如果源码没有更新，不触发构建，如果源码有修改，则触发一次新的构建。

Publish 如果是发布 Web 站点，可以直接指定需要发布站点的 csproj 文件。在 Command Line Arguments 中，使用如下参数：

```

1 /t:ResolveReferences;Compile /t:_CopyWebApplication /p:Configuration=Release /p:
   WebProjectOutputDir=C:\Jenkins_Publish /p:OutputPath=C:\Jenkins_Publish\
   bin

```

其中 WebProjectOutputDir 是 web 站点的发布路径；OutputPath 是编译输出的 dll 路径。也可以通过 Visual Studio 使用发布选项建立好了项目发布的配置 xml 文件后，提交到源代码管理中最后在参数中指定配置的 xml 文件进行发布。

```

1 /t:Rebuild /p:Configuration=Release;PublishProfile=Jenkins-DEV;DeployOnBuild=true;
   VisualStudioVersion=11.0

```

PublishProfile 指定创建的 Profile 名称 (没有扩展名)，DeployOnBuild=true 表示启用编译并发布，VisualStudioVersion=11.0 表示 VS2012。

25.5.3 邮件通知 (E-Mail Notification)

为了保证持续的构建成功，当构建失败后需要通知相关人员。以 163 邮箱配置为例，在 Jenkins 全局配置的邮件通知模块下，配置 SMTP 服务器为：smtp.163.com，勾选使用 SMTP 认证，认证的用户名填写 163 邮箱地址：dolphinjiang123@163.com。注意密码不是登录邮箱的密码，而是在 163 邮箱开通 SMTP 服务时客户端的授权码，163 客户端授权码可以重置。全局变量配置完毕后，在单个项目的构建配置增加构建后操作，添加构建失败需要通知的人员，多个邮箱地址以空格键相隔。管理员邮箱的名字需要和认证的邮箱的名字一致。

25.5.4 自动打包

25.5.5 Jenkins 执行批处理命令

在编译成功后，将编译成功的文件发布到网络文件夹。在 Jenkins 中添加构建后操作，批处理脚本如下。

```
1 xcopy D:\ClientRelease\JenkinRelease\*.* \\192.168.24.253\公共区\Client\2.0\Release /S /R  
/Y
```

初始执行时提示无效的驱动器规格。将 Jenkins 服务的启动登录账户权限修改下即可（因为 Jenkins 登录账户还没有保存登录远程账户的凭据），如下图25.6所示。

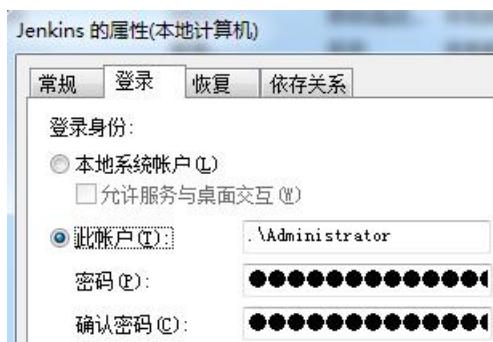


图 25.6: Jenkins 配置登录账户

有时希望每次构建的的版本有一个类似于版本号标记的标识，测试部门更加容易明白构建具体的版本，在执行 xcopy 处理命令时创建带日期的文件夹，命令如下所示：

```
1 #生成的文件夹形如: Client2.0-Release-20160524010923  
2 xcopy D:\ClientRelease\JenkinRelease\*.* \\192.168.24.253\公共区\Client\2.0\Client2.0-  
Release-%date:~0,4%-%date:~5,2%%date:~8,2%-%time:~1,1%%time:~3,2%%time  
~-6,2% /S /R /Y /I
```

其中/I 参数在目标文件夹不存在时创建目标文件夹，也可以使用 Jenkins 的变量，用构建的版本号 (Build Number) 作为文件夹名称：

```

1 #生成的文件夹形如: Client2.0-Release-20160524-315
2 xcopy D:\ClientRelease\JenkinRelease\*.* \\192.168.24.253\公共区\Client\2.0\Client2.0-
   Release-%date:~0,4%%date:~5,2%%date:~8,2%-%BUILD_NUMBER% /S /R /Y /I

```

25.6 Travis CI

25.7 自动化脚本

25.7.1 生成更新文件

```

1 @echo off
2 title 生成更新文件
3 set sourcePath=D:\软件共享\走喔热印部署软件\V0.0.4-Stable-Release20150710\Debug
4 set targetPath=D:\update
5
6 :: Copy update file
7 copy %sourcePath%\RRMall.WxPrint.Bll.dll %targetPath%
8 copy %sourcePath%\RRMall.WxPrint.Common.dll %targetPath%
9 copy %sourcePath%\RRMall.WxPrint.Dal.dll %targetPath%
10 copy %sourcePath%\RRMall.WxPrint.exe %targetPath%
11 copy %sourcePath%\RRMall.WxPrint.exe.config %targetPath%
12 copy %sourcePath%\RRMall.WxPrint.Model.dll %targetPath%
13
14 pause

```

源路径（Source Path）为需要更新的版本文件存放路径，目的路径（Target Path）为生成的更新文件存放路径。当批处理文件中有中文时，文件本身的编码需要保存为 ANSI 编码，否则脚本执行时会出现乱码的现象。

25.8 Gradle

25.8.1 安装

新建环境变量 GRADLE_HOME，值为 Gradle 的解压路径；

25.9 JIRA

JIRA 是一个优秀的问题 (or bugs,task,improvement,new feature) 跟踪及管理软件。它由 Atlassian 开发，采用 J2EE 技术. 它正被广泛的开源软件组织，以及全球著名的软件公司使用，它堪称是 J2EE 的 Bugzilla。

25.9.1 JIRA 部署

卸载安装包 如果有必要，卸载原来的 JDK。

```
1 whereis java
2 which java (java执行路径)
3 echo $JAVA_HOME
4 echo $PATH
5
6 #查看 jdk 的信息
7 rpm -qa|grep java
```

JDK 包安装 安装 JDK 的 RPM 包。

```
1 #添加执行权限
2 chmod +x jdk-8u91-linux-64.rpm
3 #安装
4 rpm -ivh jdk-7u25-linux-x64.rpm
```

配置环境变量

在 etc/profile 文件下添加

```
1 export JAVA_HOME=/usr/java/jdk1.8.0_91
2 export CLASSPATH=/usr/java/jdk1.8.0_91/lib
3 export PATH=$JAVA_HOME/bin:$PATH
```

命令 source /etc/profile 使配置文件立即生效

MySQL 安装 MySQL 的安装配置如下代码所示。

```
1 #yum 安装 mysql
2 yum -y install mysql-server
3 #设置开机启动
4 chkconfig mysqld on
5
6 c) 启动MySQL服务
7 # service mysqld start
8
9 d) 设置MySQL的root用户设置密码
10 # mysql -u root
11 mysql> select user,host,password from mysql.user;
12
13 e) 设置root用户密码
14 mysql> set password for root@localhost=password('root');
15
16 f) 删除密码为空的账户 (可选)
17 mysql>delete from mysql.user where user=' ' ;
```

```

18 mysql> exit
19
20 g) 开放远程登录权限
21
22 GRANT ALLPRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'root' WITH GRANT
   OPTION;
23 FLUSHPRIVILEGES;

```

创建 jira 相关 创建 JIRA 相关如下代码所示。

```

1 a) 创建数据库:
2 CREATE DATABASEjiradb CHARACTER SET utf8 COLLATE utf8_bin;
3
4 b) 创建用户
5 INSERT INTOmysql.user(HOST, USER, PASSWORD) VALUES("localhost","jirauser",
   PASSWORD("1234"));
6
7 c) 授权
8 GRANT SELECT,INSERT, UPDATE, DELETE, CREATE, DROP, ALTER, INDEX ON
   jiradb.* TO'jirauser'@'localhost' IDENTIFIED BY '1234';
9 GRANT ALLPRIVILEGES ON *.* TO jirauser@'%' IDENTIFIED BY '1234' WITH
   GRANT OPTION;
10 flushprivileges ;

```

当在外部无法访问 JIRA 时需要在防火墙中将 8080 端口开放。配置 MySQL 连接通过 bin 下面的 config.sh 进行。

汉化

破解

25.10 Ansible

ansible 是个什么东西呢？官方的 title 是“Ansible is Simple IT Automation”——简单的自动化 IT 工具。这个工具的目标有这么几项：让我们自动化部署 APP；自动化管理配置项；自动化的持续交付；自动化的（AWS）云服务管理。

Chapter 26

测试调试组件

26.1 WinDbg

WinDbg 是在 windows 平台下，强大的用户态和内核态调试工具。它能够通过 dmp 文件轻松的定位到问题根源，可用于分析蓝屏、程序崩溃（IE 崩溃）原因。WinDbg 是微软发布的一款相当优秀的源码级（source-level）调试工具，可以用于 Kernel 模式调试和用户模式调试，还可以调试 Dump 文件。日志文件存放在以下目录：核心内存转储路径：C:/WINDOWS/MEMORY.DMP，小内存转储文件路径：C:/WINDOWS/Minidump/

26.1.1 基本使用

WinDbg 设置 Symbol Path 符号表是 WinDbg 关键的“数据库”，如果没有它，WinDbg 无法分析出更多问题原因。运行 WinDbg-> 菜单->File->Symbol File Path 在弹出的框中输入

```

1 #srv 后的内容是设置的目的是下载该程序用到的操作系统相关的库函数的符号表到本地
2 D:\testdmp;srv*d:\symbolslocal*http://msdl.microsoft.com/download/symbols
3
4 cache*D:\MySymbol;srv*http://msdl.microsoft.com/download/symbols
5 #放到局域网共享目录 (未测试成功)
6 srv*\192.168.24.253\研发部\公用\Develop\Debug*http://msdl.microsoft.com/download/
    symbols

```

按照这样设置，WinDbg 将先从本地文件夹 C:\MySymbol 中查找 Symbol，如果找不到，则自动从 MS 的 Symbol Server 上下载 Symbols。另一种做法是从这个 Symbol 下载地址中 http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx，下载相应操作系统所需要的完整的 Symbol 安装包，并进行安装，例如我将其安装在 D:\WINDOWS\Symbols，在该框中输入“D:\WINDOWS\Symbols”。

输出内容自动换行 有时键入命令后输出的内容太长，而又不想拖动 Command 窗口的左右滚动条进行查看，此时可以设置输出内容自动换行，当超出当前屏幕展示范围时，自动从下一行开始显示，如图26.1所示。

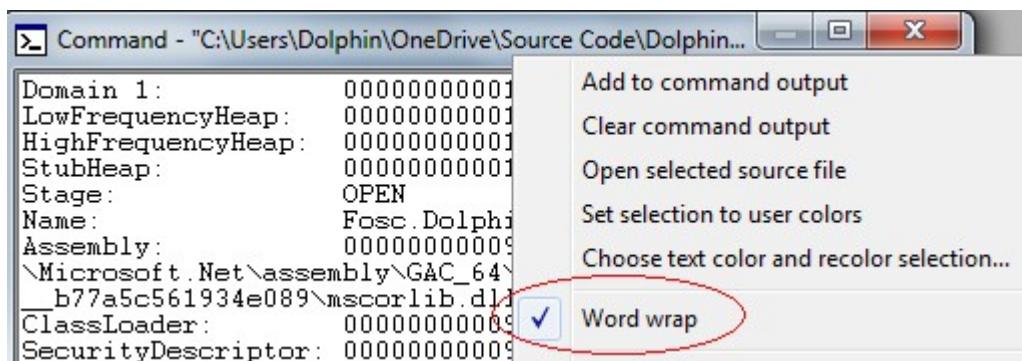


图 26.1: WinDbg 自动换行

26.1.2 dump 文件

他的英文翻译就是“转存”。也就是说把内存中或者其他输入转存到另一个位置，当然对于我们现在说的 dump 就是把内存中运行的 PE 进程的数据，从内存中抓取出来，然后在用文件的形式保存下来。根据上面的分析我们基本上得到了一个这样的思维。Dump 程序要做的事情分几个基本的步骤：1. 在系统中找到目标进程 2. 在进程中确定进程的大小 imagesize 3. 把进程中的数据保存到文件

Dump 文件是进程的内存镜像。可以把程序的执行状态通过调试器保存到 dump 文件中。Dump 文件是用来给驱动程序编写人员调试驱动程序用的，这种文件必须用专用工具软件打开，比如使用 WinDbg 打开。当我们的程序发布出去之后，在客户机上是无法跟踪自己代码的 bug 的，所以 Dump（扩展名是.dmp）文件对于我们来说特别有用。我们可以通过.dmp 文件把出现 bug 的情况再现，然后根据再现的状况（包括堆栈调用等情况），可以找到出现 bug 的行号，甚至是出现 bug 的点。

26.1.3 获取 dump 文件

由于 dump 文件记录的是进程某一时刻的具体信息，所以保存 dump 的时机非常重要。比如程序崩溃，dump 应该选在引发崩溃的指令执行时（也就是 1st chance exception 发生的时候）获取，这样分析 dump 的时候就能够看到问题的直接原因。Adplus 是跟 Windbg 在同一个目录的 VBS 脚本。Adplus 主要是用来抓取 dump 文件。使用如下的命令跟踪程序，在程序 Crash 时生成 dump 文件：

```
1 adplus.exe -crash -pn OneDrive.exe -fullonfirst -o D:\dumps
```

加上-fullonfirst 参数后，无论是 1st chance exception 还是 2nd chance exception，都会生成 full dump 文件。Adplus 更灵活的方法就是用-c 参数带配置文件。在配置文件里面，可以选择 exception 发生的时间，生成的 dump 是 mini dump 还是 full dump，还可以设定断点等等。

打开 WinDbg, File/Attach to a process/ 然后再列表中显示需要监视的进程 (.exe) 当程序崩溃之后执行 DUMP 命令产生 dmp 文件，命令有：.dump /m C:/dumps/myapp.dmp 、.dump /ma C:/dumps/myapp.dmp、.dump /mFhutwd C:/dumps/myapp.dmp

执行以上就产生了 dmp 文件

26.1.4 调试 dump 文件

26.1.5 WinDbg 加载 SOS

SOS 调试扩展 (SOS.dll) 通过提供有关内部公共语言运行库 (CLR) 环境的信息，帮助您在 WinDbg.exe 调试器和 Visual Studio 中调试托管程序。注意 Visual Studio 2013 不支持 SOS，If you are using Visual Studio 2013, SOS.dll is supported in the Windows Debugger within Visual Studio, but not in the Immediate window of the Visual Studio debugger. 使用如下命令加载

SOS:

```
1 .load C:\Windows\Microsoft.NET\Framework64\v4.0.30319\SOS.dll
2 g #开始调试
3 .loadby sos clr
```

使用.load 加载程序集时需要制定全路径，使用.loadby 加载程序集时指定程序集名称即可。Specifies the debugger extension DLL to load. If you use the .load command, DLLName should include the full path. If you use the .loadby command, DLLName should include only the file name. 如果出现了“Unable to find module ‘clr’”这样的错误。请键入 g 让调试程序运行一会儿，停下来的时候再尝试命令.loadby sos clr，这时一般都会成功。

26.1.6 symstore

SymStore (symstore.exe) 是用于创建符号存储的工具。它被包含在 Windows 调试工具包中。SymStore 按照某种格式存储符号，使得调试器可以通过时间戳、映像大小(对于.dbg 或可执行文件)、签名和寿命 (.pdb 文件) 来查找符号。使用符号存储而不是常规的符号存储格式的好处是，所有符号都可以在同一个服务器上进行存储或引用，而调试器不需要知道具有哪些产品对应的符号。注意，不同版本的.pdb 符号文件(例如共有和私有符号)不能保存在相同的符号服务器中，因为他们具有相同的签名和寿命。添加符号文件到符号服务器：

```
1 symstore.exe add /r /f D:\Source\2.0\*.pdb /s \\192.168.24.253\研发部\公用\Develop\Debug
 \Symbol /t BRProduct /z pri
```

symstore.exe 在 Windows SDK 调试文件夹下。

26.2 Fiddler(Free,Not Open Source)

Fiddler (中文名称：小提琴) 是一个 HTTP 的调试代理，以代理服务器的方式，监听系统的 Http 网络数据流动，Fiddler 可以也可以让你检查所有的 HTTP 通讯，设置断点。Fiddler 默认监听端口为 8888，有时 8888 端口会被其他程序占用，Fiddler 会选择一个随机的端口。查看 Fiddler 的监听端口在：Tools->Fiddler Options->Connections 下。如图26.2所示：

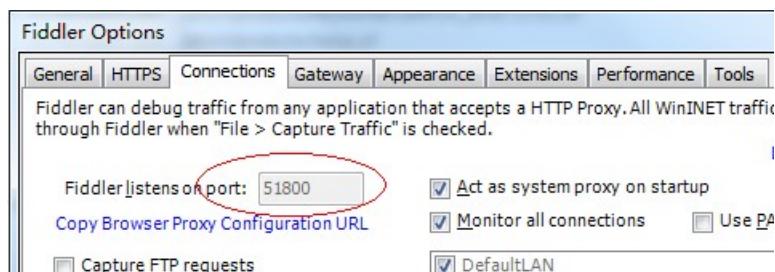


图 26.2: Fiddler 监听端口

在浏览器中将代理端口设置为 Fiddler 相应端口值即可。

26.2.1 Fiddler Capture

Fiddler 暂时没有找到暂停抓包的选项，有时为了仔细分析某个特定的请求，需要停止抓包或者说是暂停抓包（过多的包干扰分析），此时有 2 个选择，一是可以禁止左侧的请求记录列表自动滚动（AutoScroll Session List），任意 HTTP 列表记录右键单击即出现此菜单，二是在下方有展示或者抓取的进程，选择 Hide All 选项，如图26.3所示。

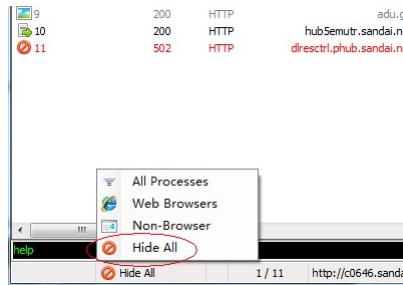


图 26.3: Fiddler 隐藏所有进程数据流向

注意在用宽带拨号上网时代理设置需要在宽带连接中进行设置，而不是设置局域网（LAN）的代理，设置局域网代理是本机处于局域网时，当电脑直接用宽带拨号时会动态分配一个广域网的 IP，重新拨号时 IP 会再分配，不再是原有 IP，宽带设置代理如图26.4所示。

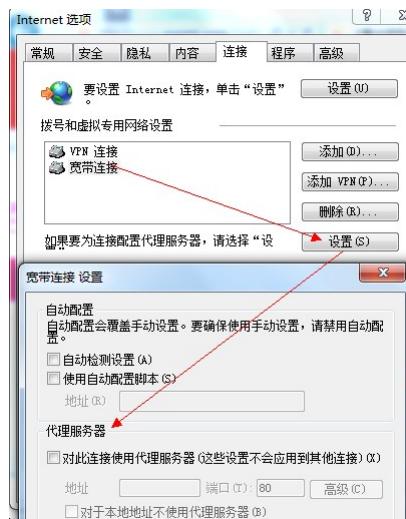


图 26.4: Fiddler 宽带连接代理设置

有时设置好后会发现 Fiddler 可以捕获 Firefox 浏览器的 HTTP 的请求，但是却无法捕获 HTTPS 请求，而 Fiddler 的 Options 中已经勾选了解析 HTTPS 流量的选项，是因为此时 Firefox 浏览器仅仅设置了 HTTP 使用本地 Fiddler 代理，而未设置 HTTPS 同样也使用本地的 Fiddler 代理，解决此问题可在 Firefox 浏览器的高级-> 网络-> 连接设置中设置“为所有协议使用相同代理”。

有时我们需要一张网页的原始码，但是无法通过浏览器直接获取，例如有的微信开发的网页，由于带有微信认证，所以只能在微信浏览器中打开，而微信浏览器到目前为止是不允许查看网页源代码的，此时就可以在微信浏览器中打开网页，在 Fiddler 中去捕获相应的网页响应数

据包，即可以获得网页的源代码。

26.2.2 Fiddler 捕获 HTTPS

要抓取走 HTTPS 的 JS 内容，Fiddler 必须解密 HTTPS 流量。但是，浏览器将会检查数字证书，并发现会话遭到窃听。为了骗过浏览器，Fiddler 通过使用另一个数字证书重新加密 HTTPS 流量。Fiddler 被配置为解密 HTTPS 流量后，会自动生成一个名为 DO_NOT_TRUST_FiddlerRoot 的 CA 证书，并使用该 CA 颁发每个域名的 TLS 证书。若 DO_NOT_TRUST_FiddlerRoot 证书被列入浏览器或其他软件的信任 CA 名单内，则浏览器或其他软件就会认为 HTTPS 会话是可信任的、而不会再弹出“证书错误”警告。菜单栏 Tools»Fiddler Options»HTTPS，将 Decrypt HTTPS Traffic 复选框勾选上，如图26.5所示：

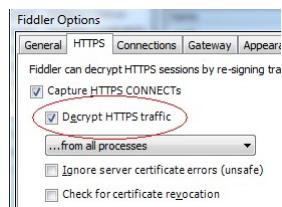


图 26.5: Fiddler 捕获 HTTPS 通信

Fiddler2 使用 man-in-the-middle(中间人) 攻击的方式来截取 HTTPS 流量。在 Web 浏览器面前 Fiddler2 假装成一个 HTTPS 服务器，而在真正的 HTTPS 服务器面前 Fiddler2 假装成浏览器。Fiddler2 会动态地生成 HTTPS 证书来伪装服务器。

26.2.3 Fiddler 应对 HSTS

HSTS 是什么 HSTS(HTTP Strict Transport Security) 国际互联网工程组织 IETE 正在推行一种新的 Web 安全协议，HSTS 的作用是强制客户端（如浏览器）使用 HTTPS 与服务器创建连接。HTTP 严格传输安全 (HSTS) 功能使 Web 服务器告知浏览器绝不使用 HTTP 访问，在浏览器端自动将所有到该站点的 HTTP 访问替换为 HTTPS 访问。如果一个 web 服务器支持 HTTP 访问，并将其重定向到 HTTPS 访问的话，那么访问者在重定向前的初始会话是非加密的。举个例子，比如访问者输入 <http://www.foo.com/> 或直接输入 foo.com 时。这就给了中间人攻击的一个机会，重定向可能会被破坏，从而定向到一个恶意站点而不是应该访问的加密页面。HTTP 严格传输安全 (HSTS) 功能使 Web 服务器告知浏览器绝不使用 HTTP 访问，在浏览器端自动将所有到该站点的 HTTP 访问替换为 HTTPS 访问。HSTS 是一个被当前大多数 Web 浏览器所支持的 Web 安全策略，它可以帮助 Web 管理员保护他们的服务器和用户避免遭到 HTTPS 降级、中间人攻击和 Cookie 劫持等 HTTPS 攻击的危害。

HSTS 可以用来抵御 SSL 剥离攻击。SSL 剥离攻击是中间人攻击的一种，由 Moxie Marlinspike 于 2009 年发明。他在当年的黑帽大会上发表的题为“New Tricks For Defeating SSL In Practice”的演讲中将这种攻击方式公开。SSL 剥离的实施方法是阻止浏览器与服务器创建 HTTPS 连接。它的前提是用户很少直接在地址栏输入 <https://>，用户总是通过点击链接或 3xx

重定向，从 HTTP 页面进入 HTTPS 页面。所以攻击者可以在用户访问 HTTP 页面时替换所有 https://开头的链接为 http://，达到阻止 HTTPS 的目的。

HSTS 可以很大程度上解决 SSL 剥离攻击，因为只要浏览器曾经与服务器创建过一次安全连接，之后浏览器会强制使用 HTTPS，即使链接被换成了 HTTP。

另外，如果中间人使用自己的自签名证书来进行攻击，浏览器会给出警告，但是许多用户会忽略警告。HSTS 解决了这一问题，一旦服务器发送了 HSTS 字段，用户将不再允许忽略警告。

当你通过一个无线路由器的免费 WiFi 访问你的网银时，很不幸的，这个免费 WiFi 也许就是由黑客的笔记本所提供的，他们会劫持你的原始请求，并将其重定向到克隆的网银站点，然后，你的所有的隐私数据都曝光在黑客眼下。

严格传输安全可以解决这个问题。如果你之前使用 HTTPS 访问过你的网银，而且网银的站点支持 HSTS，那么你的浏览器就知道应该只使用 HTTPS，无论你是否输入了 HTTPS。这样就防范了中间人劫持攻击。

使 Fiddler 可应对 HSTS 而 Fiddler 捕获 HTTPS 是正是通过中间人攻击的方式来实现的，如果首次访问过支持 HSTS 的网站比如淘宝，那么以后所有的连接都会强制使用 HTTPS 连接，反正 Fiddler 无法捕获，浏览器也会因为安全原因也无法访问淘宝站点，应对的办法就是将浏览器的第一次得到就行了，让 HSTS 始终无法生效即可。如果以前访问过淘宝，那么将来所有的请求都会使用 HTTPS 连接，要想使用 HTTP 需要删除 HSTS 设置，在 Chrome 中键入：

```
1 chrome://net-internals/#hsts
```

删除 HSTS 设定，如图所示：

Delete domain

Input a domain name to delete it from the HSTS set (yo
Domain: Delete

Query domain

Input a domain name to query the current HSTS set:
Domain: Query

Found:

- static_sts_domain: www.taobao.com
- static_upgrade_mode: UNKNOWN
- static_sts_include_subdomains:
- static_sts_observed:
- static_pkp_domain:
- static_pkp_include_subdomains:
- static_pkp_observed:
- static_spki_hashes:
- dynamic_sts_domain: www.taobao.com
- dynamic_upgrade_mode: STRICT
- dynamic_sts_include_subdomains: false
- dynamic_sts_observed: 1458266037.237044
- dynamic_pkp_domain:
- dynamic_pkp_include_subdomains:
- dynamic_pkp_observed:
- dynamic_spki_hashes:

图 26.6: 查询 HSTS 设定

将关键字链接放入 Delete Domain 中进行删除即可，如果你之前没有使用 HTTPS 访问过该站点，那么 HSTS 是不奏效的。网站需要通过 HTTPS 协议告诉你的浏览器它支持 HSTS。服务器开启 HSTS 的方法是，当客户端通过 HTTPS 发出请求时，在服务器返回的 HTTP 响应头中包含 Strict-Transport-Security 字段，如图26.7示。非加密传输时设置的 HSTS 字段无效。



图 26.7: 响应头中 Strict-Transport-Security 字段

从图中可以看出服务端设置 Strict-Transport-Security 的值为：max-age=31536000，此值正好是 365 天即 1 年经过的秒数，不过由于第一次即通过 Fiddler 代理，HSTS 此时已经无法生效了，从 Fiddler 捕获的请求可以看出，以后的请求都使用的是普通的未加密的 HTTP。

26.2.4 Fiddler 移动端 Capture

Fiddler 同样可以捕获移动端的 HTTP 请求，如果手机和电脑在同一个局域网内，在手机端设置无线代理的 IP 为 PC 端的 IP，端口为 8888，如果 PC 在广域网中的话，设置移动端的 IP 为广域网的 IP，此时手机可在任意网络（需要和 PC ping 通，不能是单独的网络）中，PC 都能够捕获到手机端的请求。另需在 Fiddler 中允许远程抓包，具体设置如图26.8所示。

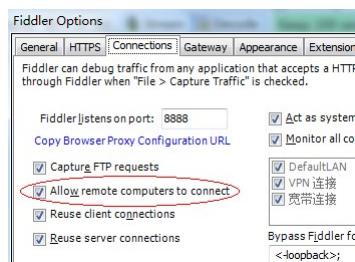


图 26.8: Fiddler 开启远程捕获

26.2.5 Fiddler 测试 API

Fiddler 的接口测试页默认是未显示的 (V4.6.2.3)，在 View->Tabs->APITest 中可打开接口测试 Tab，在请求列表中将需要测试的 Api 请求拖动到 Tab 标签中即可。测试后需要将测试记录保存为 saz 结尾的文件，下次测试的时候导入即可。

26.2.6 图标含义

<http://docs.telerik.com/fiddler/configure-fiddler/tasks/configurefiddler>

Each session is marked with an icon for quick reference:

- Request is being sent to the server
- Response is being read from the server
- Request is paused at a breakpoint
- Response is paused at a breakpoint
- Request used HTTP HEAD method; response should have no body
- Request used HTTP POST method
- Request used HTTP CONNECT method; this establishes a tunnel used for HTTPS traffic
- Response was HTML
- Response was an image
- Response was a script
- Response was Cascading Style Sheet
- Response was XML
- Response was JSON
- Response was an audio file
- Response was a video file
- Response was a Silverlight applet
- Response was a Flash applet
- Response was a font
- Generic successful response
- Response was HTTP/300,301,302,303 or 307 redirect
- Response was HTTP/304: Use cached version
- Response was a request for client credentials
- Response was a server error
- Session was aborted by the client, Fiddler, or the Server.

图 26.9: Fiddler 图标含义

26.2.7 配置过滤规则 (Configurate Filter Rule)

Fiddler 可以过滤出与某些特定网址交互的信息，如图26.10所示。注意配置主机时，不用带 http://，和左侧的列表中的主机名称一致即可。配置好过滤规则后可点击右上角的 Actions 按钮保存或者应用过滤规则。

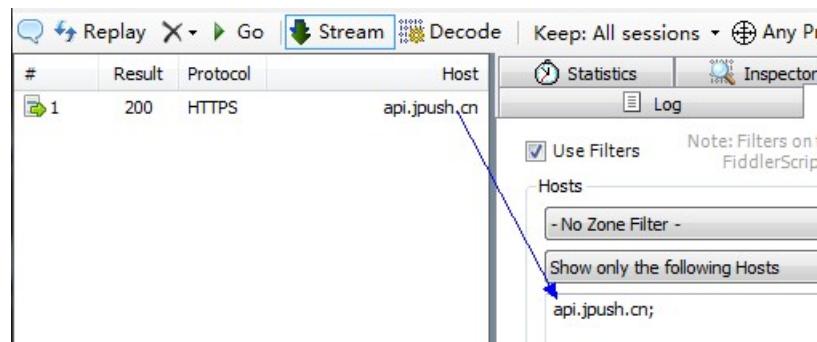


图 26.10: Fiddler Filter 过滤

26.2.8 Composer

Fiddler 的作者把 HTTP Request 发射器取名叫 Composer(中文意思是：乐曲的创造者)，很有诗意。Fiddler Composer 的功能就是用来创建 HTTP Request 然后发送。可以自定义一个 Request，也可以手写一个 Request，甚至可以在 Web 会话列表中拖拽一个已有的 Request. 来创建一个新的 HTTP Request. 能创建发送 HTTP Request 的工具很多很多。但是 Fiddler 的功能有如下的优势。

1. 能从“Web 会话列表”中拖拽一个先前捕获到的 Request，然后稍微修改一下
2. 发送 Request 后，还能设置断点，继续修改 Request.
3. 支持在 Request 中上传文件
4. 支持发送多次 Request.

在 Composor 窗口的右侧有 Request History 列表，可以右键添加 Comment，给请求的 Url 设置别名以便于识别，默认情况下 Composor 会自动过滤掉重复的 Url。有许多时候 RESTful 接口涉及到认证的过程，而 Fiddler 的请求头中默认是不带认证信息的，可将认证信息通过 FiddlerScript 统一写入，而不用每次都粘贴认证信息。在 Fiddler 的 OnBeforeRequest 方法中添加如下信息即可。

```

1 if(oSession.host.Contains("192.168.1.222:801")||oSession.host.Contains("129.89.81.118:801
2   "))
3 {
4     oSession.oRequest["auth"] = "app|e10adc3949ba59abbe56e057f20f883e";
5 }
```

在添加请求头时用 host 条件进行过滤，只需要与特定主机交互时才发送认证信息，否则在

所有的请求头中都会添加上认证信息。

26.2.9 Statistics

使用 Fiddler 的统计可以查看网页的性能信息，包括请求了多少次，耗时等，查看 Fiddler 的统计图表可以。很清楚的看到加载了哪些类型的文件，已及文件的大小，文件相对于页面的占比。有时一个页面包含非常多的请求，可以点击查找 (Find) 按钮，如图所示。

Margin Text

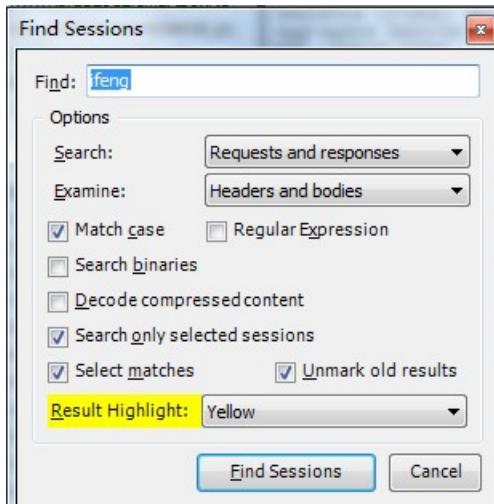


图 26.11: 查询匹配的请求

以凤凰网为例，关键字中输入“ifeng”，勾选“选中匹配选项 (Select matches)”，在右侧是统计图中即可看出当前页面的统计信息，包括图片，css，js 等文件的占比。

26.2.10 Inspectors

Inspectors 是检查员，视察员的意思，Inspectors 分为上下两个部分，上半部分是请求头部分，下半部分是响应头部分。对于每一部分，提供了多种不同格式查看每个请求和响应的内容。JPG 格式使用 ImageView 就可以看到图片，HTML/JS/CSS 使用 TextView 可以看到响应的内容。Raw 标签可以查看原始的符合 HTTP 标准的请求和响应头。Auth 则可以查看授权 Proxy-Authorization 和 Authorization 的相关信息。Cookies 标签可以看到请求的 cookie 和响应的 set-cookie 头信息。部分网站的 Inspectors 请求里可以直接看到登录网站的用户名和密码，一般 HTTPS 类型的网站会直接将明文的用户名密码放在 body 中，因为网络数据包会通过 SSL 进行加密，不够安全的 HTTP 网站也将明文直接暴露在 body 中，密码信息很容易泄漏，如图26.12所示：

一般使用 HTTPS 进行登录的网站会直接明文使用 SSL 加密，能够解密 SSL 流量，即可以看到明文密码。在 QQ 发送接收离线文件时也是通过 HTTP 进行传输，且未加密，可以通过 Fiddler 看到部分发送的文件的内容。

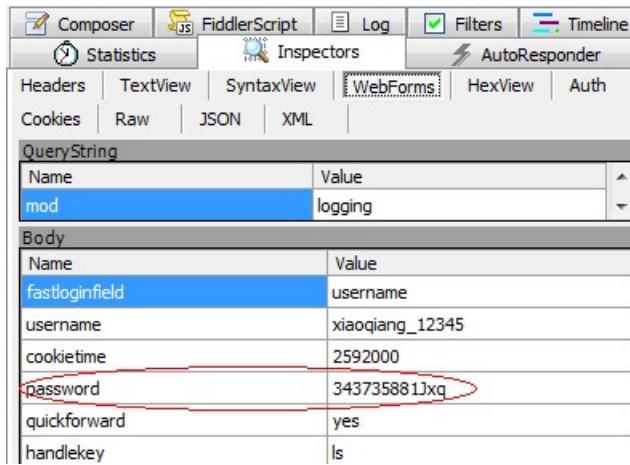


图 26.12: 请求体中的用户名密码

26.2.11 断点调试 (Breakpoint Debugging)

在启用了 Fiddler 断点调试的功能后，通过 Fiddler 代理请求的所有网站不是立即响应，在请求之前可以对请求的数据包进行编辑。

26.3 NUnit (MIT License)

NUnit 是一个专门针对.NET 来写的单元测试框架，它是 xUnit 体系中的一员，在 xUnit 体系中还有针对 Java 的 JUnit 和针对 C++ 的 CPPUnit，在开始的时候 NUnit 和 xUnit 体系中的大多数的做法一样，仅仅是将 Smalltalk 或者 Java 版本转换而来，但是在.NET2.0 之后它加入了一些特有的做法。

NUnit is a unit-testing framework for all .Net languages. Initially ported from JUnit, the current production release, version 2.6, is the seventh major release of this xUnit based unit testing tool for Microsoft .NET. It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example custom attributes and other reflection related capabilities. NUnit brings xUnit to all .NET languages.

以测试 RRMail.WxPrint.Common 命名空间下系统操作类 SystemOperHelper 的关闭计算机方法为例，介绍 NUnit 的使用。

添加 NUnit 引用 在项目的引用上添加 NUnit 的引用，示例的 NUnit 版本号为 2.6.4.14350，Visual Studio 2013，如图26.13所示。

类添加 TestFixture 在项目的类上添加 TestFixture，如图26.14所示。

TestFixture 带参数也可用于带构造参数的类的初始化。如下代码片段所示：

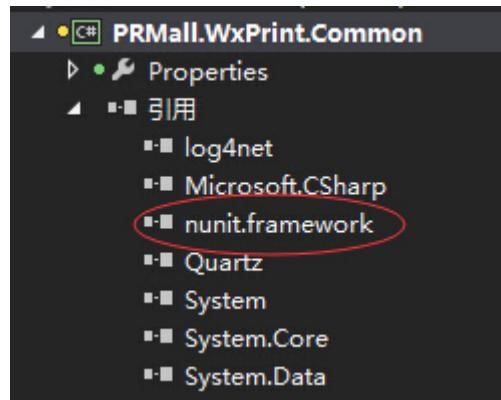


图 26.13: 添加 NUnit 引用

```
SystemOperHelper.cs  ✘  JobHelper.cs
PRMall.WxPrint.Common.Cmd.SystemOperHelper
4 | using System.Text;
5 | using NUnit.Framework;
6 |
7 |namespace PRMall.WxPrint.Common.Cmd
8 |
9 | [TestFixture]
10| public class SystemOperHelper
11| {
12|     #region Attribute
13|     private static readonly log4net.ILog logger =
14|     #endregion
15| }
```

The screenshot shows the 'SystemOperHelper.cs' file in the Visual Studio code editor. The class definition is as follows:

```
4 | using System.Text;
5 | using NUnit.Framework;
6 |
7 |namespace PRMall.WxPrint.Common.Cmd
8 |
9 | [TestFixture]
10| public class SystemOperHelper
11| {
12|     #region Attribute
13|     private static readonly log4net.ILog logger =
14|     #endregion
15| }
```

The line '9 | [TestFixture]' contains the attribute being highlighted with a red oval.

图 26.14: 类上添加 TextFixture

```

1 [TestFixture("hello", "hello", "goodbye")]
2 public class ParameterizedTestFixture
3 {
4     private string eq1;
5     private string eq2;
6     private string neq;
7
8     public ParameterizedTestFixture(string eq1, string eq2, string neq)
9     {
10         this.eq1 = eq1;
11         this.eq2 = eq2;
12         this.neq = neq;
13     }
14 }

```

方法添加 TestCase("4000") 在项目的方法上添加 TestCase，其中字符串 4000 为方法传入的参数，测试方法要求必须为 Public，如图26.15所示。



图 26.15: 方法上添加 TestCase

在单元调试时将类库类型的项目 PRMall.WxPrint.Common 设置为启动项目，在项目的属性 » 调试 » 启动操作里设置 NUNIT-GUI 的路径，如：C:/Program Files (x86)/NUnit 2.6.4/bin，如图26.16所示，设置完成即可进行单元调试，调试时会自动弹出单元调试的界面，第一次调试时左侧边框未加载任何 dll，在 Project 菜单上点击 Add Assembly 添加 dll，如图26.17所示。

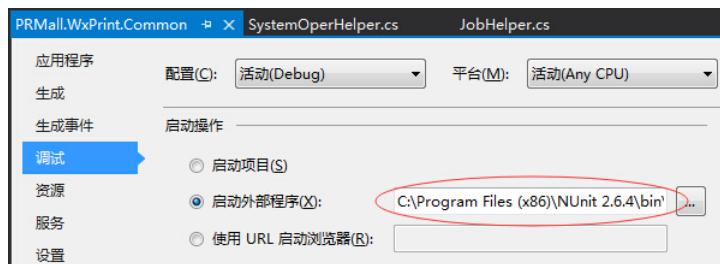


图 26.16: 设置 NUNIT-GUI 启动路径

由于种种原因，有一些测试我们不想运行。当然，这些原因可能包括你认为这个测试还没

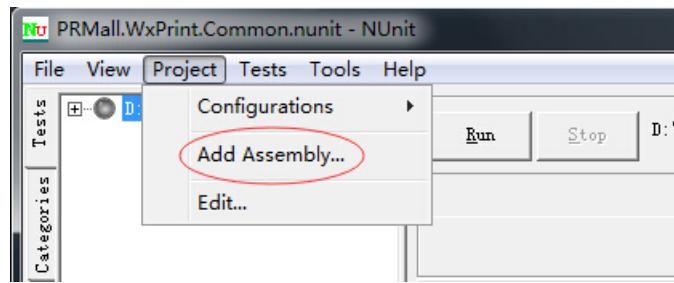


图 26.17: NUnit 添加调试 dll

有完成, 这个测试正在重构之中, 这个测试的需求不是太明确. 但你又不想破坏测试, 不然进度条可是红色的哟. 怎么办? 使用 Ignore 属性. 你可以保持测试, 但又不运行它们. 让我们标记 MultiplyTwoNumbers 测试方法为 Ignore 属性, 如图26.18所示:

```

1  #region 更新奖品状态
2  ///<summary>
3  ///
4  ///</summary>
5  ///<param name="key"></param>
6  ///<returns></returns>
7  [TestCase("ba71f653-0c65-4f3a-9344-af201c1afab2", Result = true)]
8  [Ignore]
9  public bool UpdateAwardStatus(string key)
10 {
11     string query = string.Concat(new object[]
12     {
13         "update tb_lotteryUser",
14         "set status = 1 where lotteryUser = @key"
15     });
16 }

```

图 26.18: NUnit 的 Ignore 属性

在 NUnit 上跑测试时可以添加测试用例 (TestCase) 的描述 (Description) 和所属的分类 (Category), 如下代码片段所示。

```

1 [TestCase("", Description = "房产特殊房源回复", Category = "照片打印用例")]
2 [TestCase("", Description = "抽奖关键字回复", Category = "关键字回复用例")]
3 public void GetWxMessage(string message)
4 {
5     //Execute code
6 }

```

添加完毕后在 GUI 面板的 Categories 选项卡中会看到添加的分类, 可选择某一分类的测试用例进行测试。

26.3.1 NUnit 在 .NET.Framework 4.0 调试

在使用 NUnit 版本 2.6.4 时无法在 .NET.Framework 4.0 中调试, 需要修改 NUnit 的配置文件使之支持在 .NET.Framework 4.0 中调试, 在配置文件 nunit.exe.config 的 startup 配置节中添加支持 .NET.Framework 4.0 的配置内容, 如下代码片段所示:

```

1 <!-- Comment out the next line to force use of .NET 4.0 -->
2 <requiredRuntime version="v4.0.30319" />
3 <supportedRuntime version="v4.0.30319" />
```

<supportedRuntime> 元素标记应用程序支持的运行库版本。<supportedRuntime> 元素可同时定义多个，<supportedRuntime> 元素的顺序定义系统上可用的多个运行库版本的优先级¹。查看本机.NET.Framework 4.0 具体的版本号可以在文件管理器中输入如下路径：

```
1 %systemroot%\Microsoft.NET\Framework
```

26.3.2 使用总结

在使用 NUnit 测试时用代码获取的路径较特殊，如用如下代码本意是想获取当前程序的运行路径，而实际获取的路径为 NUnit 文件的安装路径。

```
1 string tempPath = Application.StartupPath + @"\temp\";
```

获取的实际路径为 C:\Program Files (x86)\NUnit 2.6.4\bin\temp\。在使用 NUnit 进行单元测试时，建议在新的项目中引用，避免在多个项目中引用，如此，当不再需要 NUnit 进行测试时，移除 NUnit 也非常方便。

26.4 xUnit

26.4.1 使用 xUnit

注意 xUnit 从.NET Framework 4.5 开始支持，最低版本为.NET Framework 4.5。安装 xUnit 在项目右键菜单“程序包管理器”，输入 xUnit。安装 xUnit：

```

1 #安装最新版本
2 Install-Package xunit
3 #安装 2.1.0 版本
4 Install-Package xunit -Version 2.1.0
```

简单的单元测试 Demo 如下：

```

1 public class UnitTest1
2 {
3     [Fact]
4     public void PassingTest()
5     {
6         Assert.Equal(4, Add(2, 2));
```

¹ 《C# 高级编程》（第 7 版）18.6.4 节

```

7   }
8 }
```

有时在编写了 xUnit 的代码之后在单元测试管理器中依然无法看到单元测试用例，可安装 xunit.runner.visualstudio 插件，即可看到测试用例。引用 xunit.execution.desktop.dll 之后，Visual Studio 可以自动发现测试并运行。有时引用了 xunit.execution.desktop.dll 之后也未出现测试，那么可以重新生成一下项目进行尝试。调试测试用例如图26.19所示：

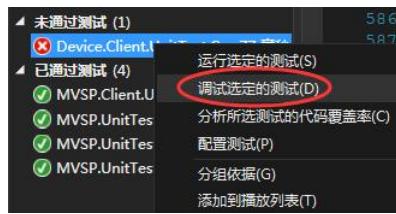


图 26.19: 调试选定测试

26.4.2 常用功能

更改测试用例名称：

```
[ Fact(DisplayName = "Max 函数测试")]
```

跳过测试用例：

```
[ Fact(Skip = "重构未完成")]
```

分组：

```
[ Trait("Group", "Category")]
```

26.5 MiniProfiler

MVC MiniProfiler 是 Stack Overflow 团队设计的一款对 ASP.NET MVC 的性能分析的小程序。可以对一个页面本身，及该页面通过直接引用、Ajax、Iframe 形式访问的其它页面进行监控，监控内容包括数据库内容，并可以显示数据库访问的 SQL（支持 EF、EF CodeFirst 等）。并且以很友好的方式展现在页面上。该 Profiler 的一个特别有用的功能是它与数据库框架的集成。除了.NET 原生的 DbConnection 类，profiler 还内置了对实体框架（Entity Framework）以及 LINQ to SQL 的支持。任何执行的 Step 都会包括当时查询的次数和所花费的时间。为了检测常见的错误，如 N+1 反模式，profiler 将检测仅有参数值存在差异的多个查询。MiniProfiler 是以 Apache License V2.0 协议发布的。

26.5.1 安装

```
1 #安装 Miniprofiler MVC 组件
```

```

2 Install-Package MiniProfiler.Mvc4
3 #安装 Miniprofiler EF 组件
4 Install-Package MiniProfiler.EF

```

26.5.2 使用

Step 1 配置 Miniprofiler 启动和停止，开始请求时 Miniprofiler 启动，结束请求时 Miniprofiler 结束，启动站点时初始化 Miniprofiler，如下代码片段所示。

```

1 protected void Application_Start()
2 {
3     MiniProfilerEF.Initialize();
4 }
5
6 void Application_BeginRequest()
7 {
8     if (Request.IsLocal)
9     {
10         MiniProfiler.Start();
11     }
12 }
13
14 void Application_EndRequest()
15 {
16     MiniProfiler.Stop();
17 }

```

Step 2 在 MVC 页面中初始化，将如下代码添加到母板页中。

```

1 @StackExchange.Profiling.MiniProfiler.RenderIncludes()

```

最终的效果如图26.20所示。

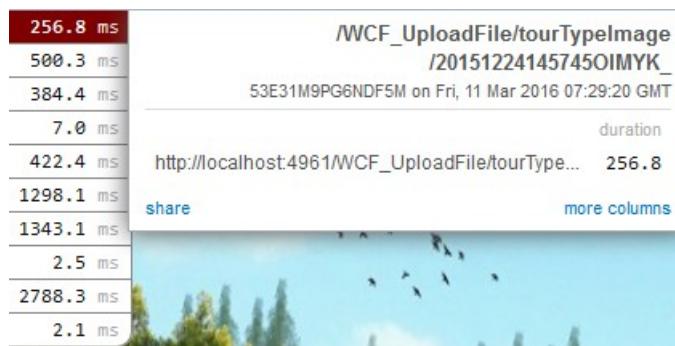


图 26.20: Miniprofiler 效果

26.6 dotTrace

一款性能测试工具，能够记录程序执行过程中各个方法的调用情况及所花时间等。
https://www.jetbrains.com/help/profiler/10.0/Introduction.html?origin=old_help

26.6.1 dotTrace 分析 ASP.NET MVC 程序性能

安装 dotTrace 完成后，从开始菜单中启动 dotTrace(run dotTrace as a standalone application)。Profile Application 选择 IIS，右侧 Open Url 填写网站本机访问的 Url。Profier Options 选择 TimeLine，接下来 dotTrace 会启动网站并进行跟踪。

26.7 CLR Profiler

CLR Profiler 的下载地址：<https://www.microsoft.com/en-us/download/details.aspx?id=16273>。The CLR Profiler allows developers to see the allocation profile of their managed applications.

26.8 JustDecompile

26.9 DebugView.exe

调试采取实时调试模式，当程序运行时，将实时信息输出到调试工具中，调试工具采用 DebugView²进行查看。程序中引入命名空间 using System.Diagnostics; 采用诸如 Debug.WriteLine("这是调试信息"); 形式输出调试信息。

26.10 Selenium

26.11 JMeter

26.12 PerfView

PerfView 能够收集 Windows 事件跟踪 (ETW) 数据来追踪程序的调用流向，这些程序通过调用哪个函数识别频率。除了配置程序性能数据 (Perfmon、PAL 和 Xperf 等工具不能轻松

²DebugView is an application that lets you monitor debug output on your local system, or any computer on the network that you can reach via TCP/IP. It is capable of displaying both kernel-mode and Win32 debug output, so you don't need a debugger to catch the debug output your applications or device drivers generate, nor do you need to modify your applications or drivers to use non-standard debug output APIs.

完成), PerfView 还能分析程序内存堆来帮助确定内存的运用是否高效。它还有一个 Diff 功能, 可以让你确定跟踪间的任意差别来帮助你认出所有逆行。最后, 该工具还有一个 Dump 功能可以生成一个程序内存转储。PerfView is an application designed to simplify the collection and analysis of performance data. PerfView 是一款免费且性能强大的工具, 他主要关注影响性能的一些深层次的问题 (磁盘 I/O, GC 事件, 内存)。我们能够抓取性能相关的 Event Tracing for Windows(ETW) 事件并能以应用程序, 进程, 堆栈, 线程的尺度查看这些信息。PerfView 能够展示应用程序分配了多少, 以及分配了何种内存以及应用程序中的函数以及调用堆栈对内存分配的贡献。

26.13 SoapUI(Apahce Licence 2.0)

SoapUI is a free and open source cross-platform Functional Testing solution. With an easy-to-use graphical interface, and enterprise-class features, SoapUI allows you to easily and rapidly create and execute automated functional, regression, compliance, and load tests. In a single test environment, SoapUI provides complete test coverage and supports all the standard protocols and technologies. There are simply no limits to what you can do with your tests. Meet SoapUI, the world's most complete testing tool!

26.14 Wireshark(GNU General Public License)

为了安全考虑, wireshark 只能查看封包, 而不能修改封包的内容, 或者发送封包。端口过滤。如过滤 80 端口, 在 Filter 中输入, `tcp.port==80`, 这条规则是把源端口和目的端口为 80 的都过滤出来。使用 `tcp.dstport==80` 只过滤目的端口为 80 的, `tcp.srcport==80` 只过滤源端口为 80 的包;

26.14.1 常用过滤语句

Wireshark 提供了两种过滤器: 捕获过滤器: 在抓包之前就设定好过滤条件, 然后只抓取符合条件的数据包。显示过滤器: 在已捕获的数据包集合中设置过滤条件, 隐藏不想显示的数据包, 只显示符合条件的数据包。使用捕获过滤器的主要原因就是性能。如果你知道并不需要分析某个类型的流量, 那么可以简单地使用捕获过滤器过滤掉它, 从而节省那些会被用来捕获这些数据包的处理器资源。当处理大量数据的时候, 使用捕获过滤器是相当好用的。在 Capture->Options->Capture filter for selected interface。

```

1 #过滤端口 6002、6003 的数据, IP 为 12.26.32.14
2 tcp port 6003 or tcp port 6002 or host 12.26.32.14

```

捕获过滤器应用于 Winpcap, 并使用 Berkeley Packet Filter (BPF) 语法, 其语法规则如下: 协议方向类型数据过滤目标 IP 语句如下:

```

1 #源端口, 目标端口 6002
2 ip.src == 192.168.24.79 && tcp.dstport == 6002
3
4 ip.dst == 192.168.1.222 or ip.dst == 183.230.47.195
5
6 #过滤出终端交互信息
7 tcp.port == 6002

```

过滤协议直接输入协议名称即可。如果想看关于 HTTP 协议的数据包，使用如下过滤语句：

```
1 http || tcp.port == 80
```

了解包列表不同协议的含义，在 View 菜单的 Coloring Rules(倒数第三个)，可以看到对应的颜色规则。

TCP 数据过滤

根据时间过滤语句如下所示，过滤出时间在 2016 年 6 月 29 日 09 点之后的数据包：状态

```
1 frame.time > "June 29, 2016 09:00:01.009638000"
```

报文长度大于 16

```

1 tcp.len > 16 and tcp.segment_data contains 7e:80:01:00
2
3 #位置信息上报报文
4 data.data contains 7e:02:00
5
6 #某一设备的位置信息上报信息
7 data.data contains 7e:02:00 and data.data contains 41:31:07
8
9 #某一设备的位置信息上报信息 (包含报警信息)
10 data.data contains 7e:02:00 and data.data contains 41:31:07 and !data.data[53,4] ==
    00:00:00:00
11
12 #Wireshark data slice
13 data[0,1-2,5-6] == 7e:02:00:01:41
14
15 #过滤所有进出路线报警
16 data[1:2] == 02:00 and data[13:4]==00:10:00:00
17
18 #过滤指定设备指定时间的报警
19 data[1:2] == 02:00 and data[13:4]==00:10:00:00 and frame.time > "July 8, 2016
    10:00:01.009638000" and frame.time < "July 8, 2016 17:00:01.009638000" and
    data[8:3] == 07:18:13

```

还有一点是，如果不知道过滤语句如何写的，那么可以在报文列表中右键点击选择 Apply as Filter，会自动生成过滤语句。

TCP 根据内容过滤 根据 TCP 的内容来过滤语法如下：

```

1  tcp[0:2]==3e:88
2
3  #Data 层满足第一位数据为 3e, 第 2 到 3 位数据为 88:90 的所有数据包
4  data[0,2-3]==3e:88:90

```

以上语句的含义为过滤 tcp 包从 0 开始的 2 个字节为 3e88 的数据包，3e 与 88 之间的冒号可以省略。

26.14.2 各层报文理解

Wireshark 捕获的报文与网络 7 层协议的报文关系如图26.21所示。从上到下依次为物理层 (Frame)、数据链路层 (Ethernet II)、网络层 (Internet Protocol Version 4)、传输层 (Transmission Control Protocol)、应用层。在 Wireshark 下方显示的是所有层的封包数据，当在列表中选择封包的不同层时，下方相应的数据包会相应的高亮显示。当在列表中选择对应层数据包详细信息 (如端口) 时，底部相应的端口数据包会高亮显示。

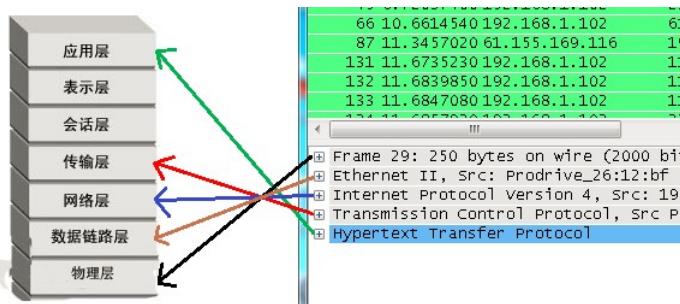


图 26.21: 网络 7 层模型和封包对应关系

数据链路层 (Data Link Layer) 传输有地址的帧以及错误检测功能 SLIP, CSLIP, PPP, ARP, RARP, MTU, 数据链路层是对物理层传输原始比特流的功能的加强，将物理层提供的可能出错的物理连接改造成为逻辑上无差错的数据链路，使之对网络层表现为一无差错的线路。如果您想用尽量少的词来记住数据链路层，那就是：“帧和介质访问控制”。数据链路层对应 Wireshark 捕获的 Ethernet II, Ethernet II Header 的结构如图26.22所示：

网络层 (Network Layer) 网络层的任务就是选择合适的网间路由和交换结点，确保数据及时传送。网络层将数据链路层提供的帧组成数据包，包中封装有网络层包头，其中含有逻辑地址信息- -源站点和目的站点地址的网络地址。如果你在谈论一个 IP 地址，那么你是在处理第 3 层的问题，这是“数据包”问题，而不是第 2 层的“帧”。IP 是第 3 层问题的一部分，此外还有一些路由协议和地址解析协议 (ARP)。有关路由的一切事情都在这第 3 层处理。地址解析和路由是 3 层的重要目的。网络层还可以实现拥塞控制、网际互连等功能。在这一层，数据的单位称为数据包 (packet)。为数据包选择路由 IP, ICMP, RIP, OSPF, BGP, IGMP，网络层对应 IP 协议，Wireshark 捕获的封包结构如图26.23所示：

Ethernet II Header				
Destination Mac Address	Source Mac Address	Type	Data	CRC Checksum
Ethernet II, Src: F5Networ_d1:96:c4 (00:01:d7:d1:96:c4), Dst: Cisco_23:a9:80 (00:12:00:23:a9:80)				
Destination: Cisco_23:a9:80 (00:12:00:23:a9:80)				
Address: Cisco_23:a9:80 (00:12:00:23:a9:80)				
....0..... = IG bit: Individual address (unicast)				
....0..... = LG bit: Globally unique address (factory default)				
Source: F5Networ_d1:96:c4 (00:01:d7:d1:96:c4)				
Address: F5Networ_d1:96:c4 (00:01:d7:d1:96:c4)				
....0..... = IG bit: Individual address (unicast)				
....0..... = LG bit: Globally unique address (factory default)				
Type: IP (0x0800)				

图 26.22: Ethernet 头

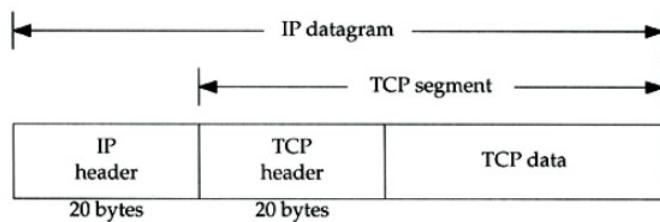


图 26.23: 网络 7 层模型和封包对应关系

传输层 (Transport Layer) 第 4 层的数据单元也称作数据包 (packets)。但是，当你谈论 TCP 等具体的协议时又有特殊的叫法，TCP 的数据单元称为段 (segments) 而 UDP 协议的数据单元称为“数据报 (datagrams)”。这个层负责获取全部信息，因此，它必须跟踪数据单元碎片、乱序到达的数据包和其它在传输过程中可能发生的危险。第 4 层为上层提供端到端（最终用户到最终用户）的透明的、可靠的数据传输服务。所为透明的传输是指在通信过程中传输层对上层屏蔽了通信传输系统的具体细节。传输层协议的代表包括：TCP、UDP、SPX 等。TCP Header 的结构如图26.24所示：

TCP Header 和 Wireshark 报文的对应关系如图所示：

应用层 (Application Layer) 应用层为操作系统或网络应用程序提供访问网络服务的接口。应用层协议的代表包括：Telnet、FTP、HTTP、SNMP 等。应用层的协议有许多，以 HTTP 协议为例。

26.14.3 删 除 Filter

在路径 C:/Users/Dolphin/AppData/Roaming/Wireshark 下的 preferences 文件中，找到 Filter Expressions 编辑即可。在 GUI 页面上的路径为：Edit->Preferences->Filter Expressions，如图所示。

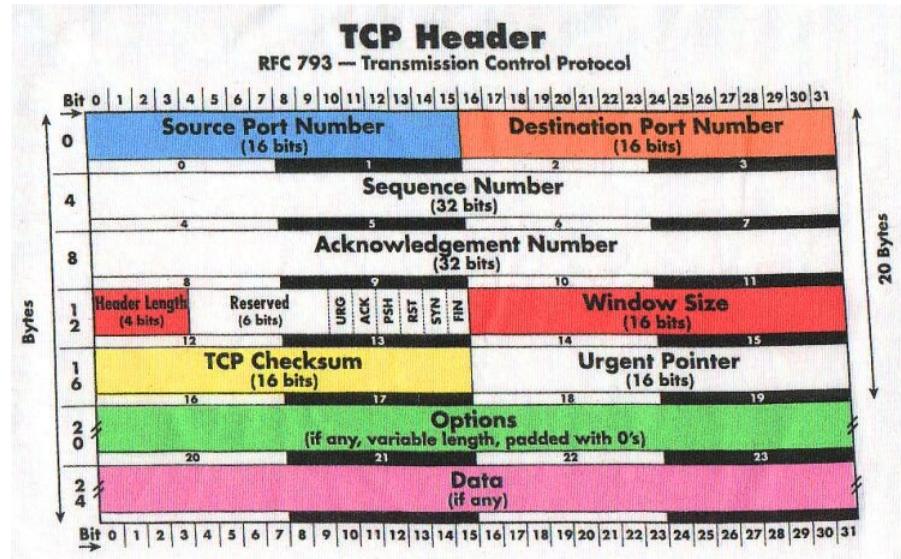


图 26.24: TCP 头

TCP Header

Source Port Number	Destination Port Number	
Sequence Number	Acknowledgement Number	
Data Offset	Flags ACK URG RST SYN etc.	Window Size
Checksum	Urgent Pointers	

Transmission Control Protocol, Src Port: 55075 (55075), Dst Port: 50100 (50100), Seq: 1381, Ack: 1, Len: 1380

Source port: 55075 (55075)
Destination port: 50100 (50100)
[Stream index: 10]
Sequence number: 1381 (relative sequence number)
(Next sequence number: 2761 (relative sequence number))
Acknowledgement number: 1 (relative ack number)
Header length: 20 bytes
Flags: 0x10 (ACK)
000..... = Reserved: Not set
...0..... = Nonce: Not set
...0..... = Congestion Window Reduced (CWR): Not set
...0..... = ECN-Echo: Not set
...0..... = Urgent: Not set
....1.... = Acknowledgement: Set
....0.... = Push: Not set
....0.... = Reset: Not set
....0.... = Syn: Not set
....0.... = Fin: Not set
Window size value: 4380
[Calculated window size: 4380]
[Window size scaling factor: 1]
Checksum: 0xfd18 [validation disabled]
[Good Checksum: False]
[Bad Checksum: False]
[SEQ/ACK analysis]
[Bytes in flight: 2760]
Data (1380 bytes)

图 26.25: TCP 头和 Wireshark 捕获对应关系

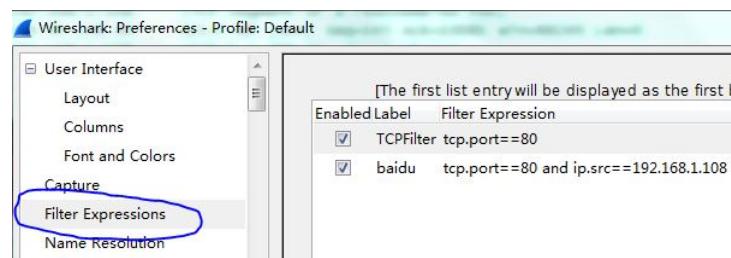


图 26.26: WireShark Filter Management

26.14.4 抓取本机包

在进行通信开发的过程中，我们往往把本机既作为客户端又作为服务器端来调试代码，使得本机自己和自己通信。但是 wireshark 此时是无法抓取到数据包的，需要通过简单的设置才可以。

1. 以管理员身份运行 cmd 2.route add 本机 ip mask 255.255.255.255 网关 ip

```
1 #打印路由表
2 route print
3 #添加路由
4 route add 192.168.24.79 mask 255.255.255.255 192.168.24.1
```

使用完毕后用 route delete 172.16.51.115 mask 255.255.255.255 172.16.1.1 删除，否则所有本机报文都经过网卡出去走一圈回来很耗性能。此时再利用 wireshark 进行抓包便可以抓到本机自己同自己的通信包，这样配置的原因是将发往本机的包发送到网关，而此时 wireshark 可以捕获到网卡驱动的报文实现抓包。但这样有一个缺点，那就是本地请求的 URL 的 IP 只能写本地的 IP 地址，不能写 localhost 或 127.0.0.1，写 localhost 或 127.0.0.1 还是抓不到包。

26.14.5 常见问题

The temporary file to which the capture would be saved ("") could not be opened:
Invalid argument 在 Capture 的 option 选项下，填写输出文件路径即可，如：C:\a.cap。

Chapter 27

辅助组件

27.1 Linux 有关工具

27.1.1 Putty

27.1.2 Vi

27.2 GMap.NET

GMap.NET is great and Powerful, Free, cross platform, open source .NET control. Enable use routing, geocoding, directions and maps from Google, Yahoo!, Bing, OpenStreetMap, ArcGIS, Pergo, SigPac, Yendux, Mappy.cz, Maps.lt, iKarte.lv, NearMap, OviMap, CloudMade, WikiMapia, MapQuest in Windows Forms & Presentation, supports caching and runs on windows mobile!

http://greatmaps.wikia.com/wiki/Main_Page

27.3 DotNetBar(Commercial)

27.3.1 Introduce

DotNetBar 是一款带有 56 个 Windows Form 控件的工具箱，使开发人员可以轻而易举地创建出专业美观的 Windows Form 应用程序用户界面，控件全部采用 C# 编写，引入了全部 Office 2007 style Ribbon 控件、Office 2003 office2010 样式、支持 windows7,Windows XP 主题等。

27.3.2 加载 DotNetBar 控件到 Visual Studio 工具箱

在 Visual Studio 工具箱中单击右键，新建 DotNetBar 分组，到 DotNetBar 安装目录下将 DevComponents.DotNetBar2.dll 文件拖动到分组下即可加载所有 DotNetBar 控件，如图27.1所示。

将 DotNetBar 控件添加到工具箱中相对来说更加直观化，重新启动 Visual Studio 后分组控件依然存在。

27.4 ildasm.exe(MSIL Disassembler)

MSIL Disassembler 的路径为：C:/Program Files (x86)/Microsoft SDKs/Windows。

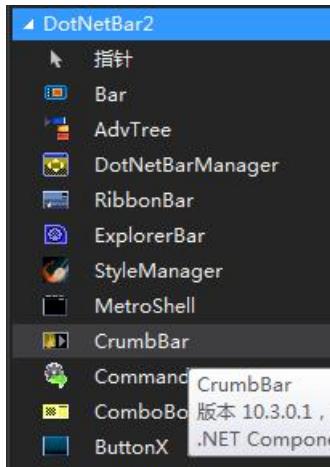


图 27.1: DotNetBar 加载到工具箱

27.5 Source Insight(Commercial)

Source Insight™ is a project-oriented program editor and code browser, with built-in analysis for C/C++, C#, and Java programs. Source Insight parses your source code and maintains its own database of symbolic information dynamically while you work, and presents useful contextual information to you automatically.

Source Insight can display reference trees, class inheritance diagrams, and call trees. Source Insight features the quickest navigation of source code and source information of any programming editor. Source Insight 3.x does not yet support UNICODE files.

27.5.1 特性 (Feature)

不用成功编译也能查看源代码的链接关系。

Call Graphs and Class Tree Diagrams The Relation Window is a Source Insight innovation that shows interesting relationships between symbols. It runs in the background and tracks what symbols you have selected. With it, you can view class hierarchies, call trees, reference trees, and more. 在 C# 代码库中看 Call Tree 如图27.2所示：

不过 class hierarchies 还不知道怎么看，或许是 C# 不支持吧。

The beauty of the Relation Window is that you don't have to do anything special. It works in the background while you work, but you can interact with it when you want to.

The Relation Window can be viewed either graphically, or in outline format. You can also have several Relation Windows open, each showing different types of information.

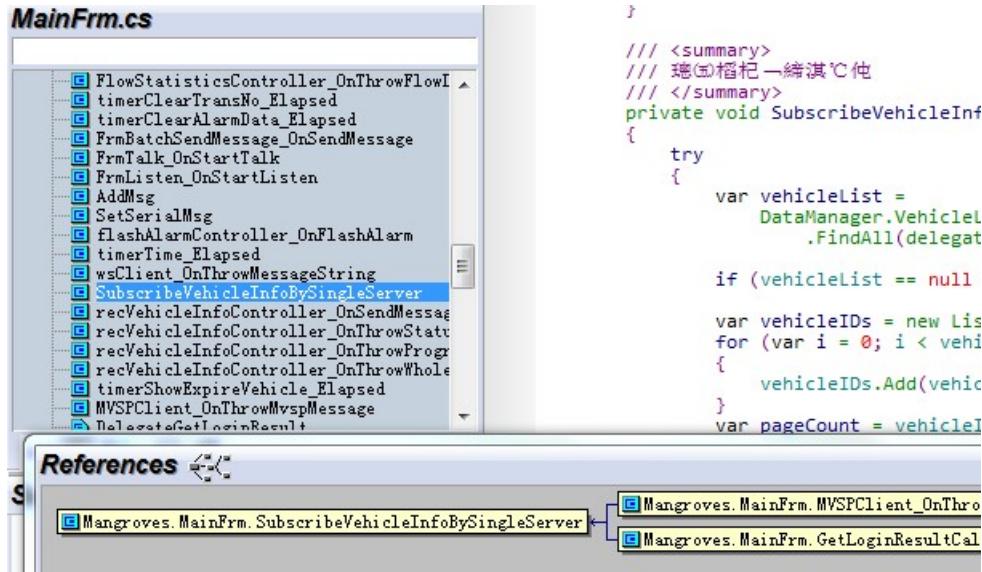


图 27.2: Call Graphs

27.6 Inno Setup

27.6.1 支持中文的安装界面

在安装完毕后制作的安装界面默认为英文，比如下一步是英文的 Next 而非中文的“下一步”。要支持中文需要找到中文的 Chinese.isl 文件，将之拷贝到安装目录下的 Languages 目录下，在生成安装包的脚本里指定 Chinese.isl 文件的位置即可，代码如下：

```

1 [Languages]
2 Name: "Chinese"; MessagesFile: "compiler:\Languages\Chinese.isl"

```

27.6.2 添加自定义界面

有时需要添加一个自定义界面展示简单的欢迎信息或者声明条款，而不是点击程序直接跳转到安装界面，创建自定义安装欢迎界面脚本如下：

```

1 [Code]
2 procedure InitializeWizard();
3 var
4     Page: TWizardPage;
5 begin
6     Page := CreateCustomPage(wpWelcome, 'Custom wizard page controls', 'TButton and
7         others');
8 end;

```

27.6.3 启用向导界面

默认情况下欢迎界面没有开启，在 Setup 代码块中添加如下脚本启动欢迎界面。

```
1 DisableWelcomePage=no
```

启动欢迎界面后效果如图27.3所示：



图 27.3: Inno Setup 启用欢迎界面

27.6.4 自动打包.NET Framework 安装程序

在开发.net 客户端时候.Net Framework 是个比较让人头疼的问题，比如一个 WPF 程序大小几百 K, 却要安装一个几十 M 的.Net Framework。但是也没办法. 这里提供两种方式，一个是将.Net Framework 打包进安装包中，一个是在线下载.Net Framework 然后安装。

27.7 Graphviz

Graphviz¹ is open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. It has important applications in networking, bioinformatics, software engineering, database and web design, machine learning, and in visual interfaces for other technical domains.

Graphviz(Graph Visualization Software 的缩写) 是一个由 AT&T 实验室启动的开源工具包，用于绘制 DOT²语言脚本描述的图形。它也提供了供其它软件使用的库。Graphviz 是一个自

¹ 下载链接: <http://www.graphviz.org/pub/graphviz/stable/windows/graphviz-2.38.msi>

² dot is a tool to generate nice-looking diagrams with a minimum of effort. It's part of GraphViz, an open

由软件，其授权为 Common Public License。其 Mac 版本曾经获得 2004 年的苹果设计奖。它的强大主要体现在“所思即所得”(WYTIWYG, what you think is what you get)，这是和 office 的“所见即所得”(WYSIWYG, what you see is what you get) 完全不同的一种方式。

在 dot 脚本里面写好图形的脚本以后，在命令行中输入以下命令生成 ps 文件。

```
1 C:\>dot -Tps a.dot -o 1.ps
```

如果需要生成 jpg 格式的图片，输入以下命令即可，Sign.dot 为源代码文件名称，Sign.jpg 为输出图片的名称。dot 的命令格式为：

```
1 dot -T[output\_filetype] [-G[option\_name]] -o [output\_filename] [input\_filename]
2
3 dot -Tjpg Sign.dot -o Sign.jpg
```

Graphviz 可以改变生成图片的质量，需要生成的图形更加清晰，参数指定 dpi 即可，编译的命令格式为：

```
1 dot -Tjpg -Gdpi=331 PhotoPrintWorkflow.dot -o ppw.jpg
```

27.7.1 中文乱码

Graphviz 默认设置下是不支持中文，如果在 dot 文件中直接写中文，会显示成乱码。解决 Graphviz 中文乱码的问题就是指定 fontname，如下代码片段所示：

```
1 digraph G{
2     size="8,8"
3     edge[fontname="FangSong"];
4     node[shape=box,fontname="FangSong",size="20,20"]
5     原始内容->摘要[label="摘要算法"];
6     摘要->数字签名[label="私钥加密"];
7     数字签名->请求数据[label="+原始内容"];
8     请求数据->发送;
9     发送->接收;
10    接收->拆分出数字签名;
11    接收->拆分出原始内容;
12    拆分出数字签名->解密出摘要[label="公钥解密"];
13    拆分出原始内容->生成摘要[label="摘要算法"];
14    解密出摘要->摘要比较;
15    生成摘要->摘要比较;
16    摘要比较->一致;
17    摘要比较->不一致;
18    一致->合法[style=bold];//连接线加粗
19    edge[color=red];//连接线的颜色为红色
```

```

20    不一致->非法;
21 }

```

Windows 系统中文字体的英文名如下所示：

新细明体	PMingLiU
细明体	MingLiU
标楷体	DFKai-SB
黑体	SimHei
宋体	SimSun
新宋体	NSimSun
仿宋	FangSong
楷体	KaiTi
仿宋 _GB2312	FangSong_GB2312
楷体 _GB2312	KaiTi_GB2312
微软正黑体	Microsoft JhengHei
微软雅黑体	Microsoft YaHei

27.7.2 箭头指向节点位置

可以指定 Node 节点的指向位置。一个小的例子写法如下：

```

1 digraph G{
2     size="8,8";
3     edge[fontname="FangSong"];
4     node[shape="Mrecord",fontname="FangSong",size="20,20",fontsize=12,color="skyblue",
5         style="filled"]
6     调取查询方法->读取app_config上次查询成功时间;
7     读取app_config上次查询成功时间->执行查询;
8     执行查询:sw->下一次查询:w[label="查询失败"];
9     执行查询->保存本次查询时间到配置文件[label="查询成功"];
10    保存本次查询时间到配置文件->获取Json;
11    获取Json->下一次查询[label="无新记录"];
12    获取Json->转换为Model[label="有新记录"];
13    转换为Model->订单是否存在;
14    订单是否存在->忽略订单[label="存在"];
15    订单是否存在->写入RR数据库shop_order表[label="不存在"];
}

```

其中 sw 指节点的左下角， w 至节点的正左边，另外还有：“n”，“ne”，“e”，“se”，“nw”，分别代表不同的含义。最终的效果如图27.4所示：

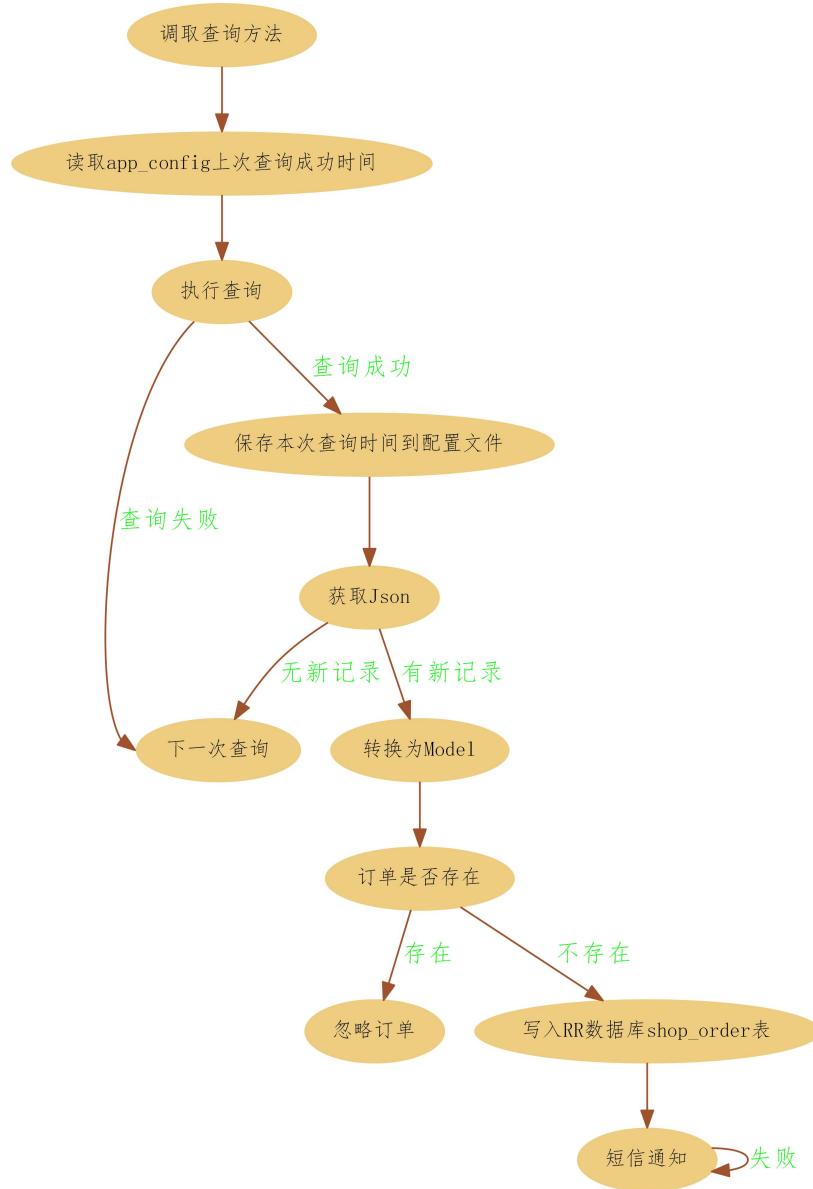


图 27.4: 有赞系统订单导入逻辑

27.8 gVim

27.8.1 乱码

用 gVim 输入的汉字，系统无法识别，在配置文件中设置文件的编码方式为 utf-8 即可。

```
1 set fileencodings=utf-8,chinese,latin-1
```

27.9 Atom

27.9.1 显示中文

27.10 Tex

27.10.1 Install

打开终端，执行下述命令安装 TexLive 和常用的一些 Latex 宏包（可以根据自己的需要增改）：

```
1 #完整安装
2 yum install texlive-scheme-full
3
4 #ubuntu
5 sudo apt-get install texlive texlive-math-extra texlive-latex-extra texlive-latex-
   recommended texlive-pictures texlive-science texlive-bibtex-extra latex-beamer
```

如果硬盘充裕的话，直接完整安装：

```
1 sudo apt-get install texlive-full latex-beamer
```

安装完后，就可以安装 CJK 的相关软件包了，如果只需要获得中文支持，那么执行：

```
1 sudo apt-get install latex-cjk-chinese ttf-arithic-* hbf-*
```

插入空白页

一种插入空白页方式采用如下代码，不过缺点是空白页上还是有页码。

```
1 \clearpage
2 \mbox{}
3 \clearpage
```

27.10.2 Tex 中注释代码

首先，创建两个变量，一个变量用于显示在代码中标记时的序号；一个用于在显示时使用的序号：

```

1 \newcounter{coderemarks}    \\ 创建变量
2 \setcounter{coderemarks}{1} \\ 设置变量初值为1
3 \newcounter{codevar}        \\ 创建变量
4 \setcounter{codevar}{1}     \\ 设置变量初值为1

```

第二步：绘制圆形标记：文本底色为白色，圆圈填充色为黑色

```

1 \newcommand{\circlemark}[1]{%
2 \tikz\node[text=white,font=\sffamily\bfseries,inner sep=0.2mm,draw,circle,fill=black]{#1};}

```

第三步：定义 makeremark 命令，调用上一步的 circlemark 命令，将变量 coderemarks 作为文本，填充至标记中，并添加注释，但并不是在这里就显示注释，而是在调用 showremarks 命令时再显示。最后递增 coderemarks 变量。

```

1 \newcommand{\makeremark}[1]{%
2 \circlemark{\arabic{coderemarks}}%
3 \global\expandafter\def\csname codebox\the\value{coderemarks}\endcsname{#1}%
4 \stepcounter{coderemarks}}

```

第四步：定义 showremarks 命令。以一个列表的形式，将代码中所有的标记列出来，并将注释也显示出来。最后，将 coderemarks 和 codevalue 都置为 1，也就是初值。这里 codevalue 的作用是一次显示出代码中用 makeremark 命令标记出的注记。详细请看下面的代码。

```

1 \newcommand{\showremarks}{%
2 \begin{list}{\circlemark{\arabic{codevar}}}%
3 {}%
4 \whiledo{\value{codevar} < \value{coderemarks}}{%
5 \item \expandafter\csname codebox\the\value{codevar}\endcsname%
6 \stepcounter{codevar}}%
7 \end{list}%
8 \setcounter{coderemarks}{1}%
9 \setcounter{codevar}{1}%
10 }

```

第五步：测试代码。在代码中，使用逃逸符号将 makeremark 命令及其中的文本包裹起来。并在 listing 环境结束后用/showremarks 命令将注记列出来。本例中，使用 lstsetescapeinside=“命令重新定义了逃逸符号为“，也就是主键盘区最左上角的那个按键。如下：

```

1 \begin{lstlisting-internal}
2 #include makeremark(命令) 导入库文件
3 makeremark 注释，位于 lstinline|| 之间;
4 int main() { makeremark 主函数，所有 C 语言可执行程序以 main 开始

```

```

5   printf("Hello, world") makeremark 调用库函数打印字符串到屏幕。
6 }
7 \end{lstlisting}
8 \showremarks

```

27.11 Rsync

rsync³, remote synchronize 顾名思意就知道它是一款实现远程同步功能的软件, 它在同步文件的同时, 可以保持原来文件的权限、时间、软硬链接等附加信息。rsync 是用 “rsync 算法” 提供了一个客户机和远程文件服务器的文件同步的快速方法, 而且可以通过 ssh 方式来传输文件, 这样其保密性也非常好, 另外它还是免费的软件。

rsync 包括如下的一些特性:

能更新整个目录和树和文件系统;

有选择性的保持符号链链、硬链接、文件属于、权限、设备以及时间等;

对于安装来说, 无任何特殊权限要求;

对于多个文件来说, 内部流水线减少文件等待的延时;

能用 rsh、ssh 或直接端口做为传输入端口;

支持匿名 rsync 同步文件, 是理想的镜像工具;

如果是 windows 的话, 需要 windows 版本 cwrsync

27.12 Docbook

27.12.1 Docbook 安装

安装路径为: d:/Docbook (可自定义路径)

Step 1: 下载 Docbook XSL 样式表 Docbook 是“内容与格式分离”的, 我们写 Docbook 文档时只关注文档的内容, 而 Docbook 如何转换成其它文档类型, 则由 Docbook XSL⁴转换样式表来定义。我们最常用的是转换到 HTML 格式的样式表, 以及转换到 FO 格式的样式表。在开源托管平台 sourceforge 下载 XSL 样式表, 链接地址为: <http://sourceforge.net/projects/docbook/files/>。下载后解压到 d:/docbook/docbook-xsl-ns-1.76.1 目录, 设置 html 输出编码, 设置为 utf-8。docbook-xsl-ns 的缺省输出编码是 ISO-8859-1, 对于中文字符, 输出编码设置为 utf-8, 否则输出的文件会显示为乱码。d:/docbook/docbook-xsl-ns-1.76.1/html/docbook.xsl

³<https://rsync.samba.org/>

⁴可扩展样式表语言: Extensible Stylesheet Language

Step 2: 下载 windows 版本 xsltproc 转换程序 下载 XML 语言转换工具。我喜欢用的转换工具是 xsltproc⁵，这是一个由 C 语言写成的 XML 转换工具，它的特点是转换速度很快，并且同时支持 Windows 和 Linux。除了这个转换工具外，还有很多 Java 写的转换工具，例如 Saxon, Xalan。下载链接：<ftp://ftp.zlatkovic.com/libxml/>。需要的几个包：

- iconv-1.9.2.win32.zip – 编码转换工具
- zlib-1.2.3.win32.zip – 压缩工具
- libxslt-1.1.24.win32.zip – xsl 和 exsl 转换工具，xsltproc 程序也在此包中
- libxml2-2.7.3.win32.zip – xml 解析工具和处理工具，同时提供了验证工具 xmllint 和 xmlcatalog

下载后，把这些包都解压到 d:/docbook/xsltproc 目录，可以看到在 d:/docbook/xsltproc/下多了三个目录：bin、include 和 lib。现在可以把 docbook 文档转换成 html 格式了。如果需要把 docbook 文档转换成 pdf 格式，继续安装把 xsl-fo 格式转换为 pdf 格式工具，我们使用 Apache FOP⁶。

Step 3: 安装 Apache FOP 到 Apache 下载 Fop 包，<http://www.apache.org/dyn/closer.cgi/xmlgraphics/fop> 下载后解压文件到 d:/docbook/fop-1.0 注意：不要下载未编译的 source 文件夹下的文件，正确的为已经编译好的 binaries 文件夹下的 zip 包，如我下载的 fop-1.0-bin.zip

3.1、安装 java 虚拟机线上安装：http://www.java.com/zh_CN/download/ 离线包安装：<http://x.139.xdowns.com/096/jre-6u4-windows-i586-p.rar> 默认安装路径：c:/Program Files/Java

3.2、提取中文字体把支持中文的字體信息從 windows 自帶系統中，提取到 fonts/simsun.xml 和 fonts/simhei.xml 文件中。

3.3、配置 fop.xconf 配置 d:/docbook/fop-1.0/conf/fop.xconf，使 apache fop 能找到提取的两种字体。详细见 d:/docbook/fop-1.0/conf/fop.xconf docbook 文档转换 html 及 pdf 环境已经配置好了，下面安装微软的制作 chm 文件的工具 HTML Help Workshop，用它把经 xsltproc 转换的 hhp 文件生成 chm。

Step 4: 下载 HTML Help Workshop <http://msdn.microsoft.com/en-us/library/ms669985%28v=VS.85%29.aspx> 下载安装好后，默认安装路径在 c:/Program Files/HTML Help Workshop 至此，编译环境配置部分已经完成，下面简单的介绍如何编写 docbook 文档和输出 html 及 pdf 文档。

⁵XSL Transport Processor

⁶Apache™ FOP (Formatting Objects Processor) is a print formatter driven by XSL formatting objects (XSL-FO) and an output independent formatter. It is a Java application that reads a formatting object (FO) tree and renders the resulting pages to a specified output. Output formats currently supported include PDF, PS, PCL, AFP, XML (area tree representation), Print, AWT and PNG, and to a lesser extent, RTF and TXT. The primary output target is PDF.

27.12.2 编写 Docbook 文档

下面是一个简单的 Docbook 5.0 文档，把这段内容保存一下，例如，存为文件 docbook.xml。

```

1 <?xml version='1.0' encoding="utf-8"?>
2 <article xmlns="http://docbook.org/ns/docbook" version="5.0" xml:lang="zh-CN"
3   xmlns:xlink='http://www.w3.org/1999/xlink'>
4   <articleinfo>
5     <title>我的第一篇Docbook 5.0文档</title>
6     <author>
7       <firstname>Easwy</firstname>
8
9       <surname>Yang</surname>
10    </author>
11  </articleinfo>
12
13  <section>
14    <title>文档介绍</title>
15
16    <para>
17      这是我的第一篇Docbook 5.0文档<link xlink:href='http://cnblogs.com/jiangxiaoqiang
18        /'>jxq的博客</link>。
19    </para>
20  </section>
21 </article>

```

27.12.3 将文档转换成 HTML 格式

在转换之前，要先把输出编码设置为 UTF-8。docbook-xsl-ns 的缺省输出编码是 ISO-8859-1，但对于中文字符，我们应该把输出编码设置为 UTF-8，否则输出的文件会显示为乱码。

更改输出编码很简单，你可以直接修改你的 C:/docbook/docbook-xsl-ns-1.74.3/html/docbook.xsl 文件，在文件中找到

```

1 <xsl:output method="html"
2   encoding="ISO-8859-1"
3   indent="no"/>

```

改为：

```

1 <xsl:output method="html"
2   encoding="UTF-8"
3   indent="no"/>

```

将 Docbook 文档转换成 HTML 格式，使用下面的命令：

```

1 xsltproc.exe -o doc.html D:\Docbook\docbook-xsl-ns-1.78.1\html\docbook.xsl docbook.
2   .xml

```

使用命令时会弹出缺少 libwinpthread-1.dll 文件，是因为 xsltproc.exe 程序是采用 MinGW 进行编译，需要将 mingwrt-4.8.1-win32-x86_64.7z 包解压到 bin 目录下或者添加到 Windows 环境变量中。

27.13 PowerDesigner(Commercial)

27.13.1 显示注释

右键- >Properties- >Columns- >Customize Columns and Filter(或直接用快捷键 Ctrl+U)->Comment(前面打勾)- >OK。在表的属性页上面的确不好寻找自定义列在哪里。如图27.5所示。

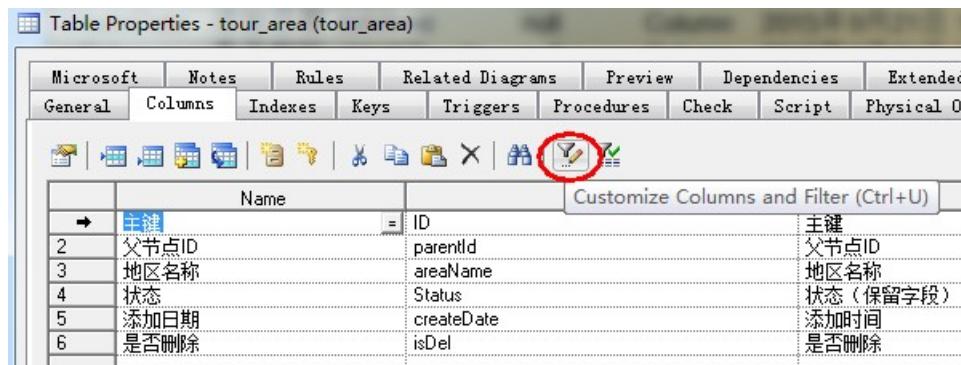


图 27.5: 自定义表的显示列

在 E-R 图的显示界面自定义显示列更加难找(不是一般的难找)。在工具 (Tools)- > 显示参数选择 (Display Preference)- >Table- >Advance- >Columns- >Search。

27.13.2 设置主键

必须是物理模型才能设置自动增长，概念模型是不能设置的。创建了概念模型以后自动更新就行了 PowerDesigner 设置主键自增方法。选中主键字段，点击进入属性设置框，勾选”Identity”，勾选后在 SQL Preview 中可以看到 identity(1,1) 证明设置标志列生效。

27.13.3 设置默认值

在 PowerDesigner 中设置列的默认值在表属性 (Table Properties) -> 列属性 (Column Properties)。-> 标准检查 (Standard Checks) 下的 Default 字段中，如图27.8所示。

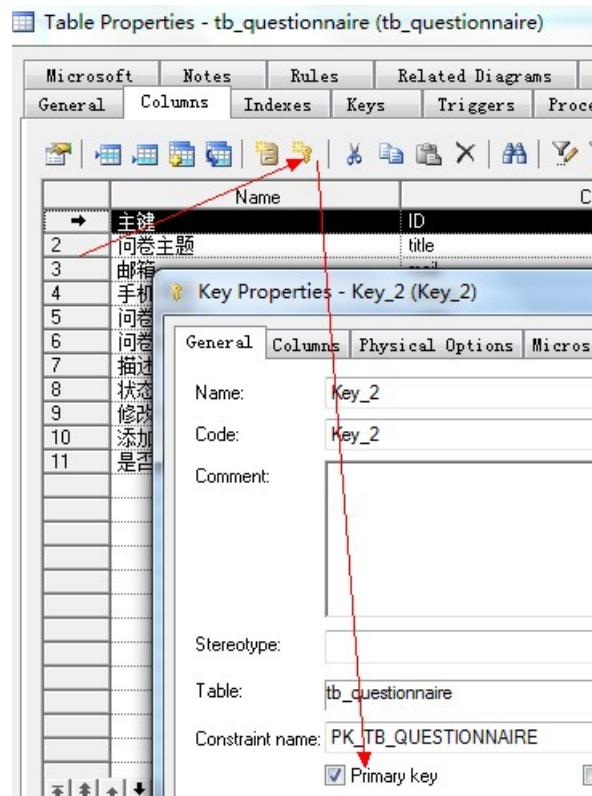


图 27.6: 设置主键

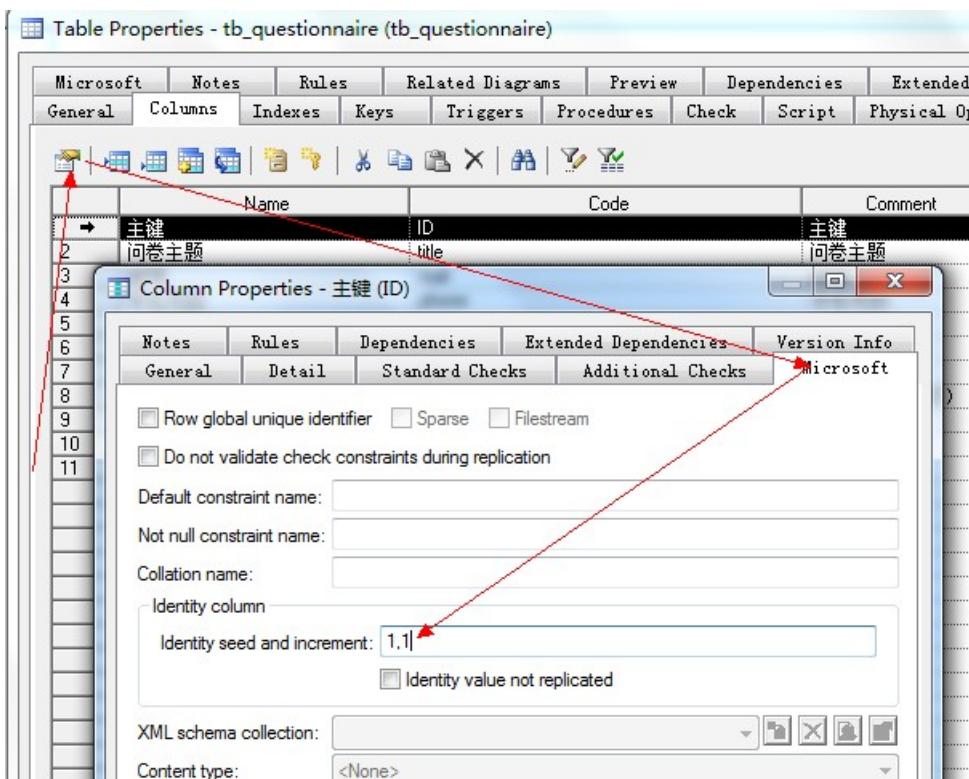


图 27.7: 设置自增列

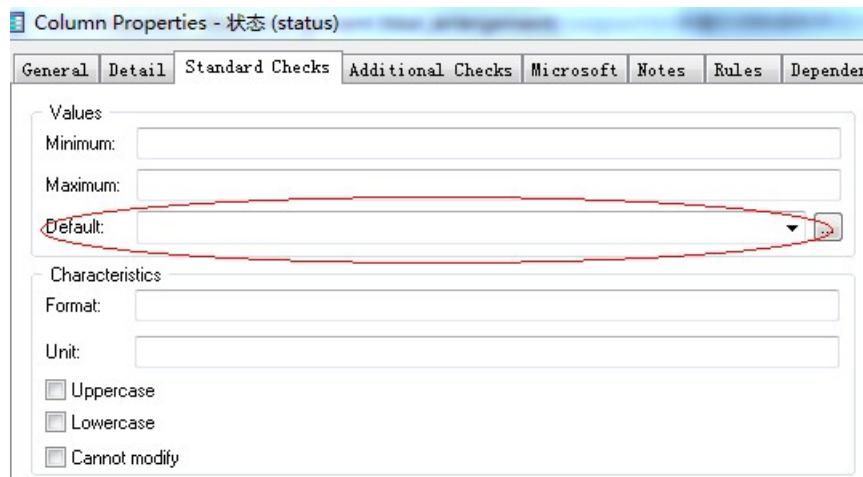


图 27.8: 设置列的默认值

27.14 WebStorm

27.14.1 配置 NodeJS

NodeJS 的配置在 Run->Edit Configurations->

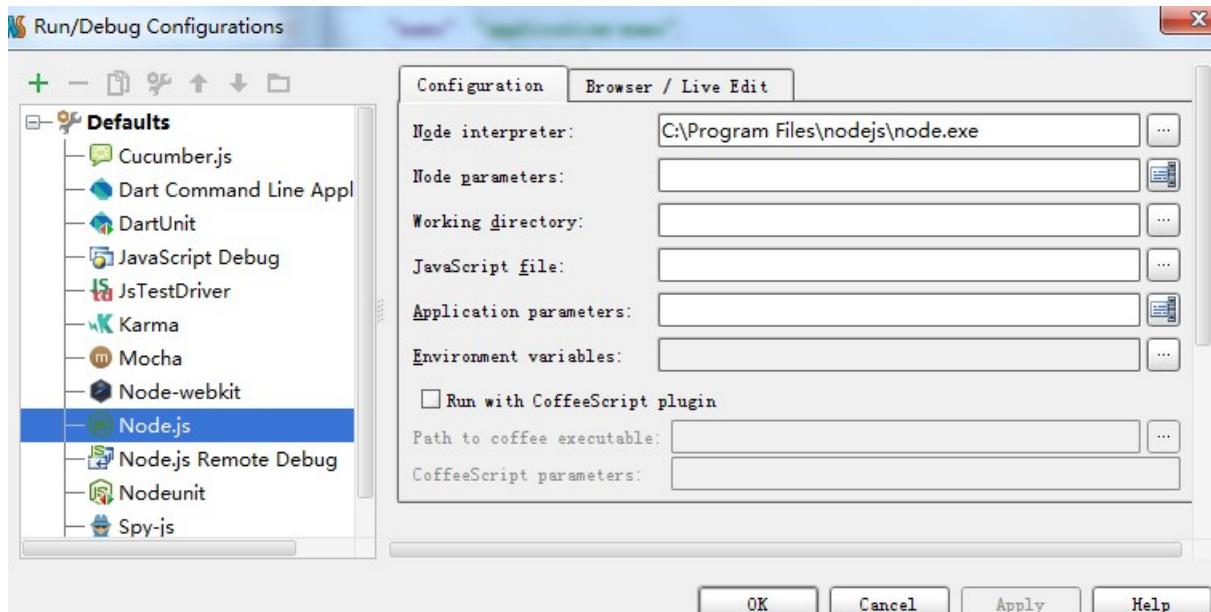


图 27.9: WebStorm 中 NodeJS 的配置

27.15 curl(CommandLine Uniform Resource Locator)

curl 是利用 URL 语法在命令行方式下工作的开源文件传输工具。它被广泛应用于 Unix、多种 Linux 发行版中，并且有 DOS 和 Win32、Win64 下的移植版本。curl is an open source

command line tool and library for transferring data with URL syntax, supporting DICT, FILE, FTP, FTPS, Gopher, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMTP, SMTPS, Telnet and TFTP. curl supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, proxies, HTTP/2, cookies, user+password authentication (Basic, Plain, Digest, CRAM-MD5, NTLM, Negotiate and Kerberos), file transfer resume, proxy tunneling and more.

27.15.1 curl for HTTP

```
1 curl -X POST -v https://api.jpush.cn/v3/push -H "Content-Type: application/json" -u
      "7d431e42dfa6a6d693ac2d04:5e987ac6d2e04d95a9d8f0d1" -d '{"platform":"all",
      "audience":"all","notification":{"alert":"Hi, JPush!"}}'
```

curl Get 经过验证的 Restful 接口数据。

```
1 curl --request GET "http://129.29.81.128:8001/active/Lottery/GetWinningList" -d "
      id=45&hour=24" --header "auth:appe10adc3949ba59abbe56e057f20f883e"
```

27.16 My Generation

MyGeneration 是一个功能很强大的代码生成工具。通过编写包含各种类型脚本(C#,VB.Net,JScript,VBScript)的模板，通过数据库的表内容，生成你需要的各种代码。你可以用它来生成 ORM 的实体类，存储过程，SQL 语句等等。我甚至用它来生成 Asp.Net 的页面。

```
1 Provider=SQLOLEDB.1;Persist Security Info=False;User ID=sa;Password=12345;Data
      Source=129.19.11.121,5002
```

27.17 Brackets(MIT License)

27.17.1 简介

Brackets 的特点是实时预览，快速编辑，跨平台，可扩展，开源，Brackets 提供网页即时预览（Live Preview）功能。使用该功能时，Brackets 调用 Chrome 浏览器打开当前页面，此后修改 html、css、javascript 并保存后，所修改的内容会即时响应到浏览器中的页面，无须手动刷新页面。这是 Brackets 最大的一个亮点，有两个显示器的 coder 有福了，可以分屏显示 Brackets 和 chrome，即时修改即时预览，无需切换编辑器/浏览器和刷新页面。目前即时预览功能的一些限制：它仅适用于 Chrome 浏览器为目标浏览器，你必须安装 Chrome。它依赖于在 Chrome 浏览器中的远程调试功能，这是一个命令行标志启用。在 Mac 上，如果你已经在使用 Chrome 浏览器，这时启动“即时预览”，Brackets 将询问你是否要重新启动 Chrome 浏

览器启用远程调试功能。只能同时对一个 HTML 文件进行预览 - 如果切换另一个 HTML 文件, Brackets 将关闭原来的预览。Remote Debugging 使得 Chrome 成为一个 WebServer, 执行命令`remote-debugging-port=1337`, 同时再开一个 Chrome 访问 `localhost:1337`, 就可以用一个 Chrome 当 Host, 一个 Chrome 当 Client, 在调试一些移动 Web 的时候非常实用。

27.17.2 常见问题

当无法在 Google Chrome 中进行即时预览 (Live Preview) 时, 尝试将 Google 浏览器设置为默认浏览器。有时可以即时预览但是需要文件保存之后才同步, 尝试将 html 文件部署到 IIS 后, 文件的修改可即时反映到浏览器界面。HTML 乱码时, 设置页面编码模式为 utf-8 即可。

```
1 <meta charset="utf-8"/>
```

<http://tid.tenpay.com/>

27.18 FileZilla

27.18.1 设置默认目录

设置默认目录后当打开某个 FTP 连接时会自动定位到本地预先设置的默认目录, 省去了手动寻找和切换目录的麻烦, 具体的设置在站点管理器中选择相应的远程站点, 在右侧的 tab 的高级选项卡中设置“默认本地目录”即可。

FileZilla 服务端设置时 IP 地址只需填写 127.0.0.1 即可, 不需设置服务器的实际 IP 地址 (实际 IP 未尝试), 注意不要有其他 FTP 服务端运行, 会造成冲突。

27.19 Yeoman

27.19.1 简介

Yeoman 是 Google 的团队和外部贡献者团队合作开发的, 他的目标是通过 Grunt (一个用于开发任务自动化的命令行工具) 和 Bower (一个 HTML、CSS、Javascript 和图片等前端资源的包管理器) 的包装为开发者创建一个易用的工作流。Yeoman 的目的不仅是要为新项目建立工作流, 同时还是为了解决前端开发所面临的诸多严重问题, 例如零散的依赖关系。Yeoman 主要有三部分组成: yo (脚手架工具)、grunt (构建工具)、bower (包管理器)。这三个工具是分别独立开发的, 但是需要配合使用, 来实现我们高效的工作流模式。

27.20 BrowserSync (MIT Licence)

27.20.1 简介

Browsersync⁷能让浏览器实时、快速响应您的文件更改 (html、js、css、sass、less 等) 并自动刷新页面。更重要的是 Browsersync 可以同时在 PC、平板、手机等设备下进项调试。您可以想象一下：“假设您的桌子上有 pc、ipad、iphone、android 等设备，同时打开了您需要调试的页面，当您使用 browsersync 后，您的任何一次代码保存，以上的设备都会同时显示您的改动”。无论您是前端还是后端工程师，使用它将提高您 30% 的工作效率。

27.20.2 安装

```
1 npm install -g browser-sync
```

27.20.3 使用

针对单纯的静态页面，我们需要使用到 browser-sync 的-server 命令。假设我的静态页面都在 C:/wamp/www/openadmin/style/html/目录下，通过控制台进入到 C:/wamp/www/openadmin(即把改目录当初 server 的 root 目录)，敲入下面的命令：

```
1 #--files 路径是相对于运行该命令的项目（目录）
2 browser-sync start --server --files "style/html/*.html"
```

本地已经搭建了服务器环境的，输入如下命令即可。

```
1 browser-sync start --proxy "192.168.1.108:8000" --files "*.*css,*.*html"
```

--files 后面带上监听的相应文件的列表，-server 表示启动一个内置的 web 服务器，-proxy 一般用在有本地的 PHP、Nginx 部署的 web 服务器下。在 windows 控制台下输入如下命令：

```
1 browser-sync start "192.168.1.108:8000" --files "index.html"
2
3 #启动已经部署完毕的网站
4 browser-sync start --proxy "192.168.1.254:7000" --files "*.*html,*.*css"
5
6 #启动本地微型服务
7 browser-sync start --server --files "**/*.*css,**/*.*html"
```

访问页面 <http://192.168.1.108:3001/help> 可以看到设置。如图27.10所示。

3000 的项目页面，和:3001 的 BrowserSync 的 UI 界面，并且每个端口都有供本地 (localhost) 和外部 (局域网 IP) 访问的 URL。需要注意的是，要先打开自己的项目页面，再打开 Remote

⁷<http://www.browsersync.io/>

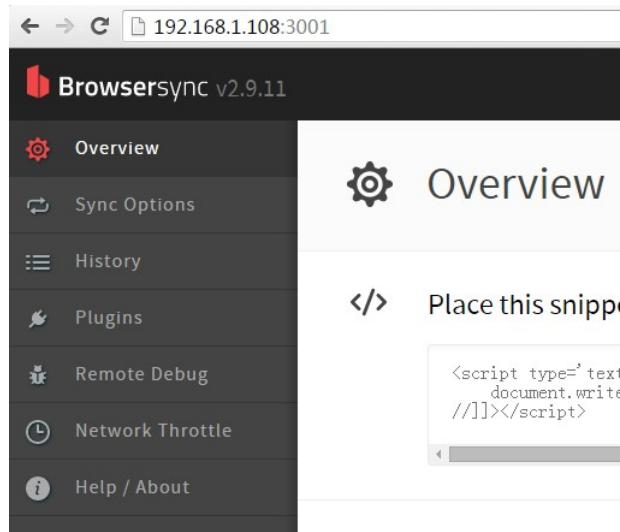


图 27.10: Browser 设置页面

Debugger，这样才能列出当前已经连接的客户端。否则的话，即使项目页面和 BrowserSync 是连接状态，Remote Debugger 也会出现捕捉不到的情况。因此，可能需要关闭再重新开启 Remote Debugger 进行调试⁸。

当网页已连接到 BrowserSync 的时候，我们可以查看一下源码，会发现它们都被添加了与 BrowserSync 有关的一段 `<script>` 代码，就像 liveReload 浏览器插件做的那样。这些代码会在浏览器和 BrowserSync 的服务器之间建立 web socket 连接，一旦有监听的文件发生变化，BrowserSync 会通知浏览器。

如果发生变化的文件是 css，BrowserSync 不会刷新整页，而是直接重新请求这个 css 文件，并更新到当前页中。

Workspace 是 Chrome DevTools 的一项功能，现在已加入到 Chrome stable 版本。Workspace 可以将 Chrome 中的网络资源与本地资源进行映射与同步，即在 Chrome 中对 CSS 的修改可以同步写到本地 css 文件中。在 chrome://flags/ 中启用 Developer Tools Experiments(开发者工具实验)⁹。

27.21 ReSharper(Commercial)

ReSharper 是一个 JetBrains 公司出品的著名的代码生成工具，其能帮助 Microsoft Visual Studio 成为一个更佳的 IDE。它包括一系列丰富的能大大增加 C# 和 Visual Basic .net 开发者生产力的特征。使用 ReSharper，你可以进行深度代码分析，智能代码协助，实时错误代码高亮显示，解决方案范围内代码分析，快速代码更正，一步完成代码格式化和清理，业界领先的自动代码重构，高级的集成单元测试方案，和强大的解决方案内导航和搜索。实质上，ReSharper 特征可用于 C#，VB.NET，XML，ASP.NET，XAML，和构建脚本。ReSharper 还为 C# 和

⁸参考链接：<http://www.codingserf.com/index.php/2015/03/browser-sync/>

⁹参考链接：<http://isux.tencent.com/chrome-workspace.html>

VB.NET 提供了增强的交叉语言功能，它使开发者可以有效的控制.net 混合项目。

27.21.1 设置 (Settings)

使用 Resharper 9 后电脑的内存蹭蹭蹭往上涨，轻松超过 1G，放弃使用那是不可能的了，无法想象没有 Resharper 的日子，可以选择关闭某些不需要的特性来降低内存的消耗。在 Resharper->Options...->Environment->Products&Features，关闭其中暂时不用的特性即可，设置完毕重新启动 Visual Studio 2013 即可，可以发现内存少了 500MB 左右。

自动换行 Resharper 长代码自动换行有时会造成阅读不便，Resharper Options=> C#=> Formatting Style=> Line Breaks and Wrapping=> Right margin(columns)=> 把这个数字改大些就好了，比如 360

27.21.2 快捷操作

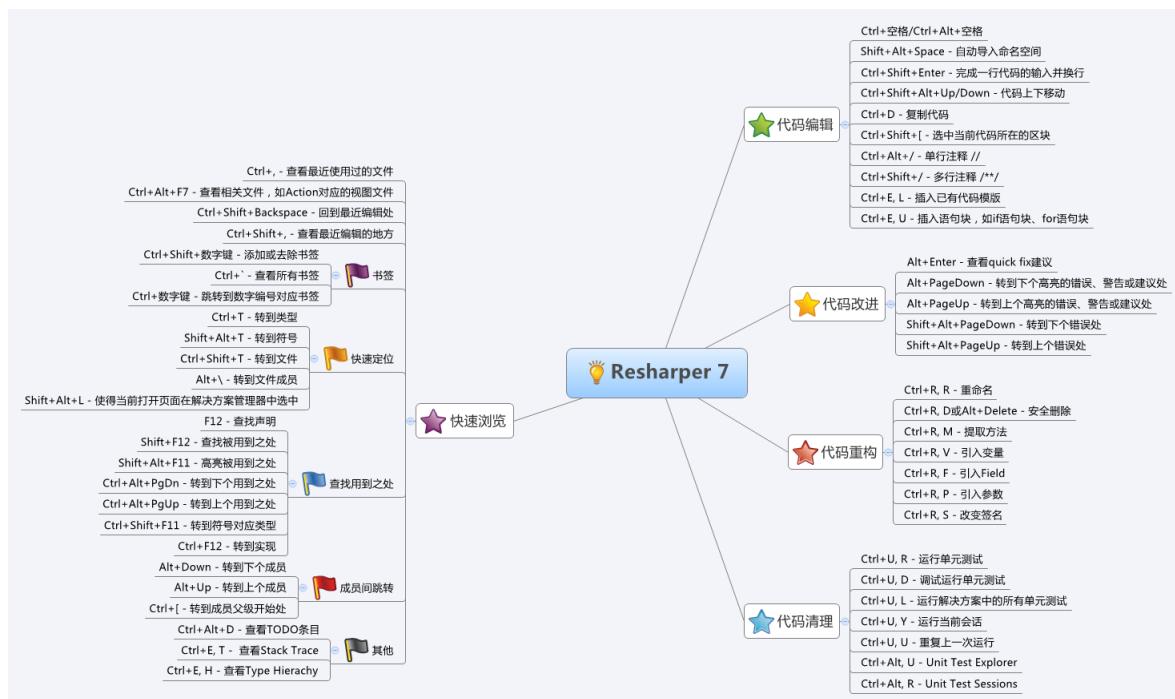


图 27.11: Resharper 快捷键

- 帮你实现某个接口或抽象基类的方法
- 提供你处理当前警告的一些建议
- 为你提供处理当前错误的一些建议（不一定是真的错误）
- 为你简化当前的臃肿代码

Ctrl+Enter 当我们看别人的代码，或者是看自己的代码的时候，总是觉得代码太多，于是我们就用 region 来把代码进行了封装注释，可是这样之后别人看代码就很郁闷，Resharper 的 File Structure 功能，就可以把 region 和你的方法都展示出来，如图27.12所示。

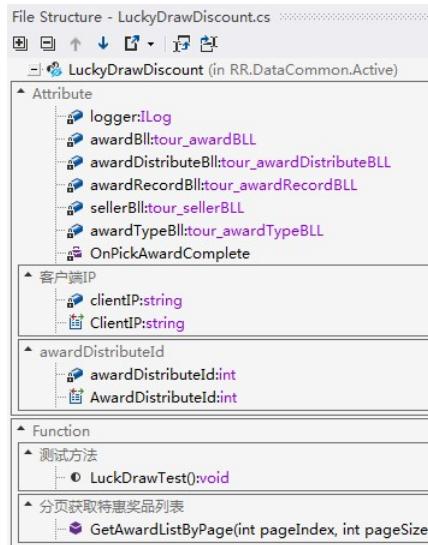


图 27.12: Resharper File Structure 功能

说了这么多，其实就是把对象浏览器和 region 的长处结合起来，既可以清晰的分类，又能一目了然的找到需要的方法。Resharper 这时帮上你的大忙了。用 Ctrl+F11，就弹出一个像右边这样的窗口来。这里面，按照你的 region 来显示，这样读你的代码的人也受益了。每个方法的参数，返回值都如 UML 一样列出来。

清洁代码 (Clean Up Code) 写了一个类之后，什么是最愉快的，就是让它顺便变干净以及变规范，这个时候，我们需要右键 Cleanup Code (Ctrl + Alt + F)。

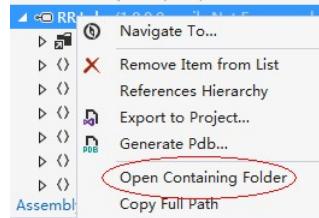


图 27.13: 查看当前 dll 的路径

查看 dll 路径

快速导航到文件 有时会查看当前编辑文件在项目树中的位置，此时最常见的操作是用鼠标点击项目解决方案树，一层一层展开，最终找到当前编辑的文件在项目树中对应的位置，而借助 Resharper 的快捷键组合 Shift+Alt+L(Locate)，可以一键展开项目树并定位到当前编辑的文件。

列出最近编辑位置 我比较常用的一个特性是列出最近编辑的文件，而在一个文件当中想快速回到之前编辑的地方，那么用快捷键 `Ctrl + Shift + Comma` 就可以列出最近编辑位置的列表。

27.21.3 Navigation and Search

Navigate To `Navigate To` is a single shortcut for all kinds of contextual navigation. `Navigate To` lists all destinations available at the current caret position. Press `Alt +~` for quick navigation links to the declaration, type declaration, base class, inheritor(s), or usage of the symbol under the caret; interface implementation for an interface; function exits for a function, and more.

Go to File Member

Go to File Member https://www.jetbrains.com/resharper/features/navigation_search.html

27.21.4 Code analysis

Call Tracking In the past, attempting to track call sequences in code could end up with lots of Find Results windows and lost context. To address this problem, ReSharper can visualize the entire call sequence in a single tool window. This is called Call Tracking, and it enables you to view and navigate through call chains in your code.

Thanks to support for events, interfaces, and closures, ReSharper's Call Tracking is a substantial improvement over Visual Studio's own Call Hierarchy.

To visualize a call sequence, choose `ReSharper | Inspect | Outgoing Calls` or `ReSharper | Inspect | Incoming Calls`, or use the `Inspect This` shortcut feature.

Navigating Between Highlighted Code Items Each error, warning, or suggestion is represented by an individual stripe on the Marker Bar. Clicking the stripe navigates you directly to the line of code that contains the error or causes the warning/suggestion.

You can navigate between errors, warnings and suggestions by pressing `Alt+PageDown` (forward) and `Alt+PageUp` (backward). You can also navigate between just errors (skipping any warnings and suggestions) by pressing `Shift+Alt+PageDown` (to next error) and `Shift+Alt+PageUp` (to previous error). A message describing the current error, warning, or suggestion is displayed in the status bar.

There's also an alternative way of navigating between code issues: you can find all code issues in a certain scope, and explore them in a dedicated tool window.

27.22 Reflexil

Reflexil is an assembly editor and runs as a plug-in for Red Gate's Reflector, IL Spy and Telerik's JustDecompile. Reflexil is using Mono.Cecil, written by Jb Evain and is able to manipulate IL code and save the modified assemblies to disk. Reflexil also supports C#/VB.NET code injection.

27.23 Gulp

27.23.1 简介

gulp.js 是一种基于流的，代码优于配置的新一代构建工具。Gulp 和 Grunt 类似。但相比于 Grunt 的频繁的 IO 操作，Gulp 的流操作，能更快地完成构建。

27.23.2 安装

```
1 npm install browser-sync gulp --save-dev
```

27.24 weinre

27.24.1 Install

```
1 npm -g install weinre
```

27.24.2 启动

```
1 weinre --boundHost 192.168.1.108
```

<http://wyqbailey.diandian.com/post/2011-11-09/20511143>

<http://liyadong.com>

27.25 Shadowsocks

27.25.1 简介

27.25.2 使用

启动时提示端口已被占用，查看本机的端口的使用情况。

```
1 netstat -ano|findstr "<端口号>"
```

27.26 NuGet

27.26.1 打开命令行界面

NuGet 的命令行管理界面在视图 > 其他窗口 > 程序包管理控制台下。



图 27.14: NuGet 进入命令行

常用的 Nuget 命令如下，可以切换当前的默认项目，选择为不同的项目安装相同或不同的组件。

```

1 #搜索可用包
2 Get-Package -listavailable -filter MiniProfiler
3
4 #安装包
5 Install-Package Swashbuckle
6
7 #卸载包
8 Uninstall-Package Swashbuckle
9
10 #列出当前可用的包

```

```
11 Get-Package -ListAvailable -Filter Swashbuckle
```

27.27 Quartz.NET

27.27.1 定时任务配置

定时任务采用 Quartz.NET 进行调度，Quartz.NET 是一个开源的作业调度框架，是 OpenSymphony 的 QuartzAPI 的.NET 移植，它用 C# 写成，可用于 winform 和 asp.net 应用中。它提供了巨大的灵活性而不牺牲简单性。你能够用它来为执行一个作业而创建简单的或复杂的调度。它有很多特征，如：数据库支持，集群，插件，支持 cron-like 表达式等等。

27.27.2 app.config

由于 quartz.net 依赖 common.logging 和 common.logging.log4net1213，在 app.config 中 common.logging 配置如下：

```
1 <configSections>
2   <section name="quartz" type="System.Configuration.NameValueSectionHandler, System,
3     Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
4   <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,
5     log4net"/>
6   <sectionGroup name="common">
7     <section name="logging" type="Common.Logging.ConfigurationSectionHandler, Common.
8       Logging" />
9   </sectionGroup>
10  </configSections>
11
12  <common>
13    <logging>
14      <factoryAdapter type="Common.Logging.Log4Net.Log4NetLoggerFactoryAdapter, Common
15        .Logging.Log4Net1213">
16        <arg key="configType" value="FILE-WATCH" />
17        <arg key="configFile" value="~/config/log4net.config" />
18      </factoryAdapter>
19    </logging>
20  </common>
```

quartz.net 易发生 dll 版本冲突的问题，建议的 dll 版本搭配方案之一为：

dll	版本
log4net	1.2.13.0
Common.Logging	3.0.0.0
Common.Logging.Log4Net1213	3.0.0.0
Quartz.net	2.3.1.0

27.27.3 quartz.config

quartz 的配置读取方式首先 app.config/web.config->quartz.config/Quartz.quartz.config->quartz_jobs.xml .

```

1 # You can configure your scheduler in either <quartz> configuration section
2 # or in quartz properties file
3 # Configuration section has precedence
4
5 quartz.scheduler.instanceName = ServerScheduler
6
7 # configure thread pool info
8 quartz.threadPool.type = Quartz.Simpl.SimpleThreadPool, Quartz
9 quartz.threadPool.threadCount = 10
10 quartz.threadPool.threadPriority = Normal
11
12 # job initialization plugin handles our xml reading, without it defaults are used
13 quartz.plugin.xml.type = Quartz.Plugin.Xml.XMLSchedulingDataProcessorPlugin, Quartz
14 quartz.plugin.xml.fileNames = ~/config/quartz_jobs.xml
15
16 # export this server to remoting context
17 quartz.scheduler.exporter.type = Quartz.Simpl.RemotingSchedulerExporter, Quartz
18 quartz.scheduler.exporter.port = 555
19 quartz.scheduler.exporter.bindName = QuartzScheduler
20 quartz.scheduler.exporter.channelType = tcp
21 quartz.scheduler.exporter.channelName = httpQuartz
22 quartz.jobStore.type = Quartz.Simpl.RAMJobStore, Quartz

```

通过 quartz.plugin.xml.fileNames 属性指定 quartz_jobs.xml 文件的保存位置。也可以将 quartz.config 的内容放到 app.config 配置文件中，注意如果在 app.config 中配置了 quartz，那么就没有必要再在单独的文件中配置。在 app.config 配置如下代码片段所示：

```

1 <quartz>
2   <add key="quartz.scheduler.instanceName" value="TaskScheduler"/>
3   <add key="quartz.scheduler.instanceId" value="AUTO"/>
4   <add key="quartz.threadPool.type" value="Quartz.Simpl.SimpleThreadPool, Quartz"
5     />
6   <add key="quartz.threadPool.threadCount" value="5"/>
7   <add key="quartz.threadPool.threadPriority" value="Normal"/>
8   <add key="quartz.plugin.xml.type" value="Quartz.Plugin.Xml.

```

```

    XMLSchedulingDataProcessorPlugin, Quartz"/>
8   <!--这里必须要保证在生成的目录下面有config这个文件夹-->
9   <add key="quartz.plugin.xml.fileNames" value="~/config/quartz_jobs.xml"/>
10  <add key="quartz.plugin.xml.scanInterval" value="600"/>
11  </quartz>

```

27.27.4 quartz_jobs.xml

Trigger 分为简单的 Simple 类型的 Trigger 和复杂的 Cron 表达式的 Trigger.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <job-scheduling-data xmlns="http://quartznet.sourceforge.net/JobSchedulingData"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0">
4      <processing-directives>
5          <overwrite-existing-data>true</overwrite-existing-data>
6      </processing-directives>
7      <schedule>
8          <job>
9              <name>ShutdownMachineJob</name>
10             <group>Group1</group>
11             <description>关闭计算机</description>
12             <!--任务类型, 任务的具体类型及所属程序集, 格式: 实现了IJob接口的包含完整命名空间
13                 的类名,程序集名称-->
14             <job-type>RRMall.WxPrint.Common.Cmd.JobHelper, RRMall.WxPrint.Common</
15                 job-type>
16             <!--<durable> (持久性) -如果一个Job是不持久的, 一旦没有触发器与之关联, 它就会
17                 被从scheduler 中自动删除-->
18             <durable>true</durable>
19             <recover>false</recover>
20         </job>
21         <trigger>
22             <!--简单任务的触发器, 可以调度用于重复执行的任务-->
23             <simple>
24                 <name>shutdownMachineTrigger</name>
25                 <group>Group2</group>
26                 <!--要调度的任务名称, 该job-name必须和对应job节点中的name完全相同-->
27                 <job-name>ShutdownMachineJob</job-name>
28                 <!--调度任务(job)所属分组, 该值必须和job中的group完全相同-->
29                 <job-group>Group1</job-group>
30                 <!--start-time(选填) 任务开始执行时间utc时间, 北京时间需要+08:00, 如: <start-
31                     time>2012-04-01T08:00:00+08:00</start-time>表示北京时间2012年4月1日上午8:00
32                     开始执行, 注意服务启动或重启时都会检测此属性, 若没有设置此属性或者start-time设置
33                     的时间比当前时间较早, 则服务启动后会立即执行一次调度, 若设置的时间比当前时间晚,
34                     服务会等到设置时间相同后才会第一次执行任务, 一般若无特殊需要请不要设置此属性-->
35             >
36             <start-time>2014-10-25T00:00:00+08:00</start-time>
37             <end-time>2015-05-10T00:00:00+08:00</end-time>
38             <!--任务执行次数, 如:<repeat-count>-1</repeat-count>表示无限次执行-->
39             <repeat-count>10</repeat-count>

```

```

31     <!--任务触发间隔(毫秒)-->
32     <repeat-interval>10000</repeat-interval>
33   </simple>
34 </trigger>
35 <job>
36   <name>ProbeNetworkStatusJob</name>
37   <group>Group3</group>
38   <description>侦测网络状态</description>
39   <job-type>RRMall.WxPrint.Common.Cmd.ProbeNetJob, RRMall.WxPrint.Common<
40     /job-type>
41   <durable>true</durable>
42   <recover>false</recover>
43 </job>
44 <trigger>
45   <!--cron复杂任务触发器使用cron表达式定制任务调度-->
46   <cron>
47     <name>ProbeNetworkStatusTrigger</name>
48     <group>Group4</group>
49     <job-name>ProbeNetworkStatusJob</job-name>
50     <job-group>Group3</job-group>
51     <start-time>2014-10-25T00:00:00+08:00</start-time>
52     <!--cron表达式-->
53     <cron-expression>0/5 * * * * ?</cron-expression>
54   </cron>
55 </trigger>
56 </schedule>
57 </job-scheduling-data>

```

表达式从左到右，依此是秒、分、时、月第几天、月、周几、年。常用的 cron 表达式为：

```

1 #每隔 5 秒执行一次
2 */5 * * * * ?
3 #每隔 1 分钟执行一次
4 0 */1 * * * ?
5 #每天凌晨 1 点执行一次
6 0 0 1 * * ?

```

许多 Trigger 可以指向同一个 Job，但一个 Trigger 仅仅能执行一个 Job。

27.27.5 实现 IJob 接口

```

1 public class SampleJob : IJob
2 {
3     private static readonly log4net.ILog logger = log4net.LogManager.GetLogger("SampleJob");
4     public void Execute(IJobExecutionContext context)
5     {
6         logger.Info("Task start...");
7     }
8 }

```

27.27.6 启动作业

```
1 private void btn_ExecuteJob_Click(object sender, EventArgs e)
2 {
3     try
4     {
5         logger.Info("开始调度作业..");
6         ISchedulerFactory schedulerFactory = new StdSchedulerFactory();
7         IScheduler scheduler = schedulerFactory.GetScheduler();
8         scheduler.Start();
9     }
10    catch (Exception ee)
11    {
12        logger.Error("启动作业调度时发生异常", ee);
13    }
14 }
```

作业启动之后，Quartz 会通过 XMLSchedulingDataProcessorPlugin 类定期去检测配置文件的变化，修改配置文件后一般不需要重新启动，例如修改了 Trigger 的 Cron 表达式执行的时间间隔，那么一段时间后即按照新的触发规则运行。

27.28 ThunderBird

通讯录确切位置在用户目录下 AppData/Thunderbird/下面的 abook.mab

27.29 PuTTY

27.29.1 PuTTY 保存密码

下载 putty 修改版。

27.29.2 psftp

```
1 #登录
2 open hostname
3 open username@hostname
4
5 #切换本地目录
6 lcd
```

27.29.3 putty 退出全屏

putty 全屏之后点击缩略图可退出全屏，如图27.15所示：

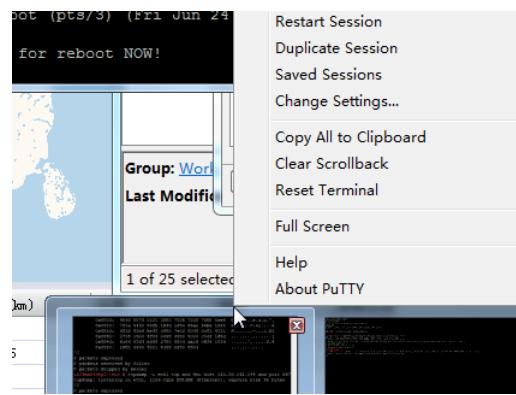


图 27.15: putty 退出全屏

Chapter 28

科学上网

28.1 Lantern

Lantern¹ is a free desktop application that delivers fast, reliable and secure access to the open Internet for users in censored regions. It uses a variety of techniques to stay unblocked, including P2P and domain fronting. Lantern relies on users in uncensored regions acting as access points to the open Internet.

28.1.1 How does Lantern work

Lantern was built to give users fast access to the blocked Internet. Lantern automatically detects whether or not a site is blocked and then accesses the blocked site either through our own servers or through Lantern users running as access points in the uncensored world. If a site is unblocked, Lantern gets out of the way, and your browser accesses it directly to give you the fastest possible access.

28.2 Shadowsocks(影梭)

¹Official Website:<https://getlantern.org/>

Part III

设计

Chapter 29

字体

29.1 字体 (Font)

29.1.1 Museo Sans

Museo Sans is based on the well-known Museo.

It is a sturdy, low contrast, geometric, highly legible sans serif typeface very well suited for any display and text use.

This OpenType font family offers also support for CE languages and even Esperanto. Besides ligatures, automatic fractions, proportional/tabular lining and old-style figures, numerators, denominators, superiors and inferiors MUSEO also has a ‘case’ feature for case sensitive forms.

29.1.2 Avenir Web

Part IV

待移除模块

29.2 配置

29.2.1 app.config

Node-Webkit

有时网页中的图片在浏览器中可以正常展示，但是在 Node-Webkit 中无法显示，是由于 Node-Webkit¹ 的缓存导致此问题，清除其缓存即可。由于 Node-Webkit 没有提供禁用缓存的 API，所以需要不停的清除缓存来解决此问题，在生成展示 html 页面的模板中添加语句：

```
1 gui.App.clearCache();
```

或者禁用缓存也可以达到同样的效果：

```
1 window.disableCache(bool);
```

程序间参数传递

在启动更新程序时，需要传递程序更新的标记，如更新视频传入标记 videoUpdate，那么更新程序即可根据传入的参数进行不同的逻辑处理。传入参数的代码片段为：

```
1 ProcessStartInfo startInfo = new ProcessStartInfo("RRMall.AutoUpdater.exe");
2 startInfo.Arguments = "videoUpdate";
3 Process.Start(startInfo);
```

同时在启动的程序中接收传入的参数：

```
1 static void Main(string[] Args)
2 {
3     Application.EnableVisualStyles();
4     Application.SetCompatibleTextRenderingDefault(false);
5     Application.Run(new FrmUpdate(Args));
6 }
```

29.2.2 定时任务

定时任务中部分任务的触发采用设计模式中的职责链（Chain Of Responsibility）模式。职责链模式可降低对象的耦合度，简化对象间的相互连接，增强给对象指派职责的灵活性，新的请求处理方便。客户端（Client）定时向服务器请求指令 Model，Model 的字段保存指令的状态信息，根据服务器的指令状态进行相应的动作。每一次执行符合命令条件的链条上的相应动作，

¹NW.js lets you call all Node.js modules directly from DOM and enables a new way of writing applications with all Web technologies. It was previously known as "node-webkit" project.

链条上的每一个节点都触发，链条上节点执行完毕后指定下一个节点，依次进行。职责链的目录结构如图29.1所示：

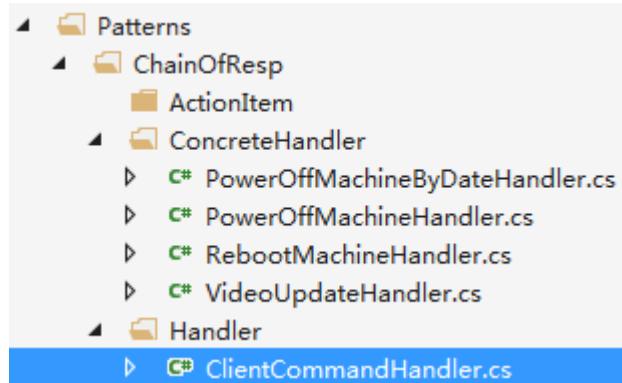


图 29.1: 职责链目录结构

动作的抽象类 ClientCommandHandler 如下代码片段所示：

```
1 public abstract class ClientCommandHandler
2 {
3     #region Attribute
4
5     /// <summary>
6     /// 触发动作执行的委托
7     /// </summary>
8     /// <param name="macData"></param>
9     public delegate void FireLeaveAction(MacData macData);
10
11    /// <summary>
12    /// 委托实例
13    /// </summary>
14    /// <param name="macData"></param>
15    public FireLeaveAction fireLeaveAction = null;
16
17    /// <summary>
18    /// 下一个处理对象（链条）
19    /// </summary>
20    public ClientCommandHandler nextHandler { get; set; }
21
22    /// <summary>
23    /// log4net实例
24    /// </summary>
25    private static readonly log4net.ILog logger = log4net.LogManager.GetLogger(MethodBase
        .GetCurrentMethod().DeclaringType);
26
27    #endregion
28
29    #region Function
30
31    /// <summary>
```

```

32  /// 需要处理的动作抽象
33  /// </summary>
34  /// <param name="macData"></param>
35  public abstract void ActionHandle(MacData macData);
36
37  /// <summary>
38  /// 委托触发的方法
39  /// </summary>
40  /// <param name="macData"></param>
41  public void LeaveAction(MacData macData)
42  {
43      if (fireLeaveAction != null)
44      {
45          fireLeaveAction(macData);
46      }
47  }
48 #endregion
49 }
```

动作的执行链条节点之一如下代码片段所示：

```

1  public class RebootMachineHandler : ClientCommandHandler
2  {
3      #region Attribute
4
5      /// <summary>
6      ///
7      /// </summary>
8      private static readonly log4net.ILog logger = log4net.LogManager.GetLogger(MethodBase
9          .GetCurrentMethod().DeclaringType);
10     #endregion
11
12     #region Function
13
14     #region Constructor
15     public RebootMachineHandler()
16     {
17         this.fireLeaveAction += new FireLeaveAction(RebootMachine_Handle);
18     }
19     #endregion
20
21     #region 处理本节点动作
22     /// <summary>
23     ///
24     /// </summary>
25     /// <param name="macData"></param>
26     private void RebootMachine_Handle(MacData macData)
27     {
28         ActionHandle(macData);
29     }
30     #endregion
31 }
```

```
30
31     #region 重启计算机
32     /// <summary>
33     /// 重启计算机
34     /// </summary>
35     /// <param name="macData"></param>
36     public override void ActionHandle(MacData macData)
37     {
38         if (macData.isReboot == 1)
39         {
40             IGUIDataService GUIDataServcie = (new GUIDataServiceFactory()).Create();
41             if (GUIDataServcie.SetMacInfo("isReboot"))
42             {
43                 ISystemOperHelper sysOper = (new SystemOperHelperFactory()).Create();
44                 sysOper.Shutdown("60", MachineOperType.Restart);
45             }
46             else
47             {
48                 logger.Error("设置isReboot标志失败");
49             }
50         }
51         else
52         {
53             if (nextHandler != null)
54             {
55                 //将任务移交下一个对象进行处理（直到有对象处理或链条结束为止）
56                 VideoUpdateHandler videoUpdateHandler = new VideoUpdateHandler();
57                 nextHandler = videoUpdateHandler;
58                 nextHandler.LeaveAction(macData);
59             }
60         }
61     }
62     #endregion
63
64     #endregion
65 }
```

界面更换验证码、调取打印数据

界面上更换验证码和定时调取打印数据通过 HTML DOM 中的 setInterval() 方法进行。setInterval() 方法会不停地调用函数，直到 clearInterval() 被调用或窗口被关闭。由 setInterval() 返回的 ID 值可用作 clearInterval() 方法的参数。如下脚本示例：

```
1 <script language=javascript>
2     var int=self.setInterval("clock()",50)
3     function clock()
4     {
5         var t=new Date()
6         document.getElementById("clock").value=t
```

```

7   }
8 </script>

```

setInterval() 方法第一个参数为调取的方法，第二个参数为间隔时间长度，单位为毫秒。

自动拨号

系统开机后无法自动连接网络，需要鼠标手动点击网络连接，为了避免反复连接鼠标带来操作上的额外步骤，系统自动集成自动拨号功能。自动拨号在程序启动后自动检测网络连接，在未连接到网络的情况下定时反复尝试自动拨号，无需人工干预。

自动关机

关闭计算机 系统间隔 10 分钟定时²检测关机设置，如检测到关机信号，则进行关机操作。

Quartz.NET 的 Job 设置如下代码片段所示：

```

1 <job>
2   <name>ShutdownMachineJob</name>
3   <group>Group1</group>
4   <description>关闭计算机</description>
5   <!--任务类型，任务的具体类型及所属程序集，格式：实现了IJob接口的包含完整命名空间的类
       名,程序集名称-->
6   <job-type>RRMall.WxPrint.Bll.Job.JobHelper, RRMall.WxPrint.Bll</job-type>
7   <!--<durable>（持久性）—如果一个Job是不持久的，一旦没有触发器与之关联，它就会被从
       scheduler中自动删除-->
8   <durable>true</durable>
9   <recover>false</recover>
10  </job>

```

自动关机的触发器设置如下代码片段所示，触发器采用简单触发器：

```

1 <trigger>
2   <!--cron复杂任务触发器使用cron表达式定制任务调度-->
3   <cron>
4     <name>ShutdownMachineTrigger</name>
5     <group>Group2</group>
6     <job-name>ShutdownMachineJob</job-name>
7     <job-group>Group1</job-group>
8     <start-time>2014-10-25T00:00:00+08:00</start-time>
9     <cron-expression>0 30 21 * * ?</cron-expression>
10    </cron>
11  </trigger>

```

cron 表达式配置每天晚上 9: 30 触发一次。

²定时检测关机状态时间和检测的时间间隔可通过 Quartz.NET 的 Crontab 表达式进行配置

```
1 #region 关闭计算机
2 /// <summary>
3 /// 关闭计算机
4 /// </summary>
5 /// <param name="seconds">启动关闭操作的时间（秒）</param>
6 public void Shutdown(string seconds)
{
7     if (string.IsNullOrEmpty(seconds))
8     {
9         logger.Error("输入的关机时间字符串为空， seconds:" + seconds);
10        return;
11    }
12    int time = 0;
13    if (int.TryParse(seconds, out time))
14    {
15        var startInfo = new System.Diagnostics.ProcessStartInfo("cmd.exe");
16        startInfo.UseShellExecute = false;
17        startInfo.RedirectStandardInput = true;
18        startInfo.RedirectStandardOutput = true;
19        startInfo.RedirectStandardError = true;
20        startInfo.CreateNoWindow = true;
21        var myProcess = new System.Diagnostics.Process();
22        myProcess.StartInfo = startInfo;
23        myProcess.Start();
24        myProcess.StandardInput.WriteLine("shutdown -f -s -t " + seconds);
25    }
26}
27else
28{
29    logger.Error("输入的字符串无法转换为关机时间， seconds:" + seconds);
30}
31}
32#endregion
```

自动关机也可以通过新建定时任务，如系统在 22: 50 关机可以通过在命令行界面运行如下语句：

```
1 at 22:50 shutdown -s
```

清空任务列表：

```
1 at /delete /yes
```

自动启动 3G 客户端

在 3G 网络连接程序未启动的情况下，程序自动启动 3G 网络连接程序。

视频更新

系统后台更新视频的流程如图29.2所示。

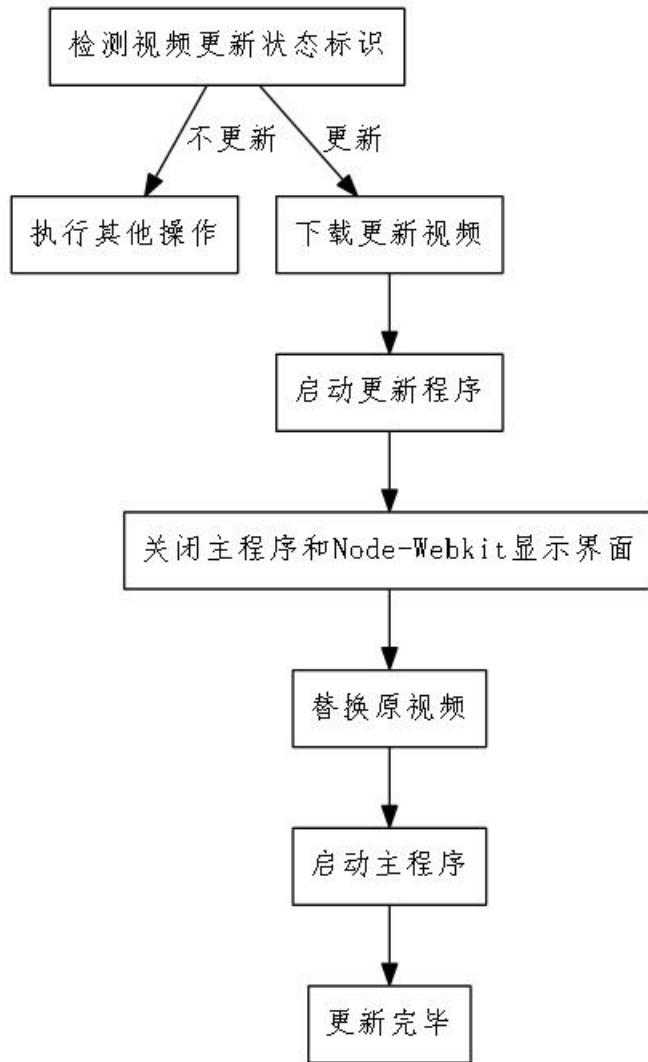


图 29.2: 视频更新流程

系统间隔一定时间探测是否有视频更新标志，在 Quartz.NET 中的 quartz_jobs.xml 文件中配置如下代码片段所示：

```

1 <job>
2   <name>ProbeVideoUpdateJob</name>
3   <group>Group5</group>
4   <description>侦测视频更新标识</description>
5   <job-type>RRMall.WxPrint.Bll.Job.ProbeVideoUpdate,RRMall.WxPrint.Bll</job-type>
6   <durable>true</durable>
7   <recover>false</recover>
8   </job>
9   <trigger>
10  <!-- cron复杂任务触发器使用cron表达式定制任务调度-->
  
```

```
11 <cron>
12   <name>ProbeVideoUpdateTrigger</name>
13   <group>Group6</group>
14   <job-name>ProbeVideoUpdateJob</job-name>
15   <job-group>Group5</job-group>
16   <start-time>2014-10-25T00:00:00+08:00</start-time>
17   <!--cron表达式:每隔一个小时触发一次-->
18   <cron-expression>0 0 */1 * * ?</cron-expression>
19   </cron>
20 </trigger>
```

配置文件里面采用了 cron 表达式 (cron-expression)。crontab 文件的格式: M H D m d cmd. M: 分钟 (0-59)。H: 小时 (0-23)。D: 天 (1-31)。m: 月 (1-12)。d: 一星期内的天 (0-6, 0 为星期天)。

每 10 秒触发一次: */10 0 * * * ?

每小时触发一次: 0 0 */1 * * ?

每一分钟触发一次: 0 0/1 * * * ?

在分钟里面的斜线为指定增量。如: “0/15” 在秒域意思是没分钟的 0, 15, 30 和 45 秒。“5/15” 在分钟域表示没小时的 5, 20, 35 和 50。符号 “*” 在 “/” 前面 (如: */10) 等价于 0 在 “/” 前面 (如: 0/10) 当检测到标志为 1 时, 启动后台视频更新进程, 从服务器数据库中读取 HTTP 链接进行视频下载, 视频下载完毕后重新启动程序, 将原有视频替换为更新视频。

```
1 #region 使用FTP协议下载单个文件
2 //下载定义
3 /// <summary>
4 /// FTP下载文件
5 /// </summary>
6 /// <param name="saveFilePath">保存文件路径</param>
7 /// <param name="saveFileName">保存文件名</param>
8 /// <param name="downloadFileName">下载文件名</param>
9 public void FTPDownloadFile(string saveFilePath, string saveFileName, string
    downloadFileName)
10 {
11     //定义FTP请求对象
12     FtpWebRequest ftpRequest = null;
13     //定义FTP响应对象
14     FtpWebResponse ftpResponse = null;
15     //存储流
16     FileStream saveStream = null;
17     //FTP数据流
18     Stream ftpStream = null;
19     string FTPFullName = string.Empty;
20     try
21     {
22         if (!Directory.Exists(saveFilePath))
23         {
24             Directory.CreateDirectory(saveFilePath);
25             logger.Info("创建路径: " + saveFilePath);
```

```

26     }
27     //生成下载文件
28     saveStream = new FileStream(saveFilePath + "\\\" + saveFileName, FileMode.Create
29     );
30     /*
31      * The FTP URI like "ftp://IP Address/new.zip",the update file located in root
32      * dictionary.
33      */
34     FTPFullFileName = "ftp://" + remoteHost + "/" + downloadFileName;
35     //生成FTP请求对象
36     ftpRequest = (FtpWebRequest)FtpWebRequest.Create(new Uri(FTPFULLFileName));
37     //设置下载文件方法
38     ftpRequest.Method = WebRequestMethods.Ftp.DownloadFile;
39     //设置文件传输类型
40     ftpRequest.UseBinary = true;
41     //设置登录FTP帐号和密码
42     ftpRequest.Credentials = new NetworkCredential(remoteUserName,
43         remoteUserPassword);
44     //生成FTP响应对象
45     ftpResponse = (FtpWebResponse)ftpRequest.GetResponse();
46     //获取FTP响应流对象
47     ftpStream = ftpResponse.GetResponseStream();
48     //响应数据长度
49     long cl = ftpResponse.ContentLength;
50     int bufferSize = 2048;
51     int readCount;
52     byte[] buffer = new byte[bufferSize];
53     //接收FTP文件流
54     readCount = ftpStream.Read(buffer, 0, bufferSize);
55     while (readCount > 0)
56     {
57         saveStream.Write(buffer, 0, readCount);
58         readCount = ftpStream.Read(buffer, 0, bufferSize);
59     }
60     logger.Info("更新文件下载成功！文件为：" + FTpfullFileName + "，保存路径为：" +
61         saveFilePath + "\\\" + saveFileName);
62 }
63 catch (Exception ex)
64 {
65     logger.Error("下载失败,FullFileName:" + FTpfullFileName, ex);
66 }
67 finally
68 {
69     if (ftpStream != null)
70     {
71         ftpStream.Close();
72     }
73
74     if (saveStream != null)
75     {
76         saveStream.Close();
77     }
78 }
```

```
73     }
74     if (ftpResponse != null)
75     {
76         ftpResponse.Close();
77     }
78 }
79 #endregion
```

后台下载视频时，开启新线程，传入 object 类型的参数。

```
1 //采用新线程进行后台下载
2 Thread t = new Thread(new ParameterizedThreadStart(DownloadFileByHTTP));
3 t.Start(macData);
```

```
1 #region 下载文件-替换Video
2 public static void DownloadFileByHTTP(object macData)
3 {
4     string fullPath = "";
5     string fileName = "";
6     MacData inputMacData = null;
7     try
8     {
9         inputMacData = (MacData)macData;
10        inputMacData.videoUrl = "http://static.zouwo.com/updater/a.wmv";
11        HttpClient httpClient = new HttpClient();
12        int position = inputMacData.videoUrl.LastIndexOf("/");
13        fileName = inputMacData.videoUrl.Substring(position, inputMacData.videoUrl.
14            Length - position);
15        string fileFolder = Application.StartupPath + @"\temp";
16        string videoStorePath = Application.StartupPath + @"\nwjs\video";
17        if (!Directory.Exists(fileFolder))
18        {
19            Directory.CreateDirectory(fileFolder);
20        }
21        fullPath = fileFolder + @"\" + fileName;
22        if (httpClient.GetSingleFile(fullFilePath, inputMacData.videoUrl))
23        {
24            logger.Info("下载完成, path:" + fullPath);
25            ISystemOperHelper sysOper = (new SystemOperHelperFactory()).Create();
26            sysOper.KillRRMallProcess();
27            if (!sysOper.ReplaceUpdateFile(fileFolder, videoStorePath))
28            {
29                logger.Error("替换文件失败,fullfilepath:" + fullPath);
30            }
31            System.Diagnostics.Process.Start("RRMall.WxPrint.exe");
32        }
33        else
34        {
35            logger.Error("下载失败, Path:" + fullPath + ",Url:" + inputMacData.videoUrl);
```

```

35     }
36   }
37   catch (Exception e)
38   {
39     StringBuilder errorMesg = new StringBuilder();
40     errorMesg.Append("后台下载文件时遇到错误,fullFilePath:" + fullFilePath);
41     errorMesg.Append(",fileName:" + fileName);
42     errorMesg.Append(",VideoUrl:" + inputMacData.videoUrl + ",详细信息为: ");
43     logger.Error(errorMesg, e);
44   }
45 }
46 #endregion

```

IIS 支持的文件格式 在采用 HTTP 下载更新视频的过程中，IIS 并不支持 MP4 文件格式的下载，返回 404 错误。需要在 IIS 中进行配置，不同版本的配置界面有所区别，这里是 IIS6.5，在站点中找到 MIME 类型。注意这里的分类有讲究，如果要下载文件，需要在分类里配置为 file/mp4。如果配置为 video/mp4，则直接在浏览器里面进行播放。

HTML5 对视频的要求 在下载视频更新后浏览器和 Node-Webkit 皆无法播放视频，而原有的视频还是可以播放，这里涉及到 HTML5 对视频的具体要求³：

当前，video 元素支持三种视频格式：

Ogg = 带有 Theora 视频编码和 Vorbis 音频编码的 Ogg 文件

MPEG4 = 带有 H.264 视频编码和 AAC 音频编码的 MPEG 4 文件

WebM = 带有 VP8 视频编码和 Vorbis 音频编码的 WebM 文件

E-

查看文件被进程占用 在删除文件时，会提示文件被进程占用，导致删除文件失败。可以借助小工具 Handle.exe 查看文件到底被哪个程序占用，如图29.3所示。

```

D:\Handle>handle D:\项目\hotmark\RRMall.WxPrint\bin\Debug\update\nwjs\video\2.mp
4

Handle v4.0
Copyright <C> 1997-2014 Mark Russinovich
Sysinternals - www.sysinternals.com

No matching handles found.

D:\Handle>

```

图 29.3: 用 Handle.exe 查看占用文件

由图中可以看出，文件 2.mp4 正在被进程 RRMall.WxPrint.vhost.exe 使用，导致文件无法删除。当文件被占用时，是无法删除文件的，可以通过先结束占用文件的进程，进而删除文件。另一个小程序 Process Explorer.exe 可以看出进程间的依赖情况。

³具体可参见：http://www.w3school.com.cn/html5/html_5_video.asp

发送指令截屏

发送指令截屏传送到服务器，检查机器运行情况。

1. 本地截取屏幕
2. 上传截图到服务器指定文件夹，通过调用接口采用 HTTP 上传
3. 将图片路径写入数据库，新建 tb_captureScreenRecord 表

重启热印机

重启热印机的流程如图所示：

29.2.3 照片打印

照片打印的流程如图29.5所示：

获取打印机状态

```

1 [DllImport("winspool.Drv", EntryPoint = "OpenPrinter", SetLastError = true, CharSet =
    CharSet.Auto, ExactSpelling = true, CallingConvention = CallingConvention.StdCall
)]
2 private static extern bool OpenPrinter([MarshalAs(UnmanagedType.LPStr)] string
    printerName, out IntPtr hPrinter, IntPtr pDefault);

```

调用此方法如下所示：

```

1 bool openSuccess=OpenPrinter(PrinterName, out hPrinter, defaults.DesiredAccess);

```

此方法打开指定的打印机，并获取打印机的句柄。如果函数成功，返回值是一个非零值。如果函数失败，返回值是零。PrinterName String，要打开的打印机的名字，指针 phPrinter Long，打开打印机或打印服务器对象的指针 pDefault PRINTER_DEFAULTS，这个结构保存要载入的打印机信息，可以为 NULL。PRINTER_DEFAULTS 数据结构成员之一 DesiredAccess ，指向由 pDefault 指定的 openprinter 句柄权限。获取打印状态的完整代码为：

```

1 #region 获取打印机状态
2 /// <summary>
3 /// 获取打印机状态
4 /// </summary>
5 /// <param name="printerName"></param>
6 /// <returns></returns>
7 [TestCase("EPSON L300 Series", Result = 0)]
8 public int GetPrinterStatusInt(string printerName)
9 {

```

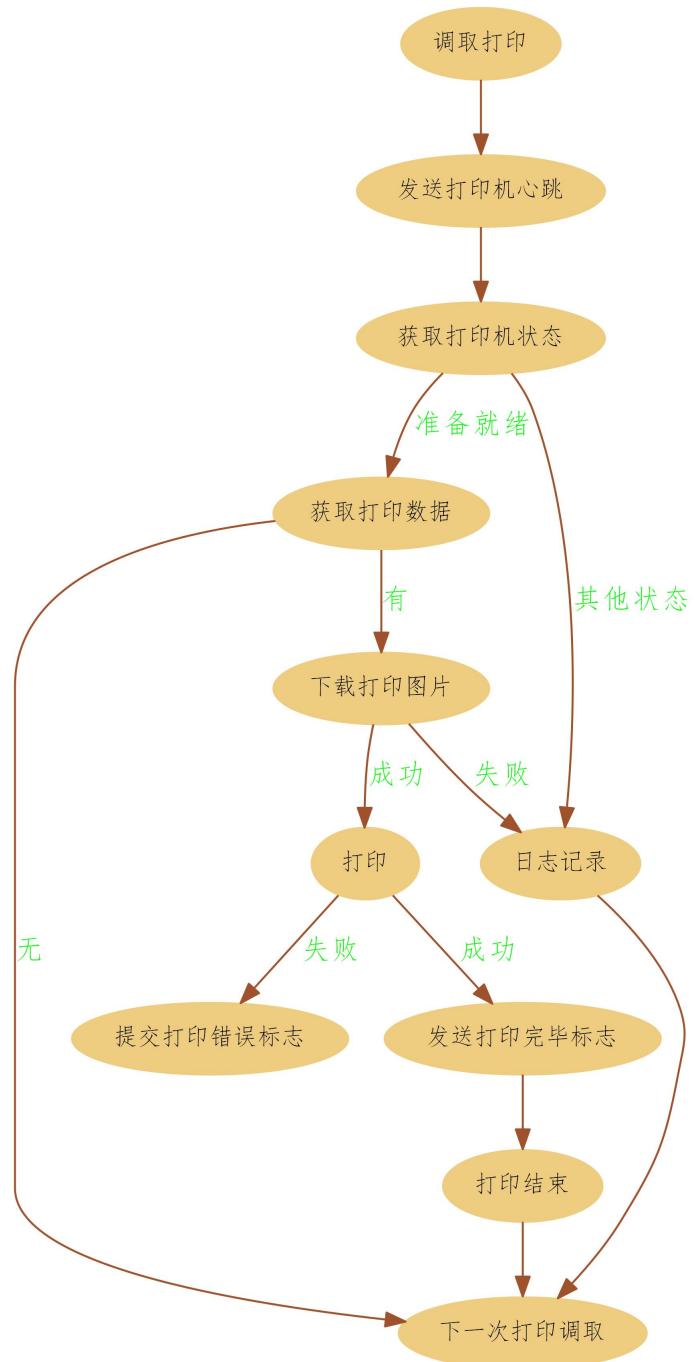


图 29.4: 照片打印流程

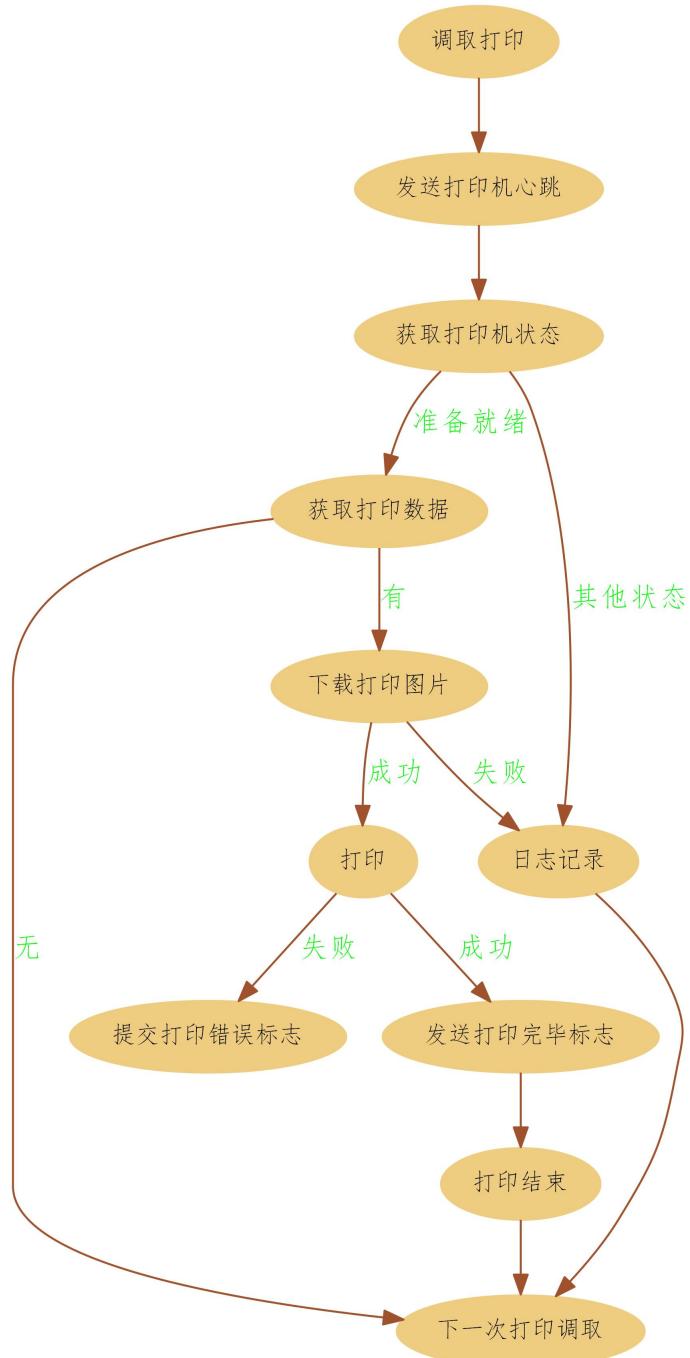


图 29.5: 照片打印流程

```

10    int intRet = 0;
11    IntPtr hPrinter;
12    PRINTERDEFAULTS defaults = new PRINTERDEFAULTS();
13    try
14    {
15        bool openSuccess = OpenPrinter(printerName, out hPrinter, defaults.DesiredAccess);
16        if (openSuccess)
17        {
18            int requestBufferSize = 0;
19            bool bolRet = GetPrinter(hPrinter, 2, IntPtr.Zero, 0, out requestBufferSize);
20            if (requestBufferSize > 0)
21            {
22                IntPtr pAddr = Marshal.AllocHGlobal((int)requestBufferSize);
23                bolRet = GetPrinter(hPrinter, 2, pAddr, requestBufferSize, out
24                requestBufferSize);
25                if (bolRet)
26                {
27                    PRINTER_INFO_2 Info2 = new PRINTER_INFO_2();
28                    Info2 = (PRINTER_INFO_2)Marshal.PtrToStructure(pAddr, typeof(
29                    PRINTER_INFO_2));
30                    intRet = System.Convert.ToInt32(Info2.Status);
31                }
32            else
33            {
34                logger.Info("Request failed!");
35            }
36            ClosePrinter(hPrinter);
37        }
38    else
39    {
40        logger.Info("Open failed:" + openSuccess);
41    }
42}
43 catch (Exception e)
44 {
45    logger.Error("Get printer status encounter an error", e);
46}
47 return intRet;
48}
#endregion

```

29.2.4 获取机器 Mac 地址

```

1 public string GetMacAddress()
2 {
3     string macAddress = string.Empty;

```

```
4  using (var macManagement = new ManagementClass("Win32_NetworkAdapterConfiguration"))
5  {
6      int macCount = 0;
7      foreach (ManagementObject managementObject in macManagement.GetInstances())
8      {
9          if (managementObject["MacAddress"] != null)
10         {
11             ++macCount;
12             macAddress = managementObject["MacAddress"].ToString();
13             if (macCount > 1)
14             {
15                 logger.Error("mac address count>1");
16             }
17         }
18     }
19 }
20 return macAddress;
21 }
```


Part V

附录

Stay Hungry. Stay Foolish.

Chapter 30

词汇

30.1 中英词汇对照

30.1.1 业务词汇对照

二维码	QR Code ¹ (Quick Response Code)
短消息业务	Short Messaging Service
括号	Parentheses
URI ²	Uniform Resource Identifier
URL ³	Uniform Resource Locators
URN ⁴	Functional Requirements for Uniform Resource Names
持续集成	Continuous Integration
电子签名	Electronic signature
幸运抽奖	Lucky Draw
重庆有线	Chongqing Cable Networks
csproj	C Sharp Project
前端开发	Font-End Development
新媒体	New Media
广告机	Advertising Machine
优惠券	Coupon
优惠券、现金凭单	Cash Voucher
折价券	Voucher
BLE	Bluetooth Low Energy
团购	Group Purchase
地理位置服务 (LBS)	Location Based Service
景区	Tourist Spots
PGP	GNU Privacy Guard
聚合	Mashup
PV(页面浏览量)	Page View
UV(独立访客量)	Unique Visitor
QPS(每秒查询率)	Query Per Second
迭代计划会	Iteration Planning Meeting
每日站会	Daily Standup
代码评审会	Code Review
回顾会议	Retrospective Meeting
结对编程	Pair Programming
现场客户	On-site Customer

语法糖	Syntactic Sugar ⁵
回调地狱	Callback Hell
DevOps	开发 (Development) & 运维 (Operations)
RAML	RESTful API 建模语言 (RESTful API Modeling Language: RAML)
UGC(用户生成内容)	User Generate Content
CC(复写的副本)	Carbon Copy
OTA(在线旅游社)	Online Travel Agent
DM(直接邮寄广告)	Direct Mail(Magazine) Advertising
Sketchy GUI	写生风格的图形界面
Stencils	(IOS UI Stencils) 模板; 钢板; [印刷] 模版印刷 (stencil 的复数形式)
Realm	领域、范围、王国 (in the realm of 在... 领域里)
RSS	Really Simple Syndication (真正简易联合)
CRUD	增加 (Create)、重新得到数据 (Retrieve)、更新 (Update) 和删除 (Delete)
HTTPS	HTTP over Secure Sockets Layer
Stems(起源于) from	起源于
pitfall	陷阱, 圈套; 缺陷; 诱惑
MVP	Minimum Viable Product(最小可用产品)
Over Engineering	过度工程
Gene-Editing	基因编辑
ludicrous	滑稽的, 荒唐的
Disposable items	一次性用品
bible	有权威的书
swagger	昂首阔步; 夸耀, 炫耀
PR(PR on line or Public Relations)	线上公关或 e 公关
LBS	位置服务 (LBS, Location Based Services)

¹ 二维码的一种类型之一, 具体参见 <http://en.wikipedia.org/wiki/Barcode>

² RFC1630, 发布于 1994 年 6 月, 被称为 “Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web”。它是一个 Informational RFC ——也就是说, 它没有获得社区的任何认可。

³ RFC1738, 发布于 1994 年 12 月, 被称为 “Uniform Resource Locators”。它是一个 Proposed Standard ——也就是说, 它是一个共识过程的结果, 虽然它还没有经过测试, 并成熟到足以成为一个完整的 Internet Standard。

⁴ RFC1737, 发布于 1994 年 12 月, 被称为 “Functional Requirements for Uniform Resource Names”。1997 年, 紧随 Proposed Standard RFC2141 (即 URN Syntax) 之后发布了 RFC1737, 它指定了另一个方案——urn: ——来加入 http:、ftp: 和其他协议中。

ACC

START 是方向盘锁，在这个位置方向盘就锁上了。
ON 是空档，一般停车就是这个也可以停到 START 位置，有的车停车是强制停到 START 位置。ACC 是一档，也就是打开一些用电器接通电源。LOCK 是点火档，着车用的。

⁵In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

30.1.2 计算机词汇中英对照

序列化	Serialization
DSN	Data Source Name
ODBC	Open DataBase Connectivity (开放数据库互联)
GGSCI	GoldenGate Software Command Interface
CPE	Extract Checkpoint File(抽取进程检查点文件)
CPR	Replicat Checkpoint Files(复制进程检查点文件)
PCE	Extract Process(抽取进程)
PCR	Replicat Process(复制进程)
PCM	Manager Process (管理进程)
RAC	实时应用集群 (Real Application Clusters)
TDM	交易数据管理 (Transactional Data Management)
DVI	设备无关 (DeVice Independent)
DWF	Design Web Format
BPEL	商业流程执行语言 (Business Process Execution Language)
STA	单线程单元 (Single-Threaded Apartment)
MTA	多线程单元 (Multi-Threaded Apartment)
ESB	企业服务总线 (Enterprise Service Bus)
SOA	面向服务的体系结构 (Service-OrientedArchitecture)
CI	持续集成 (Continuous integration)
CCB	配置管理委员会 (Configuration Control Board)
代码混淆	Obfuscated Code
测试驱动开发	Test-Driven Development (TDD)
测试驱动开发	Behavior-Driven Development (BDD)
数据持久化	Data Persistence
基础类库	Foundation Class
日志组件	Log Component
安全组件	Security Components
ADO	Activex Data Object
职责链模式	Chain Of Responsibility
面向方面设计 (AOP)	Aspect Oriented Programming
SGML	标准通用标记语言 (Standard Generalized Markup Language)
REST	表述性状态传递 (Representational State Transfer)
WSDL	Web Service Description Language
ISE	集成脚本环境 (Integrated Scripting Environment)
Ajax	Asynchronous JavaScript and XML
代码审查	Code ReView
认证	Credential

SSL	安全套接层协议 (Secure Socket Layer)
TAP	在网络分析监测领域, TAP 是 Test Access Point 的首字母缩写, 也叫分光器/分路器。分光是数据通过光纤传输; 分路是数据通过网线传输。粗浅的说, Tap 的概念类似于“三通”的意思, 即原来的流量正常通行, 同时分一股出来供监测设备分析使用。

30.2 常用术语

单线程单元 STA¹ 是 Single-threaded Apartment 的缩写, 程序每个线程都有自己独立的资源, 别的线程访问不到

多线程单元 MTA 则是.NET 程序的默认线程模型

持续集成

WebKit

Static Method Static methods are meant to be relevant to all the instances of a class rather than to any specific instance. They are similar to static variables in that sense. An example would be a static method to sum the values of all the variables of an instance for a class. For example, if there were a Product class it might have a static method to compute the average price of all products.

A static method can be invoked even if no instances of the class exist yet. Static methods are called "static" because they are resolved at compile time based on the class they are called on and not dynamically as in the case with instance methods which are resolved polymorphically based on the runtime type of the object. Therefore, static methods cannot be overridden

设计模式 Christopher Alexander 说过: “每一个模式描述了一个在我们周围不断重复发生的问题, 以及该问题的解决方案的核心。这样, 你就能一次又一次地使用该方案而不必做重复劳动” [AIS+77, 第 10 页]。尽管 Alexander 所指的是城市和建筑模式, 但他的思想也同样适用于面向对象设计模式, 只是在面向对象的解决方案里, 我们用对象和接口代替了墙壁和门窗。两类模式的核心都在于提供了相关问题的解决方案。

一、创建型

1. 工厂方法 (Factory Method): 定义一个用于创建对象的接口, 让子类决定实例化哪一个类。该模式使一个类的实例化延迟到其子类。
2. 抽象工厂 (Abstract Factory): 提供一个创建一系列相关或相互依赖对象的接口, 而无需指定它们具体的类。
3. 生成器 (Builder): 将一个复杂对象的构建与它的表示分离, 使得同样的构建过程可以创建不

¹详情请参考: http://en.wikipedia.org/wiki/Component_Object_Model#Threading

同的表示。

4. 原型模式 (Prototype): 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

5. 单例模式 (Singleton): 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

二、结构型

1. 适配器模式 (Adapter): 将一个类的接口转换成客户希望的另一个接口。该模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

2. 桥接 (Bridge): 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

3. 组合模式 (Composite): 将对象组合成树形结构以表示“部分-整体”的层次结构。该模式使得用户对单个对象和组合对象的使用具有一致性。

4. 装饰 (Decorator): 动态地给一个对象添加一些额外的职责。就增加功能来说，该模式相比生成子类更为灵活。

5. 外观 (Facade): 为子系统中的一组接口提供一个一致的界面，该模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

6. 享元 (Flyweight): 运用共享技术有效地支持大量细粒度的对象。

7. 代理 (proxy): 为其他对象提供一种代理以控制对这个对象的访问。

三、行为型

1. 职责链 (Chain of Responsibility)

意图：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

2. 命令 (Command)：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作。

3. 解释器 (Interpreter)：给定一个语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

4. 迭代器 (Iterator)：提供一种方法顺序访问一个聚合对象中的各个元素，而又不需暴露该对象的内部表示。

5. 中介者 (Mediator)：用一个中介对象封装一系列的对象交互。中介者使各对象不需要显式地相互作用，从而使其耦合松散，而起可以独立地改变它们之间的交互。

6. 备忘录 (Memento)：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

7. 观察者 (Observer)：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

8. 状态 (State)：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

9. 策略 (Strategy)：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

10. 模板方法 (Template Method)：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

11. 访问者 (Vistor)：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

框架 框架 (Framework) 是构成一类特定软件可复用设计的一组相互协作的类 [Deu89,JF88]。例如，一个框架能帮助建立适合不同领域的图形编辑器，像艺术绘画、音乐作曲和机械 C A D [V L 9 0 , J o h 9 2]。另一个框架也许能帮助你建立针对不同程序设计语言和目标机器的编译器 [J M L 9 2]。而再一个也许能帮助你建立财务建模应用 [B E 9 3]。你可以定义框架抽象类的应用相关的子类，从而将一个框架定制为特定应用。

30.3 编程语言漫谈

你应该首先回答的问题是：你准备从事什么方面的编程工作？是想创造美丽的网站还是喜欢设计 iOS 和 Android 上的手机游戏？是想进行个体用户的客户端开发还是想做大型企业软件的研发工作？金融和企业系统需要实现许多复杂的功能和高度的组织性，这需要学习 c 和 java。而与媒体和设计相关的网页和软件则需要动态、全能以及功能性的小型化编程语言，比如 Ruby, PHP, JavaScript 和 Objective-C。

Python/Ruby/PHP	网站和手机应用程序的服务器端
Objective-C	iPhone 的程序开发
HTML	网站的客户端。是构建网站的标记语言，网站的“积木”
CSS	网站的客户端。HTML 的描述语言
JavaScript	网站客户端，用于控制 HTML/CSS。jQuery 是你需要知道的
Java/.Net	Android 系统的编程。网站的服务器端，在大型企业中受欢迎
C/C++	高性能（如股票交易）或图形（电子游戏）的应用，GUI，游戏和多媒体工具包

我们可以说除了系统编程和对效率要求极高的程序之外，Java 在大部分领域优于 C++。大程度上似乎跟程序规模有关。其擅长的领域基本上于 Python 相似，在效率上无法跟 C/C++ 相提并论，在小规模的、大量使用模式匹配和编辑的项目里也无法匹敌 Perl。在小项目里，Java 显得过分强大了。我们猜测 Python 更适合小项目，而 Java 适合大项目，不过这一点并没有得到有力的证明。Python 最出色的地方在于，它鼓励清晰易读的代码，特别适合以渐进开发的方式构造大项目。其缺陷在于效率不高，太慢，不但跟编译语言相比慢，就是跟其他脚本语言相比也显得慢。一个普通的.NET 程序员，开始可能限制于 ASP.NET 的页面开发，但一旦他有了发展之心，他自然会对 ASP.NET MVC、Silverlight、WinForm、WPF 这些 UI 的开发手法感到兴趣，学习不需要多少时间，他可能就会认识这些 UI 开发只不过是一些工具，其实在开发原理上没什么区别。接着他就会向深一层的通讯模式进行了解，认识 TCP/IP、Web Service、WCF、Remoting 这些常用到的通讯方式，这时候他可能已经感觉到自己对开发技术有了进一步的了解。进而向工作流、设计模式、面向对象设计、领域驱动设计、面向服务开发等高层次进发，最后成为技术的领导者。

JAVA、C#、PHP、C++、VB……10 多种热门的开发语言，哪一种最有发展潜力呢？其实开发语言只不过是一个工具，“与其分散进攻，不如全力一击”，无论是哪一种开发语言，只要您全力地去学习，到有了一定的熟悉程度的时候，要学习另一种的语言也是轻而易举的事情。开发语言主要分为三大类：

1. 网络开发

现在网络已经成为世界通讯的一座桥梁，好像 Javascript、PHP、Ruby 这几类开发语言大部分是用作网络开发方面

2. 企业软件开发

JAVA、C#、VB 这几类开发语言都实现了面向对象开发的目标，更多时候用于企业系统的开发

3. 系统软件

C 语言、C++、Object-C 这些软件更多是用在系统软件开发，嵌入式开发的方面。

当然，这分类不是绝对，像 JAVA、C#、VB 很多时候也用于动态网站的开发。在很多开发项目都会使用集成开发的方式，同一个项目里面使用多种开发语言，各展所长，同步开发。但所以在刚入门的时候，建议您先为自己选择一种合适的开发工具，“专注地投入学习，全力一击”。

Chapter 31

经验汇总

1. 建立良好的 Profile 工具
2. 避免数据库的 join 操作，表与表之间的 join 尽量在应用层解决

31.1 Windows 搜索

如果在搜索中想让文件名称绝对匹配，可在关键字前添加等于符号 (=) 进行搜索。也可以使用 AND 和 OR 关键字。

针对接口编程，而不是针对实现编程
优先使用对象组合，而不是类继承
如果说应用程序难以设计，那么工具箱就更难了，而框架则是最难的。

31.1.1 自动属性

属性的特点：C# 属性是对类中的字段（fields）的保护，像访问字段一样来访问属性。同时也就封装了类的内部数据。每当赋值运算的时候自动调用 set 访问器，其他时候则调用 get 访问器。以帕斯卡命名不能冠以 Get/Set。静态属性是通过类名调用的！

C# 中属性这种机制使得在保证封装性的基础上实现了访问私有成员的便捷性。一个支持属性的语言将有助于获得更好的抽象。

31.1.2 批处理脚本启动服务

```
1 :: Description:This script help you start and stop database service quickly
2 :: Author:jiangtingqiang@gmail.com
3 :: Date:2013-08
4 ::
5 ::
```

```
6 :: MODIFY HISTORY
7 :: 2013-08-20 jiangxiaoqiang add oracle11g start&&stop
8 :: 2013-08-21 jiangxiaoqiang add oracle10g start&&stop
9 :: 2013-08-26 jiangxiaoqiang change bat script name to'SSDB',add database type choose
10 :: 2013-09-13 jiangxiaoqiang Program can't jump back,just jump forward
11 :: 2013-09-14 jiangxiaoqiang Add start SQL Agent,add 'begin' label
12 :: 2014-02-26 jiangxiaoqiang Add start postgreSQL database
13
14 @echo off
15 ::SSDB:Start or stop database
16 title Start and Stop DB
17 mode con cols=100 lines=40
18 color 72
19 ::@ Close the echo
20
21 :: echo[ ekəu,' ko]:n.回声, 共鸣; (言语、作风、思想等的) 重复; 重复者; [无线电]回波
22 :: vt.重复, 效仿; 随声附和; 类似; 发射 (声音等)
23
24 echo.
25 echo Function:
26 echo This script help you to start and stop SQL Server and Oracle service quickly.
27 echo Start and close software you used frequently.
28 echo.
29
30 :: set [variable=[string]]
31 :: set /p:show the variable,the variable name not contain "="
32 :Begin
33
34 echo.
35 echo Please choose which software you want to operate:
36 echo 1.SQL Server
37 echo 2.Oracle
38 echo 3.Exit
39 echo 4Shutdown computer
40 echo 5.Restart computer
41 echo 6.Sleep computer
42 echo 7.Delete system thumbnail
43 echo 8.postgreSQL
44 echo 10.etherPad
45 echo.
46
47 set /p DBType=:
48
49 if %DBType%==1 goto SQL Server
50 if %DBType%==2 goto Oracle
51 if %DBType%==3 goto Exit1
52 if %DBType%==4 goto shutdown
53 if %DBType%==5 goto Restart
54 if %DBType%==6 goto Sleep
55 if %DBType%==7 goto thumbnail
56 if %DBType%==8 goto postgreSQL
```

```
57 if %DBType%==10 goto etherPad
58 goto Error
59 ::goto end
60
61 :Oracle
62 set /p Version=Please choose your Oracle version[1:Oracle 10g ,2:Oracle 11g]:
63 echo.
64 if %Version%==1 goto Oracle10g
65 if %Version%==2 goto Oracle11g
66
67 --goto Oracle
68 echo The number you input can not be recognized.
69 goto end
70
71 :Oracle10g
72 echo Oracle10g .....
73 set /p var=Operation[1:Start , 2:Stop]:
74 echo.
75 if %var%==1 goto startOracle10g
76 if %var%==2 goto stopOracle10g
77 goto end
78
79
80 :Oracle11g
81 echo ..... oracle11g .....
82 set /p var=Operation[1:Start , 2:Stop]:
83 echo.
84 if %var%==1 goto startOracle11g
85 if %var%==2 goto stopOracle11g
86 goto Begin
87
88 :SQL Server
89 echo.
90 set /p var=Operation[1:Start , 2:Stop, 3:SQL Server Agent]:
91 echo.
92 if %var%==1 goto Start SQL Server
93 if %var%==2 goto Stop SQL Server
94 if %var%==3 goto SQL Server Agent
95 goto Error
96
97 :Start SQL Server
98 echo Start SQL Server .....
99 echo.
100 net start MSSQLSERVER
101 goto Begin
102
103 :Stop SQL Server
104 echo Stop SQL Server .....
105 echo.
106 net stop MSSQLSERVER
107 echo Stop SQL ReportServer.....
```

```
108 net stop ReportServer
109 goto Begin
110
111 :SQL Server Agent
112     set /p var=Operation[1.Start,2.Stop]:
113     if %var%==1 goto Start SQL Server Agent
114     if %var%==2 goto Stop SQL Server Agent
115     goto Error
116
117 :Start SQL Server Agent
118     echo Start SQL Server Agent.....
119     echo.
120     net start SQLServerAgent
121     goto begin
122
123 :Stop SQL Server Agent
124     echo Stop SQL Server Agent.....
125     echo.
126     net stop SQLServerAgent
127     goto begin
128
129 :startOracle10g
130     echo Start Oracle10g Listener .....
131     net start OracleOraDb10g_home1TNSListener
132     echo Start Oracle10g service ORCL.....
133     net start OracleServiceORCL
134     goto Begin
135
136 :stopOracle10g
137     echo ..... Stop Oracle10g .....
138     echo.
139     net start OracleOraDb10g_home1TNSListener
140     net start OracleServiceORCL
141     goto end
142
143
144 :startOracle11g
145     echo ..... Start Oracle11g .....
146     echo.
147     net start OracleOraDb11g_home1TNSListener
148     net start OracleServiceORCL
149     goto begin
150
151 :stopOracle11g
152     echo ..... Stop Oracle11g .....
153     echo.
154     net stop OracleOraDb11g_home1TNSListener
155     net stop OracleServiceORCL
156     goto begin
157
158 :shutdown
```

```
159 echo.
160 echo Your computer will shutdown in 1 minites.....
161 echo If you want to cancel,please input 'shutdown /a' in command line window.....
162 shutdown /f /s /t 60
163 goto Begin

164
165 :Restart
166 echo.
167 echo Your computer will restart.....
168 shutdown /r /f /t 60
169 goto begin

170
171 :Sleep
172 echo.
173 echo Your computer will go to sleep.....
174 shutdown /l
175 goto begin

176
177 :thumbnail
178 echo.
179 echo Start to delete thumbnail.....
180 FOR %%I IN (D: E: F: G: H: I: ) DO (%%I cd\\ attrib -s -h -r Thumbs.db /s /d
    >nul del Thumbs.db /s)
181 echo Complete!
182 goto begin

183
184 :postgreSQL
185 echo.
186 set /p var=Operation[1:Start]:
187 echo.
188 if %var%==1 goto Start postgreSQL
189 goto Error

190
191 :Start postgreSQL
192 echo.
193 echo Start postgreSQL....
194 cd /d H:\\Software\\DB\\pgsql\\bin
195 pg_ctl start -D H:\\Software\\DB\\pgsql\\data
196 ::pg_ctl status -D H:\\Software\\DB\\pgsql\\data
197 goto begin

198
199 :etherPad
200 echo.
201 set /p var=Operation[1.Start]:
202 if %var%==1 goto Start etherPad

203
204 :Start etherPad
205 echo Start etherPad now.....
206 pause
207 cd /d I:\\Program Files (x86)\\etherPad\\etherpad-lite-win
208 start start.bat
```

```

209  goto begin
210
211 :Error
212 echo invalid input !!!!!
213 echo.
214 goto begin
215
216 echo you can exit press any key..... &pause>nul
217 :Exit1
218 :: end should be the last row
219 :end

```

31.1.3 Windows 脚本 - % dp0 的含义

更改当前目录为批处理本身的目录有些晕吧？不急，我举例比如你有个批处理 a.bat 在 D:/qq 文件夹下 a.bat 内容为 cd /d % dp0 在这里 cd /d % dp0 的意思就是 cd /d d:/qq %0 代表批处理本身 d:/qq/a.bat dp 是变量扩充 d 既是扩充到分区号 d: p 就是扩充到路径 /qq dp 就是扩充到分区号路径 d:/qq

DebugView.exe 的使用

Net 程序的调试，是 DotNet 程序员必备的技能之一，开发出稳定的程序、解决程序的疑难杂症都需要很强大的调试能力，DotNet 调试有很多方法和技巧，熟练使用 DebugView 即是其中之一。

DebugView 是一个查看调试信息的非常棒的工具，支持 Debug、Release 模式编译的程序，甚至支持内核程序，而且能够定制各种过滤条件，让你只看到关心的输出信息，而且可以定制高亮显示的内容等等，非常方便。

自 Windows Vista 以来，调试信息在默认状态下是不显示的。为了显示调试信息，按照如下步骤设置即可：

1. 打开注册表；
2. 在 HKLM/SYSTEM/CuurentControlSet/Control/Session Manager 下新建一个名称为 Debug Print Filter 的 key；
3. 在 Debug Print Filter 下新建一个项：Default，值为 0xF.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Diagnostics;
5
6  namespace ConsoleApplication1
7  {
8      class Program

```

```
9  {
10     static void Main(string[] args)
11     {
12         Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
13         Debug.AutoFlush = true;
14         Debug.Indent();
15         Debug.WriteLine("Entering Main"); //显示在 DebugView 的信息
16         Console.WriteLine("Hello World.");
17         Debug.WriteLine("Exiting Main");
18         Debug.Unindent();
19     }
20 }
21 }
```

由于 Visual Studio 没有 Pass Through 技术，调试时可在 Visual Studio 里面查看输出的调试信息，程序实际运行时可在 DebugView 里查看调试信息。

31.2 AppDomain

AppDomain 是 CLR 的运行单元，它可以加载 Assembly、创建对象以及执行程序。AppDomain 是 CLR 实现代码隔离的基本机制。每一个 AppDomain 可以单独运行、停止；每个 AppDomain 有自己默认的异常处理；一个 AppDomain 的运行失败不会影响到其他的 AppDomain。CLR 在被 CLR Host(Windows Shell or InternetExplorer or SQL Server) 加载后，要创建一个默认的 AppDomain，程序的入口点 (Main 方法) 就是在这个默认的 AppDomain 中执行。运行一个.NET 应用程序或者一个运行库宿主时，OS 会首先建立一个进程，然后会在进程中加载 CLR(这个加载一般是通过调用 _CorExeMain 或者 _CorBindToRuntimeEx 方法来实现)，在加载 CLR 时会创建一个默认的 AppDomain，它是 CLR 的运行单元，程序的 Main 方法就是在这里执行，这个默认的 AppDomain 是唯一且不能被卸载的，当该进程消灭时，默认 AppDomain 才会随之消失。

31.2.1 AppDomain VS 进程

AppDomain 被创建在进程中，一个进程中可以有多个 AppDomain。一个 AppDomain 只能属于一个进程。2.AppDomain vs 线程其实两者本来没什么好对比的。AppDomain 是个静态概念，只是限定了对象的边界；线程是个动态概念，它可以运行在不同的 AppDomain。一个 AppDomain 内可以创建多个线程，但是不能限定这些线程只能在本 AppDomain 内执行代码。CLR 中的 System.Threading.Thread 对象其实是个 soft thread，它并不能被操作系统识别；操作系统能识别的是 hard thread。一个 soft thread 只属于一个 AppDomain，穿越 AppDomain 的是 hard thread。当 hard thread 访问到某个 AppDomain 时，一个 AppDomain 就会为之产生一个 soft thread。hard thread 有 thread local storage(TLS)，这个存储区被 CLR 用来存储这个 hard thread 当前对应的 AppDomain 引用以及 soft thread 引用。当一个 hard thread 穿越到另外一个 AppDomain 时，TLS 中的这些引用也会改变。当然这个说法很可能是和 CLR 的实现相

关的。

31.2.2 AppDomain VS Assembly

Assembly 是 .Net 程序的基本部署单元，它可以为 CLR 提供用于识别类型的元数据等等。Assembly 不能单独执行，它必须被加载到 AppDomain 中，然后由 AppDomain 创建程序集中的对象。一个 Assembly 可以被多个 AppDomain 加载，一个 AppDomain 可以加载多个 Assembly。每个 AppDomain 引用到某个类型的时候需要把相应的 assembly 在各自的 AppDomain 中初始化。因此，每个 AppDomain 会单独保持一个类的静态变量。

31.2.3 AppDomain VS 对象

任何对象只能属于一个 AppDomain。AppDomain 用来隔离对象，不同 AppDomain 之间的对象必须通过 Proxy(reference type) 或者 Clone(value type) 通信。引用类型需要继承 System.MarshalByRefObject 才能被 Marshal/UnMarshal(Proxy)。值类型需要设置 Serializable 属性才能被 Marshal/UnMarshal(Clone)。

31.3 进程间通信

31.3.1 程序间参数传递

系统更新时需要启动一个单独的更新程序，此程序为独立的 exe。在不同的更新场景下，需要更新程序具有不同的动作。比如更新视频时需要下载视频文件，更新软件时需要下载软件的依赖程序包。此处采用在启动更新程序 exe 时传入参数进行标志。

在启动程序时，传入参数，代码如下所示：

```
1 #region 启动更新程序
2 /// <summary>
3 /// Start update program
4 /// </summary>
5 /// <param name="updatorArgs"> 传入更新程序中的参数 </param>
6 public void StartUpdaterByVersionNo(string updatorArgs)
7 {
8     string updatorProgramName = string.Empty;
9     string updatorFullPath = string.Empty;
10    ProcessStartInfo processInfo = new ProcessStartInfo();
11    processInfo.Arguments = updatorArgs;
12    if (string.IsNullOrEmpty(version))
13    {
14        processInfo.FileName = UPDATORNAME;
15        Process.Start(UPDATORNAME);
16    }
17    else
```

```

18  {
19      updatorProgramName = UPDATORNAME.Substring(0, UPDATORNAME.Length -
20          4) + version + ".exe";
21      updatorFullPath = Application.StartupPath + @"\" + updatorProgramName;
22      if (File.Exists(updatorFullPath))
23      {
24          logger.Info("更新程序本地存在, 启动中..., fullPath:" + updatorFullPath);
25          processInfo.FileName = updatorProgramName;
26          Process.Start(processInfo);
27      }
28      else
29      {
30          logger.Info("本地不存在更新程序..., fullPath:" + updatorFullPath);
31          processInfo.FileName = UPDATORNAME;
32          Process.Start(processInfo);
33      }
34  }
35 #endregion

```

在 exe 的入口方法中接收参数，代码如下所示：

```

1 static void Main(string[] Args)
2 {
3     Application.EnableVisualStyles();
4     Application.SetCompatibleTextRenderingDefault(false);
5     Application.Run(new FrmUpdate(Args));
6     string updateContent = Args[0];
7 }

```

31.4 设计模式 (Design Pattern)

31.4.1 创建型模式

所有的创建型模式都有两个永恒的主旋律：第一，它们都将系统使用哪些具体类的信息封装起来；第二，它们隐藏了这些类的实例是如何被创建和组织的。外界对于这些对象只知道它们共同的接口，而不清楚其具体的实现细节。正因为如此，创建型模式在创建什么 (what)，由谁 (who) 来创建，以及何时 (when) 创建这些方面，都为软件设计者提供了尽可能大的灵活性。

简单工厂 (Simple Factory)

抽象工厂 (Abstract Factory)

工厂方法 (Factory)

在软件系统中，经常面临着“某个对象”的创建工作，由于需求的变化，这个对象的具体实现经常面临着剧烈的变化，但是它却拥有比较稳定的接口。如何应对这种变化？如何提供一种封装机制来隔离出“这个易变对象”的变化，从而保持系统中“其它依赖该对象的对象”不随着需求的改变而改变？这就可以用到工厂方法模式了。工厂方法模式定义了一个用于创建对象的接口，让其子类决定实例化哪一个对象。

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method let a class defer instantiation to subclasses.

使一个对象的创建工作延迟到其子类。工厂方法尽量遵循**依赖倒置原则¹**(Dependence Inversion Principle)。

在这里需要创建 SystemOperHelper 类，为了应对 SystemOperHelper 将来发送的变化，比如类名进行修改，避免使用 new 的方式创建对象，采用工厂方法模式。

新建 SystemOperHelper 的接口：

```

1 public interface ISystemOperHelper
2 {
3     void KillRRMallProcess();
4 }
```

实现接口：

```

1 public class SystemOperHelper : ISystemOperHelper
2 {
3     public void KillRRMallProcess()
4     {
5         //实现方法
6     }
7 }
```

新建工厂接口：

```

1 interface ISystemOperHelperFactory
2 {
3     ISystemOperHelper Create();
4 }
```

¹A. 高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。B. 抽象不应该依赖于具体，具体应该依赖于抽象。

实现创建对象的工厂方法：

```

1  public class SystemOperHelperFactory : ISystemOperHelperFactory
2  {
3      public ISystemOperHelper Create()
4      {
5          return new SystemOperHelper();
6      }
7 }
```

31.4.2 结构型模式

31.4.3 行为型模式

职责链模式 (Chain of Responsibility)

31.4.4 表驱动法 (Table-Driven Approach)

表驱动法 (查表法) 是一种编程模式 (scheme)-从表里面查找信息而不使用逻辑语句 (if 和 case)。事实上，凡是能通过逻辑语句来选择的事物，都可以通过查表来选择。对简单的情况而言，使用逻辑语句更为容易和直白。但随着逻辑链的越来越复杂，查表法也就愈发显得更具有吸引力。

31.5 逆向工程 (Reverse Engineering)

31.5.1 radare2

radare2 是一款开放源代码的逆向工程平台，它可以反汇编、调试、分析和操作二进制文件。

31.5.2 反混淆 dll(Deobfuscation dll)

目前比较常用的混淆 (加壳) 有 Dotfuscator,MaxToCode,Xenocode,ThemIDA.

反混淆的工具有很多 Dedot ,DePhe,XeCoString 等，但是这些只能剥对应算法，今天重点推荐一个工具 De4Dot

De4Dot 支持多种反混淆.Dotfuscator,MaxToCode 这两种是网友测试的,Xenocode 是我测试的，

官方介绍支持：

- Agile.NET (aka CliSecure)
- Babel.NET

- CodeFort
- CodeVeil
- CodeWall
- CryptoObfuscator
- DeepSea Obfuscator
- Dotfuscator
- .NET Reactor
- Eazfuscator.NET
- Goliath.NET
- ILProtector
- MaxtoCode
- MPRESS
- Rummage
- Skater.NET
- SmartAssembly
- Spices.Net
- Xenocode

经过混淆的 dll 如图31.1所示，命名空间的名称已经变成了无意义的字符串。

采用 De4Dot 对 dll 进行反混淆处理，在命令行下切换到 de4dot 的目录，输入如下命令：

```
1 de4dot.exe dllFullPath
```

如图31.4所示，其中 dllFullPath 为需要反混淆的 dll 完整路径。处理后会在目录下生成 dll 名称 +cleaned 为文件名的文件，此文件即输出文件，用 ILSpy 打开即可发现，命名空间已经变成有意义字符串。

31.6 开发技巧

31.6.1 搜索技巧 (Search Tricks)

准确搜索 (Exact phrase) 最简单、有效的准确搜索方式是在关键词上加上双引号，在这种情况下，搜索引擎只会反馈和关键词完全吻合的搜索结果。比方说在搜索「Joe Bloggs」的时候，

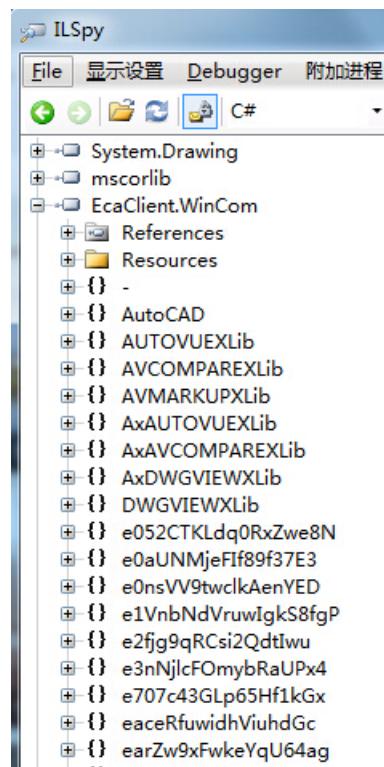


图 31.1: 经过混淆的 dll

```
E:\CrossPlatform\Software\Encrypt and Decrypt\de4dot-3.1.41592\de4dot.exe EcaClient.WinCom.dll

de4dot v3.1.41592.3405 Copyright <C> 2011-2014 de4dot@gmail.com
Latest version and source code: https://bitbucket.org/0xd4d/de4dot

Detected .NET Reactor <E:\CrossPlatform\Software\Encrypt and Decrypt\de4dot-3.1.41592\EcaClient.WinCom.dll>
Cleaning E:\CrossPlatform\Software\Encrypt and Decrypt\de4dot-3.1.41592\EcaClient.WinCom.dll
Renaming all obfuscated symbols
WARNING: Could not find property "eEitoeewG" in attribute "ef2bvWdxmP0quUCT2Gg" <02000AD4>
Saving E:\CrossPlatform\Software\Encrypt and Decrypt\de4dot-3.1.41592\EcaClient.WinCom-cleaned.dll
Ignored 2 warnings/errors
Use -v/-vv option or set environment variable SHOWALLMESSAGES=1 to see all messages
```

图 31.2: 反混淆 dll

在没有给关键词加上双引号的情况下，搜索引擎会显示所有分别和「Joe」以及「Bloggs」相关的信息，但这些显然并不是我们想要的结果。但在加上双引号后，搜索引擎则仅会在页面上反馈和「Joe Bloggs」相吻合的信息。准确搜索在排除常见但相近度偏低的信息时非常有用，可以为用户省去再度对结果进行筛选的麻烦。

排除关键词 (Exclude terms) 如果在进行准确搜索时没有找到自己想要的结果，用户可以对包含特定词汇的信息进行排除，仅需使用减号即可。例如在搜索「『Joe Bloggs』 -jeans」时，你所得到的结果反馈是不包含「jeans」字眼的「Joe Bloggs」条目。

用「Either OR」(或)逻辑进行搜索 在默认搜索下，搜索引擎会反馈所有和查询词汇相关的结果，但通过使用「OR」逻辑，你可以得到和两个关键词分别相关的结果，而不仅仅是和两个关键词都同时相关的结果。巧妙使用「OR」搜索可以让你在未能确定哪个关键词对于搜索结果起决定作用时依然可以确保搜索结果的准确性。[OR] 也可以替换成竖线 (|)。

④ 关键字前加 @ 符号，查找社交网站上相关结果。

站内搜索 站内搜索语法如下：

```
1 php site:http://stackoverflow.com/
```

site 操作符非常有用，可以让 google 搜索来自指定的网站的网页。比如，想在 sourceforge.net 网站上搜索关于编辑器 notepad++ 的网页，可以输入：“notepad++” site:sourceforge.net。这样 google 就能返回来自 sourceforge.net 的网页。又比如，想查找 csdn 网站上关于端口扫描器 Nmap 的网页，那么可以输入：“nmap” site:csdn.net。site 操作符，可用于搜索指定网站上的特殊的信息，如包含的主要资源，是否暴露出敏感文件等。

文件类型 filetype 当用户单独搜索某一类文件时，可使用该关键字。比如想搜索关于黑客技巧 Hacking 的 PPT，可以搜索：hacking filetype:ppt。如果想要搜索关于黑客大曝光系列图书的 PDF，可以尝试输入：“hacking exposed” filetype:pdf，这样就能直接搜索很多该系列图书的 PDF 文档。同理，也可以结合 site 操作符，搜索指定网站内的资源，如 word 文件、XML 文件、excel 文件、PPT 文档、txt 等等。

.. 时间范围 两点操作符用于指定一个网页的时间范围，比如用户想搜索的是 2011 年到 2012 年内，出现的关于 web hacking 的 PDF 资料。

可以输入：web hacking filetype:pdf 2011..2012

intitle:index-of 搜索某类资源的索引目录，如想搜索网站的图像、父目录等信息。可以在 google 搜索：intitle:index-of image 和 intitle:index-of “parent directory”。也可以搜索与 Admin 密切相关的网页：intitle:index-of “admin”。也可以尝试：intitle:index-of mp3/intitle:index-of password 等查询，具体查询内容就需要靠自己想象力了。

31.6.2 合理使用 Using 子句

如果你明确知道某个对象在进行完某些操作后，不再有用，需要回收，你可以使用'using'语句来销毁对象。

```

1 //以下语句:
2 using(SomeDisposableClass someDisposableObject = new SomeDisposableClass())
3 {
4     someDisposableObject.DoTheJob();
5 }
6
7 //和以下语句是一样效果:
8 SomeDisposableClass someDisposableObject = new SomeDisposableClass();
9 try
10 {
11     someDisposableObject.DoTheJob();
12 }
13 finally
14 {
15     someDisposableObject.Dispose();
16 }
```

31.6.3 使用 as 转换对象

对于软件开发人员来说，使用'(T)'做类型转换，而不是使用'as (T)'是很常见的写法。实际上，这样通常不会带来危害，因为多数对象都是可转换的。但是，如果在很低的可能性还是发生的情况下，对象不能转换，那么使用'as (T)'才是正确的。详细请查看 Prefix-casting versus as-casting in C#

```

1 //错误
2 var woman = (Woman)person;
3
4 //正确
5 var woman = person as Woman;
```

31.6.4 枚举转换为字符串

有时需要枚举数据库类型和数据库名称，当枚举数据库名称时，需要的枚举值为字符串类型。但是枚举类型默认为 int 值类型，string 类型无法转换为 int 类型，此时需要使用 Enum.GetName(Type enumType, Object value) 方法。

```

1 enum DatabaseName { System, Weixin};
2
3 public static void Main()
4 {
```

```

5   Console.WriteLine("The system database name is {0}", Enum.GetName(typeof(
6       DatabaseName), 0));
7 }
```

31.6.5 字符串拼接

字符串拼接的写法之一如下：

```

1 string message = string.Concat(new object[]
2 {
3     "Print ID not correct or duplicate, PrintID:",
4     this.printModel.printId,
5     ",Print Guid:",
6     this.printModel.guid
7 });
8 logger.Error(message);
```

31.6.6 得到每天的开始时间结束时间

```

1 //开始时间
2 DateTime.Now.ToString("yyyy-MM-dd 00:00:00");
3 //结束时间
4 DateTime.Now.ToString("yyyy-MM-dd 23:59:59");
```

31.6.7 代码优化 (Code Optimized)

使用 LINQ 替代循环 (Replace Loop with LINQ) 在代码中有时会存在许多循环迭代，使用 Linq 会让代码看起来更加优雅、简洁。不过性能上会有一定影响，如下的代码迭代获取一个 List。LINQ 简称语言集成查询，设计的目的是为了解决在.NET 平台上进行统一的数据查询

```

1 List<tour_roadLineModel> roadLineModels = new List<tour_roadLineModel>();
2 for (int i = 0; i < roadlineIdArr.Length; i++)
3 {
4     roadLineModels.Add(GetRoadLineModel(roadlineIdArr[i]));
5 }
```

可以简写为：

```

1 roadLineModels = roadlineIdArr.Select(t => GetRoadLineModel(t)).ToList();
```

进一步可以简写为 Method Group：

```

1 roadLineModels = roadlineIdArr.Select(GetRoadLineModel).ToList();
```

如下语句：

```

1 var distnateProduct = new List<ProductModel>();
2 foreach (var current in originProduct)
3 {
4     var productModel = new ProductModel();
5     productModel = HttpPost.MapModel(productModel, current);
6     distnateProduct.Add(productModel);
7 }
```

可以转化为：

```

1 var distnateProduct = (from current
2                         in originProduct
3                         let productModel = new ProductModel()
4                         select HttpPost.MapModel(productModel, current)).ToList();
```

减少嵌套 (Reducing Nesting) 在 if 语句中可直接通过返回的方式减少代码的嵌套层数，使代码更加易于阅读，减少代码行数。

```

1 if(condition)
2 {
3     //do something
4 }
5
6 //推荐
7 if(!condition) return;
8 //do something
```

显式指定 Culture(Specify Culture Explicitly) 在进行字符串比较或者字符串输出等操作时，不同的 CultureInfo 会影响某些函数的执行结果，在进行日期时间输出时，.NET 会考虑当前线程的 CultureInfo，即 Thread.CurrentThread.CurrentCulture（或者 CultureInfo.CurrentCulture），并根据 CultureInfo，进行相应地区文化的数据处理。.NET 中有一个特殊的 CultureInfo：InvariantCulture，这个 CultureInfo 有点像英语格式，但它不和国家地区挂钩，它可以提供一个可靠的在多语言环境下的规范格式化。

```

1 int i=6;
2 string str=i.ToString(CultureInfo.InvariantCulture);
```

这样不管客户端运行在什么语言环境下，输出的时间格式都是统一的，方便数据中心服务器对数据做后续处理。默认的 Culture 是 CurrentCulture。

用泛型来代替不同类型的查询 在做数据分页查询时，不同的查询对象对应有一个查询方法，可考虑采用泛型将所有对象的分页查询写在一个方法中，用一个方法去适配所有对象的分页查询，

将大大减少代码量，单个对象单个分页查询如下代码所示：

Listing 31.1: 单个对象分页查询

```

1 #region 分页获取征集图片
2 /// <summary>
3 /// </summary>
4 /// <param name="pageIndex"> 页数 </param>
5 /// <param name="pageSize"> 每页大小 </param>
6 /// <param name="totalCount"> 总数 </param>
7 /// <param name="activityId"> 所属活动 ID</param>
8 /// <returns></returns>
9 public static List<PhotoCollectionModel> GetPhotoByPage(int pageIndex, int pageSize,
    int activityId, out int totalCount)
10 {
11     var filter = "status=1 and isDel=0 and activityId=" + activityId;
12     List<PhotoCollectionModel> photoCollectionModels = null;
13     dynamic photoCollectionBll = new tour_photoCollectionBLL();
14     var photoCollectionTable = photoCollectionBll.GetListByPage(filter, "*", "sort",
        pageSize, pageIndex, out totalCount);
15     if (photoCollectionTable != null && photoCollectionTable.Rows.Count > 0)
16     {
17         photoCollectionModels = HttpPost.ConvertDataTableToList<PhotoCollectionModel>(photoCollectionTable);
18     }
19     return photoCollectionModels;
20 }
21 #endregion

```

采用泛型适配后的方法如下代码所示：

Listing 31.2: 泛型查询方法适配所有对象的分页查询

```

1 #region 分页获取对象信息
2 /// </summary>
3 /// </summary>
4 /// <param name="filter"> 过滤器 </param>
5 /// <param name="sort"> 排序 </param>
6 /// <param name="pageIndex"> 页数 </param>
7 /// <param name="pageSize"> 每页大小 </param>
8 /// <param name="totalCount"> 总数 </param>
9 /// <param name="type">Bll 实例类型 </param>
10 /// <returns></returns>
11 public static List<T> GetObjectByPage<T>(string filter, string sort, int pageIndex, int
    pageSize, Type type, out int totalCount) where T : class, new()
12 {
13     List<T> models = null;
14     dynamic entityBll = Activator.CreateInstance(type);
15     var entityTable = entityBll.GetListByPage(filter, "*", sort, pageSize, pageIndex,
        out totalCount);
16     if (entityTable != null && entityTable.Rows.Count > 0)
17     {

```

```
19     models = HttpPost.ConvertDataTableToList<T>(entityTable);
20 }
21     return models;
22 }
#endregion
```

在创建业务逻辑层的操作类时使用了反射的写法，Activator.CreateInstance 根据类型创建类的实例，Activator.CreateInstance 比直接 new 创建实例耗时在 3 倍左右，100W 次简单对象的构造在 200ms 左右（根据电脑配置会有差异），对性能会有影响，但是可以增加程序的可扩展性，在使用时需要权衡，Activator.CreateInstance 方法采用缓存的方式来提高效率。如下代码片段是对 Activator.CreateInstance 性能的测试：

```
1 var stopwatchNew = new Stopwatch();
2 stopwatchNew.Start();
3 for (var i = 0; i < 1000000; i++)
4 {
5     var bll = new fc_divisionModel();
6 }
7 stopwatchNew.Stop();
8 // Get the elapsed time as a TimeSpan value.
9 var ts = stopwatchNew.Elapsed;
10 // Format and display the TimeSpan value.
11 var elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
12     ts.Hours, ts.Minutes, ts.Seconds,
13     ts.Milliseconds / 10);
14 Console.WriteLine("RunTime using new keywords:" + elapsedTime);
15
16 var stopwatchRef = new Stopwatch();
17 stopwatchRef.Start();
18 for (var i = 0; i < 1000000; i++)
19 {
20     var bll = Activator.CreateInstance(typeof(fc_divisionModel));
21 }
22 stopwatchRef.Stop();
23 // Get the elapsed time as a TimeSpan value.
24 var ts1 = stopwatchRef.Elapsed;
25 // Format and display the TimeSpan value.
26 var elapsedTime1 = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
27     ts1.Hours, ts1.Minutes, ts1.Seconds,
28     ts1.Milliseconds / 10);
29 Console.WriteLine("RunTime using activator:" + elapsedTime1);
30 Console.ReadLine();
31
32 /*
33 RunTime using new keywords:00:00:00:06
34 RunTime using activator:00:00:00:20
35 */
```

经过如上代码的测试，增加程序的可扩展而牺牲的性能在使用 Activator.CreateInstance 是

可以考虑的，实际上 MVC 的激活 Controller 时已经在使用，在 MVC 的原始码中可以看到。使用 C# 的泛型时，在泛型类型的方法或者泛型方法中可能会使用到类型参数的类型的对象的方法、属性或成员，这时候这个类型可能并不存在要使用的方法等。这种情况下就会缺少类型安全。为了改变这种情况，可以对类型参数设置约束。设置约束的另一个作用是在编辑及编译时，使用约束后可以享受具体类型的智能感知及强类型支持，否则将只是 object 级的智能感知。

类型参数约束	代表的含义
where T : struct	T 必须是一个结构类型（值类型）
where T : class	T 必须是一个类（class）类型，引用类型
where T : new()	T 必须要有一个无参构造函数
where T : NameOfBaseClass	T 必须继承名为 NameOfBaseClass 的类
where T : NameOfInterface	T 必须实现名为 NameOfInterface 的接口

动态创建对象

1. Type.InvokeMember
2. ConstructorInfo.Invoke
3. Activator.CreateInstance(Type)
4. Activator.CreateInstance(assemblyName, typeName)
5. Assembly.CreateInstance(typeName)

最快的是方式 3，与 Direct Create 的差异在一个数量级之内，约慢 7 倍的水平。其他方式，至少在 40 倍以上，最慢的是方式 4，要慢三个数量级。

避免隐式转换 (Avoid Implicity Conversion) 隐式转换会发生装箱操作，损耗程序性能，比如函数的参数传入的值是 int 类型，但是在使用的时候为 string 类型，那么就会发生隐式转换，如图31.3所示，从反编译的代码中可以看出，所以应该避免此种写法。

```
GetProduct(int id)
    inject<ProductModel>("/tour/Product/GetProduct/" + (object) id);
```

图 31.3: 避免隐式转换

减少 Switch 条件判断个数 有时会根据条件去判断执行某个方法，如果条件分支过多，相应的 Switch 分支会变多，将对应关系存放在字典中，通过方法激活会更加简捷。也可以通过反射实现。定义一个字典表保存分支和调用方法的对应关系，如下代码所示。

```
1 /// <summary>
2 /// 保存消息类型和对应消息调用的方法
3 /// </summary>
```

```

4 private Dictionary<EnumMessageType, Action<int, Dictionary<string, IPlugin>,
    EnumMessageType, MVSPJSONMessage>> handleAction = new Dictionary<
        EnumMessageType, Action<int, Dictionary<string, IPlugin>, EnumMessageType,
        MVSPJSONMessage>>();

```

31.6.8 表驱动法

31.7 常用资源

31.7.1 常用网站搜藏

表 31.1: 常用网址搜藏

序号	网址	日期	备注
1	http://stackoverflow.com/	2015-04-22	技术问题问答
2	http://www.googto.com/	2015-06-10	Google 镜像
3	https://github.com/	2015-06-10	代码托管
4	http://www.sourceforge.net/	2015-08-10	开源项目托管平台
5	http://www.experts-exchange.com/	2015-08-16	技术问答
6	http://www.cnblogs.com/	2015-08-18	博客园
7	https://news.ycombinator.com/	2015-08-23	Hacker News
8	https://www.gitbook.com/	2015-08-28	Gitbook
9	http://www.gutenberg.org/	2015-08-31	Project Gutenberg
10	http://www.yinwang.org/	2015-09-01	王垠的博客
11	https://duckduckgo.com/	2015-09-01	搜索引擎
12	http://www.ruanyifeng.com/home.html	2015-09-04	阮一峰的个人网站
13	http://www.gnu.org/	2015-09-11	GNU 官方网站
14	http://www.fsf.org/	2015-09-11	FSF 官方网站
15	http://tool.oschina.net/	2015-09-11	在线工具
16	http://www.ted.com/	2015-09-13	Technology, Entertainment, Design
17	http://www.tedx.com/	2015-09-13	Technology, Entertainment, Design
18	http://runjs.cn/	2015-09-18	在线编辑、展示、分享、交流代码
19	http://local.joelonsoftware.com/	2015-09-21	Joel on Software
20	http://www.nodecloud.org/	2015-09-25	NodeJS Rank
21	http://island205.com/	2015-09-27	寸志的博客
22	https://linux.cn/	2015-10-07	开源中文社区
23	http://game-lab.org/	2015-10-07	一个游戏开发小伙的博客
24	http://www.kendraschaefer.com/	2015-10-08	值得阅读的博客
25	http://www.365mini.com/	2015-10-10	Code Player

26	http://www.wufoo.com/html5/	2015-10-15	HTML5
27	http://chromecj.com/	2015-10-21	Chrome Plugin
28	http://www.csszengarden.com/	2015-10-24	CSS 禅意花园
29	http://learnlayout.com/	2015-10-26	学习 Layout 布局
30	http://www.ishadowsocks.com/	2015-10-28	Shadowsocks 免费账号
31	http://html5demos.com/	2015-11-08	HTML5 Demo
32	http://googo.cf/	2015-11-09	Google Mirror
33	http://vim-adventures.com/	2015-11-13	VIM Practice
34	http://opensource.com/	2015-11-15	开源知识站点之一
35	http://referencesource.microsoft.com/	2015-11-18	Microsoft 源码阅读
36	http://wooyun.org/	2015-11-20	乌云安全平台
37	https://github.com/Microsoft	2016-01-09	Microsoft on GitHub
38	http://unbug.github.io/codelf/	2016-01-12	变量命名搜索
39	http://pagespeed.webkaka.com/docs/	2016-01-24	网站速度诊断
40	http://cn.bing.com/	2016-01-12	微软必应搜索
41	http://www.infoq.com/	2016-03-12	软件开发分享
42	http://nickcraver.com/	2016-03-12	Nick Craver 的博客 (StackOverflow)
43	http://www.jobbole.com/	2016-03-12	伯乐在线
44	https://www.darecademy.com/	2016-04-02	大人学
45	http://www.fosshub.com/	2016-04-21	Clean Software Host
46	http://wangjianshuo.com/	2016-04-22	王建硕的博客
47	https://channel9.msdn.com	2016-04-22	第九频道
48	https://www.yandex.com/	2016-07-05	俄罗斯最大的搜索引擎
49	https://freenode.net/	2016-07-08	supporting free and open source communities

31.7.2 资源地址收藏

表 31.2: 常用资源收藏

序号	网址	日期	备注
1	http://mirrors.sohu.com/	2015-04-22	搜狐镜像下载站点
2	http://www.ctex.org/documents/shredder/tex_frame.html		LaTex 宣传页

31.7.3 LDAP(Lightweight Directory Access Protocol)

LDAP 不是個軟體是個協定，由 RFC4511 所定義，他還有個較為复杂的哥哥 DAP (Directory Access Protocol)，以樹狀的架構存放資料，資料的部分可以放帳密、通讯录…… 等等階層式的存放方式。The Lightweight Directory Access Protocol (LDAP; / ldap/) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed

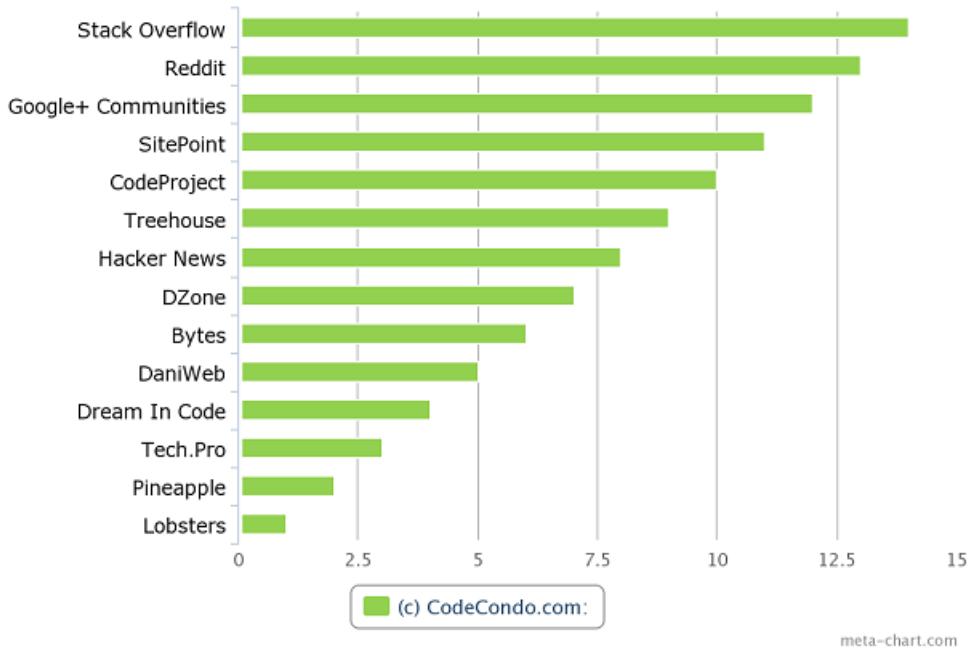


图 31.4: 参考网站

directory information services over an Internet Protocol (IP) network. Directory services play an important role in developing intranet and Internet applications by allowing the sharing of information about users, systems, networks, services, and applications throughout the network.[2] As examples, directory services may provide any organized set of records, often with a hierarchical structure, such as a corporate email directory. Similarly, a telephone directory is a list of subscribers with an address and a phone number. 使用轻量级目录访问协议 (LDAP) 构建集中身份验证系统可以减少管理成本，增强安全性，避免数据复制的问题，并提高数据的一致性。随着 Linux® 的不断成熟，已经出现了很多工具用来简化用户帐号信息到 LDAP 目录的迁移。还开发了一些工具用来在客户机和目录服务器之间启用加密通信配置，并通过复制提供容错性。OpenLDAP 是 LDAP 协议的一个开源实现。LDAP 服务器本质上是一个为只读访问而优化的非关系型数据库。它主要用做地址簿查询（如 email 客户端）或对各种服务访问做后台认证以及用户数据权限管控。（例如，访问 Samba 时，LDAP 可以起到域控制器的作用；或者 Linux 系统认证时代替 /etc/passwd 的作用。）

openldap 常用命令 openldap 默认端口是 389，openldap 常用命令

```

1 #启动 openldap
2 service slapd start
3
4 #验证配置，OpenLDAP 从传统的扁平的配置文件 (slapd.conf) 切换到 OLC 风格的配置文件
5 #并且将是缺省的配置方法。
6 #使用 OLC 风格的配置文件的一大好处是当配置需要被更改时
7 #这一动态的后台配置 (cn=config) 不需要重启服务就可以生效。
8 slaptest -v -d 1 -F /etc/openldap/slapd.d

```

区分名 (DN,Distinguished Name) 和自然界中的树不同，文件系统/LDAP/电话号码簿目录的每一片枝叶都至少有一个独一无二的属性，这一属性可以帮助我们来区别这些枝叶。在文件系统中，这些独一无二的属性就是带有完整路径的文件名。比如/etc/passwd，该文件名在该路径下是独一无二的。当然我们可以有/usr/passwd, /opt/passwd，但是根据它们的完整路径，它们仍然是唯一的。类似于 DNS 系统的 FQDN 正式域名，FQDN 也是唯一的。在 LDAP 中，一个条目的区分名称叫做“dn”或者叫做区分名。在一个目录中这个名称总是唯一的。比如，我的 dn 是”uid=aghaffar, ou=People, o=developer.ch”。不可能有相同的 dn，但是我们可以有诸如”uid=aghaffar, ou=Administrators, o=developer.ch”的 dn。这同上面文件系统中/etc/passwd 和 /usr/passwd 的例子很类似。我们有独一无二的属性，在”ou=Administrators, o=developer.ch”中 uid 和在”ou=People, o=developer.ch”中的 uid。这并不矛盾。

LDAP Account Manager (LAM)

LDAP Account Manager (LAM) is a webfrontend for managing entries (e.g. users, groups, DHCP settings) stored in an LDAP directory. LAM was designed to make LDAP management as easy as possible for the user. It abstracts from the technical details of LDAP and allows persons without technical background to manage LDAP entries. If needed, power users may still directly edit LDAP entries via the integrated LDAP browser.

配置文件的位置：

```
1 vim /var/lib/ldap-account-manager/config/lam.conf
```

Chapter 32

Open Source

32.1 参与开源项目

<https://guides.github.com/activities/contributing-to-open-source/>

32.1.1 Pull Request

首先 Fork 项目代码，将代码克隆到本地：

```
1 git clone https://github.com/jiangxiaoqiang/OpenLiveWriter.git
```


Chapter 33

FAQ

33.1 技术类

33.1.1 抽象类和接口的区别

接口用于规范，抽象类用于共性。接口中只能声明方法，属性，事件，索引器。而抽象类中可以有方法的实现，也可以定义非静态的类变量。抽象类是类，所以只能被单继承，但是接口却可以一次实现多个。抽象类可以提供某些方法的部分实现，接口不可以。抽象类的实例是它的子类给出的。接口的实例是实现接口的类给出的。再抽象类中加入一个方法，那么它的子类就同时有了这个方法。而在接口中加入新的方法，那么实现它的类就要重新编写（这就是为什么说接口是一个类的规范了）。接口成员被定义为公共的，但抽象类的成员也可以是私有的、受保护的、内部的或受保护的内部成员（其中受保护的内部成员只能在应用程序的代码或派生类中访问）。此外接口不能包含字段、构造函数、析构函数、静态成员或常量。

- 接口可以被多重实现，抽象类只能被单一继承
- 抽象基类可以定义字段、属性、方法实现。接口只能定义属性、索引器、事件、和方法声明，不能包含字段
- 抽象类是从一系列相关对象中抽象出来的概念，因此反映的是事物的内部共性；接口是为了满足外部调用而定义的一个功能约定，因此反映的是事物的外部特性
- 一般在几个类都有相同的方法时，可以考虑将这几个方法组合成一个抽象类；一般情况下想上层调用与下层代码实现分离的话，可以使用接口
- interfaces may be multiple-inherited, abstract classes may not (this is probably the key concrete reason for interfaces to exist separately from abstract classes - they permit an implementation of multiple inheritance that removes many of the problems of general MI).
- abstract classes can be inherited without implementing the abstract methods (though such a derived class is abstract itself)
- abstract classes may contain state (data members) and/or implementation (methods) item a class that implements an interface must provide an implementation of all the methods of that interface
- interfaces can have no state or implementation

详细的区别可参见¹。

33.1.2 POST 和 GET 的区别

1. 对于 Get 方式，服务器端用 Request.QueryString 获取变量的值，对于 Post 方式，服务器端用 Request.Form 获取提交的数据

¹<http://stackoverflow.com/questions/761194/interface-vs-abstract-class-general-oo>

2. Get 是向服务器发索取数据的一种请求，而 Post 是向服务器提交数据的一种请求，在 FORM (表单) 中，Method 默认为”GET”
3. 而 HTTP 中的 GET, POST, PUT, DELETE 就对应着对这个资源的查, 改, 增, 删 4 个操作。GET 一般用于获取/查询资源信息, 而 POST 一般用于更新资源信息。
4. 无法使用缓存文件 (更新服务器上的文件或数据库) 时, 使用 POST, GET 请求返回的内容可以被浏览器缓存起来。
5. 向服务器发送大量数据 (POST 没有数据量限制) 时, 使用 POST, GET 提交的数据大小有限制 (因为浏览器对 URL 的长度有限制), 而 POST 方法提交的数据没有限制.
6. 发送包含未知字符的用户输入时, POST 比 GET 更稳定也更可靠
7. GET 提交的数据会放在 URL 之后, 以? 分割 URL 和传输数据, 参数之间以 & 相连, 如 EditPosts.aspx?name=test1&id=123456. GET 请求通过 URL (请求行) 提交数据, 在 URL 中可以看到所传参数, POST 方法是把提交的数据放在 HTTP 包的 Body 中, POST 通过“请求体”传递数据, 参数不会在 URL 中显示.
8. GET 方式提交数据, 会带来安全问题, 比如一个登录页面, 通过 GET 方式提交数据时, 用户名和密码将出现在 URL 上, 如果页面可以被缓存或者其他人可以访问这台机器, 就可以从历史记录获得该用户的账号和密码。HTTP 协议中提到 GET 是安全的方法 (safe method), 其意思是说 GET 方法不会改变服务器端数据, 所以不会产生副作用。

33.1.3 == 和 equals() 的区别

来一段实例代码:

```
1 string a = new string(new char[] { 'h', 'e', 'l', 'l', 'o' });
2 string b = new string(new char[] { 'h', 'e', 'l', 'l', 'o' });
3 Console.WriteLine(a == b); //true
4 Console.WriteLine(a.Equals(b)); //true
5 object g = a;
6 object h = b;
7 Console.WriteLine("g == h:" + (g == h)); //g == h:false
8 Console.WriteLine("g equal h:" + g.Equals(h)); //g equal h:true
9 Person p1 = new Person("jia");
10 Person p2 = new Person("jia");
11 Console.WriteLine("p1 == p2:" + (p1 == p2)); //p1 == p2:false
12 Console.WriteLine("p1 equal p2:" + p1.Equals(p2)); //p1 equal p2:false
13 Person p3 = new Person("jia");
14 Person p4 = p3;
15 Console.WriteLine("p3 == p4:" + (p3 == p4)); //p3 == p4:true
16 Console.WriteLine("p3 equal p4:" + p3.Equals(p4)); //p3 equal p4:true
17 Console.ReadLine();
```

因为值类型是存储在内存中的堆栈 (以后简称栈), 而引用类型的变量在栈中仅仅是存储引用类型变量的地址, 而其本身则存储在堆中。

`==` 操作比较的是两个变量的值是否相等，对于引用型变量表示的是两个变量在堆中存储的地址是否相同，即栈中的内容是否相同。`equals`: 操作表示的两个变量是否是对同一个对象的引用，即堆中的内容是否相同。而字符串是一个特殊的引用型类型，在 C# 语言中，重载了 string 对象的很多方法方法（包括 `equals()` 方法），使 string 对象用起来就像是值类型一样。因此在上面的例子中，第一对输出，字符串 a 和字符串 b 的两个比较是相等的。对于第二对输出 `object g = a` 和 `object h = b`，在内存中两个不同的对象，所以在栈中的内容是不相同的，故不相等。而 `g.equals(h)` 用的是 sting 的 `equals()` 方法故相等。如果将字符串 a 和 b 作这样的修改：`string a="aa"; string b="aa";` 则，`g` 和 `h` 的两个比较都是相等的。这是因为系统并没有给字符串 b 分配内存，只是将“aa”指向了 b。所以 a 和 b 指向的是同一个字符串（字符串在这种赋值的情况下做了内存的优化）。对于 `p1` 和 `p2`，也是内存中两个不同的对象，所以在内存中的地址肯定不相同，故 `p1==p2` 会返回 `false`，又因为 `p1` 和 `p2` 又是对不同对象的引用，所以 `p1.equals(p2)` 将返回 `false`。对于 `p3` 和 `p4`，`p4=p3`，`p3` 将对对象的引用赋给了 `p4`，`p3` 和 `p4` 是对同一个对象的引用，所以两个比较都返回 `true`。

```

1 int age = 25;
2 short newAge = 25;
3 Console.WriteLine(age == newAge);
4 //true
5 Console.WriteLine(newAge.Equals(age));
6 //false
7 Console.ReadLine();

```

`==` 运算符被定义为带两个整形（int）或两个短整型（short）或两个长整形（long）的运算。当 “`==`” 两个参数一个是整形和一个短整型时，编译器会隐式转换 short 为 int，并比较转换后 int 值大小。对于值类型，如果对象的值相等，则相等运算符（`==`）返回 `true`，否则返回 `false`。对于 string 以外的引用类型，如果两个对象引用同一个对象，则 `==` 返回 `true`。对于 string 类型，`==` 比较字符串的值。`==` 操作比较的是两个变量的值是否相等。`equals()` 方法比较的是两个对象的内容是否一致。`equals` 也就是比较引用类型是否是对同一个对象的引用。用 `==` 操作符来比较字符串，这种方法实际上是最不推荐使用的。主要原因是由于这种方法没有在代码中显示的指定使用哪种类型去比较字符串。在 C# 中判断字符串是否相等最好使用 Equals 方法：

```

1 public bool Equals(string value);
2 public bool Equals(string value, StringComparison comparisonType);

```

第一个 Equals 方法（没有 `comparisonType` 这参数）和使用 `==` 操作符的结果是一样的，但好处是，它显式的指明了比较类型。它会按顺序逐字节的去比较字符串。在很多情况下，这正是你所期望的比较类型，尤其是当比较一些通过编程设置的字符串，像文件名，环境变量，属性等。在这些情况下，只要按顺序逐字节的比较就可以了。使用不带 `comparisonType` 参数的 Equals 方法进行比较的唯一一点不好的地方在于那些读你程序代码的人可能不知道你的比较类型是什么。使用带 `comparisonType` 的 Equals 方法去比较字符串，不仅会使你的代码更清晰，还会使你去考虑清楚要用哪种类型去比较字符串。这种方法非常值得你去使用，因为尽管在英语中，按顺序进行的比较和按语言区域进行的比较之间并没有太多的区别，但是在其他的一些语种可能会有很大的不同。如果你忽略了这种可能性，无疑是为你自己在未来的道路上挖了很

多“坑”。

33.1.4 Request.Params[]、Request[]、Request.Form[] 和 Request.QueryString[] 的区别

Request.Form: 获取以 POST 方式提交的数据(接收 Form 提交来的数据), 而 Request.Form 这个 request 的成员函数就不是那么宽泛的取值了, 仅仅是从 form 数组中取值, 也就是页面表单 <form> 标签的输入单元(比如 <input type=text> 传递过来的值。那么通过 form 就无法获取链接字符串中以 index.asp?abc=123 这样的形式传递过来的参数; Request.QueryString: 获取地址栏参数(以 GET 方式提交的数据)。

Request: 包含以上两种方式(包含 QueryString, Form, Cookies, ClientCertificate, ServerVariables, 优先获取 GET 方式提交的数据), 它会在 QueryString、Form、ServerVariable 中都搜寻一遍。而且有时候也会得到不同的结果。如果你仅仅是需要 Form 中的一个数据, 但是你使用了 Request 而不是 Request.Form, 那么程序将在 QueryString、ServerVariable 中也搜寻一遍。如果正好你的 QueryString 或者 ServerVariable 里面也有同名的项, 你得到的就不是你原本想要的值了。可以从 QueryString 中获取包含在 URL 中的一些参数, 可以从 Form 中获取用户输入的表单数据, 可以从 Cookie 中获取一些会话状态以及其它的用户个性化参数信息。除了这三大对象, HttpRequest 还提供 ServerVariables 来让我们获取一些来自于 Web 服务器变量。在 System.Web.HttpRequest。

Listing 33.1: HttpRequest 的实现

```
1 public string this[string] key
2 {
3     get
4     {
5         string text = this.QueryString[key]①;
6         if (text != null)
7         {
8             return text;
9         }
10        text = this.Form[key]②;
11        if (text != null)
12        {
13            return text;
14        }
15        HttpCookie httpCookie = this.Cookies[key]③;
16        if (httpCookie != null)
17        {
18            return httpCookie.Value;
19        }
20        text = this.ServerVariables[key]④;
21        if (text != null)
22        {
23            return text;
24        }
}
```

```

25     return null;
26   }
27 }
```

- ① 从 GET 中寻找 key
- ② 从 POST 中寻找 key
- ③ 从 Cookies 中寻找 key
- ④ 从 ServerVariables 中寻找 key

按照传递数据量来说，Request.Form 可以传递不限大小的数据，而 Request.QueryString 只能传递 2k 的数据量。按照速度来说，Request.QueryString 会略快于 Request.Form。

Request.Params[]、Request[] 这两个属性都可以让我们方便地根据一个 Key 去搜索 QueryString、Form、Cookies 或 ServerVariables 这 4 个集合。通常如果请求是用 GET 方法发出的，那我们一般是访问QueryString 去获取用户的数据，如果请求是用 POST 方法提交的，我们一般使用 Form 去访问用户提交的表单数据。而使用 Params，Item 可以让我们在写代码时不必区分是 GET 还是 POST。这两个属性唯一不同的是：Item 是依次访问这 4 个集合，找到就返回结果，而 Params 是在访问时，先将 4 个集合的数据合并到一个新集合（集合不存在时创建），然后再查找指定的结果。<http://www.cnblogs.com/fish-li/archive/2011/12/06/2278463.html>

33.1.5 HTTP Cookie 和 Session 的区别

HTTP 保持状态主要通过 Cookie、Session。Cookie 比 Session 要早添加到 HTTP 协议中。Cookie 机制采用的是在客户端保存状态的方案，而 session 机制采用的是在服务器端保持状态的方案。

- 1、Cookie 数据存放在客户的浏览器上，Session 数据放在服务器上。
- 2、Cookie 不是很安全，别人可以分析存放在本地的 Cookie 并进行 Cookie 欺骗考虑到安全应当使用 Session。
- 3、Cookie 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能考虑到减轻服务器性能方面，应当使用 Cookie。
- 4、单个 Cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 Cookie。
- 5、所以个人建议：将登陆信息等重要信息存放在 Session 其他信息如果需要保留，可以在 Cookie 中

33.1.6 string.Empty、"" 和 null 的区别

string str="" 声明变量时，和 string str=" " 比较难以区分。string str=null 声明变量时，会出现变量已赋值但仍然出现警告的情况，如图33.1所示，从实现的代码中可以看出 string.Empty

实际就是”，但是为了让代码可读性更高，推荐使用 string.Empty 的写法。

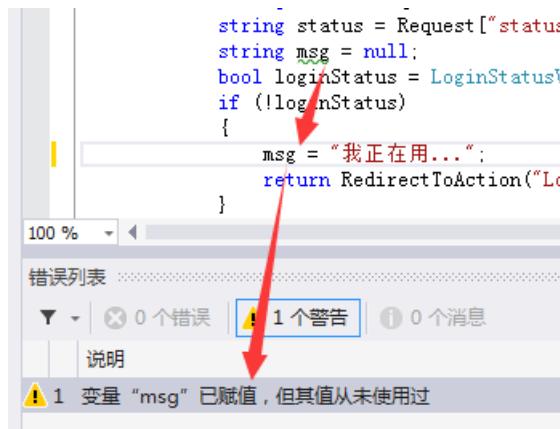


图 33.1: string=null 声明的问题

所以 string.Empty 有让代码好读，防止代码产生歧义的效果。string.Empty 的内部实现代码如片段33.2所示。

Listing 33.2: string.Empty 的内部实现

```

1 public static readonly String Empty;❶
2
3 // The Empty constant holds the empty string value.
4 //We need to call the String constructor so that the compiler doesn't mark this as a literal.
5 //Marking this as a literal would mean that it doesn't show up as a field which we can access
6 //from native.
7 public static readonly String Empty = "";❷

```

- ❶ 定义 Empty 只读静态变量，在 String 类中，String.Empty 为只读的 String 类的成员，此处的静态变量未赋值，那么为默认值 null，如果静态变量为 Int，那么默认值就是 0，如果静态变量为 bool，那么默认值就是 false，这是.NET 4.0 的内部写法
- ❷ string.Empty 在.NET 2.0 下的实现

Listing 33.3: IL 代码

```

1 string str = null;
2 IL_0001: ldnull❶
3
4 string str1 = "";
5 IL_0001: ldstr      ""❷
6
7 string str2 = string.Empty;
8 IL_0001: ldsfld     string [mscorlib]System.String::Empty❸

```

- ❶ 将空引用（O 类型）推送到计算堆栈上 (Load null)

- ② 推送对元数据中存储的字符串的新对象引用 (Load string)
- ③ 将静态字段的值推送到计算堆栈上 (Load static field)

33.1.7 SQL Server 中 exec 和 sp_executesql 的区别

sp_executesql 命令比 EXEC 命令更灵活，因为它提供一个接口，该接口及支持输入参数也支持输出参数。这功能使你可以创建带参数的查询字符串，这样就可以比 EXEC 更好的重用执行计划，sp_executesql 的构成与存储过程非常相似，不同之处在于你是动态构建代码。sp_executesql 的另一个与其接口有关的强大功能是，你可以使用输出参数为调用批处理中的变量返回值。利用该功能可以避免用临时表返回数据，从而得到更高效的代码和更少的重新编译。定义和使用输出参数的语法与存储过程类似。

33.1.8 Local system/Network service/Local Service 的区别

33.1.9 Margin 和 Padding 的区别

W3C 组织建议把所有网页上的对像都放在一个盒 (box) 中，这就是所谓的盒模型 (Box Model) 或者叫做框模型，设计师可以通过创建定义来控制这个盒的属性，这些对像包括段落、列表、标题、图片以及层。盒模型主要定义四个区域：内容 (content)、边框距 (padding，内边距)、边界 (border) 和边距 (margin，外边距)。边框以外是外边距，外边距默认是透明的，因此不会遮挡其后的任何元素。如图33.2所示。对于初学者，经常会搞不清楚 margin, background-color, background-image, padding, content, border 之间的层次、关系和相互影响。这里提供一张盒模型的 3D 示意图33.3，希望便于你的理解和记忆。

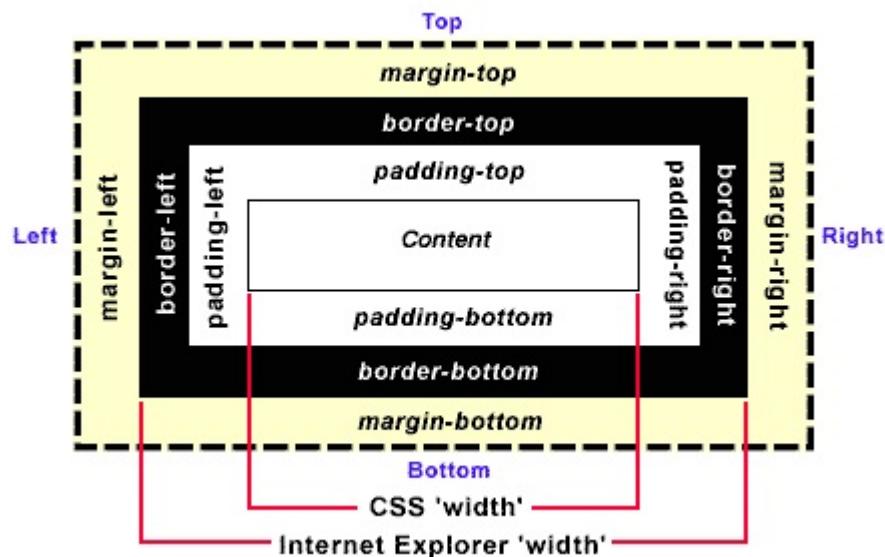


图 33.2: Margin 与 Padding 的详细图示

`margin`: 层的边框以外留的空白，`margin` 简写属性在一个声明中设置所有外边距属性，该属性可以有 1 到 4 个值，，从左到右分别为上、右、下、左。`background-color`: 背景颜

色, background-image: 背景图片, padding: 层的边框到层的内容之间的空白, border: 边框, content: 内容。在 CSS 中, width 和 height 指的是内容区域的宽度和高度。增加内边距、边框和外边距不会影响内容区域的尺寸, 但是会增加元素框的总尺寸。

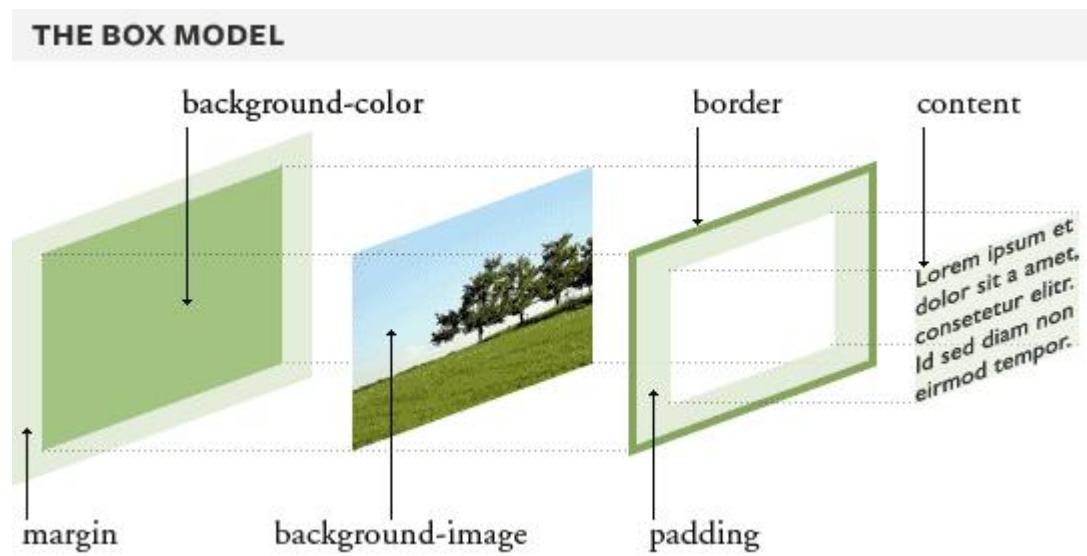


图 33.3: Margin 与 Padding 的 3D 图示

33.1.10 abbsolute 与 relative 区别

position:relative; 如果对一个元素进行相对定位, 首先它将出现在它所在的位置上。然后通过设置垂直或水平位置, 让这个元素“相对于”它的原始起点进行移动。(再一点, 相对定位时, 无论是否进行移动, 元素仍然占据原来的空间。因此, 移动元素会导致它覆盖其他框)

相对定位: relative 没有脱离正常的文档流, 被设置元素相对于其原始位置而进行定位, 其原始占位信息仍存在

position:absolute²; 表示绝对定位, 位置将依据浏览器左上角开始计算。绝对定位使元素脱离文档流, 因此不占据空间。普通文档流中元素的布局就像绝对定位的元素不存在时一样。(因为绝对定位的框与文档流无关, 所以它们可以覆盖页面上的其他元素并通过 z-index 来控制它层级次序。z-index 的值越高, 它显示的越在上层。)

33.1.11 FTP 主动模式 (Active FTP) 和被动模式 (Passive FTP) 区别

主动 FTP 对 FTP 服务器的管理有利, 但对客户端的管理不利。因为 FTP 服务器企图与客户端的高位随机端口建立连接, 而这个端口很有可能被客户端的防火墙阻塞掉。被动 (推荐模式)FTP 对 FTP 客户端的管理有利, 但对服务器端的管理不利。因为客户端要与服务器端建立两个连接, 其中一个连到一个高位随机端口, 而这个端口很有可能被服务器端的防火墙阻塞掉。

²详细信息可参考: <http://zh.learnlayout.com/position.html>

As mentioned in the main text, FTP server admins will almost definitely need to support passive FTP in order to allow the greatest number of clients to access their FTP resources. In order to support passive FTP, however, a large number of high-numbered ports on the server must be opened through a firewall. Luckily, most FTP servers allow this port range to be specified so as to limit exposure to attacks.

幸运的是，有折衷的办法。既然 FTP 服务器的管理员需要他们的服务器有最多的客户连接，那么必须得支持被动 FTP。我们可以通过为 FTP 服务器指定一个有限的端口范围来减小服务器高位端口的暴露。这样，不在这个范围的任何端口会被服务器的防火墙阻塞。虽然这没有消除所有针对服务器的危险，但它大大减少了危险。

33.1.12 const,readonly,static,static readonly 的区别

关于常量的使用，除了会用作一些算法的临时常量值以外，最重要的是定义一些全局的常量，被其他对象直接调用。而集中这些常量最好的类型是 struct (结构)。readonly 所修饰的字段，可以在声明时赋值，也可以在类的构造函数中赋值，但是在构造函数以外的地方不能赋值。使用 static 修饰符声明属于类型本身而不是属于特定对象的静态成员。static 修饰符可用于类、字段、方法、属性、运算符、事件和构造函数，但不能用于索引器、析构函数或类以外的类型。可以这么理解，static 用于指定哪些类型成员是公用的，包括字段、属性、方法、运算符、事件和构造函数；如果一个类被声明为 static，那么此类型不可被实例化，它的所有类型成员都是公用的，也即我们常说的工具类。static readonly 与 const 的应用场景已十分相似，区别在于，static readonly 修饰的是运行时常量，而 const 修饰的是编译时常量；static readonly 可以用于修饰引用型变量，而 const 修饰的引用型变量只能是 string 或 null (其他引用型变量的构造函数初始化是在运行时，而非编译时)。static readonly，在程式中只读，不过它是在运行时计算出其值的，所以还能通过静态构造函数 (Static Constructor Function) 来对它赋值。static 的意义与 const 和 readonly 迥然不同。const 仅用于常量定义，readonly 仅用于变量定义，而 static 则和常量、变量无关，它是指所定义的值与类型有关，而与对象的状态无关。

33.1.13 string 是引用类型还是值类型

C# string 类型是特殊的引用类型，它的实例是只读的。string 是引用类型，不过是不可变的。对字符串做一些操作（比如大小写的转换、`+=`），实际上是重新创建了一个字符串。这也是为什么在做大量字符串拼接的时候要使用 `StringBuilder` 而不用 `+=`。

The `string` type represents a sequence of zero or more Unicode characters. `string` is an alias for `String` in the .NET Framework. Although `string` is a reference type, the equality operators (`==` and `!=`) are defined to compare the values of `string` objects, not references. This makes testing for `string` equality more intuitive.

```

1 string a = "hello";
2 string b = "h";❶
3 b += "ello";// Append to contents of 'b'
```

```

4 Console.WriteLine(a == b);②
5 Console.WriteLine((object)a == (object)b);③

```

- ① when you write this code, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to b. The string "h" is then eligible for garbage collection.
- ② compare the value, output is true.
- ③ compare the address, output is false.

33.1.14 var,object,dynamic 的区别

var 为动态推断类型，net framework 3.5 新出的一个定义变量的类型，其实也就是弱化类型的定义，在代码转移时候确定类型。var 声明的变量在赋值的那一刻，就已经决定了它是什么类型。dynamic 动态类型，net framework 4.0 新出的一个定义变量的类型，不是在编译时候确定实际类型的，而是在运行时。However, operations that contain expressions of type dynamic are not resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time.

33.1.15 typeof 和 GetType() 的区别

typeof: The typeof operator is used to obtain the System.Type object for a type.

运算符，获得某一类型的 System.Type 对象。

```
Type t = typeof(int);
```

GetType: Gets the Type of the current instance.

方法，获取当前实例的类型。

```
int i = 10; Console.WriteLine(i.GetType());
```

在利用多态性时，GetType() 是一个有用的方法，允许根据对象的类型来执行不同的代码，而不是像通常那样，对所有的对象都执行相同的代码，例如，如果一个函数接受一个 object 类型的参数，就可以在遇到某些对象时执行额外的任务，联合使用 GetType() 和 typeof()，就可以进行比较，如下所示：

```

1 if(myobj.GetType() == typeof(MyComplexClass))
2 {
3 }
4 }
```

33.1.16 JavaScript 中 undefined 与 null 的区别

JavaScript 中有 5 种简单数据类型（也称为基本数据类型）：Undefined、Null、Boolean、Number 和 String。还有 1 种复杂数据类型——Object，Object 本质上是由一组无序的名值对组成的。首先，null 像在 Java 里一样，被当成一个对象。但是，JavaScript 的数据类型分成原始类型（primitive）和合成类型（complex）两大类，Brendan Eich 觉得表示“无”的值最好是对象。其次，JavaScript 的最初版本没有包括错误处理机制，发生数据类型不匹配时，往往是自动转换类型或者默默地失败。Brendan Eich 觉得，如果 null 自动转为 0，很不容易发现错误。因此，Brendan Eich 又设计了一个 undefined。null 是一个表示“无”的对象，转为数值时为 0；undefined 是一个表示“无”的原始值，转为数值时为 NaN。undefined 表示“缺少值”，就是此处应该有一个值，但是还没有定义。典型用法是：

- 变量被声明了，但没有赋值时，就等于 undefined。例如获取机顶盒 ID 为：

```

1 var cardId=function GetBoxId(){
2     //Get box id implement
3 }
```

当网页运行在浏览器端时，获取 ID 会失败，需要做如下比较进行判断，执行未获取到 ID 时的处理逻辑：

```

1 if (cardId === undefined) {
2     //Get id failed
3 } else {
4     //Get id success
5 }
```

- 调用函数时，应该提供的参数没有提供，该参数等于 undefined
- 对象没有赋值的属性，该属性的值为 undefined
- 函数没有返回值时，默认返回 undefined

33.1.17 for/foreach/LINQ 的区别

For 和 foreach 差异不大，它们的速度大致相同。但 foreach 循环使用更多的堆栈空间。如果数组必须在每次迭代时都访问，那么 for 循环比 foreach 更快，因为 foreach 使用更多空间，对于集合的遍历，foreach 比 for 更加简洁。对于一个集合，在遍历过程中，如果集合发生变动，foreach 会发生异常，而 for 不会。IEnumarable 接口包含一个抽象的方法 GetEnumerator()，它返回一个可用于循环访问集合的 IEnumarator 对象。IEnumarator 对象是一个真正的集合访问器，没有它，就不能使用 foreach 语句遍历集合或数组，IEnumarator 对象能访问集合中的项。

33.1.18 装箱 (Box) 和拆箱 (Unbox) 的区别

装箱时都会发生什么事情，下面是一行最简单的装箱代码。

```
1 object objValue = 1;
```

这行语句的中间代码如下所示，使用的 ildasm.exe 的版本为 Version 2.0.50727.42。

```
1 // Code size      9 (0x9)
2 .maxstack 1
3 .locals init ([0] object i)①
4 IL_0000: nop
5 IL_0001: ldc.i4.1②
6 IL_0002: box      [mscorlib]System.Int32③
7 IL_0007: stloc.0④
8 IL_0008: ret
```

- ①** 以上三行 IL 表示声明 object 类型的名称为 objValue 的局部变量
- ②** 表示将整型数 9 放到栈顶
- ③** 执行 IL box 指令，在内存堆中申请 System.Int32 类型需要的堆空间
- ④** 弹出堆栈上的变量，将它存储到索引为 0 的局部变量中

以上就是装箱所要执行的操作了，执行装箱操作时不可避免的要在堆上申请内存空间，并将堆栈上的值类型数据复制到申请的堆内存空间上，这肯定是要消耗内存和 cpu 资源的。下面的代码执行一次装箱操作和拆箱操作。

```
1 object objValue = 1;
2 int value = (int)objValue;
```

IL 代码：

```
1 // Code size      16 (0x10)
2 .maxstack 1
3 .locals init ([0] object objValue,
4             [1] int32 'value')
5 IL_0000: nop
6 IL_0001: ldc.i4.1
7 IL_0002: box      [mscorlib]System.Int32
8 IL_0007: stloc.0①
9 IL_0008: ldloc.0②
10 IL_0009: unbox.any [mscorlib]System.Int32③
11 IL_000e: stloc.1④
12 IL_000f: ret
```

- ① 弹出堆栈上的变量，将它存储到索引为 0 的局部变量中
- ② 将索引为 0 的局部变量（即 objValue 变量）压入栈
- ③ 执行 IL 拆箱指令 unbox.any 将引用类型 object 转换成 System.Int32 类型
- ④ 将栈上的数据存储到索引为 1 的局部变量即 value

装箱操作和拆箱操作是要额外耗费 cpu 和内存资源的，所以在.NET 2.0 之后引入了泛型来减少装箱操作和拆箱操作消耗。

33.1.19 程序集的加载机制

CLR (Common Language Runtime) 通过 System.Reflection.Assembly.LoadFrom 和 System.Reflection.Assembly.Load 来主动的加载程序集。前者通过位置而后者则通过唯一标识强命名程序集的 4 个元素来标识程序集。CLR 的加载机制和 Load 方法一致，其内在策略是依次通过版本策略、CodeBase 位置、应用程序域位置和应用程序位置来查找程序集。

没有做强名称签名的程序集：程序的根目录根目录下面，与被引用程序集同名的子目录根目录下面被明确定义为私有目录的子目录在目录中查找的时候，如果 dll 查找不到，则会尝试查找同名的 exe 如果程序集带有区域性，而不是语言中立的，则还会尝试查找以语言区域命名的子目录

33.1.20 CSS 文字大小单位 px、em 的区别

px 像素 (Pixel) 是相对长度单位，像素 px 是相对于显示器屏幕分辨率而言的。(引自 CSS2.0 手册) em 是相对长度单位，相对于当前对象内文本的字体尺寸。如当前对行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸。(引自 CSS2.0 手册)

因此用 px 来定义字体，无法用浏览器字体缩放的功能/* 但是 mac osx 10.9.2 测试了一下，chrome/safari/opera/firefox 都可以放大用 px 定义的字体啊，不知道现在标准是不是变了 */

使用时需要注意：

浏览器的默认：16px=1em。em 的值不是固定的，他会继承父级元素的字体大小。例如 div 标签的嵌套，设置字体大小为 1.2em，则从外层到内层的字体是逐渐变大的，也就是内层字体大小是外层字体的 1.2 倍，内层因继承 div 的字体高而变为了 1em=12px，因此需要重新计算那些被放大的字体的 em 数值，避免字体大小的重复声明；body 选择器中声明 Font-size=62.5%，换算关系就变为了：1em=16px*62.5%；Font-size=63%，可以让汉字在 IE 下正常显示。原因可能是 IE 处理汉字时，对于浮点的取值精确度有限。

33.1.21 为什么要使用 base64 编码

在 email 传输中，加密是肯定的，但是加密的目的不是让用户发送非常安全的 Email。这种加密方式主要就是“防君子不防小人”。即达到一眼望去完全看不出内容即可。基于这个目的

加密算法的复杂程度和效率也就不能太大和太低。和上一个理由类似，MIME 协议等用于发送 Email 的协议解决的是如何收发 Email，而并不是如何安全的收发 Email。因此算法复杂度要小，效率要高，否则因为发送 Email 而大量占用资源，路就有点走歪了。

但是，如果是基于以上两点，那么我们使用最简单的恺撒法即可，为什么 Base64 看起来要比恺撒法复杂呢？这是因为在 Email 的传送过程中，由于历史原因，Email 只被允许传送 ASCII 字符，即一个 8 位字节的低 7 位。因此，如果您发送了一封带有非 ASCII 字符（即字节的最高位是 1）的 Email 通过有“历史问题”的网关时就可能会出现问题。网关可能会把最高位置为 0！很明显，问题就这样产生了！因此，为了能够正常的传送 Email，这个问题就必须考虑！所以，单单靠改变字母的位置的恺撒之类的方案也就不行了。关于这一点可以参考 RFC2046。基于以上的一些主要原因产生了 Base64 编码³。应用于 URL 中，邮件传输。我们知道在计算机中任何数据都是按 ascii 码存储的，而 ascii 码的 128 ~ 255 之间的值是不可见字符。而在网络上交换数据时，比如说从 A 地传到 B 地，往往要经过多个路由设备，由于不同的设备对字符的处理方式有一些不同，这样那些不可见字符就有可能被处理错误，这是不利于传输的。所以就先把数据先做一个 Base64 编码，统统变成可见字符，这样出错的可能性就大降低了。对证书来说，特别是根证书，一般都是作 Base64 编码的，因为它要在网上被许多人下载。电子邮件的附件一般也作 Base64 编码的，因为一个附件数据往往是有不可见字符的。

33.1.22 StringBuilder

1. 固定数量的字符串连接 + 的效率是最高的；
2. 当字符串的数量不固定，并且子串的长度小于 8，用 StringBuiler 的效率高些。
3. 当字符串的数量不固定，并且子串的长度大于 8，用 List<string> 的效率高些。

List<string> 它可以转换为 string[] 后使用 string.Concat 或 string.Join，很多时候效率比 StringBuiler 更高效。List 与 StringBuilder 采用的是同样的动态集合算法，时间复杂度也是 O(n)，与 StringBuilder 不同的是：List 的 n 是字符串的数量，复制的是字符串的引用；StringBuilder 的 n 是字符串的长度，复制的数据。不同的特性决定的它们各自的适应环境，当子串比较大时建议使用 List<string>，因为复制引用比复制数据划算。而当子串比较小，比如平均长度小于 8，特别是一个一个的字符，建议使用 StringBuilder。

33.1.23 C# 中 string 和 String 有什么区别

在.NET 中 string 是 String 类型的一个别名，而 object 就是 Object 的一个别名，他们是相同的，编码中要么全用 string，要么全用 String。

33.1.24 IIS 何时会重启

无外乎下面几种情况：

³详细可参考：<http://www.ruanyifeng.com/blog/2008/06/base64.html>

- web.config 被修改
- 即时编译的 *.cs 文件被修改 (一般在 App_Code 中)
- Bin 目录被更新
- *.aspx, *.ascx, *.master 文件被修改的个数超过阀值
- 长时间的 Idle 超过阀值

33.1.25 ASP.NET 网站第一次访问速度慢原因

使用 ASP.NET 开发的网站第一次打开网站首页时特别慢，打开之后无论点哪个页面都很快。长时间无人访问，又会变慢，这对搜索引擎很不友好。如果第一次访问恰巧是搜索引擎抓取页面，由于相应时间过长，会被认为站点有问题，速度不稳定，严重的会影响排名。第一次向 IIS7 请求 ASP.NET 网页时，IIS 会启动 w3wp.exe 进程，同时在 C:/WINDOWS/Microsoft.NET/Framework/v2.0.50727/Temporary ASP.NET Files (也可在 IIS 中自定义该目录) 建立一个文件夹，并将网站相关的 DLL 等资源复制到该目录下（该步骤会占用一部分时间，我们成为 PTime-1），然后 JIT 来运行这些 DLL 文件（该步骤也会占用一部分时间，我们成为 PTime-2），最后将运行结果发送给客户端。第一次打开 aspx 页面变慢主要的时间是 (PTime-1 + PTime-2)，其它的时间可以忽略不计。如果网站内容不更新（专指 DLL 更新，aspx、ascx 等没有影响），网站重启（开机重启、超时回收重启、任务管理器中关闭 w3wp.exe 进程、执行 IISRESET 等），只有 PTime-2，没有 PTime-1。一旦内容不更新，下次再请求这个网页时又会重新创建一个临时目录，花费的时间是 (PTime-1 + PTime-2)。IIS 在这期间做了哪些工作呢？当向服务器发送一个 ASP.NET 网页的请求时，在内部执行的流程如下：

1. 检查在服务器内存中是否存在这个网页对应的本地机器代码
2. 如果存在，则执行本地机器代码，将运行结果发送到客户端
3. 如果不存在本地机器代码，则检查是否存在这个网页编译后的中间代码（以 DLL 的形式存在），如果存在，则将中间代码编译成本地机器代码，再执行本地机器代码将结果返回给客户端，并将本地机器代码缓存到内存中

如果服务器没有清理掉内存中缓存的本地机器代码而再次访问同一个页面时，服务器就会直接执行缓存中的本地机器代码，本地机器代码的运行速度是相当快的。即使本地机器代码在内存中由于某些原因被清理掉了（比如服务器重启或者服务器内存不够用时被释放），也仅仅是将中间代码编译成本地机器代码，再次运行本地机器代码将运行结果发送给客户端，然后缓存本地机器代码即可，将中间代码编译成本地机器代码的速度也是很快的。如果站点中有 HTML 等静态网页，则不需要 PTime-1、PTime-2 两个过程，执行速度都是很快的。如何解决首次加载慢的问题，目前没有好的办法，以下方案可供参考：1、延长 IIS 回收时间 2、通过百度站长等工具，设定搜索引擎抓取速度 3、做一个浏览器模拟程序，定时访问网页 4、升级到 Visual Studio 2012

但无论如何，建议每次更新网站内容后，自己立即访问一下网站。

33.1.26 ADO 和 ADO.NET 的区别

ADO 与 ADO.NET 既有相似也有区别，他们都能够编写对数据库服务器中的数据进行访问和操作的应用程序，并且易于使用、高速度、低内存支出和占用磁盘空间较少，支持用于建立基于客户端/服务器和 Web 的应用程序的主要功能。但是 ADO 使用 OLE DB 接口并基于微软的 COM 技术，而 ADO.NET 拥有自己的 ADO.NET 接口并且基于微软的.NET 体系架构。众所周知.NET 体系不同于 COM 体系，ADO.NET 接口也就完全不同于 ADO 和 OLE DB 接口，这也就是说 ADO.NET 和 ADO 是两种数据访问方式。

- ODBC – (Open Database Connectivity) 是第一个使用 SQL 访问不同关系数据库的数据访问技术。使用 ODBC 应用程序能够通过单一的命令操纵不同的数据库，而开发人员需要做的仅仅只是针对不同的应用加入相应的 ODBC 驱动。ODBC (Open Database Connectivity, 开放数据库互连) 提供了一种标准的 API (应用程序编程接口) 方法来访问 DBMS(Database Management System)。这些 API 利用 SQL 来完成其大部分任务。ODBC 本身也提供了对 SQL 语言的支持，用户可以直接将 SQL 语句送给 ODBC。ODBC 的设计者们努力使它具有最大的独立性和开放性：与具体的编程语言无关，与具体的数据系统无关，与具体的操作系统无关。微软公司在 1993 年以 DLL 集的方式发布了世界上第一个 ODBC 产品，现在成为了微软开放服务结构 (WOSA, Windows Open Services Architecture) 中，有关数据库的一个组成部分。微软的 ODBC 产品其实就是一个 ODBC 的驱动管理器，提供一个 ODBC 应用程序到某种 ODBC 驱动的接口。在 UNIX 系统上，有两个开源的 ODBC 驱动管理器，unixODBC 和 iODBC。

ODBC(Open Database Connectivity, 开放数据库互连)是微软公司开放服务结构 (WOSA, Windows Open Services Architecture) 中有关数据库的一个组成部分，它建立了一组规范，并提供了一组对数据库访问的标准 API (应用程序编程接口)。这些 API 利用 SQL 来完成其大部分任务。ODBC 本身也提供了对 SQL 语言的支持，用户可以直接将 SQL 语句送给 ODBC。

一个基于 ODBC 的应用程序对数据库的操作不依赖任何 DBMS，不直接与 DBMS 打交道，所有的数据库操作由对应的 DBMS 的 ODBC 驱动程序完成。也就是说，不论是 FoxPro、Access 还是 Oracle 数据库，均可用 ODBC API 进行访问。由此可见，ODBC 的最大优点是能以统一的方式处理所有的数据库。

- DAO - (Data Access Objects) 不像 ODBC 那样是面向 C/C++ 程序员的，它是微软提供给 Visual Basic 开发人员的一种简单地数据访问方法，用于操纵 Access 数据库。
- RDO - 在使用 DAO 访问不同的关系型数据库的时候，Jet 引擎不得不在 DAO 和 ODBC 之间进行命令的转化，导致了性能的下降，而 RDO (Remote Data Objects) 的出现就顺理成章了。
- OLE DB - 随着越来越多的数据以非关系型格式存储，需要一种新的架构来提供这种应用和数据源之间的无缝连接，基于 COM (Component Object Model) 的 OLE DB 应运而生了。

- ADO – 基于 OLE DB 之上的 ADO 更简单、更高级、更适合 Visual Basic 程序员，同时消除了 OLE DB 的多种弊端，取而代之是微软技术发展的趋势。

1、两者都将数据保存在内存中，但 ado.net 以 dataset 数据集的形式存放，而 ado 以 recordset 记录集的形式存放。

2、在 ado 中，记录集以单表的形式表现。而在 ado.net 中，数据集以一个表或多个表的形式表现。

3、ado 中，与数据库连接后，会一直保持连接，直到断开（它为连接而设计）。而在 ado.net 中，读取完需要的数据后，会自动断开连接。

当需要更新数据时，会再次连接。

4、ado.net 基于 xml 流传送数据，对数据类型没有限制。

5、ado.net 相对于 ado 减少了大量的数据转换，提高了性能。

6、ado.net 通过 xml 流，可以穿透防火墙

在开始设计.NET 体系架构时，微软就决定重新设计数据访问模型，以便能够完全的基于 XML 和离线计算模型。两者的区别主要有：

ADO 以 Recordset 存储，而 ADO.NET 则以 DataSet 表示。Recordset 看起来更像单表，如果让 Recordset 以多表的方式表示就必须在 SQL 中进行多表连接。反之，DataSet 可以是多个表的集合。ADO 的运作是一种在线方式，这意味着不论是浏览或更新数据都必须是实时的。ADO.NET 则使用离线方式，在访问数据的时候 ADO.NET 会利用 XML 制作数据的一份副本，ADO.NET 的数据库连接也只有在这段时间需要在线。

由于 ADO 使用 COM 技术，这就要求所使用的数据类型必须符合 COM 规范，而 ADO.NET 基于 XML 格式，数据类型更为丰富并且不需要再做 COM 编排导致的数据类型转换，从而提高了整体性能。

33.1.27 Javascript 如何判断是否是对象

`typeof bar === "object"` 并不能准确判断 bar 就是一个 Object。在使用 `typeof` 运算符时采用引用类型存储值会出现一个问题，无论引用的是什么类型的对象，它都返回”object”。可以通过

```
1 Object.prototype.toString.call(bar) === "[object Object]"
```

来避免这种弊端：

```
1 let obj = {};
2 let arr = [];
3
4 console.log(Object.prototype.toString.call(obj)); // [object Object]
5 console.log(Object.prototype.toString.call(arr)); // [object Array]
6 console.log(Object.prototype.toString.call(null)); // [object Null]
```

33.1.28 什么是委托？什么是事件？为什么要用委托？

C# 中的委托相当于 C++ 中的函数指针（如果之前学过 C++ 就知道函数指针是个什么概念的了），函数指针是用指针获取一个函数的入口地址，然后通过这个指针来实现对函数的操作。C# 中的委托相当于 C++ 中的函数指针，也就说两者是有区别的：委托是面向对象的，类型安全的，是引用类型（开始就说了委托是个类），所以在使用委托时首先要定义——> 声明——> 实例化——> 作为参数传递给方法——> 使用委托。

```
1 //定义
2 public delegate void InvokeDateComplete();
3 //声明
4 FrmCompleteList.InvokeDateComplete delegateInstance;
5 //实例化，作为参数传递给方法
6 delegateInstance = FrmCompleteList.LoadGridViewData;
7 //使用委托
8 delegateInstance();
```

引入委托后，编程人员可以把方法的引用封装在委托对象中（把过程的调用转化为对象的调用，充分体现了委托加强了面向对象编程的思想。），然后把委托对象传递给需要引用方法的代码，这样在编译的过程中我们并不知道调用了哪个方法，这样一来，C# 引入委托机制后，使得方法声明和方法实现的分离，充分体现了面向对象的编程思想。

C# 里面事件就是回调函数，要先声明一个代理（也就是定义一个函数指针说明接受函数的类型，参数列表等等），然后根据这个代理再定义一个事件。但是在.NET 里面一个事件可以对应多个订阅源，就是一个事件调用 N 个回调函数，这个和 MFC 里面有很大不同可以理解成一个事件就是一个 CALLBACK 列表，一旦回调就挨个调用就行了。

- 事件是委托的封装，可以理解为一种特殊的委托
- 事件里面其实就两个方法（即 add_event() 和 remove_event()）和一个私有的委托变量，这两个方法里面分别是对于这个私有的委托变量进行的合并和移除，当调用事件的 += 时其实是调用的事件里面的 add_event() 方法，同样 -= 调用的是 remove_event() 方法
- 事件只能够从对象外部增加新的响应方法和删除已知的响应方法，而不能主动去触发事件和获取其他注册的响应方法等信息。如果使用公有的 delegate 则不能做这些限制，也就是说事件对委托做了限制，使委托使用起来更加方便。也有人说事件是对委托的阉割，大概也是这个意思。

33.1.29 localhost、127.0.0.1、本机 IP 的区别

localhost 不联网不使用网卡，不受防火墙和网卡限制本机访问

127.0.0.1 不联网网卡传输，受防火墙和网卡限制本机访问

本机 IP 联网网卡传输，受防火墙和网卡限制本机或外部访问

33.1.30 Struct 和 Class 的区别

33.1.31 Javascript 中 Window 和 window 的区别

Window 是一个函数，window 是一个对象，在 FireFox 菜单-> 开发者->Web 控制台中打开浏览器 Console 界面，也可以使用 Ctrl+Shift+K 快捷键打开，输入 Window 和 window 命令如图33.4所示。Window.window returns an only read reference to the current window. The window object represents a window containing a DOM document; the document property points to the DOM document loaded in that window. A window for a given document can be obtained using the document.defaultView property.

The window object implements the Window interface, which in turn inherits from the AbstractView interface. Some additional global functions, namespaces, objects, interfaces, and constructors, not typically associated with the window, but available on it, are listed in the JavaScript Reference and DOM Reference.

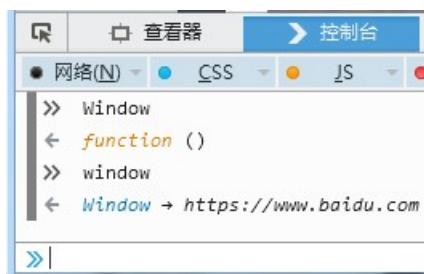


图 33.4: Window 和 window 的区别

In a tabbed browser, such as Firefox, each tab contains its own window object (and if you're writing an extension, the browser window itself is a separate window too - see Working with windows in chrome code for more information). That is, the window object is not shared between tabs in the same window. Some methods, namely window.resizeTo and window.resizeBy apply to the whole window and not to the specific tab the window object belongs to. Generally, anything that can't reasonably pertain to a tab pertains to the window instead.

33.1.32 BOM 和 DOM 的区别

Javascript 是通过访问 BOM (Browser Object Model) 对象来访问、控制、修改客户端 (浏览器)，由于 BOM 的 window 包含了 document，window 对象的属性和方法是直接可以使用而且被感知的，因此可以直接使用 window 对象的 document 属性，通过 document 属性就可以访问、检索、修改 XHTML 文档内容与结构。因为 document 对象又是 DOM (Document Object Model) 模型的根节点，如图33.5所示。

可以说，BOM 包含了 DOM(对象)，浏览器提供出来给予访问的是 BOM 对象，从 BOM

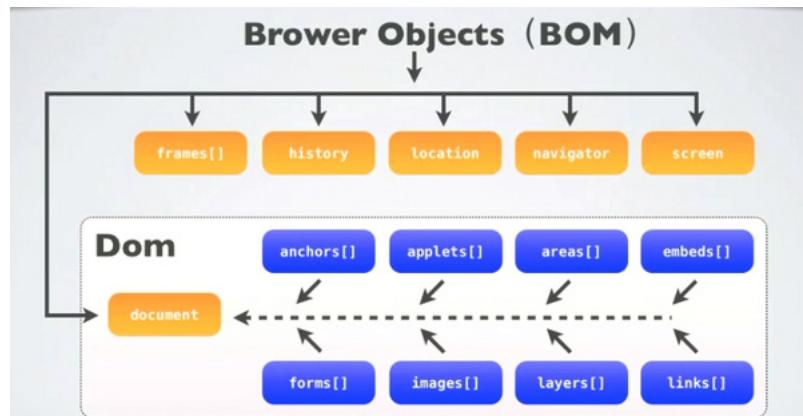


图 33.5: BOM 和 DOM 的关系

对象再访问到 DOM 对象，从而 js 可以操作浏览器以及浏览器读取到的文档。其中 DOM 包含: window, Window 对象包含属性: document、location、navigator、screen、history、frames。Document 根节点包含子节点: forms、location、anchors、images、links。从 window.document 已然可以看出，DOM 的最根本的对象是 BOM 的 window 对象的子对象。区别: DOM 描述了处理网页内容的方法和接口，BOM 描述了与浏览器进行交互的方法和接口。

33.1.33 internal 和 protect internal

被 internal 修饰的东西只能在本程序集（当前项目）内被使用，被 protected internal 修饰的属性/方法可以在其他项目中，被派生类使用。

33.1.34 IEnumarable 和 string[] 的区别

33.1.35 foreach 和 List<T>.Foreach 的区别

List<T>.Foreach 允许遍历的元素被修改，其内部是通过 for 循环来实现，ForEach method doesn't use the enumerator, it loops through the items with a for loop。List<T>.Foreach 会造成可读性上面的混淆不清，难于调试，which uses almost exactly the same characters in slightly different order. And yet the second version is harder to understand, harder to debug, and introduces closure semantics, thereby potentially changing object lifetimes in subtle ways.

When we provide two subtly different ways to do exactly the same thing, we produce confusion in the industry, we make it harder for people to read each other's code, and so on. Sometimes the benefit added by having two different textual representations for one operation (like query expressions versus their underlying method call form, or + versus String.Concat) is so huge that it's worth the potential confusion. But the compelling benefit of query expressions is their readability; this new form of "foreach" is certainly no more readable than the "normal" form and is arguably worse. 当需要执行的代码量较短时，比如仅仅调取一个方法，使用 List<T>.Foreach 是非常不错的选择，因为代码只有一行而 foreach 有 3 行。

33.1.36 HashTable Hashmap 区别

首先 C# 只有 Hashtable,Hashtable 表示键/值对的集合，这些键/值对根据键的哈希代码进行组织。C# 中没有 HashMap, 而 HashMap 是 Java1.2 引进的 Map interface 的一个实现。

33.1.37 3 种主流 Web 开发架构

WEB 应用开发中每种技术独有的资源组织形式（包括文件，数据库，HTTP 请求处理等）。

基于“WEB 页面/文件” 例如 CGI 和 PHP/ASP 程序。程序的文件分别存储在不同的目录里，与 URL 相对应。当 HTTP 请求提交至服务器时，URL 直接指向某个文件，然后由该文件来处理请求，并返回响应结果。

基于“动作”（Action） 这是 MVC 架构的 WEB 程序所采用的最常见的方式。目前主流的 WEB 框架像 Struts、Webwork(Java), Ruby on Rails(Ruby), Zend Framework(PHP) 等都采用这种设计。URL 映射到控制器 (controller) 和控制器中的动作 (action)，由 action 来处理请求并输出响应结果。这种设计和上面的基于文件的方式一样，都是请求/响应驱动的方案，离不开 HTTP。

基于“组件”（Component，GUI 设计也常称控件）、事件驱动的架构 最常见的是微软的.NET。基本思想是把程序分成很多组件，每个组件都可以触发事件，调用特定的事件处理器来处理（比如在一个 HTML 按钮上设置 onClick 事件链接到一个 PHP 函数）。这种设计远离 HTTP，HTTP 请求完全抽象，映射到一个事件。

事实上这种设计原本最常应用于传统桌面 GUI 程序的开发，例如 Delphi, Java Swing 等。所有表现层的组件比如窗口，或者 HTML 表单都可以由 IDE 来提供，我们只需要在 IDE 里点击或拖动鼠标就能够自动添加一个组件，并且添加一个相应的事件处理器。

33.1.38 ViewBag 和 ViewData 的区别

ViewData 和 ViewBag 的简单对比如表33.1所示：

ViewBag 和 ViewData 只在当前 Action 中有效，等同于 View。ViewBag 是动态类型，使用时直接添加属性赋值即可，如 ViewBag.myName。ViewData 为 object 型，而 ViewBag 为 dynamic 型。而 dynamic 型与 object 型的区别则是在使用时它会自动根据数据类型转换，而 object 型则需要我们自己去强制转换。比如上面我们遍历 ViewBag.Items 时，它自动根据数据类型转换，而 ViewData 则需要我们强制转换。

表 33.1: 修改记录

ViewData	ViewBag
它是 Key/Value 字典集合	它是 dynamic 类型对像
从 ASP.NET MVC 1 就有了	ASP.NET MVC3 才有
基于 ASP.NET 3.5 framework	基于 Asp.net 4.0 与 .net framework
ViewData 比 ViewBag 快	ViewBag 比 ViewData 慢
在 ViewPage 中查询数据时需要转换合适的类型	在 ViewPage 中查询数据时不需要类型转换
有一些类型转换代码	可读性更好

33.1.39 常用数据结构及复杂度

Array 访问一个数组元素的时间复杂度为 $O(1)$ ，因此对数组的访问时间是恒定的。也就是说，与数组中包含的元素数量没有直接关系，访问一个元素的时间是相同的。

Array ($T[]$)

ArrayList Array 在存储值类型时是采用未装箱 (unboxed) 的方式。由于 ArrayList 的 Add 方法接受 object 类型的参数，导致如果添加值类型的值会发生装箱 (boxing) 操作。这在频繁读写 ArrayList 时会产生额外的开销，导致性能下降。

当元素的数量是固定的，并且需要使用下标时。Linked list ($LinkedList<T>$)

当元素需要能够在列表的两端添加时。否则使用 List $<T>$ 。Resizable array list ($List<T>$)

当元素的数量不是固定的，并且需要使用下标时。Stack ($Stack<T>$)

当需要实现 LIFO (Last In First Out) 时。Queue ($Queue<T>$)

当需要实现 FIFO (First In First Out) 时。Hash table ($Dictionary<K,T>$)

当需要使用键值对 (Key-Value) 来快速添加和查找，并且元素没有特定的顺序时。Tree-based dictionary ($SortedDictionary<K,T>$)

当需要使用价值对 (Key-Value) 来快速添加和查找，并且元素根据 Key 来排序时。Hash table based set ($HashSet<T>$)

当需要保存一组唯一的值，并且元素没有特定顺序时。Tree based set ($SortedSet<T>$)

当需要保存一组唯一的值，并且元素需要排序时。

33.1.40 进程 (Process) 与线程 (Thread) 的区别

计算机的核心是 CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，单个 CPU 一次只能运行一个任务。操作系统的设计，因此可以归结为三点：

- 以多进程形式，允许多个任务同时运行
- 以多线程形式，允许单个任务分成不同的部分运行
- 提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源

在使用 Visual Studio 调试网站的时候，打开 Visual Studio 的进程窗口（调试-> 窗口-> 进程），也可以使用快捷键 Ctrl+D(Debug)+P(Process) 打开，如图33.6所示：

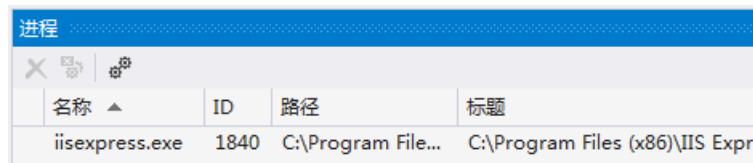


图 33.6: ASP.NET MVC 网站运行的进程

可以看到调试进程的 ID(PID) 为 1840，在 Windwos 任务管理器中也可以看到 iisexpress 进程的 ID 为 1840。然后再打开 Visual Studio 调试的线程窗口（调试-> 窗口-> 线程），也可以使用快捷键 Ctrl+D(Debug)+T(Thread)，可以查看当前进程中包含的所有线程，如图33.7所示

	ID	托管 ID	类别	名称
▲ 进程 ID: 1840 (46 线程)				
▼	6060	0	主线程	主线程
▼	6556	0	工作线程	ntdll.dll!_DbgUiRemoteBreakin@40
▼	7284	0	工作线程	nativrd2.dll!NOTIFICATION_THREAD::ThreadProc()
▼	7352	0	工作线程	ntdll.dll!_TppWaiterpThread@40
▼	7396	0	工作线程	w3tp.dll!THREAD_MANAGER::ThreadManagerThread()
▼	7400	0	工作线程	w3tp.dll!THREAD_MANAGER::ThreadManagerThread()
▼	7404	0	工作线程	w3tp.dll!THREAD_MANAGER::ThreadManagerThread()
▼	7408	0	工作线程	w3tp.dll!THREAD_MANAGER::ThreadManagerThread()
▼	7412	0	工作线程	ntdll.dll!_TppWorkerThread@40

图 33.7: ASP.NET MVC 网站进程中运行的线程

可以看到此时进程包含了 46 个线程。进程和线程都包含有同样的状态，当进程所需的资源没有到位时会是阻塞状态，当进程所需的资源到位时但 CPU 没有到位时是就绪状态，当进程既有所需的资源，又有 CPU 时，就为运行状态。进程都有自己独立的内存地址空间，而线程共享进程的内存地址空间。在运行于 32 位处理器上的 32 位 Windows 操作系统中，可将一个进程视为一段大小为 4GB (232 字节) 的线性内存空间，它起始于 0x00000000 结束于 0xFFFFFFFF。一个线程包含以下内容。一个指向当前被执行指令的指令指针；一个栈；一个寄存器值的集合，定义了一部分描述正在执行线程的处理器状态的值；一个私有的数据区。线程 (Thread) 是进程的一个实体，是 CPU 调度和分派的基本单位。进程是操作系统分配资源的单位。

33.1.41 Form 表单中 onclick/submit/onsubmit 的关系

这几个函数的执行顺序：

- 1) onclick: Y();
- 2) onsubmit: X();
- 3) submit();

也就是说，只要 onclick 未 return false 那么就继续执行 onsubmit，只要 onsubmit 未 return false 那么表单就被提交出去了。

onsubmit: You can override this event by returning false in the event handler. Use this capability to validate data on the client side to prevent invalid data from being submitted to the server. If the event handler is called by the onsubmit attribute of the form object, the code must explicitly request the return value using the return function, and the event handler must provide an explicit return value for each possible code path in the event handler function. The submit method does not invoke the onsubmit event handler.

submit: The submit method does not invoke the onsubmit event handler. Call the onsubmit event handler directly. When using Microsoft? Internet Explorer 5.5 and later, you can call the fireEvent method with a value of onsubmit in the sEvent parameter.

33.1.42 Model, ORM, DAO 和 Active Record 的区别

DAO DAO 是一个软件设计的指导原则，在核心 J2EE 模式中是这样介绍 DAO 模式的：为了建立一个健壮的 J2EE 应用，应该将所有对数据源的访问操作抽象封装在一个公共 API 中。用程序设计的语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口在逻辑上对应这个特定的数据存储。顾名思义就是与数据库打交道，夹在业务逻辑与数据库资源中间，是 DAL 的具体实现。简单的说 Dao 层就是对数据库中数据的增删改查等操作封装在专门的类里面，在业务逻辑层中如果要访问数据的时候，直接调用该 dao 类（包括了如何访问数据库和数据的增删改查等等代码），就可以返回数据，而不需要再在业务逻辑层中写这些代码。

ORM ORM 也是一种对数据库访问的封装，然而 ORM 不像 DAO 只是一种软件设计的指导原则，强调的是系统应该层次分明，更像是一种工具，有着成熟的产品，比如 JAVA 界非常有名的 Hibernate，以及很多 PHP 框架里自带的 ORM 库。他们的好处在于能将你程序中的数据对象自动地转化为关系型数据库中对应的表和列，数据对象间的引用也可以通过这个工具转化为表之间的 JOIN。使用 ORM 的好处就是使得你的开发几乎不用接触到 SQL 语句。创建一张表，声明一个对应的类，然后你就只用和这个类的实例进行交互了，至于这个对象里的数据该怎么存储又该怎么获取，通通不用关心。

33.1.43 TCP 长连接与短连接的区别

TCP 短连接 模拟一下 TCP 短连接的情况，client 向 server 发起连接请求，server 接到请求，然后双方建立连接。client 向 server 发送消息，server 回应 client，然后一次读写就完成了，这时候双方任何一个都可以发起 close 操作，不过一般都是 client 先发起 close 操作。为什么呢，一般的 server 不会回复完 client 后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在 client/server 间传递一次读写操作

短连接的优点是：管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段

TCP 长连接 接下来我们再模拟一下长连接的情况，client 向 server 发起连接，server 接受 client 连接，双方建立连接。Client 与 server 完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

首先说一下 TCP/IP 详解上讲到的 TCP 保活功能，保活功能主要为服务器应用提供，服务器应用希望知道客户主机是否崩溃，从而可以代表客户使用资源。如果客户已经消失，使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，则服务器将永远等待客户端的数据，保活功能就是试图在服务器端检测到这种半开放的连接。

33.1.44 HTTP 的长连接和短连接

HTTP1.1 规定了默认保持长连接（HTTP persistent connection，也有翻译为持久连接），数据传输完成了保持 TCP 连接不断开（不发 RST 包、不四次握手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。

HTTP 首部的 Connection: Keep-alive 是 HTTP1.0 浏览器和服务器的实验性扩展，当前的 HTTP1.1 RFC2616 文档没有对它做说明，因为它所需要的功能已经默认开启，无须带着它，但是实践中可以发现，浏览器的报文请求都会带上它。如果 HTTP1.1 版本的 HTTP 请求报文不希望使用长连接，则要在 HTTP 请求报文首部加上 Connection: close。《HTTP 权威指南》提到，有部分古老的 HTTP1.0 代理不理解 Keep-alive，而导致长连接失效：客户端-> 代理-> 服务端，客户端带有 Keep-alive，而代理不认识，于是将报文原封不动转给了服务端，服务端响应了 Keep-alive，也被代理转发给了客户端，于是保持了“客户端-> 代理”连接和“代理-> 服务端”连接不关闭，但是，当客户端第发送第二次请求时，代理会认为当前连接不会有请求了，于是忽略了它，长连接失效。书上也介绍了解决方案：当发现 HTTP 版本为 1.0 时，就忽略 Keep-alive，客户端就知道当前不该使用长连接。其实，在实际使用中不需要考虑这么多，很多时候代理是我们自己控制的，如 Nginx 代理，代理服务器有长连接处理逻辑，服务端无需做 patch 处理，常见的是客户端跟 Nginx 代理服务器使用 HTTP1.1 协议 & 长连接，而 Nginx 代理服务器跟后端服务器使用 HTTP1.0 协议 & 短连接。

在实际使用中，HTTP 头部有了 Keep-Alive 这个值并不代表一定会使用长连接，客户端和服务器端都可以无视这个值，也就是不按标准来，譬如我自己写的 HTTP 客户端多线程去下载文件，就可以不遵循这个标准，并发的或者连续的多次 GET 请求，都分开在多个 TCP 通道中，每一条 TCP 通道，只有一次 GET，GET 完之后，立即有 TCP 关闭的四次握手，这样写代码

更简单，这时候虽然 HTTP 头有 Connection: Keep-alive，但不能说是长连接。正常情况下客户端浏览器、web 服务端都有实现这个标准，因为它们的文件又小又多，保持长连接减少重新开 TCP 连接的开销很有价值。

33.1.45 为什么使用强名称（强命名程序集）

强名称 (Strong Name) 由一个程序集的标识组成并通过公钥和数字签名（针对该程序集生成）加强的名称，其中的标识包括程序集的简单文本名称、版本号和区域性信息（如果提供的话）。在 Windows 下开发程序时常会遭遇著名的“DLL Hell”问题，即动态链接库的向后兼容问题。微软在 .Net 产生前曾尝试使用 COM 组件的方式来解决 DLL Hell 问题，即使用 Guid 来唯一的标识每一个 COM 组件。但是，实际上使用 COM 组件（包括版本升级）也是一件颇为麻烦的事：为了运行 COM 组件就必须在组册表中对其进行注册，重新编译有可能破坏 Guid 从而导致原来引用此 COM 组件的程序不能正确运行，等等。在 .Net 中，微软引入了一种新的解决方案：强命名程序集 (Strong Name)，以及与之配套的全局程序集缓存 (GAC) 来解决这个问题。我们知道，.Net 使用 Name、Version、Culture、PublicToken 四个属性来唯一标识一个程序集，而不同产品前三个属性 (Name、Version 和 Culture) 完全相同的情况是有可能发生的，如此一来，这唯一标识程序集可重任就落到 PublicToken 的头上了。强命名的程序集正是使用 RSA 来保证 PublicToken 的唯一性，因为在理论上，非对称算法 RSA 生成的公钥/私钥对不会重复。.Net 正是通过在编译项目时将指定的公钥/私钥对写入程序集来保证其唯一性。对于全局程序集缓存 (GAC)，MSDN 是这样介绍的：安装有公共语言运行库的每台计算机都具有称为全局程序集缓存的计算机范围内的代码缓存。全局程序集缓存中存储了专门指定给由计算机中若干应用程序共享的程序集。在开发一般的、非共享的程序时，我们不需要使用强命名的程序集，仅将项目 (Project) 编辑成 .DLL 或者 .EXE 即可。但是，如果我们开发的是组件库、框架时，通过对程序集进行强命名，并使用将其部署到 GAC 中，可以保证我们的程序集不会出现版本问题。

33.1.46 Invoke 与 BeginInvoke 的区别

* Control.Invoke 方法 (Delegate)：在拥有此控件的基础窗口句柄的线程上执行指定的委托。

* Control.BeginInvoke 方法 (Delegate)：在创建控件的基础句柄所在线程上异步执行指定委托。

控件上的大多数方法只能从创建控件的线程调用。如果已经创建控件的句柄，则除了 InvokeRequired 属性以外，控件上还有四个可以从任何线程上安全调用的方法，它们是：Invoke、BeginInvoke、EndInvoke 和 CreateGraphics。在后台线程上创建控件的句柄之前调用 CreateGraphics 可能会导致非法的跨线程调用。对于所有其他方法调用，则应使用调用 (invoke) 方法之一封送对控件的线程的调用。调用方法始终在控件的线程上调用自己的回调。

注：在主线程中直接调用 Invoke、BeginInvoke、EndInvoke 都会造成阻塞。所以应该利用副线程（支线程）调用。

* Control 的 Invoke 和 BeginInvoke 的委托方法是在主线程，即 UI 线程上执行。（也就是说如果你的委托方法用来取花费时间长的数据，然后更新界面什么的，千万别在主线程上调用

Control.Invoke 和 Control.BeginInvoke，因为这些是依然阻塞 UI 线程的，造成界面的假死)

* Invoke 会阻塞主线程，BeginInvoke 只会阻塞主线程，不会阻塞支线程！因此 BeginInvoke 的异步执行是指相对于支线程异步，而不是相对于主线程异步。

33.1.47 Activex、OLE、COM、OCX、DLL 之间的区别

熟悉面向对象编程和网络编程的人一定对 ActiveX、OLE 和 COM/DCOM 这些概念不会陌生，但是它们之间究竟是什么样的关系，对许多们还是比较模糊的。在具体介绍它们的关系之间，我们还是先明确组件 (Component) 和对象 (Object) 之间的区别。组件是一个可重用的模块，它是由一组处理过程、数据封装和用户接口组成的业务对象 (Business Object)。组件看起来像对象，但不符合对象的学术定义。它们的主要区别是：1) 组件可以在另一个称为容器 (有时也称为承载者或宿主) 的应用程序中使用，也可以作为独立过程使用；2) 组件可以由一个类构成，也可以由多个类组成，或者是一个完整的应用程序；3) 组件为模块重用，而对象为代码重用。现在，比较流行的组件模型有 COM (Component Object Model, 对象组件模型) /DCOM (Distributed COM, 分布式对象组件模型) 和 CORBA (Common Object Request Broker Architecture, 公共对象请求代理体系结构)。到这里，已经出现了与本文相关的主题 COM，而 CORBA 与本文无关，就不作介绍。之所以从组件与对象的区别说起，是想让大家明确 COM 和 CORBA 是处在整个体系结构的最底层，如果暂时对此还不能理解，不妨继续往下看，最后在回过头看一看就自然明白了。现在开始阐述 ActiveX、OLE 和 COM 的关系。首先，让大家有一个总体的概念，从时间的角度讲，OLE 是最早出现的，然后是 COM 和 ActiveX；从体系结构角度讲，OLE 和 ActiveX 是建立在 COM 之上的，所以 COM 是基础；单从名称角度讲，OLE、ActiveX 是两个商标名称，而 COM 则是一个纯技术名词，这也是大家更多的听说 ActiveX 和 OLE 的原因。既然 OLE 是最早出现的，那么就从 OLE 说起，自从 Windows 操作系统流行以来，“剪贴板” (Clipboard) 首先解决了不同程序间的通信问题（由剪贴板作为数据交换中心，进行复制、粘贴的操作），但是剪贴板传递的都是“死”数据，应用程序开发者得自行编写、解析数据格式的代码，于是动态数据交换 (Dynamic Data Exchange, DDE) 的通信协定应运而生，它可以让应用程序之间自动获取彼此的最新数据，但是，解决彼此之间的“数据格式”转换仍然是程序员沉重的负担。对象的链接与嵌入 (Object Linking and Embedding, OLE) 的诞生把原来应用程序的数据交换提高到“对象交换”，这样程序间不但获得数据也同样获得彼此的应用程序对象，并且可以直接使用彼此的数据内容，其实 OLE 是 Microsoft 的复合文档技术，它的最初版本只是瞄准复合文档，但在后续版本 OLE2 中，导入了 COM。由此可见，COM 是应 OLE 的需求而诞生的，所以虽然 COM 是 OLE 的基础，但 OLE 的产生却在 COM 之前。COM 的基本出发点是，让某个软件通过一个通用的机构为另一个软件提供服务。COM 是应 OLE 的需求而诞生，但它的第一个使用者却是 OLE2，所以 COM 与复合文档间并没有多大的关系，实际上，后来 COM 就作为与复合文档完全无关的技术，开始被广泛应用。这样一来，Microsoft 就开始“染指”通用平台技术。但是 COM 并不是产品，它需要一个商标名称。而那时 Microsoft 的市场专家们已经选用了 OLE 作为商标名称，所以使用 COM 技术的都开始贴上了 OLE 的标签。虽然这些技术中的绝大多数与复合文档没有关系。Microsoft 的这一做法让人产生这样一个误解 OLE 是仅指复合文档呢？还是不单单指复合文档？其实 OLE 是 COM 的商标名称，自然不仅仅指复合文档。但 Microsoft 自己恐怕无法解释清楚，这要花费相当的精力

和时间。于是，随着 Internet 的发展，在 1996 年春，Microsoft 改变了主意，选择 ActiveX 作为新的商标名称。ActiveX 是指宽松定义的、基于 COM 的技术集合，而 OLE 仍然仅指复合文档。当然，ActiveX 最核心的技术还是 COM。ActiveX 和 OLE 的最大不同在于，OLE 针对的是桌面上应用软件和文件之间的集成，而 ActiveX 则以提供进一步的网络应用与用户交互为主。到这里，大家应该对 ActiveX、OLE 和 COM 三者的关系有了一个比较明确的认识，COM 才是最根本的核心技术，所以下面的重点 COM。让对象模型完全独立于编程语言，这是一个非常新奇的思想。这一点从 C++ 和 Java 的对象概念上，我们就能有所了解。但所谓 COM 对象究竟是什么呢？为了便于理解，可以把 COM 看作是某种（软件）打包技术，即把它看作是软件的不同部分，按照一定的面向对象的形式，组合成可以交互的过程和以组支持库。COM 对象可以用 C++、Java 和 VB 等任意一种语言编写，并可以用 DLL 或作为不同过程工作的执行文件的形式来实现。使用 COM 对象的浏览器，无需关心对象是用什么语言写的，也无须关心它是以 DLL 还是以另外的过程来执行的。从浏览器端看，无任何区别。这样一个通用的处理技巧非常有用。例如，由用户协调运行的两个应用，可以将它们的共同作业部分作为 COM 对象间的交互来实现（当然，现在的 OLE 复合文档也能做到）。为在浏览器中执行从 Web 服务器下载的代码，浏览器可把它看作是 COM 对象，也就是说，COM 技术也是一种打包可下载代码的标准方法（ActiveX 控件就是执行这种功能的）。甚至连应用与本机 OS 进行交互的方法也可以用 COM 来指定，例如在 Windows 和 Windows NT 中用的是新 API，多数是作为 COM 对象来定义的。可见，COM 虽然起源于复合文档，但却可有效地适用于许多软件问题，它毕竟是处在底层的基础技术。用一句话来说，COM 是独立于语言的组件体系结构，可以让组件间相互通信。随着计算机网络的发展，COM 进一步发展为分布式组件对象模型，这就是 DCOM，它类似于 CORBA 的 ORB，本文对此将不再做进一步的阐述。通过上面的讲述相信大家一定对 ActiveX、OLE 和 COM/DCOM 的关系有了一个清楚的了解。

Activex,OLE,COM 都是微软的一些技术标准。Ole 比较老后来发展成 Activex，再后来发展成为 COM OCX，DLL 是扩展名。Activex 有两种扩展名 OCX 和 DLL。实际上你可以把它们的扩展名调换。COM 作为 ActiveX 的更新技术，扩展名也有可能是 DLL DLL 文件还有可能是动态链接库。主要是装载一些函数，可以动态加载。activex、ole、ocx 都是基于 com 技术的。这三种实际上都是 com 的具体表现。至于 dll，和其它四种没什么关系，虽然 Activex、OLE、COM、OCX 的扩展名也可以是 dll，但也可以是 exe，或是 ocx/vbx 等等。dll 的种类就很多了，com 组件有很多是以 dll 形式提供的，当然，普通的 api 一般也是被封在 dll 中。甚至还有.net 程序。实际上，微软在.net 之前，只有两种基本的组件技术，一个是 com（在 com 的基础上出现了很多其他的技术，如 activex），另一个就是普通的 api 了，如 windows kernel api。不过有了.net，尽量使用.net 的技术。com 注册太麻烦，弄不好就会掉进 com hell 里去了。

33.1.48 Hadoop 和 Spark 区别

解决问题的层面不一样 首先，Hadoop 和 Apache Spark 两者都是大数据框架，但是各自存在的目的不尽相同。Hadoop 实质上更多是一个分布式数据基础设施：它将巨大的数据集分派到一个由普通计算机组成的集群中的多个节点进行存储，意味着您不需要购买和维护昂贵的服务器硬件。

同时，Hadoop 还会索引和跟踪这些数据，让大数据处理和分析效率达到前所未有的高度。

Spark，则是那么一个专门用来对那些分布式存储的大数据进行处理的工具，它并不会进行分布式数据的存储。

两者可合可分 Hadoop 除了提供为大家所共识的 HDFS 分布式数据存储功能之外，还提供了叫做 MapReduce 的数据处理功能。所以这里我们完全可以抛开 Spark，使用 Hadoop 自身的 MapReduce 来完成数据的处理。

相反，Spark 也不是非要依附在 Hadoop 身上才能生存。但如上所述，毕竟它没有提供文件管理系统，所以，它必须和其他的分布式文件系统进行集成才能运作。这里我们可以选择 Hadoop 的 HDFS，也可以选择其他的基于云的数据系统平台。但 Spark 默认来说还是被用在 Hadoop 上面的，毕竟，大家都认为它们的结合是最好的。

以下是从网上摘录的对 MapReduce 的最简洁明了的解析：

我们要数图书馆中的所有书。你数 1 号书架，我数 2 号书架。这就是“Map”。我们人越多，数书就更快。现在我们到一起，把所有人的统计数加在一起。这就是“Reduce”。Spark 数据处理速度秒杀 MapReduce

Spark 因为其处理数据的方式不一样，会比 MapReduce 快上很多。MapReduce 是分步对数据进行处理的：“从集群中读取数据，进行一次处理，将结果写到集群，从集群中读取更新后的数据，进行下一次的处理，将结果写到集群，等等…”Booz Allen Hamilton 的数据科学家 Kirk Borne 如此解析。

反观 Spark，它会在内存中以接近“实时”的时间完成所有的数据分析：“从集群中读取数据，完成所有必须的分析处理，将结果写回集群，完成，”Born 说道。Spark 的批处理速度比 MapReduce 快近 10 倍，内存中的数据分析速度则快近 100 倍。

如果需要处理的数据和结果需求大部分情况下是静态的，且你也有耐心等待批处理的完成的话，MapReduce 的处理方式也是完全可以接受的。

但如果你需要对流数据进行分析，比如那些来自于工厂的传感器收集回来的数据，又或者说你的应用是需要多重数据处理的，那么你也许更应该使用 Spark 进行处理。

大部分机器学习算法都是需要多重数据处理的。此外，通常会用到 Spark 的应用场景有以下方面：实时的市场活动，在线产品推荐，网络安全分析，机器日记监控等。

灾难恢复 两者的灾难恢复方式迥异，但是都很不错。因为 Hadoop 将每次处理后的数据都写入到磁盘上，所以其天生就能很有弹性的对系统错误进行处理。

Spark 的数据对象存储在分布于数据集群中的叫做弹性分布式数据集 (RDD: Resilient Distributed Dataset) 中。“这些数据对象既可以放在内存，也可以放在磁盘，所以 RDD 同样也可以提供完成的灾难恢复功能，”Borne 指出。

33.1.49 B 树、B-树、B+ 树、B* 树都是什么

B 树即二叉搜索树,B 树的搜索, 从根结点开始, 如果查询的关键字与结点的关键字相等, 那么就命中; 否则, 如果查询关键字比结点关键字小, 就进入左儿子; 如果比结点关键字大, 就进入右儿子; 如果左儿子或右儿子的指针为空, 则报告找不到相应的关键字;

如果 B 树的所有非叶子结点的左右子树的结点数目均保持差不多 (平衡), 那么 B 树的搜索性能逼近二分查找; 但它比连续内存空间的二分查找的优点是, 改变 B 树结构 (插入与删除结点) 不需要移动大段的内存数据, 甚至通常是常数开销;

实际使用的 B 树都是在原 B 树的基础上加上平衡算法, 即 “平衡二叉树”; 如何保持 B 树结点分布均匀的平衡算法是平衡二叉树的关键; 平衡算法是一种在 B 树中插入和删除结点的策略; B-树是一种多路搜索树 (并不是二叉的), B+ 树是 B-树的变体, 也是一种多路搜索树,B+ 树是应文件系统所需而产生的一种 B-树的变形树。

****B***** 树 ** 是 B+ 树的变体, 在 B+ 树的非根和非叶子结点再增加指向兄弟的指针;

****B+ 树在数据库中的应用 ****

1. 索引在数据库中的作用 ** 在数据库系统的使用过程当中, 数据的查询是使用最频繁的一种数据操作。最基本的查询算法当然是顺序查找 (linear search), 遍历表然后逐行匹配行值是否等于待查找的关键字, 其时间复杂度为 $O(n)$ 。但时间复杂度为 $O(n)$ 的算法规模小的表, 负载轻的数据库, 也能有好的性能。但是数据增大的时候, 时间复杂度为 $O(n)$ 的算法显然是糟糕的, 性能就很快下降了。好在计算机科学的发展提供了很多更优秀的查找算法, 例如二分查找 (binary search)、二叉树查找 (binary tree search) 等。如果稍微分析一下会发现, 每种查找算法都只能应用于特定的数据结构之上, 例如 ** 二分查找要求被检索数据有序, 而二叉树查找只能应用于二叉查找树上 **, 但是数据本身的组织结构不可能完全满足各种数据结构 (例如, 理论上不可能同时将两列都按顺序进行组织), 所以, ** 在数据之外, 数据库系统还维护着满足特定查找算法的数据结构, 这些数据结构以某种方式引用 (指向) 数据, 这样就可以在这些数据结构上实现高级查找算法。这种数据结构, 就是索引。 索引是对数据库表中一个或多个列的值进行排序的结构。与在表中搜索所有的行相比, 索引用指针指向存储在表中指定列的数据值, 然后根据指定的次序排列这些指针, 有助于更快地获取信息。通常情况下, 只有当经常查询索引列中的数据时, 才需要在表上创建索引。索引将占用磁盘空间, 并且影响数据更新的速度。但是在多数情况下, 索引所带来的数据检索速度优势大大超过它的不足之处。**2. B+ 树在数据库索引中的应用 **

目前 ** 大部分数据库系统及文件系统都采用 B-Tree 或其变种 B+Tree 作为索引结构 **
1) 在数据库索引的应用在数据库索引的应用中, B+ 树按照下列方式进行组织: 叶结点的组织方式。B+ 树的查找键是数据文件的主键, 且索引是稠密的。也就是说, 叶结点中为数据文件的第一个记录设有一个键、指针对, 该数据文件可以按主键排序, 也可以不按主键排序; 数据文件按主键排序, 且 B + 树是稀疏索引, 在叶结点中为数据文件的每一个块设有一个键、指针对; 数据文件不按键属性排序, 且该属性是 B + 树的查找键, 叶结点中为数据文件里出现的每个属性 K 设有一个键、指针对, 其中指针执行排序键值为 K 的记录中的第一个。 非叶结点的组织

方式。B+ 树中的非叶结点形成了叶结点上的一个多级稀疏索引。每个非叶结点中至少有 $\text{ceil}(m/2)$ 个指针，至多有 m 个指针。2) B+ 树索引的插入和删除 在向数据库中插入新的数据时，同时也需要向数据库索引中插入相应的索引键值，则需要向 B+ 树中插入新的键值。** 即上面我们提到的 B-树插入算法。** 当从数据库中删除数据时，同时也需要从数据库索引中删除相应的索引键值，则需要从 B+ 树中删除该键值。** 即 B-树删除算法 **

** 为什么使用 B-Tree (B+Tree) ** 二叉查找树进化品种的红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用 B-/+Tree 作为索引结构。一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘 I/O 消耗，相对于内存存取，I/O 存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘 I/O 操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘 I/O 的存取次数。为什么使用 B-/+Tree，还跟磁盘存取原理有关。局部性原理与磁盘预读由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分之一，因此为了提高效率，要尽量减少磁盘 I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高 I/O 效率。预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为 4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

我们上面分析 B-/+Tree 检索一次最多需要访问节点：** $h = \lceil \log_m N \rceil$ **

** 数据库系统巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全载入。** 为了达到这个目的，在实际实现 B-Tree 还需要使用如下技巧：每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个 node 只需一次 I/O。B-Tree 中一次检索最多需要 $h-1$ 次 I/O（根节点常驻内存），渐进复杂度为 $O(h) = O(\log_m N)$ 。一般实际应用中，m 是非常大的数字，通常超过 100，因此 h 非常小（通常不超过 3）。综上所述，用 B-Tree 作为索引结构效率是非常高的。而红黑树这种结构，** h ** 明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的 I/O 渐进复杂度也为 $O(h)$ ，效率明显比 B-Tree 差很多。

****MySQL 的 B-Tree 索引（技术上说 B+Tree）****

在 MySQL (<http://lib.csdn.net/base/14>) 中，主要有四种类型的索引，分别为：B-Tree 索引，Hash 索引，Fulltext 索引和 R-Tree 索引。我们主要分析 B-Tree 索引。B-Tree 索引是 MySQL 数据库中使用最为频繁的索引类型，除了 Archive 存储引擎之外的其他所有的存储引擎都支持 B-Tree 索引。Archive 引擎直到 MySQL 5.1 才支持索引，而且只支持索引单个 AUTO_INCREMENT 列。不仅仅在 MySQL 中是如此，实际上在其他的很多数据库管理系统

中 B-Tree 索引也同样是作为最主要的索引类型，这主要是因为 B-Tree 索引的存储结构在数据库的数据检索中有非常优异的表现。一般来说，MySQL 中的 B-Tree 索引的物理文件大多都是以 Balance Tree 的结构来存储的，也就是所有实际需要的数据都存放于 Tree 的 Leaf Node(叶子节点)，而且到任何一个 Leaf Node 的最短路径的长度都是完全相同的，所以我们大家都称之为 B-Tree 索引。当然，可能各种数据库（或 MySQL 的各种存储引擎）在存放自己的 B-Tree 索引的时候会对存储结构稍作改造。如 Innodb 存储引擎的 B-Tree 索引实际使用的存储结构实际上是 B+Tree，也就是在 B-Tree 数据结构的基础上做了很小的改造，在每一个 Leaf Node 上面除了存放索引键的相关信息之外，还存储了指向与该 Leaf Node 相邻的后一个 LeafNode 的指针信息（增加了顺序访问指针），这主要是为了加快检索多个相邻 Leaf Node 的效率考虑。

动态查找树主要有：二叉查找树（Binary Search Tree），平衡二叉查找树（Balanced Binary Search Tree），红黑树（Red-Black Tree），B-tree/B+-tree/ B*-tree** **(B Tree)。前三者是典型的二叉查找树结构，其查找的时间复杂度 $O(\log_2 N)$ 与树的深度相关，那么降低树的深度自然对查找效率是有所提高的；还有一个实际问题：就是大规模数据存储中，实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致二叉查找树结构由于树的深度过大而造成磁盘 I/O 读写过于频繁，进而导致查询效率低下（为什么会出现这种情况，待会在外部存储器-磁盘中有所解释），那么如何减少树的深度（当然是不能减少查询的数据量），一个基本的想法就是：采用多叉树结构（由于树节点元素数量是有限的，自然该节点的子树数量也就是有限的）。这样我们就提出了一个新的查找树结构——多路查找树。根据平衡二叉树的启发，自然就想到平衡多路查找树结构，也就是这篇文章所要阐述的主题 B tree(B 树结构)，B-tree 这棵神奇的树是在 Rudolf Bayer, Edward M. McCreight(1970) 写的一篇论文《Organization and Maintenance of Large Ordered Indices》中首次提出。具体介绍可以参考 wikipedia 中的介绍：<http://en.wikipedia.org/wiki/B-tree>，其中还阐述了 B-tree 名字来源以及相关的开源地址。

33.1.50 .ocx 是什么

OCX 是对象类别扩充组件（Object Linking and Embedding (OLE) Control eXtension）；是不可执行的文件；是 .ocx 控件的扩展名，与 .exe、.dll 同属于 PE 文件。如果你用过 Visual Basic 或者 Delphi 一类的可视化编程工具，那么对控件这个概念一定不会陌生，就是那些工具条上的小按钮，如 EditBox, Grid, ImageBox, Timer 等等。每个控件都有自己的事件、方法和属性。使用了控件的编程非常容易。在程序的设计阶段可以设置一些属性，如大小，位置，标题（caption）等等，在程序运行阶段，可以更改这些属性，还可以针对不同的事件，调用不同的方法来实现对该控件的控制。控件就好像一块块的积木，程序要做的事只是将这些积木搭起来。控件的最大好处是可以重复使用，甚至可以在不同的编程语言之间使用，例如，你可以在 VB 中嵌入用 VC 开发的控件。

33.1.51 什么是面向对象？为什么要面向对象？

面向对象程序设计（英语：Object-oriented programming，缩写：OOP）是种具有对象概念的程序编程范型，同时也是一种程序开发的方法。它可能包含数据、属性、代码与方法。对象则

指的是类的实例。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性，对象里的程序可以访问及经常修改对象相关连的数据。在面向对象程序编程里，计算机程序会被设计成彼此相关的对象。

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是一系列对电脑下达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。目前已经被证实的是，面向对象程序设计推广了程序的灵活性和可维护性，并且在大型项目设计中广为应用。此外，支持者声称面向对象程序设计要比以往的做法更加便于学习，因为它能够让人们更简单地设计并维护程序，使得程序更加便于分析、设计、理解。反对者在某些领域对此予以否认。

当我们提到面向对象的时候，它不仅指一种程序设计方法。它更多意义上是一种程序开发方式。在这一方面，我们必须了解更多关于面向对象系统分析和面向对象设计（Object Oriented Design，简称 OOD）方面的知识。许多流行的编程语言是面向对象的，它们的风格就是会透由对象来创出实例。重要的面向对象编程语言包含 Common Lisp、Python、C++、Objective-C、Smalltalk、Delphi、Java、Swift、C#、Perl、Ruby 与 PHP.

面向对象相对面向过程的优点 1) 结构清晰。使人们的编程与实际的世界更加接近，所有的对象被赋予属性和方法，结果编程就更加富有人性化。2) 封装性。减小外部对内部的影响。封装将对象有关的数据和行为封装成整体来处理，使得对象以外的部分不能随意存取对象的内部属性，从而有效地避免了外部错误对它的影响，大大减小了查错和排错的难度。3) 容易扩展，代码重用率高。容易扩展，在大框架不变的情况下很容易就开发出适合自己的功能，实现简单，可有效地减少程序的维护工作量，软件开发效率高。

面向对象相对面向过程的缺点 1) 增加工作量。如果一味地强调封装，当进行修改对象内部时，对象的任何属性都不允许外部直接存取，则要增加许多没有其他意义、只负责读或写的行为。这会为编程工作增加负担，增加运行开销，并且使程序显得臃肿。2) 性能低。由于面向更高的逻辑抽象层，使得面向对象在实现的时候，不得不做出性能上面的牺牲，计算时间和空间存储大小的都开销很大。

33.1.52 Why is Dictionary preferred over HashTable

Dictionary 是类型安全的，HashTable 不是。Dictionary 没有装箱拆箱，HashTable 可能有。

33.1.53 dll 和 COM 的区别

1) dll 是以函数集合的方式来调用的，是编程语言相关的，如：VC 必须加上 extern "C"。而 COM 是以 interface 的方式提供给用户使用的是一种二进制的调用规范，是与编程语言无关的。2) DLL 只有 DLL 一种形势，里面可任意定义函数无限制，只能运行在本机上，而 COM 有

DLL 和 EXE 两种存在形势。3) COM 所在的 DLL 中必须导出四个函数:1. dllgetobjectclass, 2. dllregisterserver, 3. dllunregisterserver, 4. dllunloadnow. Com 补充: COM 解决了版本、模块化开发, 所有语言使用, 当然只能在 WINDOWS 平台上。COM 载体: DLL、EXE (不常用), OCX (用于 activex 控件), activex 实际上是 COM 的一种变体, 但本质上没变, 当然 ACTIVE-X 控件也能以 DLL 作载体。DLL, ACTIVE-X, COM, 插件区别: DLL (基于名字导入的, 名字就是符号, DLL 有符号表的。根据约定好的名字调用函数) 接口是按照规划定义的规则集合简单说来呢, 这好比一棵树, COM (组件是基于接口的, 根据约定好的接口对 COM 对象进行控制) 是树根, 组件 (软件的组成部分.) 是树干, 控件 (具有用户界面的组件) 和 ActiveX 都是树枝, 插件 (网页中用到的, flash 插件, 没有它浏览器不能播放 flash.) 就是树上引来的一只鸟。ACTIVE-X 和 COM 的区别: 两者没有质的区别, 前者主要用于客户端, 后者用于服务器端。前者可以有界面而后者决没有界面 ActiveX 的作用: 可轻松方便的在 Web 页中插入多媒体效果、交互式对象、以及复杂程序, ActiveX 插件安装的一个前提是必须经过用户的同意及确认。

33.1.54 Func<T, TResult> 委托的由来和调用和好处

Func<T, TResult> 是系统的内置委托的中最常用的一个。特点就是必须有一个返回值。(func 委托有多个重载, 所有重载的最后一个参数就是返回值的类型, 前面的是参数类型)。注: 没有返回值的系统内置委托是 Action<T>

Func 委托的作用就是当我们需要传入 n 参数并有返回值时, 我们不用再去定义一个委托, 直接调用 func 即可。

33.1.55 为什么要使用 LINQ 技术

为了实现数据的查询, 并跟上技术的进步, 开发人员不得不针对特定查询语言所支持的数据源或数据格式, 而花费大量时间学习如何使用它们。然而, LINQ (语言集成查询) 通过采用一种跨多种数据源和数据格式使用数据的一致模型, 简化了这一情况。在 LINQ 查询中, 始终会用到对象。为此, 可以使用相同的基本编码模式来查询和转换 XML 文档、SQL 数据库、ADO.NET 数据集、.NET 集合中的数据, 以及对其有 LINQ 提供程序可用的任何其他格式的数据。LINQ 把查询和设置等操作封装起来, 实现了更加灵活的数据查询机制。它定义了一组多用途的标准查询操作符, 可以在任何一种基于.NET 平台的开发语言查询操作符中, 采用一些简洁明了的语法实现数据访问、过滤、发送等操作。同时, 为了满足 LINQ 需要, C# 也进行了较大调整, 诸如隐式类型化局部变量和匿名类, 都被添加到语言之中。

LINQ 是.NET 3.5 導入的一種新的資料存取技術, 這個技術可以讓開發人員透過一致的方法, 存取各種型態的資料來源。

.NET 2.0 導入了對泛型的支援, LINQ 在泛型基礎上, 從集合類型資料的搜尋功能出發, 逐步發展支援物件、資料庫、XML 以及實體資料等等不同資料來源的新技術。

針對集合類型資料的操作, 無論是篩選、計量、關聯或是切割等等, LINQ 為我們提供了一套出色的解決方案。

你可以將 LINQ 視為一套專門用來對付集合類型資料的搜尋操作語法, 只要將資料儲存於

集合，就便可以如同資料庫一般進行操作，例如，你可以載入硬碟檔案系統資訊，與資料庫中的資料透過 LINQ 合併作處理。

有了 LINQ，我們可以針對關聯式資料庫或是 XML 等各種不同結構的資料統一以 LINQ 進行處理，在原有的基礎上提供更好的資料整合能力，更有效率的處理各種不同來源的資料，而當你只是要針對特定的資料來源進行存取，原來的技術還是可以運作的很好。

所以，重點不在於 LINQ 可以存取這些資料來源，LINQ 真正的精神，在於它終於正視了集合物件元素的存取操作這件事。

電腦這門行業之所以稱為資訊工業，在於它是人類為了處理資料而發展出來的一門技術，資料只有經過有效的整理歸納與萃取之後，才能成為有用的資訊，我們所開發的軟體，都是為了處理各種資料而設計，而為了應付資料處理的需求，所以有了資料庫，有了 XML 文件，然後有了存取這些資料來源的相關的技術。

回到源頭，早期程式語言剛開始發展出來的時候，~~沒有~~ 有上述談到的這些東西，最早期的資料處理，一般均透過所謂的陣列與集合進行操作，一直到現在，只要資料被載入記憶體處理，我們所依賴的依然是這些技術。

了解問題在那裏了嗎，集合，長久以來，一直沒有足夠「好」的技術，可以針對儲存在集合中的元素執行存取維護作業，.NET 2.0 導入了對泛型的支援，也僅是讓資料的存取更為嚴謹靈活，如此而已，一直到 LINQ，這個問題終於有了好的解答。

你很難想像，將集合物件當作資料庫來操作，無論是篩選、計量、關聯或是切割等資料操作，這些 SQL 的基本功能，傳統的程式語言都很難、或是不容易在集合身上辦到，LINQ 為我們提供了一套出色的解決方案。

你可以將 LINQ 視為一套專門用來對付集合物件的資料操作語法，想像有了 LINQ，只要將資料儲存於集合，就可以如同資料庫一般進行操作，例如，你可以載入硬碟檔案系統資訊，與資料庫中的資料透過 LINQ 合併作處理，多麼美好的技術，不是嗎。

既然對於集合物件元素的存取有了好的方案，SQL 與 XML 同時一併納入處理最大的好處，是可以在原有的基礎上，提供更好的資料整合能力，而非我們原先認爲的，LINQ 的出現只是為了取代這些技術，相反的，它可以讓我們更有效的處理各種不同來源的資料，而當你只是要針對特定的資料來源進行存取，原來的技術還是可以運作的很好。

在 Linq To Sql 正式推出之前，很多人只是把 sql 语句形成一个 string，然后，通过 ADO.NET 传给 SQL Server，返回结果集。这里的缺陷就是，假如你 sql 语句写的有问题，只有到运行时才知道。而且并不是所有的人都懂数据库的。Linq To SQL 在一切围绕数据的项目内都可以使用。特别是在项目中缺少 sql server 方面的专家时，Linq To SQL 的强大的功能可以帮我们快速的完成项目。Linq To SQL 的推出，是让大家从繁琐的技术细节中解脱出来，更加关注项目的逻辑。Linq To Sql 的出现，大大降低了数据库应用程序开发的门槛，它实质是事先为你构架了数据访问层，势必将加快数据库应用程序的开发进度。Linq To Sql 解放了众多程序员，让他们的把更多的精力放到业务逻辑以及 code 上，而不是数据库。对于初学者来讲，Linq To Sql 可以让他们迅速进入数据库应用程序开发领域，节约了培训成本。

33.1.56 c# 引用类型与值类型的区别

值类型包括 C# 的基本类型（用关键字 int、char、float 等来声明），结构（用 struct 关键字声明的类型），枚举（用 enum 关键字声明的类型）；而引用类型包括类（用 class 关键字声明的类型）和委托（用 delegate 关键字声明的特殊类）。C# 中的每一种类型要么是值类型，要么是引用类型。所以每个对象要么是值类型的实例，要么是引用类型的实例。值类型的实例通常是在线程栈上分配的（静态分配），但是在某些情形下可以存储在堆中。引用类型的对象总是在进程堆中分配（动态分配）。

33.2 非技术类

团队水准/文化

你在这里工作多久了？

是什么让你在众多选择中选择了这家公司？

到目前为止，你碰到过的最充实/最精彩/技术最复杂的项目是什么？

你希望你的工作有什么不同？

你离开团队的频率？是什么让你加入你现在所在的团队？你离开团队的必要条件是什么？

(如果是初创企业)，你和创始人的最后一次互动是什么时候？有什么收获？创始人的日常工作是什么？

工程项目

每天标准的日常工作情况？

你们的技术堆栈是什么？

特定技术堆栈背后的基本原理是什么？

添加新工具的频率？

更趋向于自己推出解决方案呢，还是依靠第三方工具？

使用哪种测试覆盖？

更务实或更理论化地描述你的工程文化？

花在钻研新事物的时间多还是迭代已有的内容时间多？

发布周期要多久？

近期发生的最糟糕的技术错误是什么？你们是怎么处理的？事后采取什么措施以确保不会再发生？

早期最昂贵的，现如今与公司息息相关的技术决策是什么？

在业务方面的产品语音/可视化

你现在正在做什么？因为什么原因结束当前任务？

如果你想要构建新的东西，那么需要怎么做？

最后一个版本发布的新内容主要关于什么？功能点子源于哪里？产品理念一般又来自于哪里？

在此领域的主要竞争对手是谁？我们有哪些他们没有的优势？

摩擦

工程师平均留在公司多久？

最近几个人离开是因为什么原因？

创业

有多少前雇员离开公司自己去创业了？

对他们这样的行为是不是友好对待的？

相关的企业文化如何鼓励创业？具体的例子？

33.2.1 Idea

绿色食品

产品整个生命周期平台 客户可以反馈产品的整个生命周期使用情况，从产品下单到产品丢弃或者回收的整个生命链条，用户在使用的进程中对产品的评价，评价可以在数月、数年、数十年甚至更长时间里，用户可以分享使用感受，用户可在不同时期对产品评分，产品不针对普通大众，只针对少数人群，他们需要是独立思考，有相似的见地，他们属于少部分人。

Bibliography

- [1] 蒋金楠.《ASP.NET MVC 5 框架揭秘》.电子工业出版社,2014.
- [2] 阮一峰.《RESTful API 设计指南》.http://www.ruanyifeng.com/blog/2014/05/restful_api.html.
- [3] 蔡 学 镛.《Metadata 的 格 式 和 意 义》.<https://msdn.microsoft.com/zh-tw/library/dd229216.aspx>.