

目录

云南大学信息学院无线创新实验室，中国，昆明

翻译人：杨俊东 黄铭 李文键 代海涛

内容目录

前言.....	10
第1章 引言.....	11
1.1 原则.....	12
1.2 Web 框架.....	12
1.3 模型-视图-控制器 (Model-View-Controller)	13
1.4 为什么选择 web2py.....	15
1.5 安全性.....	16
1.6 框架内容.....	17
1.7 授权.....	18
1.8 致谢.....	19
1.9 关于本书.....	19
1.10 风格要素.....	20
第2章 Python 语言.....	22
2.1 关于 Python.....	22
2.2 启动.....	22
2.3 help 和 dir 命令.....	23
2.4 类型.....	24
2.4.1 str (字符串)	24
2.4.2 list (列表)	25
2.4.3 tuple (元组)	25
2.4.4 dict (字典)	26
2.5 关于缩进.....	27
2.6 for...in.....	27
2.7 while.....	28
2.8 if...elif...else.....	28
2.9 try...except...else...finally.....	29
2.10 def...return.....	30
2.10.1 lambda 函数.....	32
2.11 class (类)	33
2.12 特殊属性、方法和运算符.....	34
2.13 文件输入/输出.....	34
2.14 exec 和 eval 函数.....	35
2.15 导入 (import)	35
2.15.1 os 模块.....	36
2.15.2 sys 模块.....	36
2.15.3 datetime 模块.....	36
2.15.4 time 模块.....	37
2.15.5 cPickle 模块.....	37
第3章 概述.....	38
3.1 启动.....	38
3.2 问好.....	40
3.3 计数.....	42

3.4	访问我的名字.....	43
3.5	回传.....	44
3.6	图像博客.....	45
3.7	添加 CRUD.....	52
3.8	添加认证.....	53
3.8.1	添加网格.....	54
3.9	配置布局.....	55
3.10	wiki 维基.....	55
3.10.1	关于 date, datetime 和 time 属性格式.....	62
3.11	关于 admin 的更多内容.....	62
3.11.1	site 页面.....	62
3.11.2	about 页面.....	64
3.11.3	edit 页面.....	64
3.11.4	errors 页面.....	66
3.11.5	Mercurial 版本.....	68
3.11.6	Admin 向导（实验性的）.....	68
3.11.7	配置 admin.....	69
3.12	更多关于 appadmin.....	70
第4章	核心.....	72
4.1	命令行选项.....	72
4.2	工作流.....	74
4.3	调度.....	76
4.4	库.....	78
4.5	应用程序.....	82
4.6	API.....	83
4.6.1	从 Python 模块访问 API.....	84
4.7	request 请求对象.....	85
4.8	response 响应对象	87
4.9	session 会话对象.....	89
4.9.1	单独的会话.....	90
4.10	cache 缓存对象.....	91
4.11	URL 统一资源定位器	93
4.11.1	绝对 url.....	94
4.11.2	数字签名的 url.....	94
4.12	HTTP 和重定向 redirect.....	95
4.13	T 和 国际化.....	96
4.14	Cookies.....	97
4.15	应用程序 init.....	98
4.16	URL 重写.....	99
4.16.1	基于参数的系统.....	99
4.16.2	基于模式的系统.....	100
4.17	出错路由 Routes on Error.....	103
4.18	后台运行任务.....	104
4.18.1	Cron.....	104
4.18.2	自制任务队列.....	106
4.18.3	调度程序（实验性的）.....	106

4.19	第三方模块.....	108
4.20	执行环境.....	109
4.21	协作.....	110
4.22	日志.....	111
4.23	WSGI (Web 服务器网关接口)	111
4.23.1	外部中间件.....	112
4.23.2	内部中间件.....	112
4.23.3	调用 WSGI 应用.....	112
第 5 章	视图.....	114
5.1	基本语法.....	115
5.1.1	for...in.....	115
5.1.2	while.....	116
5.1.3	if...elif...else.....	116
5.1.4	try...except...else...finally.....	116
5.1.5	def...return.....	117
5.2	HTML 帮助对象.....	117
5.2.1	XML (可扩展标记语言)	119
5.2.2	内置帮助对象.....	120
5.2.3	自定义帮助对象.....	130
5.3	美化.....	131
5.4	服务器端 DOM 和解析.....	131
5.4.1	elements 方法.....	131
5.4.2	componets 方法.....	132
5.4.3	parent 方法.....	132
5.4.4	flatten 方法.....	133
5.4.5	解析.....	133
5.5	页面布局.....	133
5.5.1	默认页面布局.....	135
5.5.2	定制的默认布局.....	139
5.5.3	移动部署.....	139
5.6	视图中的函数.....	140
5.7	视图中的块.....	140
第 6 章	数据库抽象层.....	142
6.1	依赖性 dependencies	142
6.2	连接字符串.....	143
6.2.1	连接池.....	143
6.2.2	连接失败.....	144
6.2.3	复制的数据库 replicated database	144
6.3	保留关键词.....	144
6.4	DAL、Table 和 Field.....	145
6.5	记录表示.....	145
6.6	迁移.....	148
6.7	修复损坏迁移.....	149
6.8	插入方法 insert	149
6.9	提交和回滚 commit and rollback	150
6.10	原始的 sql.....	151

6.10.1	定时查询 Timing queries	151
6.10.2	executesql 方法.....	151
6.10.3	_lastsql 文件.....	151
6.11	删除 drop	151
6.12	索引 Indexes	151
6.13	传统数据库和键表.....	152
6.14	分布式事务.....	152
6.15	手册上传.....	152
6.16	Query（查询）、Set（集合）和 Rows 对象.....	153
6.17	查询 select	153
6.17.1	快捷方式 Shotcuts	154
6.17.2	得到一个 Row 对象.....	155
6.17.3	递归查询 Recrusive selects	155
6.17.4	视图中序列化 Rows 对象.....	156
6.17.5	排序、分组、限制、区别 orderby, groupby, limitby, distinct ..	157
6.17.6	逻辑运算符.....	158
6.17.7	count, isempty, delete 和 update 方法.....	159
6.17.8	表达式.....	159
6.17.9	update_record 方法	159
6.17.10	first 和 last 方法.....	160
6.17.11	as_dict 和 as_list 方法.....	160
6.17.12	find、exclude 和 sort 对象.....	160
6.18	其它方法.....	161
6.18.1	update_or_insert 方法.....	161
6.18.2	validate_and_insert 和 validate_and_update 方法.....	161
6.18.3	smart_query 方法（实验性的）	161
6.19	计算字段.....	162
6.20	虚拟字段.....	162
6.20.1	旧风格虚拟字段.....	162
6.20.2	新风格的虚拟字段（实验性的）	164
6.21	一对多关系.....	164
6.21.1	内联 Inner joins	165
6.21.2	左外联 Left outer join	166
6.21.3	分组和计数 Grouping and counting	166
6.22	多对多关系.....	167
6.23	多对多、list:<type>和容器.....	167
6.24	其它运算符.....	168
6.24.1	like, startswith, contains, upper 和 lower 运算符.....	169
6.24.2	year, month, day, hour, minutes 和 seconds 运算符.....	169
6.24.3	belongs 运算符	170
6.24.4	sum, min, max 和 len 运算符.....	170
6.24.5	子字符串 Substrings	170
6.24.6	coalesce 默认值 和 coalesce_zero.....	170
6.25	生成原始 sql.....	171
6.26	导出和导入数据.....	171
6.26.1	CSV（一张表一次）	171

6.26.2	CSV (所有表一次)	172
6.26.3	CSV 和远程数据库同步	172
6.26.4	HTML 与 XML (一张表一次)	174
6.26.5	数据表示	174
6.27	缓存查询	175
6.28	自引用和别名	175
6.29	高级特性	176
6.29.1	表继承 Table inheritance	176
6.29.2	公共 fields 和 multi-tenancy 多分租	177
6.29.3	通用过滤器 Common filters	177
6.29.4	定制 Field 类型 (实验性的)	178
6.29.5	不定义表使用 DAL	178
6.29.6	从一个 db 复制数据到另外一个	178
6.29.7	新的 DAL 和适配器的注意事项	179
6.29.8	Gotchas 陷阱	181
第 7 章	表单和验证器 Forms and validators	183
7.1	表单 Forms	183
7.1.1	process 和 validate 方法	186
7.1.2	隐藏字段 Hidden fields	187
7.1.3	keepvalues 参数	187
7.1.4	onvalidation 参数	187
7.1.5	检测记录修改	188
7.1.6	表单和重定向	188
7.1.7	多表单每页面	189
7.1.8	共享表单	189
7.2	SQLFORM	190
7.2.1	SQLFORM 和 insert/update/delete	194
7.2.2	HTML 中的 SQLFORM	194
7.2.3	SQLFORM 和上传	195
7.2.4	存储源文件名	197
7.2.5	autodelete 属性 (自动删除)	197
7.2.6	链接到引用记录	197
7.2.7	预填充表单	199
7.2.8	添加额外表单元素到 SQLFORM	199
7.2.9	无数据库 IO 的 SQLFORM	199
7.3	SQLFORM.factory	200
7.3.1	单个表单多表 One form for multiple tables	200
7.4	CRUD	201
7.4.1	设置 Settings	202
7.4.2	消息 Messages	203
7.4.3	方法 Methods	203
7.4.4	记录版本 Record versioning	205
7.5	定制表单 Custom forms	205
7.5.1	CSS 规定 CSS conventions	206
7.5.2	隐藏错误 Hide errors	206
7.6	验证器 Validators	207

7.6.1	验证器 Validators.....	208
7.6.2	数据库验证器 Database validators.....	213
7.6.3	自定义验证器 Custom validators.....	215
7.6.4	依赖的验证器 Validators with dependencies.....	215
7.7	Widgets 小工具.....	216
7.7.1	自动完成widget Autocomplete widget.....	217
7.8	SQLFORM.grid 和 SQLFORM.smartgrid (实验性的)	217
第8章	电子邮件和短信系统	223
8.1	设置电子邮件.....	223
8.1.1	给 Google App Engine 配置电子邮件.....	223
8.1.2	x509 和 PGP 加密.....	223
8.2	发送电子邮件.....	224
8.2.1	简单文本电子邮件.....	224
8.2.2	HTML 电子邮件	224
8.2.3	文本和HTML 混合电子邮件	224
8.2.4	抄送和密送 cc and bcc emails.....	224
8.2.5	附件 Attachments.....	225
8.2.6	多附件 Multiple attachments.....	225
8.3	发送短信	225
8.4	用模板系统生成消息.....	225
8.5	用后台任务发送消息.....	226
第9章	访问控制 Access Control	228
9.1	认证 Authentication	229
9.1.1	注册限制 Restrictions on registration.....	231
9.1.2	集成 OpenID, Facebook 等	231
9.1.3	CAPTCHA 和 reCAPTCHA.....	232
9.1.4	定制 Auth.....	233
9.1.5	重命名 Auth 表.....	234
9.1.6	其它登录方法和登录表单.....	235
9.2	Mail 和 Auth 类.....	238
9.3	授权 Authorization	239
9.3.1	装饰器 Decorators	240
9.3.2	组合要求 Combining requirements	241
9.3.3	授权和 CRUD Authorization and CRUD	241
9.3.4	授权和下载 Authorization and downloads	242
9.3.5	访问控制和基本认证.....	242
9.3.6	手动认证 Manual Authentication	243
9.3.7	设置和消息 Settings and messages	243
9.4	集中认证服务 Central Authentication Service	247
9.4.1	用 web2py 授权非 web2py 应用	248
第10章	服务 Services	250
10.1	呈现字典.....	250
10.1.1	HTML、XML 和 JSON.....	250
10.1.2	通用视图 Generic views	251
10.1.3	呈现 Rows 对象	252
10.1.4	自定义格式	252

10.1.5	RSS	253
10.1.6	CSV.....	254
10.2	远程过程调用 Remote procedure calls	254
10.2.1	XMLRPC.....	256
10.2.2	JSONRPC.....	257
10.2.3	JSONRPC 和 Pyjamas.....	257
10.2.4	Amfrpc.....	260
10.2.5	SOAP.....	261
10.3	低级别API 和其它方法.....	262
10.3.1	simplejson.....	262
10.3.2	PyRTF.....	263
10.3.3	ReportLab 和 PDF.....	263
10.4	Restful Web 服务.....	264
10.4.1	parse_as_rest 方法(实验性的).....	266
10.4.2	smart_query 方法(实验性的).....	269
10.4.3	访问控制.....	269
10.5	服务与认证 Services and Authentication.....	269
第11章	jQuery 和 Ajax.....	271
11.1	web2py_ajax.html.....	271
11.2	jQuery 效果.....	274
11.2.1	表单的条件域	276
11.2.2	删除确认.....	277
11.3	ajax 函数	278
11.3.1	Eval target.....	279
11.3.2	自动完成 Auto-completion.....	279
11.3.3	Ajax 表单提交.....	281
11.3.4	投票和打分 Voting and rating.....	282
第12章	组件和插件 Components and plugins.....	284
12.1	组件 Components.....	284
12.1.1	客户-服务器组件通信.....	287
12.1.2	捕获 Ajax 链接	288
12.2	插件 Plugins	288
12.2.1	组件插件 Component plugins	290
12.2.2	插件管理对象 Plugin manager	291
12.2.3	布局插件 Layout plugins	292
12.3	plugin_wiki.....	293
12.3.1	MARKMIN 语法.....	294
12.3.2	页面权限 Page permissions.....	296
12.3.3	特殊页面 Special pages.....	296
12.3.4	配置 plugin_wiki	298
12.3.5	当前小工具 Current widgets	298
12.3.6	扩展 widget.....	302
第13章	部署方法.....	304
13.0.7	文件 anyserver.py.....	306
13.1	Linux 和 Unix.....	306
13.1.1	生产部署一步到位.....	306

13.1.2	Apache 设置.....	306
13.1.3	mod_wsgi 模块.....	307
13.1.4	mod_wsgi 模块和 SSL（安全套接层）.....	309
13.1.5	mod_proxy 模块（代理模块）.....	310
13.1.6	开启 Linux 守护进程.....	312
13.1.7	Lighttpd 开源服务器.....	312
13.1.8	具有 mod_python 模块的共享式主机.....	313
13.1.9	具有 FastCGI 的 cherokee 服务器.....	314
13.1.10	Postgresql 数据库.....	315
13.2	Windows 系统.....	316
13.2.1	Apache 和 mod_wsgi.....	316
13.2.2	开始为 Windows 服务.....	318
13.3	安全会话和 admin(管理).....	319
13.4	高效性和扩展性.....	319
13.4.1	高效策略.....	320
13.4.2	数据库中的会话.....	320
13.4.3	HAProxy 一种高可用性、负载均衡器.....	321
13.4.4	清除会话.....	322
13.4.5	上传文件于数据库.....	322
13.4.6	收集票据.....	323
13.4.7	Memcache 缓存.....	323
13.4.8	memcache 会话.....	324
13.4.9	Redis 的高速缓存.....	324
13.4.10	删除应用.....	324
13.4.11	使用复制数据库.....	325
13.5	部署在 Google App Engine.....	325
13.5.1	配置.....	326
13.5.2	运行和部署.....	327
13.5.3	配置处理器.....	328
13.5.4	避免文件系统.....	329
13.5.5	Memcache 缓存.....	329
13.5.6	数据存储问题.....	329
13.5.7	GAE 和 https.....	330
13.6	Jython.....	330
第 14 章	其他方法.....	331
14.1	升级.....	331
14.2	如何以二进制文件发布您的应用程序.....	331
14.3	建立一个最低限度的 web2py.....	332
14.4	取出一个外部 URL.....	332
14.5	漂亮的日期.....	332
14.6	地理编码 Geocoding.....	333
14.7	分页.....	333
14.8	httpserver.log 和日志文件格式.....	334
14.9	用虚拟数据填充数据库.....	334
14.10	接受信用卡付款.....	335
14.10.1	Google 钱包.....	335

14.10.2	Paypal.....	336
14.10.3	Stripe.com.....	336
14.10.4	Authorize.Net.....	337
14.11	Dropbox API.....	338
14.12	Twitter API.....	338
14.13	流虚拟文件.....	339
参考文献:	340

前言

web2py 于 2007 年发布，现在经过 4 年持续发展，我们已经完成了期待已久的本书第 4 版撰写。在这期间，web2py 成功赢得了成千上万学识渊博用户和一百多位开发人员的喜爱。我们共同的努力创造了现有功能最全的开源 web 框架之一。

我最初将 web2py 作为一种教学工具，因为我相信，构建高品质 web 应用的能力对于一个自由开放社会发展是至关重要的。发展 web 应用能防止信息垄断。这一动机始终是对的，现在看起来更迫切。

一般来说，web 框架是为了让 web 开发更简单、更快捷，并降低开发者的失误，尤其是涉及安全的方面。在 web2py 中，我们用三个主要目标解决这些问题：

容易使用是 web2py 的首要目标。对我们来说，这意味着缩短学习和部署的时间，这就是 web2py 采用全堆栈无依赖性的原因，它无需安装和配置文件。在 web2py 中，每一项功能都即开即用，包括 web 服务器配置、数据库开发和基于 web 的集成开发环境使用。API 包含 12 个核心对象，这方便了用户记忆和使用，它能与绝大多数的 web 服务器、数据库以及所有的 Python 库进行交互。

快速开发是 web2py 的第二目标。web2py 中每个函数都有一个默认的行为（该行为可被重写），例如，一旦你指定了数据模型，你就可以访问一个基于 web 的数据库管理面板。web2py 还能自动为你的数据生成表单，这允许你方便的将数据以 HTML、XML、JSON、RSS 等形式表现出来。

安全是 web2py 的核心，这里我们的目标是锁定一切以保持系统和数据安全。因此，我们的数据层消除 SQL 注入。模板语言防止跨站点脚本漏洞，web2py 生成的表单提供了字段验证，阻止跨站点请求伪造。密码总是在经过哈希运算之后才存储。默认时，会话被存储在服务器端，以阻止 cookie 篡改；会话 cookie 采用 uuid，以阻止 cookie 窃取。

web2py 始终是从用户角度出发而设计，通过长期的内部优化变得更快和更精简，并保证向后兼容性。

web2py 是免费使用的。如果您从中受益，我们希望您能以您选择的任何形式回报社会。

2011 年，InfoWorld 杂志评论了六个最流行的基于 Python 的全堆栈 web 框架，web2py 名列第一。同年，web2py 赢得最佳开源开发软件奖 Bossie Award。

第 1 章 引言

web2py[1]是一种免费的、开源的 web 开发框架，用于敏捷地开发安全的、数据库驱动的 web 应用；web2py 采用 Python[2]语言编写，并且可以使用 Python 编程。web2py 是一个完整的堆栈框架，也就是说它包含了开发完整功能的 web 应用所需的所有组件。web2py 被设计来指导 web 开发人员遵循良好的软件工程实践，如使用模型(Model)、视图(View)、控制器(Controller) (MVC) 模式。web2py 将数据表达 (the model) 从数据表示 (the view) 和应用逻辑及工作流 (the controller) 中分开。web2py 提供的库可以帮助开发者分别设计、实施和测试 MVC 的每一部分，并能使它们一起工作。web2py 是为了安全而构建的。这意味着遵循成熟的方法，它能自动处理许多可能导致安全漏洞的问题。例如，web2py 验证所有输入（防止注入攻击），转义所有输出（防止跨站点脚本攻击），重命名上传文件（防止目录遍历攻击）。在与安全有关的方面，web2py 没有留给应用程序开发人员选择的余地。web2py 中包含数据库抽象层 (DAL)，它能够动态写入 SQL，因此开发人员不需要自己写。DAL 知道如何透明地生成支持

SQLite[sqlite]、MySQL[mysql]、PostgreSQL[postgres]、MSSQL[mssql]、FireBird[firebird]、Oracle[oracle]、IBM DB2[db2]、Informix[informix]以及 Ingres[ingresdb]的 SQL 语句。当在谷歌 App Engine (GAE)^[gae]上运行时，DAL 也能生成函数调用 Google Datastore。实验时，我们支持更多的数据库。请查看 web2py 网站和邮件列表，获取最新的支持。一旦有一个或多个数据库表被定义，web2py 也能生成一个全功能的基于 web 的数据库管理接口来访问数据库和表。web2py 与其它 web 框架的不同之处在于，它是唯一全面支持 web2.0 范例的框架，在这里 web 就是计算机。实际上，web2py 不需要安装或配置，它能在任何支持 Python 的体系结构 (Windows, Windows CE, Mac OS X, iOS, Unix/Linux) 上运行，应用程序的开发、部署和维护可以通过本地或远程 web 接口完成。web2py 支持 CPython (C 语言实现) 或 Jython (Java 语言实现)，虽然官方声称仅支持 2.5 版本，但实际支持的版本包括 2.4, 2.5, 2.6, 2.7，这保证了应用程序的后向兼容性。web2py 提供了一个票据系统。如果出现错误，系统会发出一个票据给用户，并记录错误信息供管理员查看。web2py 是开源的，在 LGPL 版本 3 许可证下发布。

web2py 的另一个特点是开发者承诺在未来版本中保持后向兼容性。从 2007 年 10 月 web2py 首次发布至今，我们一直都是这样做的。尽管 web2py 增加了新功能，修复了漏洞，然而如果一个程序在 web2py 1.0 上能运行，那么它现在还能运行。

下面给出一些 web2py 语句的例子来展示它的功能和简洁性。代码如下：

```
1 db.define_table('person', Field('name'), Field('image', 'upload'))
```

以上代码创建了一个“person”数据库表，该表包含两个字段，即“name”字符串和“image”，image 是需要上传的图片（实际图片）。如果该表已经存在，但是与上述定义不匹配，则该表将会被妥善更改。

给定上述定义表，代码如下：

```
1 form = crud.create(db.person)
```

创建了一个插入表单，该表允许用户上传图片。它还会验证提交的表单，以安全的方式重命名上传的图片，并将图片存储到文件中。同时，向数据库中插入相应记录，以防止重复提交，如果用户提交的数据未能通过验证，将通过添加错误信息来修改表单，代码如下：

```
1 @auth.requires_permission('read', 'person')
2 def f(): ....
```

上述代码将阻止访问者进入函数 f，除非访问者所在的组有权限读取“person”中的记

录。如果访问者未登录，他将被定向到 login（登录）页面（由 web2py 默认提供）。

下面的代码将嵌入页面组件：

```
1 {{=LOAD('other_controller','function.load',ajax=True, ajax_trap=True)}}
```

上述代码指示 web2py 以视图形式加载其它控制函数生成的内容（适用于任何函数）。它通过 Ajax 加载内容，并将内容嵌入当前网页（使用当前布局而不是其它控制函数布局），这样就能捕获加载内容中的所有表单，这样不需要重新加载网页也能通过 Ajax 提交表单。它也能 LOAD 非 web2py 应用的内容。

LOAD 助手允许应用程序的模块化设计；本书最后一章将对此进行论述。

1.1 原则

Python 编程通常遵循以下基本原则：

- 不重复自己（DRY）。
- 仅有一种实现方式。
- 明确比含蓄更好。

在 web2py 中，通过强制开发者使用可靠成熟的软件工程实践，遏制代码重复，保证完全遵守前两条原则。web2py 能指导开发者完成几乎所有 web 应用开发中的常见任务（创建和处理表单、管理会话、小甜饼“cookie”、错误等等）。

web2py 对第 3 个原则的处理与其它框架有所不同，有时与前两个原则相冲突。尤其是 web2py 不会导入用户应用，而是在预定义的情况下执行。这会暴露 Python 和 web2py 关键字。

对某些人来说，这看起来就像魔术，但它不是这样的。简单地说，在实践中有些模块已经自动导入了，而不需要开发者导入。web2py 试图避免其它框架下存在的令人讨厌的特征，即开发者需要在每个模型和控制器的顶部导入相同的模块。web2py 通过导入自有模块节约了时间，避免了错误，这遵循了不重复自己和仅有一种实现方式的精髓。

如果开发者想使用其它 Python 模块或第三方模块，这些模块必须明确导入，就像开发任何其它 Python 程序一样。

1.2 Web 框架

在其最基本的层面上，web 应用包含了一组程序（或者函数），当用户访问相应的 URL 时，该程序将被执行。同时，程序的输出返回给用户，并呈现在浏览器中。

web 框架是为了让开发者更快、更简便、无差错的开发新应用。它通过提供 API 和开发工具，以减少代码编写量。

开发 web 应用的两个经典方法是：

- 通过编程生成 HTML 代码[[html:w](#),[html:o](#)]。
- 将代码嵌入 HTML 页面中。

早期的 CGI 脚本遵循第一种模型。下列脚本遵循第二种模型，例如 PHP[[php](#)]（代码用 PHP 编写，类似 C 语言）、ASP（代码用 Visual Basic 编写）以及 JSP（代码用 Java 编写）脚本。

这里举一个 PHP 程序的例子，执行时，从数据库中获得数据，并返回一个显示选中记录的 HTML 页面。

```
1 <html><body><h1>Records</h1><?
2 mysql_connect(localhost,username,password);
3 @mysql_select_db(database) or die( "Unable to select database");
4 $query="SELECT * FROM contacts";
5 $result=mysql_query($query);
```

```

6 mysql_close();
7 $i=0;
8 while ($i < mysql_numrows($result)) {
9 $name=mysql_result($result,$i,"name");
10 $phone=mysql_result($result,$i,"phone");
11 echo "<b>$name</b><br>Phone:$phone<br /><br /><hr /><br />";
12 $i++;
13 }
14 ?></body></html>

```

这种方法的问题在于，程序代码嵌入到 HTML 中，但是这个程序在生成额外的 HTML 的同时，还要生成 SQL 语句查询数据库，应用的不同层次交织在一起，代码变得难以阅读和难以维护。对于 Ajax 应用程序，情况就更糟了，随着应用页数（文件）的增加，复杂性也增加。

上述例子的功能，在 web2py 中可用两行 Python 代码来表达：

```

1 def index():
2 return HTML(BODY(H1('Records'), db().select(db.contacts.ALL)))

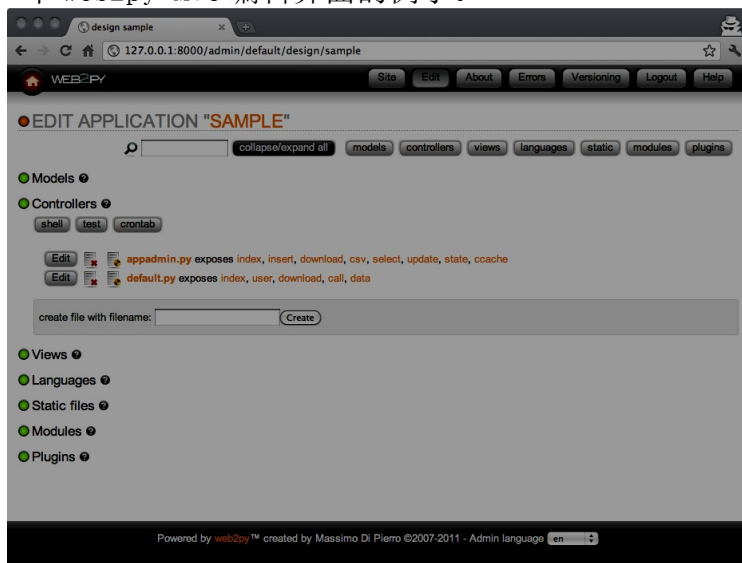
```

在这个简单的例子中，页面结构由 HTML，BODY 和 H1 对象程序化表示；通过 select 语句查询数据库 db；最后，所有结果都被序列化为 HTML 代码。注意 db 不是关键字，而是一个用户定义的变量。为了避免混淆，我们将始终使用 db 这一术语来指代数据库连接。

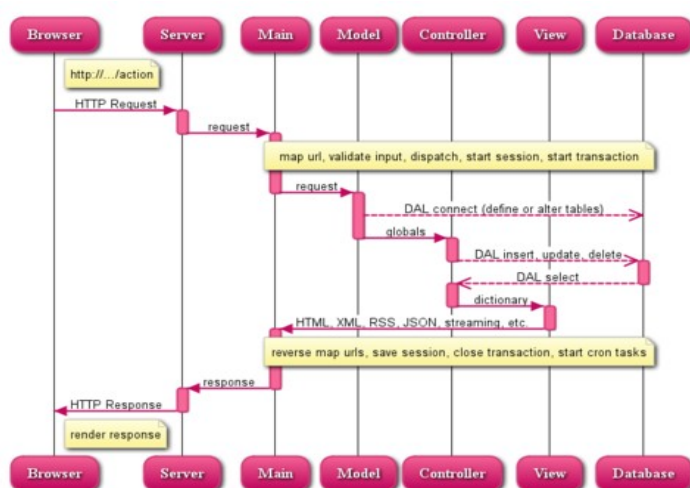
web 框架通常分为两种类型：一种是“胶水”框架，通过组合（粘合）几个第三方组件来构造；另一种是“全功能栈”框架，通过创建特别设计的紧密整合和协作工作的组件来构造。web2py 是一个全堆栈框架，几乎所有组件都是从头构建的，并被设计成协同工作，但是这些组件在 web2py 框架之外同样能发挥功能。例如，数据库抽象层(DAL)或模板语言都能独立于 web2py 框架使用，只要将 gluon.dal 或 gluon.template 导入你的 Python 应用即可。gluon 是包含系统库的 web2py 模块的名称，一些 web2py 库依赖 web2py 的其它部分，例如，建立和处理来自数据库表格的表单。web2py 也能够同第三方 Python 库配合使用，包括其它模板语言和 DAL，但它们之间的结合没有原配组件那么紧密。

1.3 模型-视图-控制器（Model-View-Controller）

web2py 鼓励开发人员将数据表达（Model）、数据表示（View）和应用 workflow（Controller）分离。让我们再考虑前面的例子，看看如何围绕该例建立一个 web2py 应用。下面是一个 web2py MVC 编辑界面的例子。



web2py 中一个请求的典型 workflow 描述如下：



在图中：

- 服务器可以是 web2py 内置服务器或第三方服务器，例如 Apache。服务器可以处理多线程。
- “main”是主要的 WSGI 应用。它负责处理所有常见任务和封装用户应用，它处理 cookies、sessions、transactions、URL 地址解析及反向地址解析和分发。

如果 web 服务器没有处理的话，它能服务和流静态文件。

- Model、View、Controller 组件构成了用户应用。
- 同一个 web2py 实例可以承载多个应用。
- 虚线箭头表示与数据库引擎的通信，数据库查询可以使用 SQL 语言（不推荐）或使用 web2py DAL 语言（推荐），这样 web2py 应用代码不依赖于特定数据库引擎。
- 分发器将请求的 URL 映射成控制器中的函数调用，函数的输出可以是字符串或符号字典（哈希表），字典中的数据将被呈现成视图。如果用户请求 HTML 页面（默认情况），字典将被呈现成 HTML 页面；如果用户以 XML 请求同一页面，web2py 将会尝试找到一个能将字典呈现成 XML 格式的视图。开发人员可以创建视图将页面呈现成任何已经支持的协议（HTML、XML、JSON、RSS、CSV、RTF）或者另外的自定义协议。
- 所有的调用都被封装到一个事务(transaction)之中，并且任何未捕获到的异常都将导致事务回滚。如果请求成功，事务将被提交。
- web2py 还能自动处理 sessions 和 session cookies，并且当事务被提交的时候，相应的 session 也被保存，除非另有说明。
- 还能注册经常性的任务（通过 cron）以定时和/或在特定的任务完成(action)之后执行，用这种方式可以在不影响用户浏览的情况下，在后台运行耗时长、计算量大的任务。

这里给出一个最小的、完整的 MVC 应用，它由 3 个文件组成：

“db.py”是模型：

```
1 db = DAL('sqlite://storage.sqlite')
2 db.define_table('contact',
3   Field('name'),
4   Field('phone'))
```

它连接数据库（在本例中是指存储在 storage.sqlite 文件中的一个 SQLite 数据库），并定义了一个名为 contact 的表。如果该表不存在，web2py 将在后台透明的创建它，并生成适用于特定数据库引擎的 SQL 语句。开发人员可以看到生成的 SQL， 如果用

MySQL、PostgreSQL、MSSQL、FireBird、Oracle、DB2、Informix、Interbase、Ingres 或谷歌 App Engine (SQL 和 NoSQL) 数据库代替默认数据库 SQLite，就不需要修改数据库后台的代码。

当表格被定义并创建好之后，*web2py* 还会生成一个功能完整的基于 *web* 的数据库管理界面，该界面称作 *appadmin*，通过它访问数据库和表。

“*default.py*”是控制器：

```
1 def contacts():
2     grid=SQLFORM.grid(db.contact, user_signature=False)
3     return locals()
```

在 *web2py* 中，URL 被映射成 Python 模块和函数调用。在本例中，控制器仅包含一个名为 *contacts* 的函数（或“action”）。Action 可能返回字符串（返回的网页）或 Python 字典（一组键：值对）或一组局部变量（如同本例）。如果函数返回字典，它将被传送给视图，该视图与控制器/函数同名，并返回一个网页，在本例中，函数 *contacts* 生成一个表 *db.contact* 的选择（select）/搜索（search）/创建（create）/更新（update）/删除（delete）网格，并将该网格返回给视图。

“*default/contacts.html*”是视图：

```
1 {{extend 'layout.html'}}
2 <h1>Manage My Contacts</h1>
3 {{=grid}}
```

在相应的控制器函数（action）被执行后，*web2py* 会自动调用视图，该视图的作用是将返回字典中的变量呈现成 HTML。视图文件是用 HTML 语言编写的，并用分隔符 `{{and}}` 分隔嵌入的 Python 代码，这完全不同于 PHP 代码，因为嵌入到 HTML 中的码是“表示层”码。“*layout.html*”文件由 *web2py* 提供，并在视图文件的开始被引用，该文件构成了所有 *web2py* 应用的基本布局，该布局文件可以很容易地被修改或替换。

1.4 为什么选择 web2py

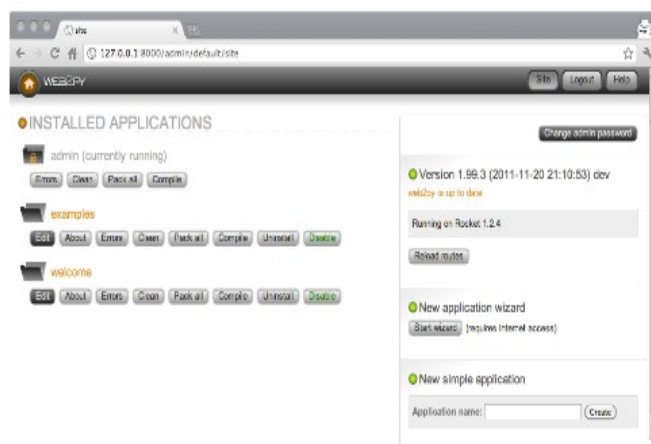
web2py 是众多 web 应用框架中的一种，但是它有引人注目的、独特的功能。*web2py* 最初被开发成一种教学工具，最初的开发动机如下：

- 在不牺牲功能的前提下，方便用户学习服务器端 web 开发。为此，*web2py* 被设计成无需安装、无需配置，无依赖性（除了源代码的发行版要求 Python 2.5 和它的标准库模块外），并通过 Web 浏览器界面公开绝大部分功能。
- *web2py* 从刚推出起一直到今天都保持稳定，因为它遵循自上而下的设计原则，即在它被编写以前所有编程接口（API）都已经被设计好，甚至加入了新功能，它的后向兼容性也不会被破坏，即便将来增加了新功能，也能实现兼容。
- *web2py* 前瞻性地解决许多重要的安全问题，这些问题困扰着许多现代 Web 应用，将在下面开放 Web 软件安全工程 [owasp] 中介绍。
- *web2py* 是轻量级的。其核心库，包括数据库抽象层、模板语言和所有帮助对象加在一起只有 1.4MB。整个源代码包括示例应用和图像在内，也只有 10.4MB。
- *web2py* 占用资源少，运行速度快。它使用由 Timothy Farrell 开发的 Rocket^[rocket] WSGI 服务器，它与采用 *mod_wsgi* 的 Apache 一样快。我们的测试表明，在一台普通的 PC 上，不访问数据库的动态网页平均响应时间大约 10ms，DAL 开销小，通常小于 3%。
- *web2py* 在模块、控制器和视图中采用 Python 句法，但并不导入模块和控制器（其余 Python 框架采用导入方式），而是去执行它们。这意味着不必重启 web 服务器即可进行应用的安装、卸载和修改，不同的应用可以共存而不会导致模块的互相干扰。
- *web2py* 使用数据库抽象层取代对象关系映射 (ORM)。从概念角度来说，这意味着不同的

数据库表被映射成不同的 Table 类实例，而不是不同的类，同时记录被映射成 Row 类的实例，而不是相应的 Table 类的实例。从实用的角度来看，这意味着 SQL 句法与 DAL 句法几乎一一对应，DAL 在底层没有复杂的元类（metaclass）编程，这与流行的 ORM 不同，复杂的编程将增加延迟。

WSGI ^[wsgi:wsgi:o]（Web 服务器网关接口）是一种新兴的 Web 服务器和 Python 应用之间通信的 Python 标准。

下面是 web2py *admin*（主要管理）界面的截图：



1.5 安全性

开放 Web 应用安全工程^[owasp]（OWASP 的）是一个自由和开放的全球社区，专注于改善应用软件的安全性。

OWASP 列出了 web 应用安全方面的十大问题。在这里给出该列表，并陈述 web2py 是如何解决这些问题的：

- “跨站点脚本攻击 (XSS)：当应用获取用户提交的数据并返回 web 浏览器时，如果不首先进行验证或编码，XSS 漏洞就可能出现。XSS 允许攻击者在受害者浏览器中执行脚本，从而劫持用户会话，毁损网站，还可能引入蠕虫。”在默认情况下，web2py 将会转意呈现在视图中的所有变量，防止跨站点脚本攻击。
- “注入攻击 (Injection Flaws)：注入攻击在 Web 应用中非常普遍，特别是 SQL 注入，当用户提交的数据被作为命令或查询的一部分发送到解释器时，注入攻击就可能发生，攻击者的恶意数据欺骗解释器执行异常的命令或更改数据。”web2py 通过采用数据库抽象层使得 SQL 注入攻击不可能发生。通常情况下，SQL 语句并不是由开发人员编写的，而是由 DAL 动态生成的，从而确保所有插入的数据都被适当地转意。
- “执行恶意文件” (Malicious File Execution)：脆弱的远程文件包含的代码 (RFI) 可能被攻击者加入恶意代码和数据，造成毁灭性的攻击，例如服务器瘫痪。”web2py 只允许运行对外暴露的函数，从而防止恶意文件的执行，导入的函数绝不会被暴露，暴露的仅有行为 (action)。web2py 采用了基于 WEB 的管理接口，使得非常容易跟踪暴露的行为。
- “不安全的直接对象引用 (Insecure Direct Object Reference)：当开发者把内部引用对象，例如文件、目录、数据库记录或密钥，作为 URL 地址或表单的参数时，不安全的直接对象引用攻击就可能发生，攻击者能够操控这些引用，在未经授权的情况下访问其它对象。”web2py 没有暴露任何内部对象，此外 web2py 还会验证所有的 URL，从而防止目录遍历攻击。web2py 还提供了一个简单的机制，以创建自动验证所有输入的值。
- “跨站点请求伪造 (Cross Site Request Forgery)：CSRF 攻击迫使一个已经登录的受

害者浏览器，向脆弱的网络应用发送一个预先验证的请求，该请求又迫使受害者的浏览器执行有利于攻击者的故意行为，web 应用有多强大，CSRF 就有多强大。” web2py 通过在表单中加入一次性随机令牌，防止 CSRF 攻击和偶然的表单重复提交，另外 web2py 对会话 cookie 使用了 UUID。

- “信息泄露和错误处理不当 (Information Leakage and Improper Error Handling)：应用可能无意中泄露它们的配置、内部运作的信息，或者在各种应用中侵犯隐私，攻击者可以通过该缺陷窃取获得敏感数据或发起更严重的攻击。” web2py 中包含一个票据系统，任何错误都不会导致代码暴露给用户，所有的错误都被记录，框架会发送一个票据给用户，用来进行错误追踪。只有系统管理员才能访问错误和源代码。
- “验证和会话管理中断 (Broken Authentication and Session Management)：用户账户信息和会话令牌常常没有被妥善加以保护，攻击者通过获取用户密码、密钥或者验证令牌冒用其它用户的身份。” web2py 提供了内置的管理员验证机制，它能为每一个应用独立地管理会话。当客户端不是 `localhost` 时，管理接口也能够强制使用安全的会话 cookie，对于应用来说，web2py 框架提供了功能强大的基于角色的接入控制 API。
- “不安全加密存储 (Insecure Cryptographic Storage)：Web 应用程序很少使用合理的加密算法保护数据和凭证，攻击者利用加密不足的数据进行身份盗用和其它犯罪，例如信用卡诈骗。” web2py 使用 MD5 或 HMAC+SHA-512 哈希算法来保护用户存储的密码，也可以采用其它算法。
- “不安全通信 (Insecure Communications)：当需要保护敏感通信时，应用常常不能加密网络数据流。” web2py 包含支持 SSL^[ssl] 的 Rocket WSGI 服务器，它也能使用 Apache 或 Lighttpd 和 mod-ssl 进行通信 SSL 加密。
- “未能限制 URL 访问” (Failure to Restrict URL Access)：应用经常通过不显示的链接或 URL 来阻止未授权用户，仅为保护敏感功能。攻击者正是利用这一漏洞，通过直接接入这些 URL 来进行未授权的操作。” web2py 将 URL 请求映射到 Python 模块和函数。web2py 包含了一套验证机制，可以定义那些函数是公有的，那些是需要经过验证和授权才能访问，基于角色的接入控制 API 允许开发者限制访问那些基于登录、群成员或接入许可群的函数。权限是非常精细的，并且可以结合 CRUD 允许访问，例如，允许访问特定的表和记录。web2py 还支持数字签名的 URL，并且提供了 API 对 ajax 回调进行数字签名。对 web2py 的安全性评论，你可以在参考文献 [\[pythonsecurity\]](#) 中找到评论性结果。

1.6 框架内容

你可以通过官方网站下载 web2py：

1 <http://www.web2py.com>

web2py 由如下组件构成：

- 库 (libraries)：提供 web2py 核心功能，可通过编程访问。
- web 服务器：Rocket WSGI web 服务器。
- 管理 (admin) 应用：用于创建、设计和管理其它 web2py 应用，admin 提供了一个完整的基于 web 的集成开发环境 (IDE)，用于开发 web2py 应用，它还包括其它功能，如基于 web 的测试和 web 的壳 (shell)。
- 示例 (examples) 应用：包含文档和交互示例。应用示例是官方网站 web2py.com 的副本，并包含 epydoc 文档。
- 欢迎 (welcome) 应用：是其它应用的基本构建模板。默认时，它包含一个纯 CSS 层叠菜单和用户认证（在第九章讨论）。

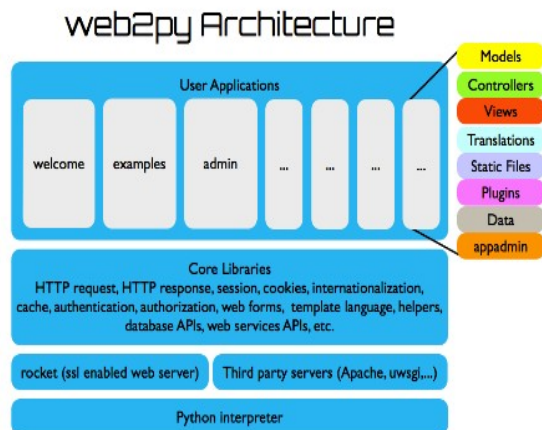
web2py 以源代码或二进制形式发行，适用于微软 windows 或 Mac OS X 操作系统。

源代码发行版可以在任何支持 Python 的平台上运行，并且包含了上述所有组件。为了

运行源代码，你需要预先安装 Python 2.5。同时，还需要安装一种支持的数据库引擎，为了测试和轻量级需求的应用，你可以使用内置于 Python 2.5 中的 SQLite 数据库。

web2py 的二进制版本（适用于 Windows 和 Mac OS X）包含 Python2.5 解释器和 SQLite 数据库。从技术上讲，这两个部分并不是 web2py 的组件，将它们包含在二进制发行版中，是为了使您能够直接运行 web2py。

下图描绘了 web2py 的整体结构：



1.7 授权

web2py 在 LGPL 版本 3 许可证下授权，关于该许可证的全部内容，请参阅参考文献 [\[lglp13\]](#)。

依照 LGPL 许可，用户可以：

- 将 web2py 与您的应用一起重新发行（包括官方 web2py 二进制版本）
- 在您希望的任何许可证下，发布您开发的使用官方 web2py 库文件的应用，与此同时，您必须遵守：
 - 在文档中明确表示，您的应用使用了 web2py
 - 依照 LGPLv3 许可证，发布对 web2py 库的任何修改

LGPLv3 许可证包含以下声明：

在适用法律允许的范围内，本程序没有担保。除另有书面注明外，版权持有人和/或其它方按现状提供程序，没有任何形式的明示或暗示的担保，包括但不限于针对特定用途的适销性和适用性暗示担保。程序的质量和性能的全部风险由你承担，如果程序被证明有缺陷，你承担所有必要的维修、修理或更正的费用。

除非适用法律要求或书面同意，按照上述允许修改和/或传达程序的任何版权持有人或任何其它方都不承担损害赔偿责任，包括因为使用或没有能力使用程序而产生的任何一般的、特殊的、偶然或必然损害（包括但不限于数据丢失、数据呈现不准确、由您或第三方遭受的损失或程序未能与任何其它程序协作），即使持有人或其它方已被告知此类损害的可能性。

早期版本

web2py 的早期版本 1.0.*-1.90.* 在 GPL2 许可下发布，商业授权例外，为了实用目的，它非常类似于当前的 LGPLv3。

第三方软件

在 gluon/contrib/文件夹、各种 JavaScript 和 CSS 文件中，web2py 包含第三方软件。在文件中表述的原许可证下，这些文件与 web2py 一起发布。

1.8 致谢

web2py 最初由 Massimo Di Pierro 开发并拥有版权，2007 年 10 月发行第一个版本 (1.0)。此后，web2py 被许多用户使用，其中一些用户也贡献了相关的漏洞报告、测试、调试、补丁和本书的校对。

按照名字的字母顺序，一些主要贡献者如下：

Alexey Nezhdanov, Alvaro Justen, Andrew Willimott, Angelo Compagnucci, Anthony Bastardi, Antonio Ramos, Arun K. Rajeevan, Attila Csipa, Bill Ferret, Boris Manojlovic, Branko Vukelic, Brian Meredyk, Bruno Rocha, Carlos Galindo, Carsten Haese, Chris Clark, Chris Steel, Christian Foster Howes, Christopher Smiga, CJ Lazell, Cliff Kachinske, Craig Younkins, Daniel Lin, David Harrison, David Wagner, Denes Lengyel, Douglas Soares de Andrade, Eric Vicenti, Falko Krause, Farsheed Ashouri, Fran Boon, Francisco Gama, Fred Yanowski, Gilson Filho, Graham Dumpleton, Gyuris Szabolcs, Hamdy Abdel-Badeea, Hans Donner, Hans Murx, Hans C. v. Stockhausen, Ian Reinhart Geiser, Ismael Serratos, Jan Beilicke, Jonathan Benn, Jonathan Lundell, Josh Goldfoot, Jose Jachuf, Josh Jaques, José Vicente de Sousa, Keith Yang, Kenji Hosoda, Kyle Smith, Limodou, Lucas D'Ávila, Marcel Leuthi, Marcel Hellkamp, Marcello Della Longa, Mariano Reingart, Mark Larsen, Mark Moore, Markus Gritsch, Martin Hufsky, Martin Mulone, Mateusz Banach, Miguel Lopez, Michael Willis, Michele Comitini, Nathan Freeze, Niall Sweeny, Niccolo Polo, Nicolas Bruxer, Olaf Ferger, Omi Chiba, Ondrej Such, Ovidio Marinho Falcao Neto, Pai, Paolo Caruccio, Patrick Breitenbach, Phyo Arkar Lwin, Pierre Thibault, Ramjee Ganti, Robin Bhattacharyya, Ross Peoples, Ruijun Luo, Ryan Seto, Scott Roberts, Sergey Podlesnyi, Sharrieff Aina, Simone Bizzotto, Sriram Durbha, Sterling Hankins, Stuart Rackham, Telman Yusupov, Thadeus Burgess, Tim Michelsen, Timothy Farrell, Yair Eshel, Yarko Tymciurak, Younghyun Jo, Vidul Nikolaev Petrov, Vinicius Assef, Zahariash.

若有遗漏，我表示抱歉。

我特别感谢 Jonathan, Mariano, Bruno, Martin, Nathan, Simone, Thadeus, Tim, Iceberg, Denes, Hans, Christian, Fran 和 Patrick 对 web2py 作出的重大贡献。感谢 Anthony, Alvaro, Bruno, Denes, Felipe, Graham, Jonathan, Hans, Kyle, Mark, Michele, Richard, Robin, Roman, Scott, Shane, Sharrieff, Sriram, Sterling, Stuart, Thadeus 对本书所做的校对，他们的贡献是无价的。如果你在本书中发现任何错误，责任完全归咎于我，错误可能是由于是编辑匆忙造成的，我还要感谢 Wiley Custom Learning Solutions 的 Ryan Steffen 帮助我出版了这本书的第一版。

web2py 中包含以下作者的代码，在此我也表示感谢：

Guido van Rossum for Python^[python], Peter Hunt, Richard Gordon, Timothy Farrell for the Rocket^[rocket] web server, Christopher Dolivet for EditArea^[editarea], Bob Ippolito for simplejson^[simplejson], Simon Cusack and Grant Edwards for pyRTF^[pyrtf], Dalke Scientific Software for pyRSS2Gen^[pyrss2gen], Mark Pilgrim for feedparser^[feedparser], Trent Mick for markdown2^[markdown2], Allan Saddi for fcgi.py, Evan Martin for the Python memcache module^[memcache], John Resig for jQuery^[jquery].

本书封面由 Young Designers 的 Peter Kirchner 设计。

感谢 Helmut Epp (DePaul University 教务长)、David Miller (the College of

Computing and Digital Media of DePaul University 院长) 和 Estia Eichten (MetaCryption LLC 会员) 对我一直的信赖和支持。

最后, 我想感谢我的妻子 Claudia 和儿子 Marco, 他们体谅我花了许多时间来开发 web2py、与用户和合作者交换电子邮件及撰写本书。这本书是献给他们的。

1.9 关于本书

除引言外, 本书还包含以下章节:

- 第 2 章简要介绍了 Python。假定读者具有编程的基本知识和面向对象编程的基本概念, 例如循环、条件转移、函数调用和类, 本章主要包括 Python 的基本句法和本书所用的 Python 模块的例子。如果您已经熟悉 Python 语言, 可以跳过该章。
- 第 3 章展示如何启动 web2py, 介绍了管理界面, 引导读者熟悉了以下复杂性不断增加的例子: 返回字符串应用、计数器应用、图片日志、完善的 wiki 应用, 该应用允许上传图片、用户评论, 提供验证、授权、web 服务和 RSS 订阅。阅读本章时, 您可能需要参考第 2 章有关 Python 句法和后面章节中所用函数的详细介绍。
- 第 4 章系统化介绍 web2py 的核心结构和库文件: URL 映射、请求、响应、会话、缓存、cron、国际化和一般工作流程。
- 第 5 章介绍用于创建视图的模板语言。展示了如何向 HTML 中嵌入 Python 代码, 演示了帮助对象的使用方法 (能生成 HTML 的对象)。
- 第 6 章介绍数据库抽象层或 DAL。通过一系列的例子介绍了 DAL 句法。
- 第 7 章介绍表单、表单验证和表单处理。FORM 是低层次的用于创建表单的帮助对象, SQLFORM 是高层次的用于创建表单的帮助对象, 本章还讨论了创建/读取/更新/删除 (CRUD) API。
- 第 8 章介绍了与发送邮件和 SMS 有关的信息。
- 第 9 章介绍 web2py 中的用户验证、用户授权和扩展的基于角色的访问控制机制。还介绍了用于用户验证的 mail 设置和 CAPTCHA。在本书第 3 版中, 我们增加了有关与第三方验证机制集成的内容, 如 OpenID、OAuth、Google、Facebook、LinkedIn 等。
- 第 10 章涉及 web2py 创建 web 服务的内容。我们提供了一些有关集成的例子, 例如通过 Pyjamas 和 Google Web Toolkit 集成, 通过 PyAMF 和 Adobe Flash 集成。
- 第 11 章涉及 web2py 和 jQuery 使用的方法。web2py 主要用于服务器端编程, 但它包含 jQuery, 因为我们发现 jQuery 是最好的开源 JavaScript 库, 能用于效应和 Ajax。本章介绍如何有效地使用 jQuery 和 web2py。
- 第 12 章介绍 web2py 组件和插件以及构建模块化应用的方法。我们给出了一个使用插件实现许多常用功能的例子, 例如图表、注释、标注和 wiki。
- 第 13 章涉及 web2py 应用的生产部署。我们主要陈述了 3 种可能的生产环境: 在一个 Linux web 服务器或一组服务器 (我们考虑的主要替代部署), 在微软的 Windows 环境下运行, 在 Google 应用引擎上部署。在本章中, 我们还讨论了安全性和扩展性问题。
- 第 14 章介绍了一些用于解决特定问题的方法, 包括升级、地理编码、分页、Twitter API 等等。

本书只介绍 web2py 的基础功能及 web2py 附带的 API。本书不对 web2py appliances (即现成的应用) 作介绍。

您可以从相应的网站^[appliances]下载 web2py 应用范例。

您可以在 AlterEgo^[alterego]上找到讨论的其它主题, 进行交互式 web2py 常见问题解答。

本书的写作采用了 markmin 句法, 它能自动转换成 HTML、LaTeX 和 PDF 文档。

1.10 风格要素

在使用 Python 编程时，PEP8^[style] 包含优良的样式实践。你会发现 web2py 并不总是遵循这些规范。这不是因为遗漏或疏忽，但是我们认为 web2py 的用户应该遵循这些规则，并且我们鼓励这样做。在定义 web2py 帮助对象时，我们选择不遵循 PEP8 的某些规则，以尽量减少用户自定义对象与帮助对象发生命名冲突的概率。

例如，表示<div>的类被称为 DIV，然而根据 Python 样式参考，它应该被称为 Div，我们相信，对于这个具体的例子，使用全大写的“DIV”是一个更自然的选择。此外，如果程序员愿意的话，这种方法让他们自由选择创建一个称为“Div”的类，我们的句法能自然地映射成大多数浏览器（包括，例如，火狐）使用的 DOM 标记。

根据 Python 的样式指南，常量的命名应使用全大写字符串，但变量的命名不能这么做。继续我们的例子，即使考虑 DIV 是一个类，它也是一个用户不应该修改的特殊类，因为修改它会破坏其它的 web2py 应用。因此，我们认为 DIV 类应该被当做常量对待，进一步证明了我们选择的记号是合理的。

总之，下列习惯应该被遵守：

- HTML 帮助对象和验证都采用大写（例如 DIV, A, FORM, URL），这么做的理由前面已经讨论过了。
- 转换对象 T 采用大写，尽管事实上它只是类的一个实例而不是一个类。从逻辑上说，转换(translator)对象执行的动作类似于 HTML 帮助对象，它会影响演示文稿的呈现。此外，在代码中需要对 T 轻松地定位，它必须有一个简短的名称。
- DAL 类遵循 Python 样式指南（首字母大写），例如 Table、Field、Query、Row、Rows 等等。

对于所有其它情况，我们相信已尽可能多的遵循了 Python 样式指南（PEP8）。例如，所有的实例对象都是小写（request, response, session, cache），所有的内部类都是大写。

在本书的所有例子中，web2py 关键字都以粗体显示，字符串和注释都以斜体显示。

第 2 章 Python 语言

2.1 关于 Python

Python 是一种通用的高级编程语言。它的设计思想是强调程序员的工作效率及代码的可读性。它有一个最低限度的核心语法，包括很少的基本命令以及简单语义，但与此同时它又有一个大规模的全面的标准库，包括许多面向底层操作系统 (OS) 函数的应用编程接口 (API)，Python 代码，在简约的同时，定义了内置对象，例如列表 (list)、元组 (tuple)、哈希表 (字典 dict) 以及任意长度整型数 (long)。

Python 支持多重编程架构，包括面向对象 (class)、指令 (def)、以及函数 (lambda) 编程，Python 还有一套动态类型系统以及通过引用计数实现的（类似于 Perl、Ruby 和 Scheme）自动内存管理。

Python 最初由 Guido van Rossum 于 1991 年发布。该语言是开源的、基于社区开发模型的，由非盈利 Python 软件基金会管理，可以实现 Python 语言的解释器及编译器有许多，包括 Java 版的 Jython，不过在此简要回顾中，我们特指由 Guido 开发的 C 语言实现。

在 Python 官方网站上 [2]，您可以找到该语言的许多教程、官方文档以及库指南。

若要获取更多参考文献，我们推荐参考文献 [37] 和 [38] 中的书。

如果您已经熟悉 Python 语言，可以跳过本章节。

2.2 启动

Web2py 的二进制发行版支持微软 Windows 或苹果 OS X 操作系统，该发行文件中包含内置的 Python 解释器。

在 Windows 操作系统下，可以使用如下命令来启动 web2py (在 DOS 提示符下输入)：

```
1 web2py.exe -S welcome
```

在苹果 OS X 操作系统下，在终端窗口中输入如下命令（假如您与 web2py.app 在同一文件夹中）：

```
1 ./web2py.app/Contents/MacOS/web2py -S welcome
```

在 Linux 或其他 Unix 操作系统下，很可能 Python 已经安装过了，如果是这样的话，在 shell 提示符下输入：

```
1 python web2py.py -S welcome
```

如果您没有预先安装 Python 2.5 (或者更新的版本 2.x)，在运行 web2py 之前，您需要下载并安装新的 Python。

-S welcome 命令行选项指示 web2py 运行交互式 shell (壳)，就像 welcome 应用中命令在控制器内被执行，即 web2py 基本构建应用，这向用户暴露了几乎所有的 web2py 的类、对象及函数，这是 web2py 交互式命令行与普通 Python 命令行的唯一不同之处。

对于每个应用，admin 管理接口提供了一个基于 web 的 shell。

对于“welcome”应用，您可以访问 shell。

```
1 http://127.0.0.1:8000/admin/shell/index/welcome
```

您可以使用普通 shell 或基于 web 的 shell 来尝试本章中的所有例子。

2.3 help 和 dir 命令

Python 语言提供了两条命令来获取定义在当前作用域内对象的有关文档，包括内置的和用户定义的。

我们可以寻求有关对象的 help，例如“1”：

```
1 >>> help(1)
2 Help on int object:
3
4 class int(object)
5 |     int(x[, base]) -> integer
6 |
7 |     Convert a string or number to an integer, if possible. A floating point
8 |     argument will be truncated towards zero (this does not include a string
9 |     representation of a floating point number!) When converting a string, use
10 |    the optional base. It is an error to supply a base when converting a
11 |    non-string. If the argument is outside the integer range a long object
12 |    will be returned instead.
13 |
14 |    Methods defined here:
15 |
16 |    __abs__(...)
17 |    x.__abs__() <==> abs(x)
18 ...
```

并且由于“1”是一个整数，我们得到 int 类及其所有方法的描述。这里输出被截断了，因为它很长非常详细。

类似的，我们用命令 dir 获取对象“1”的方法列表：

```
1 >>> dir(1)
2 ['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
3  '__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
4  '__floordiv__', '__getattr__', '__getnewargs__', '__hash__', '__hex__',
5  '__index__', '__init__', '__int__', '__invert__', '__long__', '__lshift__',
6  '__mod__', '__mul__', '__neg__', '__new__', '__nonzero__', '__oct__',
7  '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__',
8  '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
9  '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__',
10 '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
11 '__str__', '__sub__', '__truediv__', '__xor__']
```

2.4 类型

Python 是一种动态类型语言，这意味着变量没有类型，因此也不需要被声明。另一方面，值是有类型的，您可以查询变量包含的值的类型：

```
1 >>> a = 3
```



```
2 >>> print type(a)
3 <type 'int'>
4 >>> a = 3.14
5 >>> print type(a)
6 <type 'float'>
7 >>> a = 'hello python'
8 >>> print type(a)
9 <type 'str'>
```

Python 也包括了原生的数据结构，如列表和字典。

2.4.1 str（字符串）

Python 支持两种不同类型字符串的使用：ASCII 字符串和 Unicode 字符串，ASCII 字符串使用 '...'、"..."、'...' 或 """...""" 分隔，三重引号分隔多行字符串，Unicode 字符串以一个 u 起始，包含 Unicode 字符的字符串跟在后面。通过选择编码，可以将 Unicode 字符串转换成 ASCII 字符串，例如：

```
1 >>> a = 'this is an ASCII string'
2 >>> b = u'This is a Unicode string'
3 >>> a = b.encode('utf8')
```

执行完这三条命令之后，得到的结果 a 是一个存储 UTF8 编码字符的 ASCII 字符串。按照设计，web2py 内部使用 UTF8 编码的字符串。

可以用多种方法将变量写入字符串：

```
1 >>> print 'number is ' + str(3)
2 number is 3
3 >>> print 'number is %s' % (3)
4 number is 3
5 >>> print 'number is %(number)s' % dict(number=3)
6 number is 3
```

最后一种标记是首选，它更明确、不容易出错。

使用 str 或 repr 命令，许多 Python 对象可以被序列化为字符串，例如数字。

这两条命令非常相似，但会产生略有不同的输出。例如：

```
1 >>> for i in [3, 'hello']:
2     print str(i), repr(i)
3 3 3
4 hello 'hello'
```

对于用户定义的类，使用特殊操作符 __str__ 和 __repr__，str 和 repr 可以被定义/重定义，后面会对此作简要介绍，更多相关内容，请参阅 Python 官方文档，repr 总有一个默认值。

Python 字符串的另一重要特征是，像列表一样，它是一个遍历对象。

```
1 >>> for i in 'hello':
2     print i
3 h
4 e
5 l
6 l
7 o
```

2.4.2 list（列表）

Python 列表的主要方法包括添加 (append)、插入 (insert)、删除 (delete)。


```
>>> a = [1, 2, 3]
>>> print type(a)
<type 'list'>
>>> a.append(8)
>>> a.insert(2, 7)
>>> del a[0]
>>> print a
[2, 7, 3, 8]
>>> print len(a)
4
```

列表可以被分片：

```
1 >>> print a[:3]
2 [2, 7, 3]
3 >>> print a[1:]
4 [7, 3, 8]
5 >>> print a[-2:]
6 [3, 8]
```

还可以被连接：

```
1 >>> a = [2, 3]
2 >>> b = [5, 6]
3 >>> print a + b
4 [2, 3, 5, 6]
```

列表可以被遍历，可以通过循环访问：

```
1 >>> a = [1, 2, 3]
2 >>> for i in a:
3     print i
4 1
5 2
6 3
```

列表中的元素不必是相同类型，它们可以是Python对象的任意类型。

有一种很常见的情况，可以使用列表解析（list comprehension）。请看如下代码：

```
1 >>> a = [1, 2, 3, 4, 5]
2 >>> b = []
3 >>> for x in a:
4     if x % 2 == 0:
5         b.append(x * 3)
6 >>> b
7 [6, 12]
```

这段代码清楚地处理了一个项目列表，选择和修改了输入列表的一个子表，并创建了一个新的结果列表。用下列列表解析，可以完全代替上述代码：

```
1 >>> a = [1, 2, 3, 4, 5]
2 >>> b = [x * 3 for x in a if x % 2 == 0]
3 >>> b
4 [6, 12]
```

2.4.3 tuple（元组）

元组类似于列表，但是它的大小和元素是不可变的，而列表的大小和元素是可变的。如

果元组元素是一个对象，则对象的属性是可变的，元组由圆括号分隔。

```
1 >>> a = (1, 2, 3)
```

因此，如下代码适用于列表：

```
1 >>> a = [1, 2, 3]
2 >>> a[1] = 5
3 >>> print a
4 [1, 5, 3]
```

元素赋值不能用于元组：

```
1 >>> a = (1, 2, 3)
2 >>> print a[1]
3 2
4 >>> a[1] = 5
5 Traceback (most recent call last):
6 File "<stdin>", line 1, in <module>
7 TypeError: 'tuple' object does not support item assignment
```

类似于列表，元组是一个遍历对象。请注意，一个元素构成的元组必须包含一个结尾逗号，如下所示：

```
1 >>> a = (1)
2 >>> print type(a)
3 <type 'int'>
4 >>> a = (1,)
5 >>> print type(a)
6 <type 'tuple'>
```

由于元组的不变性，对高效封装对象，元组是非常有用的，而且括号往往是可选的。

```
1 >>> a = 2, 3, 'hello'
2 >>> x, y, z = a
3 >>> print x
4 2
5 >>> print z
6 hello
```

2.4.4 dict（字典）

Python 字典是一个哈希表，将键对象映射成值对象。例如：

```
1 >>> a = {'k': 'v', 'k2': 3}
2 >>> a['k']
3 v
4 >>> a['k2']
5 3
6 >>> a.has_key('k')
7 True
8 >>> a.has_key('v')
9 False
```

键可以是任何哈希类型（整型、字符串，或类能实现`_hash_`方法的任何对象），值可以是任何类型，在同一字典中，不同键和值不一定是相同的类型，如果键是字母数字字符，也能用替代句法声明字典：

```
1 >>> a = dict(k='v', h2=3)
2 >>> a['k']
3 v
4 >>> print a
5 {'k': 'v', 'h2': 3}
```

实用的方法是 `has_key`、`keys`、`values` 和 `items`：

```
1 >>> a = dict(k='v', k2=3)
2 >>> print a.keys()
3 ['k', 'k2']
4 >>> print a.values()
5 ['v', 3]
6 >>> print a.items()
7 [('k', 'v'), ('k2', 3)]
```

`Items` 方法产生一元组列表，每个元组包含一个键和它的对应值。

字典元素和列表元素可以用 `del` 命令删除。

```
1 >>> a = [1, 2, 3]
2 >>> del a[1]
3 >>> print a
4 [1, 3]
5 >>> a = dict(k='v', h2=3)
6 >>> del a['h2']
7 >>> print a
8 {'k': 'v'}
```

在内部，Python 用 `hash` 运算符将对象转换成整数，并用得到的整数确定值存储在那里。

```
1 >>> hash("hello world")
2 -1500746465
```

2.5 关于缩进

Python 使用缩进分隔代码块，一个代码块起始于一行以冒号结尾的代码，下面所有缩进相同或更多的行都被视为同一个代码块。例如：

```
1 >>> i = 0
2 >>> while i < 3:
3 >>>     print i
4 >>>     i = i + 1
5 >>>
6 0
7 1
8 2
```

通常每级缩进使用四个空格。不混用 `tab` 和空格是一个好的策略，那样可能会导致（不可见的）混乱。

2.6 for...in

在 Python 中，你可以循环遍历对象：

```
1 >>> a = [0, 1, 'hello', 'python']
2 >>> for i in a:
```

```
3         print i
4 0
5 1
6 hello
7 python
```

一种常用快捷方式是 xrange，它可以产生一个遍历范围而无需存储整个列表元素。

```
1 >>> for i in xrange(0, 4):
2         print i
3 0
4 1
5 2
6 3
```

这与 C/C++/C#/Java 句法是等价的：

```
1 for(int i=0; i<4; i=i+1) { print(i); }
```

另一种实用的命令是 enumerate，在循环的同时能计数：

```
1 >>> a = [0, 1, 'hello', 'python']
2 >>> for i, j in enumerate(a):
3         print i, j
4 0 0
5 1 1
6 2 hello
7 3 python
```

还有一个关键词 range(a, b, c) 可以返回一个整数列表，从值 a 开始，递增 c，结束值小于 b，其中 a 的默认值是 0，c 的默认值是 1。xrange 与此类似，但不会真正生成列表，只生成一个列表的遍历，因此对于循环来说它更好。

你可以使用 break 跳出循环。

```
1 >>> for i in [1, 2, 3]:
2         print i
3         break
4 1
```

使用 continue，你可以不执行完整个代码块就跳转到下一个循环迭代。

```
1 >>> for i in [1, 2, 3]:
2         print i
3         continue
4         print 'test'
5 1
6 2
7 3
```

2.7 while

在 Python 中，while 循环的工作方式与其它语言类似，在执行循环体前先判断循环次数和测试条件。如果条件为假，循环就终止。

```
1 >>> i = 0
2 >>> while i < 10:
3         i = i + 1
4 >>> print i
```

Python 中没有 loop...until 结构。

2.8 if...elif...else

在 Python 中使用条件是直观的：

```
1 >>> for i in range(3):
2 >>>     if i == 0:
3 >>>         print 'zero'
4 >>>     elif i == 1:
5 >>>         print 'one'
6 >>>     else:
7 >>>         print 'other'
8 zero
9 one
10 other
```

“elif”意味着“else if”，elif 和 else 从句都是可选的，可以有多个 elif 语句，但只能有一个 else 声明。使用 not, and 和 or 运算符可以构造复杂条件。

```
1 >>> for i in range(3):
2 >>>     if i == 0 or (i == 1 and i + 1 == 2):
3 >>>         print '0 or 1'
```

2.9 try...except...else...finally

Python 可以抛出中断，引发异常：

```
1 >>> try:
2 >>>     a = 1 / 0
3 >>> except Exception, e:
4 >>>     print 'oops: %s' % e
5 >>> else:
6 >>>     print 'no problem here'
7 >>> finally:
8 >>>     print 'done'
9 oops: integer division or modulo by zero
10 done
```

如果引起异常，它会被 except 语句捕获，except 语句会被执行，而 else 语句不被执行，如果未引起异常，except 语句不被执行，而 else 语句会被执行，finally 语句始终会被执行。

对于可能出现的不同异常，可以有多种 except 语句。

```
1 >>> try:
2 >>>     raise SyntaxError
3 >>> except ValueError:
4 >>>     print 'value error'
5 >>> except SyntaxError:
6 >>>     print 'syntax error'
7 syntax error
```

else 和 finally 语句都是可选的。

以下是 Python 内置异常+HTTP(由 web2py 定义)列表。

```

1 BaseException
2 +-- HTTP (defined by web2py)
3 +-- SystemExit
4 +-- KeyboardInterrupt
5 +-- Exception
6 +-- GeneratorExit
7 +-- StopIteration
8 +-- StandardError
9 | +-- ArithmeticError
10 | | +-- FloatingPointError
11 | | +-- OverflowError
12 | | +-- ZeroDivisionError
13 | +-- AssertionError
14 | +-- AttributeError
15 | +-- EnvironmentError
16 | | +-- IOError
17 | | +-- OSError
18 | | +-- WindowsError (Windows)
19 | | +-- VMSError (VMS)
20 | +-- EOFError
21 | +-- ImportError
22 | +-- LookupError
23 | | +-- IndexError
24 | | +-- KeyError
25 | +-- MemoryError
26 | +-- NameError
27 | | +-- UnboundLocalError
28 | +-- ReferenceError
29 | +-- RuntimeError
30 | | +-- NotImplementedError
31 | +-- SyntaxError
32 | | +-- IndentationError
33 | | +-- TabError
34 | +-- SystemError
35 | +-- TypeError
36 | +-- ValueError
37 | | +-- UnicodeError
38 | | +-- UnicodeDecodeError
39 | | +-- UnicodeEncodeError
40 | | +-- UnicodeTranslateError
41 +-- Warning
42 +-- DeprecationWarning
43 +-- PendingDeprecationWarning
44 +-- RuntimeWarning
45 +-- SyntaxWarning
46 +-- UserWarning
47 +-- FutureWarning
48 +-- ImportWarning
49 +-- UnicodeWarning

```

关于以上列表中每一项的详细介绍，请参阅 Python 官方文档。

web2py 仅暴露一个新异常，该异常被称作 HTTP，当该异常出现时，它会使程序返回一个 HTTP 错误页面（更多相关内容请参阅第 4 章）。

任何对象都可以引起异常，但推荐的做法是引起异常的对象是内置异常类的扩展。

2.10 def...return

函数声明使用 def，下面是一个典型的 Python 函数：

```
1 >>> def f(a, b):
2     return a + b
3 >>> print f(4, 2)
4 6
```

没有必要（或者方法）指定参数类型或返回类型。这个例子中，定义了一个有两个参数的函数 f。

函数是本章描述的第一个代码句法特征，这是为了引入域和命名空间的概念。在上面的例子中，标识符 a 和 b 在函数 f 的域之外是没有定义的：

```
1 >>> def f(a):
2     return a + 1
3 >>> print f(1)
4 2
5 >>> print a
6 Traceback (most recent call last):
7   File "<pyshell#22>", line 1, in <module>
8     print a
9 NameError: name 'a' is not defined
```

定义在函数域之外的标识符是可以在函数内访问的，请观察下面代码中标识符 a 是如何被处理的：

```
>>> a = 1
>>> def f(b):
>>>     return a + b
>>> print f(1)
2
>>> a = 2
>>> print f(1) # new value of a is used
3
>>> a = 1 # reset a
>>> def g(b):
>>>     a = 2 # creates a new local a
>>>     return a + b
>>> print g(2)
4
>>> print a # global a is unchanged
1
```

如果 a 被修改了，随后的函数调用将使用全局变量 a 的新值，因为函数定义绑定了标识符 a 的存储位置，而不是在函数声明时标识符 a 本身的值；但是，如果 a 被分配到函数 g 内，全局变量 a 将不会受影响因为新的局部变量 a 覆盖了全局变量的值。在创建闭包

（closure）时，可以使用外部域参考。

```
1 >>> def f(x):
2     def g(y):
3         return x * y
4     return g
5 >>> doubler = f(2) # doubler is a new function
6 >>> tripler = f(3) # tripler is a new function
7 >>> quadrupler = f(4) # quadrupler is a new function
8 >>> print doubler(5)
9 10
10 >>> print tripler(5)
```

```
11 15
12 >>> print quadrupler(5)
13 20
```

函数 `f` 创建新函数，并且注意名称 `g` 的域完全在 `f` 内部，闭包是非常强大的。
函数参数可以有默认值，并且能够返回多个结果。

```
1 >>> def f(a, b=2):
2     return a + b, a - b
3 >>> x, y = f(5)
4 >>> print x
5 7
6 >>> print y
7 3
```

利用函数名，函数参数可以被显式传递，这意味着在函数调用中指定的参数顺序可以不同于函数定义时的参数顺序：

```
1 >>> def f(a, b=2):
2     return a + b, a - b
3 >>> x, y = f(b=5, a=2)
4 >>> print x
5 7
6 >>> print y
7 -3
```

函数也能取一个运行时可变数目的参数：

```
1 >>> def f(*a, **b):
2     return a, b
3 >>> x, y = f(3, 'hello', c=4, test='world')
4 >>> print x
5 (3, 'hello')
6 >>> print y
7 {'c':4, 'test':'world'}
```

这里在元组 `a` 中存储的参数没有被 `(3, 'hello')` 传递，而字典 `b` 中存储的参数被 `(c 和 test)` 传递。

在相反的情况下，将一个列表或元组传递给函数，它需要通过拆分得到单个位置参数。

```
1 >>> def f(a, b):
2     return a + b
3 >>> c = (1, 2)
4 >>> print f(*c)
5 3
```

并且可以将字典拆分传递关键词参数：

```
1 >>> def f(a, b):
2     return a + b
3 >>> c = {'a':1, 'b':2}
4 >>> print f(**c)
5 3
```

2.10.1 lambda 函数

`lambda` 提供了一种创建非常短的未命名函数的简易方法：

```
1 >>> a = lambda b: b + 2
2 >>> print a(3)
```


表达式“lambda [a]: [b]”直译为“一个带参数[a] 返回[b]的函数”，lambda 表达式本身是匿名的，但是该函数通过绑定标识符 a 获得了名称，def 的域规则同样适用于 lambda，实际上就 a 而言，上面的代码等同于使用 def 的函数声明：

```
1 >>> def a(b):
2     return b + 2
3 >>> print a(3)
4 5
```

lambda 唯一的优点是简洁；然而，在特定情况下简洁是十分方便的。考虑一个名为 map 的函数，它对列表中的所有项应用一个函数来创建一个新列表：

```
1 >>> a = [1, 7, 2, 5, 4, 8]
2 >>> map(lambda x: x + 2, a)
3 [3, 9, 4, 7, 6, 10]
```

在上面的代码中，如果使用 def 取代 lambda，代码将扩大一倍。lambda 的主要缺点是（在 Python 实现下）句法只允许一个单一表达式，然而对于更长的函数可以使用 def，因为提供函数名的额外成本随着函数长度增加而减少，同 def 一样，lambda 也可用来 curry 函数：可以通过封装已有函数创建新函数，这样新函数就会携带一组不同的参数。

```
1 >>> def f(a, b): return a + b
2 >>> g = lambda a: f(a, 3)
3 >>> g(2)
4 5
```

很多情况下 curry 功能是有用的，但其中一种在 web2py 中是直接有用的：缓存 (caching)。假设你有一个高负荷函数，检查参数是否为素数：

```
1 def isprime(number):
2     for p in range(2, number):
3         if (number % p) == 0:
4             return False
5 return True
```

显然，该函数是耗时的。

假设你有一个缓存函数 cache.ram，它带有 3 个参数：键、函数和秒数。

```
1 value = cache.ram('key', f, 60)
```

该函数初次被调用时，它将调用函数 f()，在内存中的字典里保存输出（比方说“d”），并且返回它，因此值 (value) 是：

```
1 value = d['key']=f()
```

该函数第二次被调用时，如果键在字典中，而且存在的时间不多于指定的秒数(60)，它将返回相应的值而不执行函数调用。

```
1 value = d['key']
```

对任意输入，如何缓存函数 isprime 的输出呢？这里提供了一种方法：

```
1 >>> number = 7
2 >>> seconds = 60
3 >>> print cache.ram(str(number), lambda: isprime(number), seconds)
4 True
5 >>> print cache.ram(str(number), lambda: isprime(number), seconds)
6 True
```

输出总是一样的，但是 cache.ram 第一次被调用时，isprime 会被调用；第二次时，就不调用了。

使用 def 或 lambda 创建的 Python 函数，允许根据不同参数组重构已有函数，cache.ram 和 cache.disk 是 web2py 缓存函数。

2.11 class (类)

因为Python是动态类型，Python类和对象似乎有点异样。事实上，当声明类时不需要定义成员变量（属性），而且同一个类的不同实例可以有不同属性。属性通常与实例相关，而不是类（除了被声明为类属性，这与C++/Java中的静态成员变量是一样的）。

下面是一个例子：

```
1 >>> class MyClass(object): pass
2 >>> myinstance = MyClass()
3 >>> myinstance.myvariable = 3
4 >>> print myinstance.myvariable
5 3
```

注意pass是一条不作任何操作的命令。在本例中它被用来定义MyClass()类，该类不包含任何内容，MyClass()调用类构造函数（在本例中是默认构造），并返回一个对象，即类的一个实例，在类定义中，(object)表明我们的类扩展内置的object类，这不是必需的，但却是好的做法。

下面是一个更复杂的类：

```
1 >>> class MyClass(object):
2 >>>     z = 2
3 >>>     def __init__(self, a, b):
4 >>>         self.x = a, self.y = b
5 >>>     def add(self):
6 >>>         return self.x + self.y + self.z
7 >>> myinstance = MyClass(3, 4)
8 >>> print myinstance.add()
9 9
```

在类中被声明的函数是方法。有些方法有特殊的保留名称，例如，__init__是构造，除了方法外声明的变量，其余所有变量都是方法的局部变量。例如，z是一个类变量，等同于一个C++中的静态成员变量，该变量对类的所有实例保持相同的值。

注意__init__带有3个参数，add带有1个参数，但我们分别用2个和0个参数调用它们。按照惯例，第一个参数表示局部名称，该名称被内部方法使用来指示当前对象。这里我们用self指示当前对象，但也可以使用其它名称，self起到相同的作用，正像*this在C++中或者this在Java中一样，但self并不是一个保留的关键词。

当声明嵌套类的时候，为了避免歧义，采取这种语法是有必要的，例如一个类在另一个类中是局部方法。

2.12 特殊属性、方法和运算符

以双下划线开头的类属性、方法和运算符通常被视作私有的（即在内部使用但不暴露在类的外部），虽然这是一种惯例，但解释器并不强制。

其中有些是保留关键词，并具有特殊的含义。

其中3个例子是：

- __len__
- __getitem__
- __setitem__

例如，可以用它们创建一个功能类似于列表的容器对象：

```

1 >>> class MyList(object):
2 >>> def __init__(self, *a): self.a = list(a)
3 >>> def __len__(self): return len(self.a)
4 >>> def __getitem__(self, i): return self.a[i]
5 >>> def __setitem__(self, i, j): self.a[i] = j
6 >>> b = MyList(3, 4, 5)
7 >>> print b[1]
8 4
9 >>> b.a[1] = 7
10 >>> print b.a
11 [3, 7, 5]

```

其它特殊运算符包括 `__getattr__` 和 `__setattr__`、`__sum__` 和 `__sub__`，前两个定义类的 `get` 和 `set` 属性，后两个重载了算数运算符。关于这些运算符的使用，我们向读者推荐这一主题的更深的书籍，前面我们已经提到了特殊运算符 `__str__` 和 `__repr__`。

2.13 文件输入/输出

在 Python 中，你可以用如下代码打开和写入文件：

```

1 >>> file = open('myfile.txt', 'w')
2 >>> file.write('hello world')
3 >>> file.close()

```

类似地，你可以用如下代码读文件：

```

1 >>> file = open('myfile.txt', 'r')
2 >>> print file.read()
3 hello world

```

另外，你可以使用标准 C 标记，在二进制模式下用 `"rb"` 读取，在二进制模式下用 `"wb"` 写入，用 `"a"` 在附加模式下打开文件。

`read` 命令有一个可选参数，它是字节数。你还可以在文件中用 `seek` 跳转到任意位置。

你可以用 `read` 读取文件，

```

1 >>> print file.seek(6)
2 >>> print file.read()
3 world

```

而且你可以用如下方法关闭文件：

```

1 >>> file.close()

```

Python 标准发行版被称为 CPython，变量是引用计数的，包括那些持有的文件句柄，所以当打开文件句柄的引用计数减少到 0 时，CPython 是知道的，这时就可以关闭文件和处理变量。然而，Python 的其它实现如 PyPy 中，使用垃圾回收取代引用计数，这意味着同一时间内可能积聚过多的打开文件句柄，在 gc 有机会关闭和处理它们之前引发错误。因此，当不再需要的时候，最好明确关闭文件句柄，web2py 在 `gluon.fileutils` 命名空间内提供两个帮助函数 `read_file()` 和 `write_file()`，它们封装文件访问，并确保正在实用的文件句柄被正确关闭。

在使用 web2py 时，你不能确定当前目录的位置，因为这依赖于 web2py 的配置，变量 `request.folder` 包含了当前应用的路径，使用 `os.path.join` 命令可以级联路径，将在后面讨论。

2.14 exec 和 eval 函数

与 Java 不同的是：Python 是真正的解释型语言，这意味着它能执行存储在字符串中的 Python 语句。例如：

```
1 >>> a = "print 'hello world'"
2 >>> exec(a)
3 'hello world'
```

发生了什么呢？exec 函数告诉解释器调用它自己，执行参数传递的字符串内容，它也可以执行词典中的符号定义的范围内一个字符串的内容。

```
1 >>> a = "print b"
2 >>> c = dict(b=3)
3 >>> exec(a, {}, c)
4 3
```

当执行字符串 a 时，这里解释器看到定义在 c 中的符号（本例中是 b），但不能看到 c 或 a 本身，这与受限环境不同，因为 exec 不限制内部代码可以做什么；它只是定义了一组代码可见的变量。

与此相关的函数是 eval，它的工作方式非常像 exec，除了它期待参数能计算得到一个值，并且它将返回这个值。

```
1 >>> a = "3*4"
2 >>> b = eval(a)
3 >>> print b
4 12
```

2.15 导入 (import)

Python 真正的强大之处是它的库模块，这些模块对许多系统库提供了一组大量的、一致性的应用编程接口 API（常常以一种独立于操作系统的方式）。

例如，如果你需要使用一个随机数生成器，可以采用如下方法：

```
1 >>> import random
2 >>> print random.randint(0, 9)
3 5
```

它打印了一个 0 到 9 之间（包括 9）的随机数，本例中是 5。randint 函数是定义在模块 random 中的，它也可以将来自模块的一个对象导入到当前命名空间：

```
1 >>> from random import randint
2 >>> print randint(0, 9)
```

或者将来自模块的全部对象导入到当前命名空间：

```
1 >>> from random import *
2 >>> print randint(0, 9)
```

或者将全部内容导入新定义的命名空间：

```
1 >>> import random as myrand
2 >>> print myrand.randint(0, 9)
```

在本书的其余部分，我们将主要使用定义在 os, sys, datetime 和 cPickle 模块中的对象。

通过胶子 (gluon) 模块可以访问全部 web2py 对象，这是后面章节中的主题。web2py 在内部使用许多 Python 模块（例如 thread），但很少时候需要直接访问它们。

在以下小节中，我们介绍了几个最实用的模块。

2.15.1 os 模块

该模块提供了一个操作系统 API 接口。例如：

```
1 >>> import os
2 >>> os.chdir('../')
3 >>> os.unlink('filename_to_be_deleted')
```

一些 os 函数，如 chdir，千万不能用在 web2py 中，因为它们不是线程安全的。

os.path.join 是非常实用的；它允许以独立于操作系统的方式级联路径：

```
1 >>> import os
2 >>> a = os.path.join('path', 'sub_path')
3 >>> print a
4 path/sub_path
```

通过如下方法，可以访问系统环境变量：

```
1 >>> print os.environ
```

这是一个只读字典。

2.15.2 sys 模块

sys 模块包含许多变量和函数，我们使用最多的一个是 sys.path，它包含了一个路径列表，Python 在那里搜索模块，当我们试图导入一个模块时，Python 会在 sys.path 列出的所有文件夹中查找。如果你在一些位置安装额外模块，并且希望 Python 能找到它们，你需要将到那个地址的路径添加到 sys.path 中。

```
1 >>> import sys
2 >>> sys.path.append('path/to/my/modules')
```

在运行 web2py 时，Python 常驻在内存中，并且只有一个 sys.path，然而有多个线程为 HTTP 请求服务。为了避免内存泄露，在添加路径前，最好先检查一下该路径是否已经存在。

```
1 >>> path = 'path/to/my/modules'
2 >>> if not path in sys.path:
3 sys.path.append(path)
```

2.15.3 datetime 模块

datetime 模块的使用最好通过一些例子来说明：

```
1 >>> import datetime
2 >>> print datetime.datetime.today()
3 2008-07-04 14:03:90
4 >>> print datetime.date.today()
5 2008-07-04
```

有时你需要的时间戳数据可能是基于 UTC 时间，而不是本地时间。在这种情况下，你可以使用如下函数：

```
1 >>> import datetime
2 >>> print datetime.datetime.utcnow()
3 2008-07-04 14:03:90
```

datetime 模块包含各种类：date，datetime，time 和 timedelta。两个 date 或两个 datetime 或两个 time 对象之间的区别就是 timedelta。

```
1 >>> a = datetime.datetime(2008, 1, 1, 20, 30)
2 >>> b = datetime.datetime(2008, 1, 2, 20, 30)
3 >>> c = b - a
```

```
4 >>> print c.days
5 1
```

在 web2py 中，当传递到数据库或从数据库中返回时，date 和 datetime 可用于存储相应的 SQL 类型。

2. 15. 4 time 模块

time 模块不同于 date 和 datetime，因为它表示的时间是从纪元（从 1970 年开始）开始的秒数。

```
1 >>> import time
2 >>> t = time.time()
3 1215138737.571
```

关于用秒表示的时间和用 datetime 表示的时间之间的转换函数，请参阅 Python 文档。

2. 15. 5 cPickle 模块

这是一个非常强大的模块，它提供的函数可以序列化几乎所有 Python 对象，包括自引对象。例如，让我们构建一个特殊的对象：

```
1 >>> class MyClass(object): pass
2 >>> myinstance = MyClass()
3 >>> myinstance.x = 'something'
4 >>> a = [1, 2, {'hello': 'world'}, [3, 4, [myinstance]]]
```

现在键入：

```
1 >>> import cPickle
2 >>> b = cPickle.dumps(a)
3 >>> c = cPickle.loads(b)
```

本例中，b 是 a 的一个字符串表示，c 是 a 通过反序列化 b 产生的一个副本。

cPickle 也可以序列化到文件和从文件反序列化；

```
1 >>> cPickle.dump(a, open('myfile.pickle', 'wb'))
2 >>> c = cPickle.load(open('myfile.pickle', 'rb'))
```

第 3 章 概述

3.1 启动

web2py 提供用于 Windows 和 Mac OS X 的二进制包。二进制版本包含了 Python 解释器，因此用户不需要预先安装，源代码版本能在 Windows、Mac、Linux 和其它 Unix 系统上运行。Windows 和 OS X 上的二进制版本包括必要的 Python 解释器，源代码包版本假定计算机安装了 Python，web2py 不需要安装。启动时，针对特定的操作系统，解压下载的压缩（zip）文件，并运行相应的 web2py 文件。

在 Windows 上，运行：

```
1 web2py.exe
```

在 OS X 上，运行：

```
1 open web2py.app
```

在 Unix 和 Linux 上，通过输入以下命令从源代码运行：

```
1 python2.5 web2py.py
```

为了在 Windows 上从源代码运行 web2py，需要先安装 Mark Hammond's "Python for Windows extensions，之后运行：

```
1 python2.5 web2py.py
```

web2py 程序接受各种命令行选项，这将在后面讨论。

默认情况下，在启动时，web2py 会显示一个启动窗口，然后显示一个 GUI 部件，要求您选择一个一次性管理员密码、网络接口的 IP 地址和端口号，其中 IP 地址用于 web 服务器，端口号用于服务请求。默认情况下，web2py 在 127.0.0.1:8000（本地主机 localhost 上的 8000 端口）上运行它的 Web 服务器，但它也能够任何可用的 IP 地址和端口上运行。你可以查询网络接口的 IP 地址，方法是在 Windows 下打开命令行、键入 ipconfig 或在 OS X 和 Linux 下键入 ifconfig。从现在开始，我们假定 web2py 在 localhost（127.0.0.1:8000）上运行，使用 0.0.0.0:80 可以在任何网络接口上运行公开 web2py。



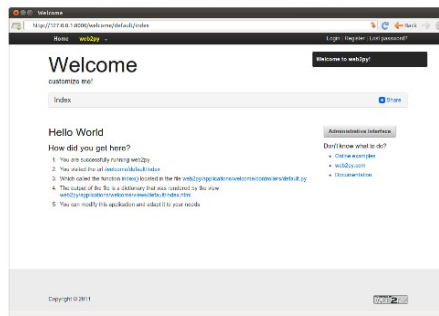
如果你不提供管理员密码，管理界面是不可用的。这是为了防止暴露管理界面而采取的安全措施。

管理界面 *admin* 仅能通过 localhost（本地主机）访问，除非你在后台使用带有 mod_proxy 的 Apache 服务器上运行 web2py。如果 admin 检测到代理，会话 cookie 被设为安全，admin 登录不起作用，除非客户端和代理之间采用 HTTPS 通信这一安全措施。客户端和 *admin* 之间的所有通信必须始终是本地的或加密的；否则，攻击者将能发动中间攻击或重放（replay）攻击，并在服务器端执行任意代码。

管理密码设置后，web2py 在如下页面启动 web 浏览器：

1 <http://127.0.0.1:8000/>

如果计算机没有默认的浏览器，打开一个 web 浏览器并输入 URL。

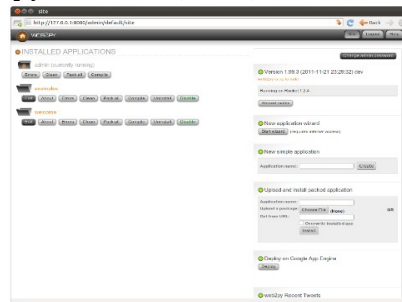


点击“administrative interface”，将带你到管理界面的登录页面。



管理员密码是您在启动时选择的密码。请注意，管理员只有一个，因此也只有一个管理员密码，出于安全原因，web2py 每次启动时，开发人员都要选择一个新密码，除非 <recycle> 选项被指定，这有别于 web2py 应用中的身份验证机制。

管理员登录 web2py 之后，浏览器被定向到“site”页面。



该页面列出了全部已安装的 web2py 应用，并允许管理员管理它们。web2py 带有 3 个应用：

- admin 应用，即你正在使用的应用。
- examples 应用，包含在线交互式文档和一个 web2py 官方网站的副本。
- welcome 应用，它是任何其它 web2py 应用的基本模板，也被称为基本构建应用（scaffolding application），也是当启动时欢迎用户的应用。

立即可用的 web2py 应用被称作 web2py appliance，你可以在 [appliances](#) 下载许多免费提供的应用范例，鼓励 web2py 用户提交新的应用范例，无论是以开源还是闭源（编译和包装）的形式。

从 admin 应用的 site 页面，你可以执行以下操作：

- 通过填写页面右下角的表单，安装应用；给应用命名，选择包含封装应用的文件夹或应用所处位置的 URL，并点击“submit”。
- 通过点击相应按钮，卸载应用，有一个确认页面。
- 通过选择一个名称，并点击“create”，即可创建一个新应用。
- 通过点击相应的按钮，打包应用发行。下载的应用是一个包括数据库的完整压缩（tar）文件，不用解压这个文件，使用 admin 安装后 web2py 会自动将它解压。
- 清理应用临时文件，例如会话、错误和缓存文件。

- 编辑应用。

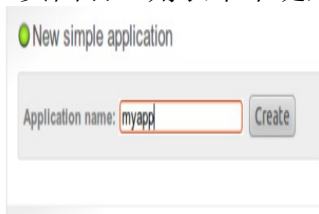
在你用 *admin* 创建新应用时，它开始是作为带有一个“models/db.py”的“welcome”基本构建应用的克隆，“models/db.py”创建一个 SQLite 数据库，并连接到该数据库，实例化 Auth、Crud 和 Service，并配置它们。它还提供了一个“controller/default.py”，为用户管理暴露了动作“index”、“download”和“user”，为服务暴露了“call”。下面，我们假设这些文件已被删除，我们将从头开始创建应用。

web2py 中配备的向导，在本章后面介绍，基于网络上提供的布局、插件和模型的高层次描述，向导能为用户编写替代框架代码。

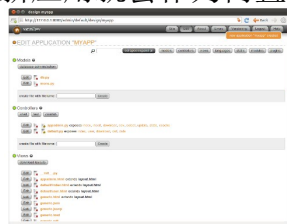
3.2 问好

一个例子，我们创建一个简单的 Web 应用，它向用户显示“Hello from MyApp”，我们把这个应用称为“myapp”，还将增加一个统计用户访问相同页面次数的计数器。

通过在 *admin* 内 *site* 页面右上角表单中键入应用的名称，你就可以创建一个新应用。



按下[create]之后，新应用就会作为内置 welcome 的副本被创建。



要运行新应用，请访问：

1 <http://127.0.0.1:8000/myapp>

现在，你有一个 welcome 应用的副本了。

为了编辑应用，可以点击新创建应用的 edit 按钮。

edit 页面告诉你应用内部有些什么，每个 web2py 应用包括一些特定文件，其中大部分可归纳为以下六类：

- *models*: 描述数据表示。
- *Controllers*: 描述应用逻辑和 workflow。
- *views*: 描述数据表达。
- *languages*: 描述如何将应用表达翻译成其它语言。
- *modules*: 属于应用的 Python 模块。
- *static files*: 静态图像，CSS `[css:w.css;o.css:school]` 文件，JavaScript `[js:w.js;b]` 文件等等。
- *plugins*: 用于设计协同工作的文件组。

遵循 MVC 设计模式，应用设计井然有序。*edit* 页面中的每个部分对应于应用文件夹的一个子文件夹。

注意，节标题将切换其内容，静态文件的文件夹名称也是可折叠的。

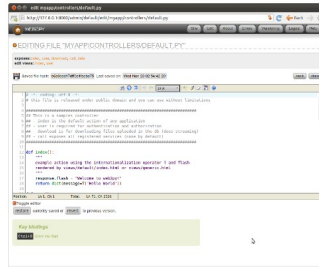
节中列出的每个文件对应于子文件夹中的实体文件。通过 *admin* 界面对文件的任何操作（创建、编辑、删除），都可以使用您偏爱的编辑器直接从 shell 执行。

该应用包含其它类型的文件（数据库、会话文件、错误文件等），但它们并没有在编辑页面中列出，因为它们不是由管理员创建或修改，而是由应用本身创建和修改。

控制器包含应用逻辑和 workflows，每个 URL 都被映射成控制器（actions）中某个函数的调用。有两个默认控制器：“appadmin.py”和“default.py”，appadmin 提供数据库管理界面，目前我们还不需要它，“default.py”是你需要编辑的控制器，当 URL 中没有指定控制器时，默认调用它。编辑“index”函数如下：

```
1 def index():
2     return "Hello from MyApp"
```

下面是在线编辑器的图示：

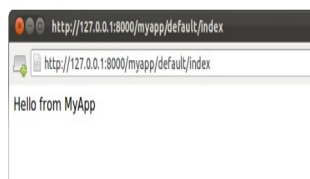


保存并返回到 *edit* 页面。点击 index 链接访问新创建的页面。

当你访问 URL 时

```
1 http://127.0.0.1:8000/myapp/default/index
```

myapp 应用中的默认控制器的 index 行为将被调用，它返回一个在浏览器中显示的字符串，如下图所示：



现在，编辑“index”函数如下：

```
1 def index():
2     return dict(message="Hello from MyApp")
```

从 *edit* 页面，编辑视图“default/index.html”（视图文件与 action 有关），并且用如下内容完全替换该文件的当前内容：

```
1 <html>
2     <head></head>
3     <body>
4         <h1>{{=message}}</h1>
5     </body>
6 </html>
```

现在 action 返回一个定义 message 的字典，当 action 返回一个字典时，web2py 会查找一个名称如下的视图：

```
1 [controller]/[function].[extension]
```

并执行它。这里 [extension] 是请求的扩展名，如果没有指定扩展名，默认为“html”，这里我们也将这样假设，在这个假设下，视图是一个使用特殊标签 {{ }} 并嵌入 Python 代码的 HTML 文件，特别是，在本例中 {{=message}} 指示 web2py 用 action 返回的 message 值取代标签代码。注意，这里 message 不是一个 web2py 关键字，但被定义在 action 中。到目前为止我们没有使用任何 web2py 关键字。

如果 web2py 没有找到请求的视图，它将使用通用视图“generic.html”。

如果指定“html”以外的扩展名（例如“json”），并且视图文件“[controller]/

[function].json” 未被找到，web2py 将查找“generic.json”视图。web2py 带有 generic.html、generic.json、generic.xml 和 generic.rss。应用可以单独修改这些通用视图，并且可以很容易的添加其它视图。

通用视图是一种开发工具。在生产中每个 action 都应有自己的视图，实际上，默认时通用视图只能从 localhost 启用。

你也可以使用 `response.view = 'default/something.html'` 指定一个视图。

在第 10 章中，你也可以看到有关这个主题的更多内容。

如果你回到“EDIT” 并点击 index，你将看到如下 HTML 页面：



为了调试目的，你可以随时添加如下代码

```
1  {{=response.toolbar()}}
```

到视图中，它会向你显示一些有用信息，包括请求、响应和会话对象，并列出所有数据库查询及其时间。

3.3 计数

我们现在添加一个计数器到页面中，该计数器将统计相同访问者访问页面的次数，web2py 将自动、透明的使用会话和 cookie 追踪访问者。对于新访问者，它创建一个会话并分配唯一的“session_id”；会话是一个存储服务器端变量的容器，唯一的 id 通过 cookie 发送到浏览器，当访问者请求同一应用中的另一页面时，浏览器发回 cookie，它被 web2py 回收并且相应的会话被恢复。

为了使用会话，修改默认控制器如下：

```
1  def index():
2      if not session.counter:
3          session.counter = 1
4      else:
5          session.counter += 1
6      return dict(message="Hello from MyApp", counter=session.counter)
```

注意 counter 不是 web2py 关键字但 session 是，我们要求 web2py 检查会话中是否有计数器变量，如果没有创建一个并将其设为 1，如果有计数器变量，我们要求 web2py 把计数器的值加 1。最后，我们把计数器的值传递给视图。

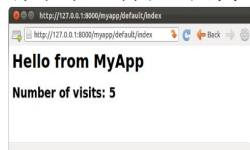
更紧凑的实现相同功能的代码如下：

```
1  def index():
2      session.counter = (session.counter or 0) + 1
3      return dict(message="Hello from MyApp", counter=session.counter)
```

现在修改视图，添加一行显示计数器值的代码：

```
1  <html>
2      <head></head>
3      <body>
4          <h1>{{=message}}</h1>
5          <h2>Number of visits: {{=counter}}</h2>
6      </body>
7  </html>
```

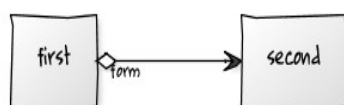
当你再次访问 index 页面时，你应该得到以下 HTML 页面：



计数器与每个访问者相关，访问者每次重载页面时计数器递增 1，不同访问者看到的计数器不同。

3.4 访问我的名字

现在创建两个页面(first 和 second)，其中 first 页面创建一个表单，询问访问者的名字，并重定向到 second 页面，second 页面通过名字问候访问者。



在默认控制器中写入相应 action(动作)：

```
1 def first():
2     return dict()
3
4 def second():
5     return dict()
```

然后为 first action 创建一个视图"default/first.html"，并输入：

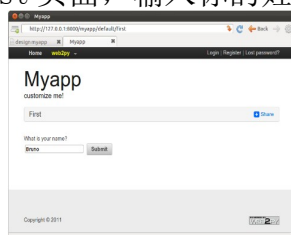
```
1 {{extend 'layout.html'}}
2 What is your name?
3 <form action="second">
4 <input name="visitor_name" />
5 <input type="submit" />
6 </form>
```

最后，为 second action 创建一个视图"default/second.html"：

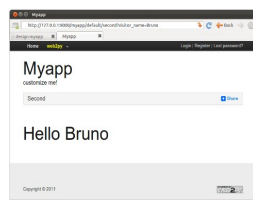
```
1 {{extend 'layout.html'}}
2 <h1>Hello {{=request.vars.visitor_name}}</h1>
```

在这两个视图中，我们都扩展了 web2py 自带的基本视图"layout.html"，layout 视图保持两页面的外观和感觉一致，因为 layout 文件主要包含 HTML 代码，因此可以轻松地编辑和替换。

如果你现在访问 first 页面，输入你的姓名：



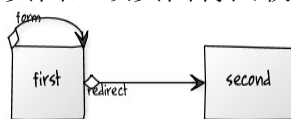
并提交表单，你将收到一个问候：



3.5 回传

我们前面使用的表单提交机制是很常见的，但这并不是好的编程习惯。在上面的例子中，应当验证所有的输入，验证的负担会落在 second action 上，执行验证的 action 与生成表单的 action 不同，这往往造成代码冗余。

一个更好的表单提交模式是将表单提交给生成它们的 action，在我们的例子中是“first”，“first” action 应该接受变量、处理变量和将变量存储在服务器端，并将访问者重新定向到“second”页面，该页面将回收变量，这种机制被称为回传（postback）。



修改默认控制器以实现自提交：

```
1 def first():
2     if request.vars.visitor_name:
3         session.visitor_name = request.vars.visitor_name
4         redirect(URL('second'))
5     return dict()
6
7 def second():
8     return dict()
```

之后，修改视图“default/first.html”：

```
1 {{extend 'layout.html'}}
2 What is your name?
3 <form>
4 <input name="visitor_name" />
5 <input type="submit" />
6 </form>
```

并且“default/second.html”视图需要从 session 而非 request.vars 恢复数据：

```
1 {{extend 'layout.html'}}
2 <h1>Hello {{=session.visitor_name or "anonymous"}}</h1>
```

从访问者的角度，自提交与先前的实现方法表现完全相同。我们还没有加入验证，但现在很清楚验证应该由 first action 执行。

这种方法比较好，因为访问者的名字留在 session 中，应用中所有 action 和 view 都可以用无显式传递方式访问它。

注意：在访问者名被设定之前，如果“second” action 被调用，它将显示“Hello anonymous”，这是因为 session.visitor_name 返回为 None。另外，我们可以在控制器中添加下面的代码（在 second 函数内）：

```
1 if not request.function=='first' and not session.visitor_name:
2     redirect(URL('first'))
```

这是在控制器上强制授权的一种普遍机制，更有效的方法参阅第 9 章。

使用 web2py，我们可以更进一步请求它为我们生成包括验证在内的表单，web2py 提供帮助对象 (FORM、INPUT、TEXTAREA 和 SELECT/OPTION)，它们与等效的 HTML 标签同名，且它们可以用于在控制器或视图中构建表单。

例如，这里是改写 first action 的一种可能方式：

```
1 def first():
2 form = FORM(INPUT(_name='visitor_name', requires=IS_NOT_EMPTY()),
3 INPUT(_type='submit'))
4 if form.process().accepted:
5 session.visitor_name = form.vars.visitor_name
6 redirect(URL('second'))
7 return dict(form=form)
```

这里我们说的 FORM 标签包含两个 INPUT 标签，input 标签的属性由以下划线开头的命名参数指定。requires 参数不是一个标签属性（因为它不以下划线开头），但为 visitor_name 的值设置了一个验证。

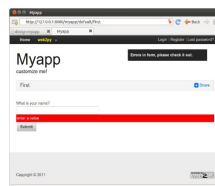
这里是创建相同表单的另一种更好的方式：

```
1 def first():
2 form = SQLFORM.factory(Field('visitor_name', requires=IS_NOT_EMPTY()))
3 if form.process():
4 session.visitor_name = form.vars.visitor_name
5 redirect(URL('second'))
6 return dict(form=form)
```

通过在视图“default/first.html”中嵌入 form 对象，可以很容易的在 HTML 中将它序列化。

```
1 {{extend 'layout.html'}}
2 What is your name?
3 {{=form}}
```

form.process() 方法使用了验证器并返回表单本身，如果表单已经被处理并通过验证，form.accepted 变量将被设为 True；如果自提交表单通过验证，它将在会话中存储变量并像以前一样重定向；如果表单没有通过验证，错误消息会被插入到表单中，并以如下方式显示给用户：

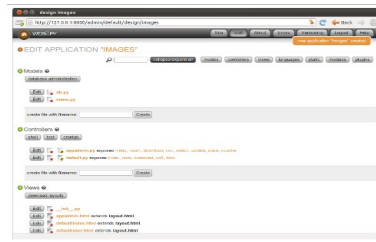


下一节，我们将展示如何从模型自动生成表单。

3.6 图像博客

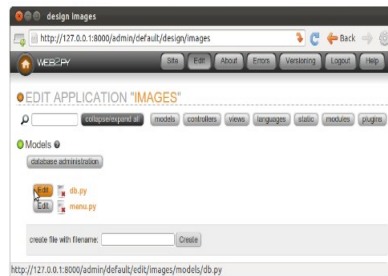
作为另一个例子，在这里我们希望创建一个 Web 应用，它允许管理员发布图片并给它们命名，同时允许网站访问者查看命名的图片并提交评论。

像以前一样，从 admin 中的 site 页面创建一个命名为 images 的新应用，并导航到 edit 页面：



我们开始创建一个模型，即应用中持久数据的表示(上传的图像、它们的名字和评论)。首先，你需要创建/编辑模型文件，由于缺乏想象力，我们称之为“db.py”，我们假设下面的代码将取代“db.py”中任何现有的代码，模型和控制器必须具有.py扩展名，因为它们都是Python代码。如果没有提供扩展名，web2py会自动添加，视图具有.html扩展名，因为它们主要包含HTML代码。

通过点击相应的“edit”按钮，可以编辑“db.py”文件：



并输入以下代码：

```
1 db = DAL("sqlite://storage.sqlite")
2
3 db.define_table('image',
4 Field('title', unique=True),
5 Field('file', 'upload'),
6 format = '%(title)s')
7
8 db.define_table('comment',
9 Field('image_id', db.image),
10 Field('author'),
11 Field('email'),
12 Field('body', 'text'))
13
14 db.image.title.requires = IS_NOT_IN_DB(db, db.image.title)
15 db.comment.image_id.requires = IS_IN_DB(db, db.image.id, '%(title)s')
16 db.comment.author.requires = IS_NOT_EMPTY()
17 db.comment.email.requires = IS_EMAIL()
18 db.comment.body.requires = IS_NOT_EMPTY()
19
20 db.comment.image_id.writable = db.comment.image_id.readable = False
```

让我们来逐行分析。

第1行定义一个表示数据库连接的称为db的全局变量。在本例中，它是一个到SQLite数据库的连接，该数据库存储在“applications/images/databases/storage.sqlite”文件中，在SQLite情况下，如果数据库不存在则创建它，可以更改文件和全局变量db的名称，但给予它们相同的名称更便于记忆。

第3-5行定义一个“image”表。define_table是db对象的一个方法，第一个参数“image”是我们定义的表的名称，其它参数都是属于该表的字段，该表有一个“title”字段，一个“file”字段和一个“id”字段，并且“id”字段作为表的主键（“id”没有显式声明，因为默认时所有表都有一个id字段），“title”字段是一个字符串，字段“file”是“upload”类型，“upload”是一个特殊的字段类型，web2py数据抽象层（DAL）用它来存储上传文件的名称。

称。web2py 知道如何上传文件（如果文件大则用流媒体）、安全的重命名文件和存储文件。

当定义表时，web2py 可能采取如下动作之一：

- 如果表不存在则创建它；
- 如果表存在但不符合定义，表将被相应的修改，如果字段有不同的类型，web2py 会设法转换其内容；
- 如果表存在并且符合定义，web2py 不执行任何操作。

这种行为被称为“migration”。在 web2py 中迁移（migration）是自动的，但通过传递 `migrate=False` 作为 `define_table` 的最后一个参数，可以禁用每个表的迁移。

第 6 行为表定义了一个格式字符串，它决定了如何将记录表示为一个字符串。注意，`format` 参数也可以是一个函数，该函数需要一个记录并返回一个字符串。例如：

```
1 format=lambda row: row.title
```

8-12 行定义另一个名为“comment”的表。一条评论包含“author”、“email”（我们打算存储评论者的电子邮件地址）、“text”类型的“body”（我们打算用它来存储作者发布的评论）和引用类型的通过“id”字段指向 `db.image` 的“image_id”字段。

在第 14 行，`db.image.title` 表示“image”表的“title”字段。属性需要允许您设置规定/限制，该规定/限制将被 web2py 表单强制设置，这里我们要求“title”是唯一的：

```
IS_NOT_IN_DB(db, db.image.title)
```

注意这是可选的，鉴于 `Field('title', unique=True)` 它将被自动设置。

表示这些限制的对象被称为验证，多个验证可以组合在一个列表中，并且根据出现的顺序执行验证。`IS_NOT_IN_DB(a, b)` 是一个特殊验证，它检查新纪录字段 `b` 的值是否已经在 `a` 中。

第 15 行要求表“comment”的“image_id”字段在 `db.image.id` 中。就涉及的数据库而言，当定义表“comment”时我们已经声明了这一点，现在我们明确地告诉模型，在表单处理层当有新评论发布时，该条件应当由 web2py 强制执行，这样无效值就不会从输入表单传递到数据库。

我们还要求用相应记录的“title”，即 `'%(title)s'` 来代表“image_id”。

第 20 行表明“comment”表的字段“image_id”不应显示在表单中，`writable=False`，甚至只读形式也不行，`readable=False`。

15-17 行中验证的含义应该是显而易见的。

注意如下验证：

```
1 db.comment.image_id.requires = IS_IN_DB(db, db.image.id, '%(title)s')
```

如果我们为引用的表指定格式，可以省略（并且是自动的）它：

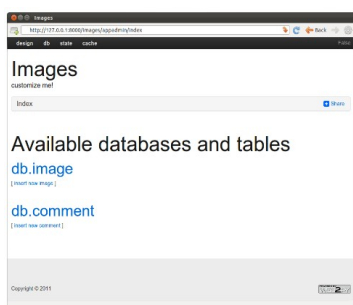
```
1 db.define_table('image', ..., format='%(title)s')
```

这里该格式可以是一个字符串或一个函数，它接收一个记录并返回一个字符串。

一旦模型被定义，如果没有错误，web2py 创建一个应用管理界面来管理数据库，可以通过 `edit` 页面中的“database administration”链接来访问它，或者直接访问：

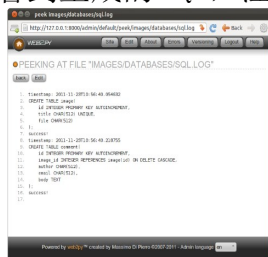
```
1 http://127.0.0.1:8000/images/appadmin
```

下面是 appadmin 界面的截图：



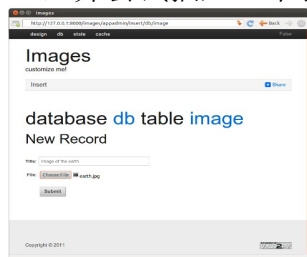
此界面在控制器“appadmin.py”和相应的视图“appadmin.html”中编码。从现在起，我们将把这一界面简单的称作 *appadmin*，它允许管理员插入新的数据库记录、编辑和删除现有记录、浏览表和进行数据库连接。

appadmin 第一次被访问时，执行模型并创建表，web2py DAL 将 Python 代码翻译成针对选定数据库后台（本例中是 SQLite）的 SQL 语句，通过点击“models”下的“sql.log”链接，你可以从 *edit* 页面看到生成的 SQL，注意在表被创建之前链接是不存在的。



如果你要编辑模型，并再次访问 appadmin，web2py 会生成 SQL 来修改现有的表，生成的 SQL 被记录在“sql.log”中。

现在回到 appadmin，并尝试插入一个新的图片记录：



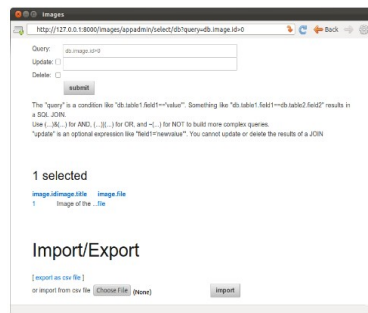
web2py 翻译 db.image.file 的“upload”字段为文件的上传表单，当表单被提交并且图像文件被上传时，将以安全的保留扩展名的方式重命名文件，用新文件名存储该文件到应用“uploads”文件夹中，新文件名将被保存在 db.image.file 字段中，该过程被设计来防止目录遍历攻击。

注意，每个字段类型由一个部件 (*widget*) 呈现，默认部件可能被重写。

当你在 *appadmin* 中点击表名时，web2py 在当前表中执行选择所有记录，并由 DAL 查询确定：

```
1 db.image.id > 0
```

且呈现结果如下。



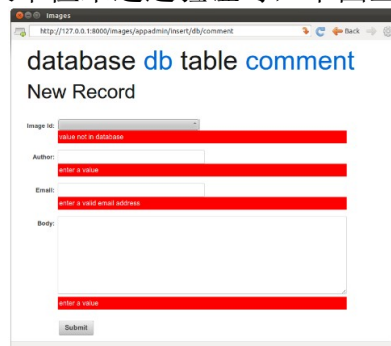
通过编辑 SQL 查询并按 [Submit]，你可以选择不同的记录集。

若要编辑或删除单个记录，可以点击记录的 id 号。

由于 IS_IN_DB 验证，引用字段“image_id”将被下拉菜单呈现，下拉条目将被存储为键(db.image.id)，但由 db.image.title 表示，正如验证指定的一样。

验证器是功能强大的对象，它知道如何表示字段、过滤字段值、生成错误和格式化从字段中提取的值。

当你提交一个表单但未通过验证时，下图显示了发生的情况：



由 appadmin 自动生成的相同表单，也可以通过 SQLFORM 帮助对象编程产生并嵌入用户应用中，这些表单都是 CSS 友好的且可被定制。

每个应用都有自己的 appadmin；因而，appadmin 本身可以在不影响其它应用下被修改。

到目前为止，应用知道如何存储数据，并且我们已经看到如何通过 appadmin 访问数据库。仅限管理员访问 appadmin，它的目的不是作为一个应用的生产网络接口；故下一部分逐步解说。具体来说，我们要创建：

- 一个“index”页面，按页面的标题排序，列出所有可用的图像以及图像详细信息页面的链接。
- 一个“show/[id]”页面，显示访问者请求的图像并允许访问者查看和发表评论。
- 一个“download/[name]”动作，下载上传的图片。

用示意图表示如下：



返回到编辑页面并编辑“default.py”控制器，取代其内容如下：

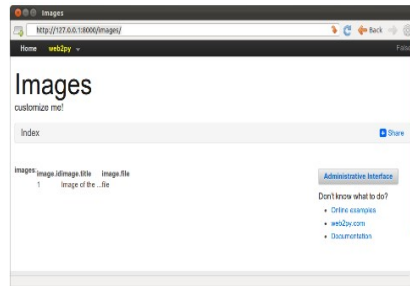
```

1 def index():
2     images = db().select(db.image.ALL, orderby=db.image.title)
3     return dict(images=images)
  
```

此动作返回一个字典。字典中的条目的键被解释为变量，传递给与动作相关的视图，开发时，如果没有视图，动作被“generic.html”视图呈现，每个 web2py 应用都提供该视图。

index 动作执行对表 image 中所有字段(db.image.ALL)的选择，按照 db.image.title 排序，选择的结果是一个 Rows 对象，其中包含选中的记录，将它赋值给一个局部变量，该局部变量叫做 images，由动作返回给视图。images 是迭代的，它的元素是选定的行，每行的列可以作为字典访问：images[0]['title'] 或等价于 images[0].title。

如果你没有写视图，字典将被“views/generic.html”呈现并对 index 动作的调用看起来将是这样的：



还没有为该动作创建视图，因此 web2py 用普通表格形式呈现记录组。

着手为 index 动作建立一个视图。返回 admin，编辑“default/index.html”并将其内容替换如下：

```
1 {{extend 'layout.html'}}
2 <h1>Current Images</h1>
3 <ul>
4 {{for image in images:}}
5 {{=LI(A(image.title, _href=URL("show", args=image.id))}}
6 {{pass}}
7 </ul>
```

首先要注意的是，视图是有特殊标签 {{...}} 的纯 HTML，嵌入 {{...}} 中的代码是纯 Python 代码，需要注意一点：缩进是无关紧要的。代码块开始于以冒号(:) 结束的行，并终止于以关键词 pass 结尾的行。在某些情况下，块的结尾从上下文看是很明显的，不需要使用 pass。

5-7 行循环 image 的行，并为每行显示：

```
1 LI(A(image.title, _href=URL('show', args=image.id))
```

这是一个...标签，其中包含...标签，该标签中包含 image.title，超文本引用(href 属性)的值是：

```
1 URL('show', args=image.id)
```

即同一应用中的 URL 和控制器，该控制器作为当前请求调用函数“show”，传递一个参数给函数，args=image.id。LI、A 等是 web2py 帮助对象，它们映射到相应的 HTML 标签，它们的未命名参数被解释为需要序列化的对象并被插入标签内的 HTML，以下划线开始的命名参数被解释为不带下划线的标记属性。例如，_href 是 href 属性，_class 是 class 属性等。

作为一个例子，下面的语句：

```
1 {{LI(A('something', _href=URL('show', args=123)))}}
```

被呈现为:

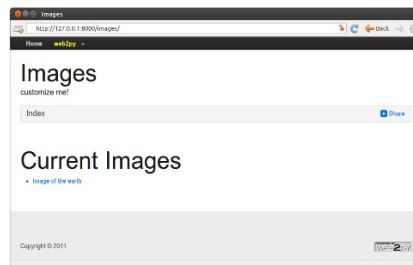
```
1 <li><a href="/images/default/show/123">something</a></li>
```

少数帮助对象 (INPUT, TEXTAREA, OPTION 和 SELECT) 也支持一些特殊的不以下划线开始的命名属性 (value 和 requires), 对于构建自定义表单它们是重要的并将在稍后讨论。

返回 *edit* 页面, 它现在表明 “default.py exposes (暴露) index”, 通过点击 “index”, 你可以访问新建的页面:

```
1 http://127.0.0.1:8000/images/default/index
```

它看起来像:



如果你点击图片名链接, 你将被定向到:

```
1 http://127.0.0.1:8000/images/default/show/1
```

这将导致错误, 因为你还没有在控制器 “default.py” 中创建名为 “show” 的动作。

让我们编辑 “default.py” 控制器, 并将其内容替换如下:

```
1 def index():
2     images = db().select(db.image.ALL, orderby=db.image.title)
3     return dict(images=images)
4
5 def show():
6     image = db(db.image.id==request.args(0)).select().first()
7     db.comment.image_id.default = image.id
8     form = SQLFORM(db.comment)
9     if form.process().accepted:
10         response.flash = 'your comment is posted'
11         comments = db(db.comment.image_id==image.id).select()
12         return dict(image=image, comments=comments, form=form)
13
14 def download():
15     return response.download(request, db)
```

控制器包含两个动作: “show” 和 “download”, “show” 动作从请求 args 中解析得到的 id, 选择图片和与其相关的所有评论, 之后 “show” 将一切传递给视图 “default/show.html”。

引用图片 id:

```
1 URL('show', args=image.id)
```

在 “default/index.html” 中, 可以作为: “show” 动作中的 request.args(0) 被访问。

“download”动作期待 request.args(0) 中的文件名，建立一个路径到文件应该存在的位置，并把它发送回客户端，如果文件过大，它流式传输该文件，而不产生任何内存开销。

注意以下语句：

- 第 7 行仅使用指定字段，为表 db.comment 创建一个插入表单 SQLFORM。
- 第 8 行设定引用字段的值，它不是输入表单的一部分，因为它不在上面指定的字段列表中。
- 第 9 行在当前会话（会话用来防止重复提交，强制执行导航）中处理提交的表单（提交的表单变量在 request.vars 中）。如果提交的表单变量经过验证，在表 db.comment 中插入新评论；否则，表单被修改为包括错误消息（例如，如果作者的电子邮件地址无效），这一切都是在第 9 行中完成的！
- 表单被接受到第 10 行才会被执行，在记录被插入数据库表之后，response.flash 是一个 web2py 变量，它被显示在视图中并被用来通知访问者发生了一些事情。
- 第 11 行选择所有引用当前图片的评论。

在基本构建应用的“default.py”控制器中，已经定义了“download”动作。

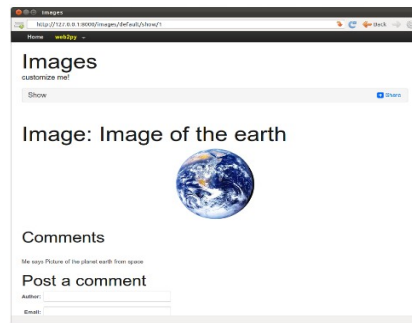
“download”动作不返回字典，因此它不需要视图。不过，“show”动作应该有视图，因此返回 *admin* 并创建一个新的视图叫做“default/show.html”。

编辑这个新文件，并替换为以下内容：

```
1 {{extend 'layout.html'}}
2 <h1>Image: {{=image.title}}</h1>
3 <center>
4 
6 </center>
7 {{if len(comments):}}
8   <h2>Comments</h2><br /><p>
9   {{for comment in comments:}}
10    <p>{{=comment.author}} says <i>{{=comment.body}}</i></p>
11   {{pass}}</p>
12 {{else:}}
13   <h2>No comments posted yet</h2>
14 {{pass}}
15 <h2>Post a comment</h2>
16 {{=form}}
```

该视图通过调用标签内的“download”动作显示 image.file。如果有评论，对它们循环，并显示每一个。

下面是显示给访问者的效果：



当访问者通过此页提交评论，评论将被存储在数据库中并附加在页面底部。

3.7 添加 CRUD

web2py 还提供 CRUD (Create/Read/Update/Delete) API，这样更加简化了表单，使用 CRUD，有必要在某处定义它，例如在文件“db.py”中：

```
1 from gluon.tools import Crud
2 crud = Crud(db)
```

crud 对象提供高层次的方法，例如：

```
1 form = crud.create(table)
```

可以用来取代编程模式：

```
1 form = SQLFORM(table)
2 if form.process().accepted:
3 session.flash = '...'
4 redirect('...')
```

这里，我们使用 crud 对象改写以前的“show”动作，并做更多的改进：

```
1 def show():
2 image = db.image(request.args(0)) or redirect(URL('index'))
3 db.comment.image_id.default = image.id
4 form = crud.create(db.comment,
5 message='your comment is posted',
6 next=URL(args=image.id))
7 comments = db(db.comment.image_id==image.id).select()
8 return dict(image=image, comments=comments, form=form)
```

首先，注意我们使用如下语法

```
1 db.image(request.args(0)) or redirect(...)
```

来获取所需的记录。因为如果记录未找到，table(id) 返回 None，这种情况下我们可以在一行代码中使用 or redirect(...)

crud.create 的 next（下一个）参数是表单被接受之后重定向的 URL，message 参数在验收后显示，你可以阅读第 7 章中有关 CRUD 的内容。

3.8 添加认证

web2py 基于角色的访问控制 API 是相当复杂的，但是现在我们会限制自己，限制通过身份验证的用户访问 show 动作，在第 9 章将进行更详细的讨论。

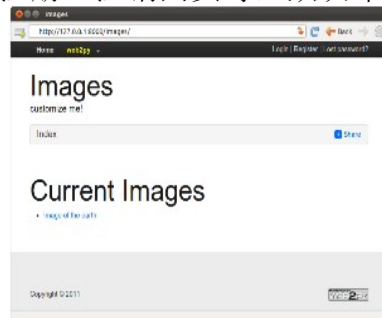
为了限制通过身份验证的用户访问，我们需要完成三个步骤。在一个模型中，例如“db.py”，我们需要添加：

```
1 from gluon.tools import Auth
2 auth = Auth(db)
3 auth.define_tables()
```

在我们的控制器中，我们需要添加一个动作：

```
1 def user():
2 return dict(form=auth())
```

这足以启用登录、注册、注销网页等，默认布局也将在右上角显示相应页面的选项。



现在我们可以修饰要限制的函数，例如：

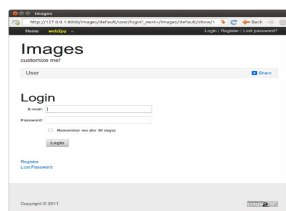
```
1 @auth.requires_login()
2 def show():
3 image = db.image(request.args(0)) or redirect(URL('index'))
4 db.comment.image_id.default = image.id
5 form = crud.create(db.comment, next=URL(args=image.id),
6 message='your comment is posted')
7 comments = db(db.comment.image_id==image.id).select()
8 return dict(image=image, comments=comments, form=form)
```

任何企图访问

```
1 http://127.0.0.1:8000/images/default/show/[image_id]
```

都需要登录。如果用户没有登录，用户将被重定向到

```
1 http://127.0.0.1:8000/images/default/user/login
```



user 函数暴露其他的一些，其中包括下列动作：

```
1 http://127.0.0.1:8000/images/default/user/logout
2 http://127.0.0.1:8000/images/default/user/register
3 http://127.0.0.1:8000/images/default/user/profile
4 http://127.0.0.1:8000/images/default/user/change_password
5 http://127.0.0.1:8000/images/default/user/request_reset_password
6 http://127.0.0.1:8000/images/default/user/retrieve_username
7 http://127.0.0.1:8000/images/default/user/retrieve_password
8 http://127.0.0.1:8000/images/default/user/verify_email
9 http://127.0.0.1:8000/images/default/user/impersonate
10 http://127.0.0.1:8000/images/default/user/not_authorized
```

现在，为了能够登录并阅读或发表评论，初次使用者需要注册。

auth 对象和 user 函数都已经定义在基本构建应用中，auth 对象高度可定制并且可以处理电子邮件验证，注册审批，CAPTCHA 和通过插件替代登录方法。

3.8.1 添加网格

使用 SQLFORM.grid 和 SQLFORM.smartgrid 小工具我们可以进一步改善，为我们的应用创建一个管理界面：

```
1 @auth.requires_membership('manager')
2 def manage():
3     grid = SQLFORM.smartgrid(db.image)
4     return dict(grid=grid)
```

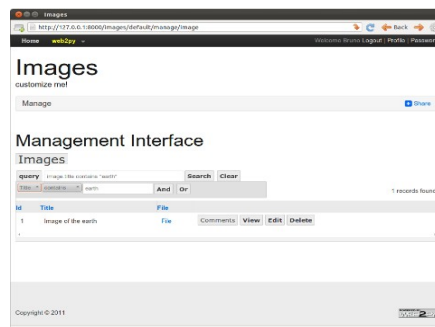
带有相关的“views/default/manage.html”

```
1 {{extend 'layout.html'}}
2 <h2>Management Interface</h2>
3 {{=grid}}
```

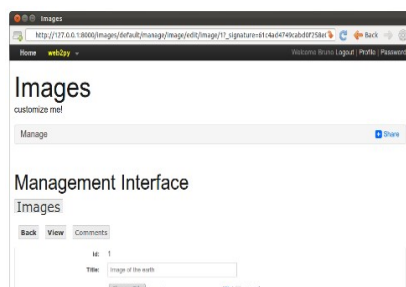
使用 appadmin 创建组“manager”并使一些用户成为该组的成员，他们将不能访问

```
1 http://127.0.0.1:8000/images/default/manage
```

和浏览、搜索：



创建、更新和删除图像以及它们的评论：



3.9 配置布局

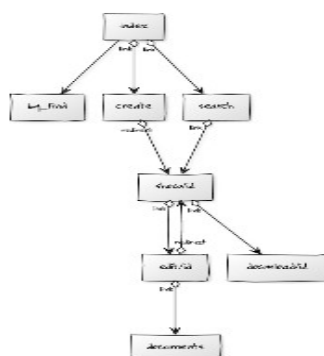
可以通过编辑“views/layout.html”来配置默认布局，而且也可以配置它而无需编辑 HTML。实际上，“static/base.css”样式表在第 5 章中有详细描述。可以改变颜色、列、大小、边界和背景而无需编辑 HTML，如果要编辑菜单、标题或副标题，可以在任何模型文件中操作，基本构建应用能在文件“models/menu.py”中设置这些参数的默认值：

```
1 response.title = request.application
2 response.subtitle = T('customize me!')
3 response.meta.author = 'you'
4 response.meta.description = 'describe your app'
5 response.meta.keywords = 'bla bla bla'
6 response.menu = [ [ 'Index', False, URL('index') ] ]
```

3.10 wiki 维基

在本节中，我们建立了一个 wiki，从无到有，并且不使用第 12 章中介绍的 plugin_wiki 提供的扩展功能。访问者能够创建页面，搜索（按照标题）并编辑它们，还能发表评论（与之前的应用完全相似），上传文件（作为网页的附件）并从网页链接。作为惯例，我们的 wiki 语法采用 Markmin 语法，还将使用 Ajax 实现一个搜索页面，该页面的 RSS 订阅，以及一个通过 XML-RPC **xmlrpc** 搜索该页面的句柄。

以下图表列出我们需要实现的动作以及我们打算建立的它们之间的链接。



开始创建一个新的基本构建应用，命名为“mywiki”。

该模型必须包含三个表：page（页面），comment（评论）和 document（文档），评论和文档都引用页面，是因为它们属于页面，文档包含 upload 类型的字段 file，正如前面的

images 应用一样。

下面是完整的模型：

```
1 db = DAL('sqlite://storage.sqlite')
2
3 from gluon.tools import *
4 auth = Auth(db)
5 auth.define_tables()
6 crud = Crud(db)
7
8 db.define_table('page',
9     Field('title'),
10    Field('body', 'text'),
11    Field('created_on', 'datetime', default=request.now),
12    Field('created_by', db.auth_user, default=auth.user_id),
13    format='%(title)s')
14
15 db.define_table('comment',
16    Field('page_id', db.page),
17    Field('body', 'text'),
18    Field('created_on', 'datetime', default=request.now),
19    Field('created_by', db.auth_user, default=auth.user_id))
20
21 db.define_table('document',
22    Field('page_id', db.page),
23    Field('name'),
24    Field('file', 'upload'),
25    Field('created_on', 'datetime', default=request.now),
26    Field('created_by', db.auth_user, default=auth.user_id),
27    format='%(name)s')
28
29 db.page.title.requires = IS_NOT_IN_DB(db, 'page.title')
30 db.page.body.requires = IS_NOT_EMPTY()
31 db.page.created_by.readable = db.page.created_by.writable = False
32 db.page.created_on.readable = db.page.created_on.writable = False
33
34 db.comment.body.requires = IS_NOT_EMPTY()
35 db.comment.page_id.readable = db.comment.page_id.writable = False
36 db.comment.created_by.readable = db.comment.created_by.writable = False
37 db.comment.created_on.readable = db.comment.created_on.writable = False
38
39 db.document.name.requires = IS_NOT_IN_DB(db, 'document.name')
40 db.document.page_id.readable = db.document.page_id.writable = False
41 db.document.created_by.readable = db.document.created_by.writable = False
42 db.document.created_on.readable = db.document.created_on.writable = False
```

编辑控制器“default.py” 并创建以下动作：

- index：列出所有的wiki 页面
- create：发布其它wiki 页面
- show：显示一个wiki 页面以及它的评论，并追加评论
- edit：编辑已有页面
- documents：管理附加到一个页面的文件
- download：下载文件（如在 images 例子中）

- search: 显示一个搜索框并且通过 Ajax 回调，在访问者输入的同时返回所有匹配的标题
 - callback: Ajax 回调函数，它返回在访问者输入时被嵌入搜索页面的 HTML。
- 这里是“default.py”控制器:

```
1 def index():
2     """ this controller returns a dictionary rendered by the view
3     it lists all wiki pages
4     >>> index().has_key('pages')
5     True
6     """
7     pages = db().select(db.page.id, db.page.title, orderby=db.page.title)
8     return dict(pages=pages)
9
10 @auth.requires_login()
11 def create():
12     """creates a new empty wiki page"""
13     form = crud.create(db.page, next=URL('index'))
14     return dict(form=form)
15
16 def show():
17     """shows a wiki page"""
18     this_page = db.page(request.args(0)) or redirect(URL('index'))
19     db.comment.page_id.default = this_page.id
20     form = crud.create(db.comment) if auth.user else None
21     pagecomments = db(db.comment.page_id==this_page.id).select()
22     return dict(page=this_page, comments=pagecomments, form=form)
23
24 @auth.requires_login()
25 def edit():
26     """edit an existing wiki page"""
27     this_page = db.page(request.args(0)) or redirect(URL('index'))
28     form = crud.update(db.page, this_page,
29 next=URL('show', args=request.args))
30     return dict(form=form)
31
32 @auth.requires_login()
33 def documents():
34     """browser, edit all documents attached to a certain page"""
35     page = db.page(request.args(0)) or redirect(URL('index'))
36     db.document.page_id.default = page.id
37     db.document.page_id.writable = False
38     grid = SQLFORM.grid(db.document.page_id==page.id, args=[page.id])
39     return dict(page=page, grid=grid)
40
41 def user():
42     return dict(form=auth())
43
44 def download():
45     """allows downloading of documents"""
46     return response.download(request, db)
47
48 def search():
49     """an ajax wiki search page"""
50     return dict(form=FORM(INPUT(_id='keyword', _name='keyword',
51 _onkeyup="ajax('callback', ['keyword'], 'target');")),
52 target_div=DIV(_id='target'))
```

```

53
54 def callback():
55     "an ajax callback that returns a <ul> of links to wiki pages"
56     query = db.page.title.contains(request.vars.keyword)
57     pages = db(query).select(orderby=db.page.title)
58     links = [A(p.title, _href=URL('show',args=p.id)) for p in pages]
59     return UL(*links)

```

2-6 行为 index 动作提供了一个评论，评论中的 4-5 行由 python 解释成测试代码 (doctest)，通过管理界面可以运行测试，在本例中，测试验证 index 动作运行没有错误。

第 18、27 和 35 行尝试用 request.args(0) 中的 id 获取一个 page 记录。

第 13、20 行为新页面和新评论定义和处理创建表单。

第 28 行定义和处理一个 wiki 页面的更新表单。

第 38 行创建一个 grid 对象来浏览、添加和更新链接到页面的评论。

一些奇妙的事情发生在第 51 行，INPUT 标签“keyword”的 onkeyup 属性被设定，每次访问者放开按键，onkeyup 属性中的 JavaScript 代码在客户端被执行。下面是 JavaScript 代码：

```

1 ajax('callback', ['keyword'], 'target');

```

ajax 是一个在文件“web2py.js”中定义的 JavaScript 函数，默认的“layout.html”包含该文件，它有三个参数：动作的 URL，执行同步回调；列表中变量的 ID，被发送到回调(["keyword"]); 和响应要插入的 ID("target")。

只要你在搜索框中键入内容，并放开按键，客户端调用服务器并发送'keyword' 字段的内容，而且当服务器响应时，响应被作为'target' 标签的内部 HTML 嵌入到页面中。

'target' 标签是一个 DIV 被定义在第 52 行中，也可以在视图中定义它。

这里是视图“default/create.html”的代码：

```

1 {{extend 'layout.html'}}
2 <h1>Create new wiki page</h1>
3 {{=form}}

```

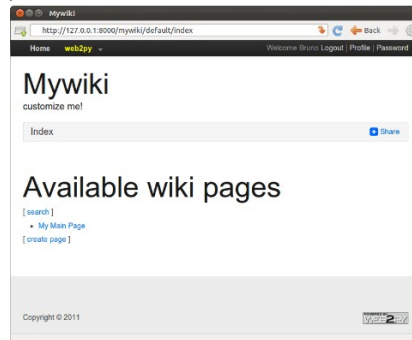
如果你访问 create 页面，你将看到以下内容：



这里是视图“default/index.html”的代码：

```
1 {{extend 'layout.html'}}
2 <h1>Available wiki pages</h1>
3 [ {{=A('search', _href=URL('search'))}} ]<br />
4 <ul>{{for page in pages:}}
5 {{=LI(A(page.title, _href=URL('show', args=page.id))}}
6 {{pass}}</ul>
7 [ {{=A('create page', _href=URL('create'))}} ]
```

它将生成以下页面：



这里是视图“default/show.html”的代码：

```
1 {{extend 'layout.html'}}
2 <h1>{{=page.title}}</h1>
3 [ {{=A('edit', _href=URL('edit', args=request.args))}}
4 | {{=A('documents', _href=URL('documents', args=request.args))}} ]<br />
5 {{=MARKMIN(page.body)}}
6 <h2>Comments</h2>
7 {{for comment in comments:}}
8 <p>{{=db.auth_user[comment.created_by].first_name}} on {{=comment.created_on}}
9 says <i>{{=comment.body}}</i></p>
10 {{pass}}
11 <h2>Post a comment</h2>
12 {{=form}}
```

如果你想使用 markdown 语法取代 markmin 语法：

```
1 from gluon.contrib.markdown import WIKI
```

并使用 WIKI 取代 MARKMIN 帮助对象。另外，你可以选择原始的 HTML 取代 markmin 句法。在本例中，你将取代：

```
1 {{=MARKMIN(page.body)}}
```

用：

```
1 {{=XML(page.body)}}
```

（这样 XML 没有得到转义，正如 web2py 默认行为）。

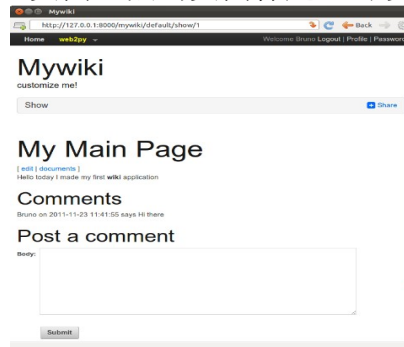
更好的做法如下：

```
1 {{=XML(page.body, sanitize=True)}}
```

通过设定 sanitize=True，告诉 web2py 转义不安全的 XML 标签例如“<script>”，这样防

止 XSS 漏洞。

现在如果从 index 页面，点击页面标题，可以看到你创建的页面：



下面是视图“default/edit.html”的代码：

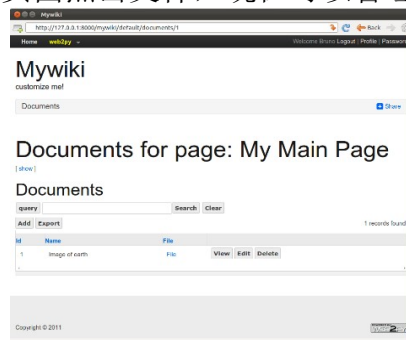
```
1 {{extend 'layout.html'}}
2 <h1>Edit wiki page</h1>
3 [ {{=A('show', _href=URL('show', args=request.args))}} ]<br />
4 {{=form}}
```

它产生一个看起来与 create 页面几乎相同的页面。

这里是视图“default/documents.html”的代码：

```
1 {{extend 'layout.html'}}
2 <h1>Documents for page: {{=page.title}}</h1>
3 [ {{=A('show', _href=URL('show', args=request.args))}} ]<br />
4 <h2>Documents</h2>
5 {{=grid}}
```

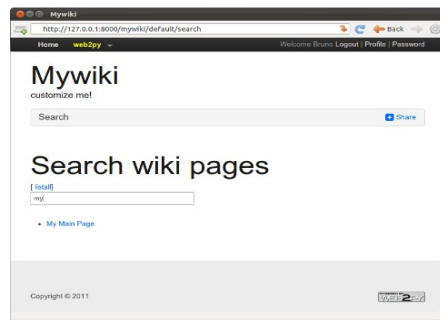
如果从“show”页面点击文件，现在可以管理附加到该页面的文件。



最后这里是视图“default/search.html”的代码：

```
1 {{extend 'layout.html'}}
2 <h1>Search wiki pages</h1>
3 [ {{=A('listall', _href=URL('index'))}} ]<br />
4 {{=form}}<br />{{=target_div}}
```

它生成如下 Ajax 搜索表单：



你还可以尝试直接调用 callback 动作，例如通过访问如下 URL：

```
1 http://127.0.0.1:8000/mywiki/default/callback?keyword=wiki
```

如果你看页面源代码，你会看到 callback 返回的 HTML：

```
1 <ul><li><a href="/mywiki/default/show/4">I made a Wiki</a></li></ul>
```

使用 web2py 从存储的页面生成 RSS 订阅很容易是因为 web2py 包含 gluon.contrib.rss2。

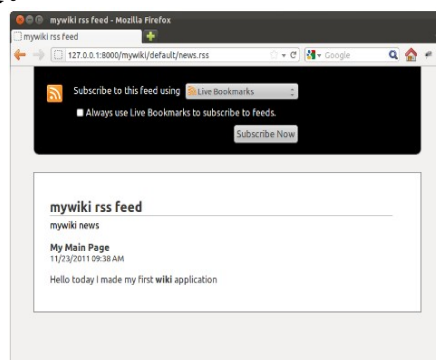
只需在默认控制器中追加如下动作：

```
1 def news():
2     "generates rss feed form the wiki pages"
3     reponse.generic_patterns = ['.rss']
4     pages = db().select(db.page.ALL, orderby=db.page.title)
5     return dict(
6         title = 'mywiki rss feed',
7         link = 'http://127.0.0.1:8000/mywiki/default/index',
8         description = 'mywiki news',
9         created_on = request.now,
10        items = [
11            dict(title = row.title,
12                link = URL('show', args=row.id),
13                description = MARKMIN(row.body).xml(),
14                created_on = row.created_on
15            ) for row in pages])
```

当访问如下页面

```
1 http://127.0.0.1:8000/mywiki/default/news.rss
```

你会看到订阅（确定的输出取决于订阅的读者），注意字典被自动转换成 RSS，得益于 URL 中的 .rss 扩展。



web2py 中包含 feedparser 以读取第三方订阅。

最后，让我们添加一个 XML-RPC 句柄，它允许以编程方式搜索 wiki：

```
1 service = Service()
2
3 @service.xmlrpc
4 def find_by(keyword):
5     "finds pages that contain keyword for XML-RPC"
6     return db(db.page.title.contains(keyword).select()).as_list()
7
8 def call():
9     "exposes all registered services, including XML-RPC"
10    return service()
```

这里，handler 动作只是发布（通过 XML-RPC）列表中指定的函数。本例中，find_by.find_by 不是一个动作（因为它需要一个参数），它使用.select() 查询数据库，使用.response 提取记录作为一个列表并返回该列表。

下面是从外部 Python 程序访问 XML-RPC 句柄的一个例子。

```
1 >>> import xmlrpclib
2 >>> server = xmlrpclib.ServerProxy(
3     'http://127.0.0.1:8000/mywiki/default/call/xmlrpc')
4 >>> for item in server.find_by('wiki'):
5     print item['created_on'], item['title']
```

可以从许多其它理解 XML-RPC 的编程语言访问句柄，包括 C、C++、C# 和 Java。

3.10.1 关于 date, datetime 和 time 属性格式

每个字段类型的 date、datetime 和 time 有三种不同的表示：

- 数据库表示
- 内部 web2py 表示
- 表单和表中的字符串表示

数据库表示是一个内部问题并且不会影响代码。内部，在 web2py 层面，它们分别被作为 datetime.date、datetime.datetime 以及 datetime.time 对象存储，可以如下操纵它们：

```
1 for page in db(db.page).select():
2     print page.title, page.day, page.month, page.year
```

使用 ISO 表示将日期转换成表单中的字符串

```
%Y-%m-%d %H:%M:%S
```

但这一表示是全局化的而且你可以使用 admin 翻译页面将格式改为备用。例如：

```
%m/%b/%Y %H:%M:%S
```

注意默认为英语而不是翻译的，因为 web2py 假定应用已经用英文编写，如果想让全局化适用于英语，需要创建翻译文件（使用 admin）而且需要声明该应用的当前语言是除英语以

外的其他东西，例如：

```
T.current_languages = ['null']
```

3.11 关于 admin 的更多内容

管理界面提供了附加功能，我们这里简要回顾。

3.11.1 *site* 页面

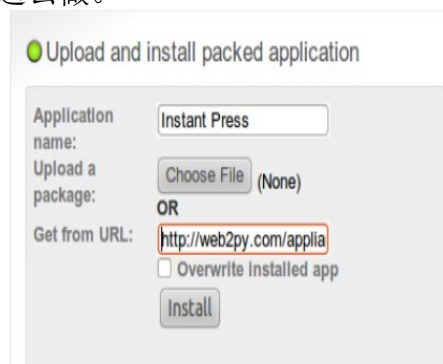
该页面列出所有安装的应用，底部有两个表单。

其中第一个表单允许通过指定名字创建新应用。

第二个表单允许从本地文件或远程 URL 上传已有应用。当你上传一个应用，你需要为它指定一个名字，可以是它的原始名字，但并不需要这样。这允许安装同一个应用的多个副本。例如，可以尝试从如下 URL 上传由 Martin Mulone 创建的 Instant Press CMS：

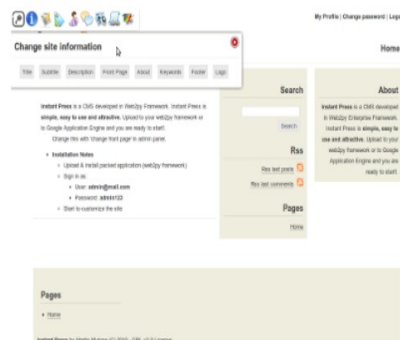
[1 http://code.google.com/p/instant-press/](http://code.google.com/p/instant-press/)

Web2py 文件打包成.w2p 文件，这些是 tar gzip 压缩文件，Web2py 使用.w2p 扩展代替.tgz 扩展来防止浏览器下载之后解压，可以使用 tar zxvf [filename]手动解压它们，尽管从来没有必要这么做。



成功上传之后，web2py 显示上传文件的 MD5 校验，可以用它验证文件在上传时有没有被损坏。InstantPress 会出现在已安装应用列表中。

在 admin 中，点击名字 InstantPress 来使它启动并运行。



可以通过如下 URL 阅读更多关于 Instant Press 的内容：

<http://code.google.com/p/instant-press/>

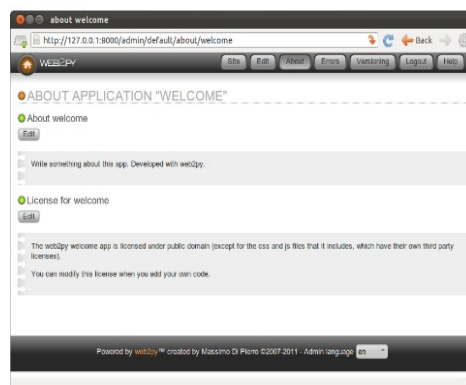
对每个应用，*site* 页面允许：

- 卸载应用。
- 跳转到 *about* 页面（阅读下面）。
- 跳转到 *edit* 页面（阅读下面）。
- 跳转到 *errors* 页面（阅读下面）。
- 清理临时文件（会话、错误以及 *cache.disk* 文件）。
- 打包全部。这返回一个包含完整应用副本的 *tar* 文件，我们建议在打包应用之前清理临时文件。
- 编译应用。如果没有错误，该选项将会字节码编译（*bytecode*）所有模型，控制器和视图。因为视图可以在一个树中扩展和包含其它视图，在字节码编译之前，每个控制器的视图树都被折叠成一个单一的文件，净效应是一个字节码编译应用，其速度更快，因为在运行时没有模版解析和字符串替换。
- 包编译完成。该选项仅存在于字节码编译完成的应用，无需源代码即可打包应用作为封闭源代码发布。注意 Python（与其它任何编程语言一样）可以从技术上反编译；因此编译不能提供对源代码的完整保护。不过，反编译可能是困难的并且可能是违法的。
- 移除已编译。它简单的从应用中移除字节码编译的模型，视图和控制器。如果应用打包带有源代码或者被本地编辑，移除字节码编译文件没有任何坏处，并且应用会继续工作，如果应用是从打包的编译文件安装的，那么这是不安全的，因为不能恢复到源代码，并且应用将不再工作。

web2py admin 站点页面的所有功能都可以通过定义在 *gluon/admin.py* 模型中的 API 编程访问，只需打开一个 Python shell（外壳）并导入该模型。

3.11.2 *about* 页面

about 选项卡允许编辑应用的描述以及它的许可证，它们分别被写入应用文件夹中的 ABOUT 和 LICENSE 文件。



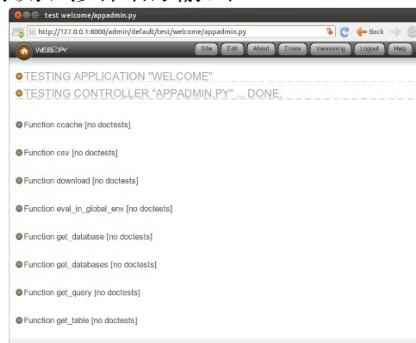
对这些文件，可以使用 MARKMIN 或 *gluon.contrib.markdown.WIKI* 语法，正如参考 [\[markdown2\]](#) 中描述的一样。

3.11.3 *edit* 页面

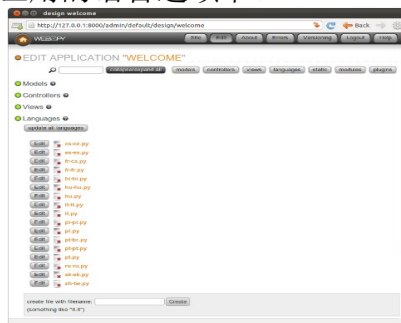
本章中已经使用了 *edit* 页面，这里我们想指出 *edit* 页面的一些更多功能。

- 如果点击任意文件名，可以看到语法高亮显示的文件内容。
- 如果点击 edit（编辑），可以通过 web 界面编辑文件。
- 如果点击 delete（删除），可以删除文件（永久性的）。
- 如果点击 test（测试），web2py 会运行测试，测试是开发人员使用 Python doctests 编写的，每个函数都应该有其自己的测试。
- 可以添加语言文件，扫描应用来发现所有字符串，并通过 web 界面编辑字符串翻译。
- 如果静态文件组织在文件夹和子文件夹中，可以通过点击文件夹名称切换文件夹层次结构。

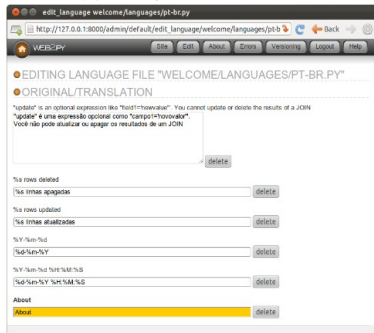
下图显示 *welcome* 应用测试页面的输出。



下图显示 *welcome* 应用的语言选项卡。



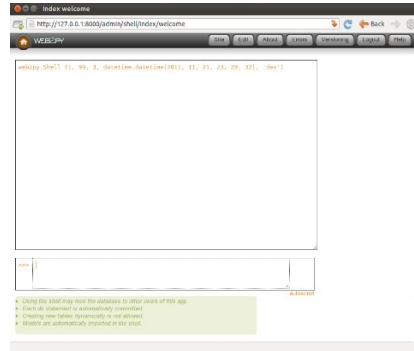
下图显示如何编辑语言文件，本例中是 *welcome* 应用的“it”（意大利语）语言。



shell

如果在 edit 页面中点击控制器选项卡下的“shell”链接，web2py 会打开一个基于 web 的

Python shell 并且会执行当前应用的模型，这使得你能交互的与你的应用对话。



crontab

编辑页面中同样是在控制器选项卡下有一个“crontab”链接，通过点击该链接将能够编辑 web2py 的 crontab 文件，它遵循的语法与 unix crontab 相同，但不依赖于 unix。实际上，它只需要 web2py，并且能在 Windows 上运行，允许你注册那些需要在预定时间后台执行的动作。

如需更多相关信息，请参见下一章。

3.11.4 *errors* 页面

在编程 web2py 时，难免不犯错误和引入漏洞。web2py 提供了两方面的帮助：1) 它允许你为每个函数创建测试，可以在浏览器中从 edit 页面运行；2) 当错误表现出来时，发出一张票据给访问者并且错误被记录。

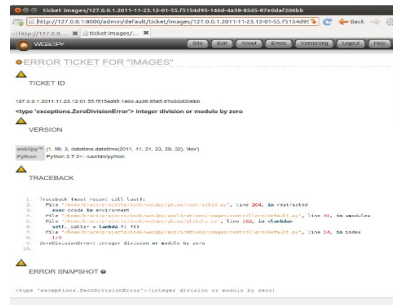
如下，在 images 应用中故意引入错误：

```
1 def index():  
2     images = db().select(db.image.ALL, orderby=db.image.title)  
3     1/0  
4     return dict(images=images)
```

当访问 index 动作时，得到如下票据：



只有管理员才能访问票据：



票据显示回溯，文件的内容引发了问题和系统的完整状态（变量、请求、会话等），如果错误出现在视图中，web2py 显示从 HTML 转换成 Python 代码的视图，这可以很容易地确定文件的逻辑结构。

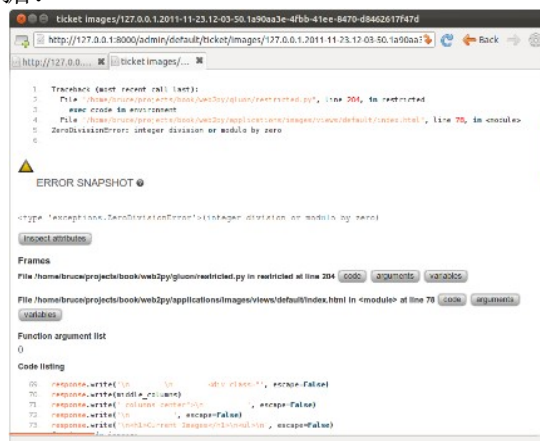
默认情况下，票据存储在文件系统中并按照回溯分组，管理界面提供聚合视图（回溯的类型和发生的次数）和一个详细视图（所有票据按照票据 id 列出），管理员可以切换两个视图。

注意 *admin* 处显示语法高亮的代码（例如，在错误报告中，*web2py* 关键字显示为黄色）。如果点击 *web2py* 关键字，你会被重定向到有关该关键字的文档页面。

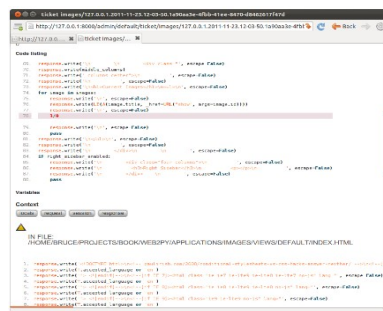
如果 *index* 动作中修复了除以零的错误并在 *index* 视图中引入一个：

```
1 {{extend 'layout.html'}}
2
3 <h1>Current Images</h1>
4 <ul>
5 {{for image in images:}}
6 {{1/0}}
7 {{=LI(A(image.title, _href=URL("~/show", args=image.id))}}
8 {{pass}}
9 </ul>
```

会得到如下票据：



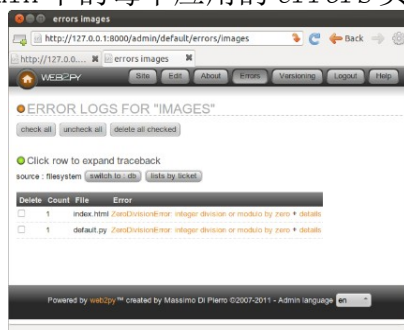
注意 *web2py* 已经将视图从 HTML 转换成 Python 代码，并且票据中描述的错误指的是生成的 Python 代码而不是原始视图文件：



最初这可能看起来令人困惑，但在实践中它使调试变得更容易，因为 Python 缩进突出显示嵌入视图中代码的逻辑结构。

代码显示在同一页面的底部。

所有票据都在 admin 下的每个应用的 *errors* 页面被列出：



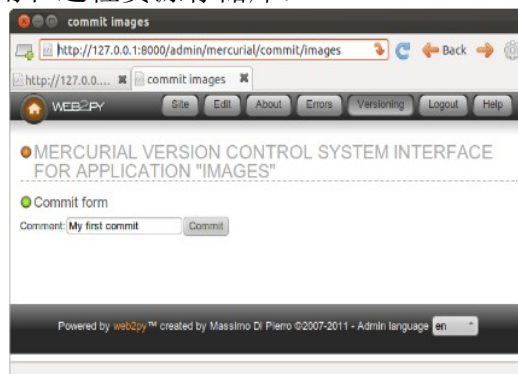
3.11.5 Mercurial 版本

如果从源代码运行并安装了有 Mercurial 版本的控制库：

`easy_install mercurial`

然后，管理界面显示一个称为“mercurial”的菜单项，它自动地为应用创建一个本地 Mercurial 存储库。在页面中按下“commit”按钮将提交当前应用，Mercurial 创建并存储那些你在应用子文件夹中的隐藏文件夹“.hg”中对代码所做改动的信息。每个应用都有自己的“.hg”文件夹和“.hgignore”文件（告知 Mercurial 忽略哪些文件）。

Mercurial web 界面允许你浏览以前的提交以及 diff 文件但我们强烈建议直接从 shell 或许多基于 GUI 的 Mercurial 客户端来使用 Mercurial，因为它们功能更强大，例如它们允许你同步你的应用和远程资源存储库：

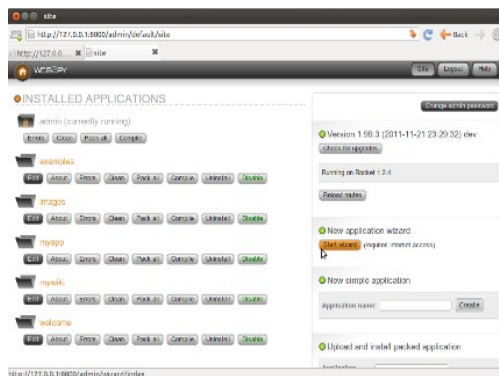


可以在如下地址阅读关于 Mercurial 的更多内容：

<http://mercurial.selenic.com/>

3.11.6 Admin 向导（实验性的）

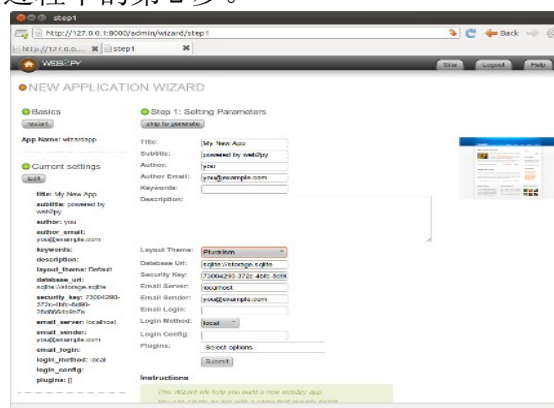
admin 界面包含一个向导，它能帮助创建新应用，可以从“sites”页面访问向导，如下图所示：



向导将指导你通过一系列创建新应用所涉及到的步骤：

- 为应用选择名称
- 配置应用并选择需要的插件
- 构建需要的模型（它将为每个模型创建 CRUD 页面）
- 允许使用 MARKMIN 语法编辑页面视图

下图显示该过程中的第 2 步。



你会看到一个下拉菜单来选择一个布局插件（从 web2py.com/layouts），一个多选下拉菜单来检查其它插件（从 web2py.com/plugins），还有一个“login config”字段在那儿放置的 Janrain “domain:key”。

其它步骤都非常不言自明。

向导能很好地完成它的工作，但它却被视为一种实验性功能，原因有两个：

- 使用向导创建和手动编辑过的应用，以后不能被向导修改。
- 向导界面会随时间而变，包括支持更多的功能和更容易的可视化开发。

无论如何，向导是一个用来快速制作原型的得心应手的工具，它可以用来引导带有备用布局和可选插件的新应用。

3.11.7 配置 admin

通常无需进行任何 admin 配置，但一些自定义是可能的。在登录 admin 之后，可以通过如下 URL 编辑 admin 配置文件：

```
http://127.0.0.1:8000/admin/default/edit/admin/models/0.py
```

注意 admin 可以用来编辑本身，实际上 admin 是一个与其它任何应用一样的应用。

文件“0.py”在很大程度上是自我记录，并且如果你正在打开可能你已经知道你正在寻找什么。无论如何，有一些自定义比其它更重要：

```
GAE_APPCFG = os.path.abspath(os.path.join('/usr/local/bin/appcfg.py'))
```

这应该指向“appcfg.py”文件的位置，该文件配备 Google App Engine SDK。如果已安装了 SDK 你可能需要将配置参数改为正确的值，这将允许你从 admin 界面部署到 GAE。

还可以在演示模式下设置 web2py admin：

```
DEMO_MODE = True
FILTER_APPS = ['welcome']
```

只有列出在过滤应用中的应用才能访问，并且只能在只读模式下访问。

如果你是一名教师，需要公开管理界面给学生，使学生可以分享他们的项目管理界面（可认为是虚拟实验室），可以如下设置：

```
MULTI_USER_MODE = True
```

这样，学生需要登录并且通过 admin 将只能访问自己的应用，而你作为第一用户/老师，将能够访问全部应用。

注意这个机制仍然假定所有的用户都是可信任的，admin 下创建的所有应用都运行在相同文件系统的相同凭据下，一名学生创建的应用可能访问来自另一名学生创建的应用的数据和源代码。

3.12 更多关于 appadmin

appadmin 的目的不是被公开曝光，它旨在帮助你提供一个容易的数据库访问。它只包含两个文件：控制器“appadmin.py”和视图“appadmin.html”，它们被控制器中的所有动作使用。

appadmin 控制器相对较小且可读，它提供了一个设计数据库接口的例子。

appadmin 显示可用的数据库和每个数据库中存在的表，可以插入记录并为每个表单独列出所有记录。appadmin 分页输出，一次 100 条记录。

一旦一组记录被选中，页头就会变化，允许更新或删除选中的记录。

更新记录，在查询字符串字段输入一个 SQL 任务：

```
1 title = 'test'
```

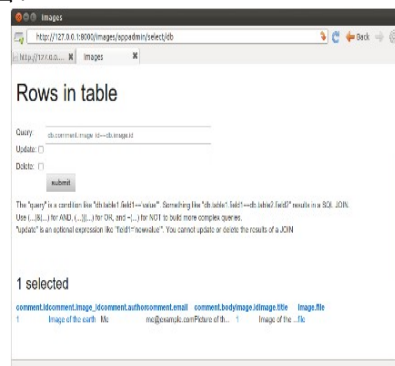
字符串值必须括在单引号内，可以用逗号分隔成多个字段。

删除一条记录，单击相应的复选框以确认你确实想删除。

appadmin 还可以进行连接，如果 SQL FILTER（过滤器）包含的 SQL 条件涉及两个或多个表。例如，尝试：

```
1 db.image.id == db.comment.image_id
```

web2py 将它传递到 DAL，并且 DAL 理解该查询链接的两个表；因此，两个表都被 INNER JOIN 选中。下面是输出：



如果点击 id 字段的数字，会得到相应 ID 记录的编辑页面。

如果点击参考字段的数字，会得到一个编辑页面所引用的记录。

不能更新或删除由联合查询选定的行，因为它们涉及多个表中的记录并且这将是含糊不清的。

除了其数据库管理能力，appadmin 还能让你查看有关应用程序缓存（在 / yourapp/ appadmin/ ccache 中）内容的详细信息，以及当前 request、response 和 session 对象（在 / yourapp/ appadmin/ state 中）的内容。

appadmin 用自己的菜单取代 response.menu，它提供了到 admin 应用的 edit 界面、db（数据库管理）界面、state 界面和 cache 界面的链接。如果你的应用布局不会使用 response.menu 生成菜单，那么将看不到 appadmin 菜单。在这种情况下，可以修改

appadmin.html 文件并添加 `{{=MENU(response.menu)}}` 以显示菜单。

第 4 章 核心

4.1 命令行选项

可以跳过 GUI 直接从命令行启动 web2py，方法如下：

```
1. python web2py.py -a 'your password' -i 127.0.0.1 -p 8000
```

当 web2py 启动时，它会创建一个文件名为“parameters_8000.py”，其中存储散列的密码，如果使用密码“<ask>”，web2py 会提示。

为了额外安全性，可以如下启动 web2py：

```
1. python web2py.py -a '<recycle>' -i 127.0.0.1 -p 8000
```

在这种情况下，web2py 重用以前存储的哈希密码，如果没有提供密码或“parameters_8000.py”文件被删除，基于 web 的管理界面会被禁用。

一些 Unix/ Linux 系统上，如果密码是

```
1. <pam_user:some_user>
```

web2py 使用操作系统帐户 some_user 的 PAM 密码来认证管理员，除非被 PAM 配置阻止。

web2py 通常运行 CPython (Python 解释器的 C 语言实现由 Guido van Rossum 创建)，但它也能运行 Jython (解释器的 Java 实现)，后者的可能性允许 web2py 在 J2EE 基础架构的情况下使用。使用 Jython，只需更换“python web2py.py ...”为“jython web2py.py”，可以在第 14 章中找到关于安装 Jython 和访问数据库需要的 zxJDBC 模块的详细信息。

“web2py.py”脚本可以采取许多命令行参数，指定最大线程数，启用 SSL 等。如下获取完整列表：

```
1 >>> python web2py.py -h
2 Usage: python web2py.py
3
4 web2py Web Framework startup script. ATTENTION: unless a password
5 is specified (-a 'passwd'), web2py will attempt to run a GUI.
6 In this case command line options are ignored.
7
8 Options:
9 --version show program's version number and exit
10 -h, --help show this help message and exit
11 -i IP, --ip=IP ip address of the server (127.0.0.1)
12 -p PORT, --port=PORT port of server (8000)
13 -a PASSWORD, --password=PASSWORD
14 password to be used for administration (use -a
15 "<recycle>" to reuse the last password)
16 -c SSL_CERTIFICATE, --ssl_certificate=SSL_CERTIFICATE
```

```
17 file that contains ssl certificate
18 -k SSL_PRIVATE_KEY, --ssl_private_key=SSL_PRIVATE_KEY
19 file that contains ssl private key
20 -d PID_FILENAME, --pid_filename=PID_FILENAME
21 file to store the pid of the server
22 -l LOG_FILENAME, --log_filename=LOG_FILENAME
23 file to log connections
24 -n NUMTHREADS, --numthreads=NUMTHREADS
25 number of threads (deprecated)
26 --minthreads=MINTHREADS
27 minimum number of server threads
28 --maxthreads=MAXTHREADS
29 maximum number of server threads
30 -s SERVER_NAME, --server_name=SERVER_NAME
31 server name for the web server
32 -q REQUEST_QUEUE_SIZE, --request_queue_size=REQUEST_QUEUE_SIZE
33 max number of queued requests when server unavailable
34 -o TIMEOUT, --timeout=TIMEOUT
35 timeout for individual request (10 seconds)
36 -z SHUTDOWN_TIMEOUT, --shutdown_timeout=SHUTDOWN_TIMEOUT
37 timeout on shutdown of server (5 seconds)
38 -f FOLDER, --folder=FOLDER
39 folder from which to run web2py
40 -v, --verbose increase --test verbosity
41 -Q, --quiet disable all output
42 -D DEBUGLEVEL, --debug=DEBUGLEVEL
43 set debug output level (0-100, 0 means all, 100 means
44 none; default is 30)
45 -S APPNAME, --shell=APPNAME
46 run web2py in interactive shell or IPython (if
47 installed) with specified appname (if app does not
48 exist it will be created). APPNAME like a/c/f (c,f
49 optional)
50 -B, --bpython run web2py in interactive shell or bpython (if
51 installed) with specified appname (if app does not
52 exist it will be created). Use combined with --shell
53 -P, --plain only use plain python shell; should be used with
54 --shell option
55 -M, --import_models auto import model files; default is False; should be
56 used with --shell option
57 -R PYTHON_FILE, --run=PYTHON_FILE
58 run PYTHON_FILE in web2py environment; should be used
59 with --shell option
60 -K SCHEDULER, --scheduler=SCHEDULER
61 run scheduled tasks for the specified apps
62 -K app1, app2, app3 requires a scheduler defined in the
63 models of the respective apps
64 -T TEST_PATH, --test=TEST_PATH
65 run doctests in web2py environment; TEST_PATH like
66 a/c/f (c,f optional)
67 -W WINSERVICE, --winservice=WINSERVICE
68 -W install/start/stop as Windows service
69 -C, --cron trigger a cron run manually; usually invoked from a
70 system crontab
71 --softcron triggers the use of softcron
72 -N, --no-cron do not start cron automatically
```

```

73 -J, --cronjob identify cron-initiated command
74 -L CONFIG, --config=CONFIG
75 config file
76 -F PROFILER_FILENAME, --profiler=PROFILER_FILENAME
77 profiler filename
78 -t, --taskbar use web2py gui and run in taskbar (system tray)
79 --nogui text-only, no GUI
80 -A ARGS, --args=ARGS should be followed by a list of arguments to be passed
81 to script, to be used with -S, -A must be the last
82 option
83 --no-banner Do not print header banner
84 --interfaces=INTERFACES
85 listen on multiple addresses:
86 "ip:port:cert:key;ip2:port2:cert2:key2;..." (:cert:key
87 optional; no spaces)

```

小写选项用于配置 web 服务器，-L 选项告诉 web2py 从文件中读取配置选项，-W 安装 web2py 作为一个 Windows 服务，而 -S、-P 和 -M 选项启动交互式 Python shell，-T 选项发现并在 web2py 执行环境中运行控制器 doctests。例如下面的例子，从“welcome”应用中的所有控制器运行 doctests：

```
1. python web2py.py -vT welcome
```

如果运行 web2py 作为 Windows 服务，通过使用命令行参数传递配置来说，-W 是不方便的。出于这个原因，web2py 文件夹中有一个对内部 web 服务器的样本“options_std.py”配置文件：

```

1 import socket
2 import os
3
4 ip = '0.0.0.0'
5 port = 80
6 interfaces=[('0.0.0.0',80)]
7 #interfaces.append(('0.0.0.0',443,'ssl_private_key.pem','ssl_certificate.pem'))
8 password = '<recycle>' # ## <recycle> means use the previous password
9 pid_filename = 'httpserver.pid'
10 log_filename = 'httpserver.log'
11 profiler_filename = None
12 minthreads = None
13 maxthreads = None
14 server_name = socket.gethostname()
15 request_queue_size = 5
16 timeout = 30
17 shutdown_timeout = 5
18 folder = os.getcwd()
19 extcron = None
20 nocron = None

```

这个文件包含了 web2py 的默认，如果编辑这个文件，需要用 -L 命令行选项明确地导入它，它仅当作为一个 Windows 服务运行 web2py 才适用。

4.2 工作流

web2py 工作流如下：

- HTTP 请求到达 web 服务器（内置 Rocket 服务器或通过 WSGI 连接到 web2py 的不同服务器或另外的适配器），该 web 服务器在其自己的线程中，并行处理每个请求。
- HTTP 请求标头被解析并传递到调度器（在本章稍后解释）。
- 调度器将决定由哪一个安装应用处理请求，并把 URL 中的 PATH_INFO 映射成函数调用，每一个 URL 对应一个函数调用。
- 对于静态文件夹中的文件请求被直接处理，并且大文件被自动地流式传输到客户端。
- 除静态文件之外的请求被映射成动作（例如：在请求应用中的一个控制器文件中的函数）。
- 调用动作之前，发生的几件事：如果请求标头包含应用的会话 cookie，会话对象被恢复；如果没有，会话 id 被创建（但会话文件直到后来才被保存）；创建请求的执行环境，在这种环境下模型被执行。
- 最后，控制器动作在预先构建的环境中执行。
- 如果动作返回一个字符串，它被返回客户端（或者如果该动作返回一个 web2py 的 HTML 帮助对象，它被序列化并返回给客户端）。
- 如果动作返回一个可迭代工作流，它被用来循环并将数据分流传送到客户端。
- 如果动作返回一个字典，web2py 会试图找到一个视图呈现该字典，视图必须与动作名称相同（除非另有规定）并且扩展名与被请求的页面相同（默认为.html）；失败的话，web2py 可能采用通用视图（如果可用并且已启用），视图看到定义在模型中和由动作返回的字典中的每个变量，但没有看到在控制器中定义的全局变量。
- 整个用户代码在单一事务中执行，除非另有规定。
- 如果用户代码成功，事务被提交。
- 如果用户代码失败，回溯被存储在票据中，并且票据 ID 被发送给客户端。只有系统管理员能搜索和阅读票据中的回溯。

有一些注意事项要牢记：

- 同一文件夹/子文件夹中的模型按字母顺序执行。
- 模型中定义的任何变量对按字母顺序的模型、控制器和视图都是可见的。
- 子文件夹中的模型被有条件的执行。例如，如果用户请求“/a/c/f”，这儿“a”是应用，“c”是控制器，“f”是函数（动作），那么如下模型会被执行：

```
applications/a/models/*.py
applications/a/models/c/*.py
applications/a/models/c/f/*.py
```

- 请求的控制器被执行并且请求的函数被调用，这意味着控制器中的所有顶层代码在对该控制器的每个请求中被执行。
- 只有动作返回一个字典时，视图才被调用。
- 如果找不到视图，web2py 会试图使用通用视图。默认情况下，通用视图被禁用，虽然 ‘welcome’ 应用在 /models/db.py 中包含一行代码，该代码在 localhost 上启用它们。每个扩展名以及每个动作能启用它们（使用 response.generic_patterns）。一般说来，通用视图是一个开发工具，通常不在生产中使用。如果你想让某些动作使用通用视图，将那些动作列在 response.generic_patterns 中（更多细节在 Services 章节中讨论）。

动作的可能行为有以下几种：

返回一个字符串

```
def index(): return 'data'
```

为视图返回一个字典：

```
def index(): return dict(key='value')
```

返回所有局部变量：

```
def index(): return locals()
```

将访问者重定向到另一页面:

```
def index(): redirect(URL('other_action'))
```

返回“200 OK”之外的HTTP页面:

```
def index(): raise HTTP(404)
```

返回一个帮助对象(例如, 一个表单):

```
def index(): return FORM(INPUT(_name='test'))
```

(这主要用于Ajax回调和组件, 请参阅第12章)

当动作返回一个字典, 其中可能包含帮助对象生成的代码, 包括基于数据库表的表单或来自工厂的表单, 例如:

```
def index(): return dict(form=SQLFORM.factory(Field('name')).process())
```

(web2py生成的所有表单都使用回传, 见第3章)

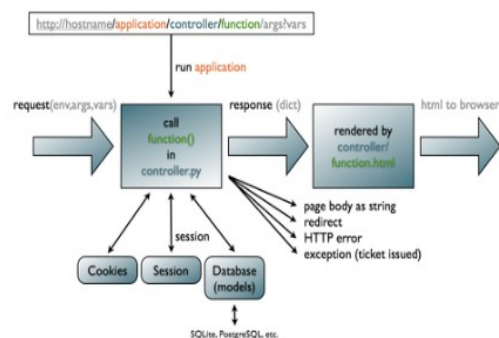
4.3 调度

web2py将映射成如下形式的URL:

```
1. http://127.0.0.1:8000/a/c/f.html
```

到应用“a”控制器“c.py”中的函数f(), 如果f不存在, web2py默认使用index控制器函数; 如果c不存在, web2py默认使用“default.py”控制器; 并且如果a不存在, web2py默认使用init应用; 如果没有init应用, web2py尝试运行welcome应用。示意图如下:

默认应用、控制器和函数的名称可以在routes.py中重写, [Default Application, Controller 和 Function](#) 如下:



默认情况下, 任何新请求都创建一个新会话。除此之外, 会话cookie被返回客户端浏览器来追踪会话。

扩展名.html是可选的, .html被假定为默认, 该扩展名决定了用于呈现控制器函数f()的输出的视图的扩展名, 它允许以多种格式(html、xml、json、rss等)服务相同的内容。

带参数或双下划线开始的函数是不公开的并且只能由其它函数调用。

如下形式的URL例外:

```
1. http://127.0.0.1:8000/a/static/filename
```

没有被称为“static”的控制器, web2py把这解释成请求应用“a”子文件夹“static”中名为“filename”的文件。

当静态文件被下载, web2py并不创建会话, 也不发出cookie或执行模型, web2py总是以1MB的块数据流静态文件, 以及当客户端发送RANGE请求文件的一个子集时, 发送

PARTIAL CONTENT。web2py 还支持 IF_MODIFIED_SINCE 协议，如果文件已经存储在浏览器缓存中并且版本没有改变就不发送文件。

当链接到静态文件夹中的一个音频或视频文件时，如果想强制浏览器下载文件，而不是通过媒体播放器的音频/视频流，将?附件添加到 URL，它告诉 web2py 将 Content-Disposition 标头设置为“attachment”。例如：

```
1. <a href="/app/static/my_audio_file.mp3?attachment">Download</a>
```

当上面的链接被点击后，浏览器会提示用户下载 MP3 文件，而不是立即流媒体音频，（正如[下面](#)所讨论，还可以通过分配标头名字典和它们的值给 response.headers. 来直接设置 HTTP 请求标头。）

web2py 映射如下形式的 GET/POST 请求：

```
1. http://127.0.0.1:8000/a/c/f.html/x/y/z?p=1&q=2
```

到应用 a 控制器“c.py”中的函数 f，并且它在 request 变量中存储 URL 参数如下：

```
1. request.args = ['x', 'y', 'z']
```

和：

```
1. request.vars = {'p':1, 'q':2}
```

以及：

```
1. request.application = 'a'
2. request.controller = 'c'
3. request.function = 'f'
```

在上面的例子中，request.args[i] 和 request.args(i) 都能用来检索 request.args 的第 i 个元素，如果列表没有这样的索引，前者会引发异常，后者在这种情况下返回 None。

```
1. request.url
```

存储当前请求的完整 URL（不包括 GET 变量）。

```
1. request.ajax
```

默认为 False，但如果 web2py 决定动作是 Ajax 请求调用的，它就是 True。

如果请求是 Ajax 请求并且是由 web2py 组件发起的，可以找到该组件的名称：

```
1. request.cid
```

在第 12 章详细讨论了组件。

如果 HTTP 请求是一个 GET，那么 request.env.request_method 被设置为“GET”；如果它是一个 POST，request.env.request_method 被设置为“POST”。URL 查询变量存储在 request.vars 存储字典中，它们还存储在 request.get_vars（跟随 GET 请求）或 request.post_vars 中（跟随 POST 请求）。

web2py 在 request.env 中存储 WSGI 和 web2py 环境变量，例如：

```
1. request.env.path_info = 'a/c/f'
```

和 HTTP 标头到环境变量，例如：

```
1. request.env.http_host = '127.0.0.1:8000'
```

请注意，web2py 验证所有的 URL 以防止目录遍历攻击。

URL 只允许包含字母数字字符，下划线和斜线，args 可以包含非连续点。验证前空格替换为下划线，如果是无效的 URL 语法，web2py 会返回一个 HTTP400 错误消息 `[http:w,http:o]`。

如果该 URL 对应一个静态文件请求，web2py 简单的读取和返回（流式传输）请求的文件。

如果 URL 不请求静态文件，web2py 按以下顺序处理要求：

- 解析 cookie。
- 创建一个环境，在其中执行函数。
- 初始化请求、响应、缓存。
- 打开现有会话或创建一个新的。
- 执行属于请求应用的模型。
- 执行请求的控制器动作函数。
- 如果该函数返回一个字典，执行相关的视图。
- 如果成功，提交所有开放事务。
- 保存会话。
- 返回一个 HTTP 响应。

注意，控制器和视图在同样环境的不同副本下执行；因此，视图不会看到控制器，但它会看到模型并且它能看到控制器动作函数返回的变量。

如果引发异常（非 HTTP），web2py 会执行以下操作：

- 将回溯存储在一个错误文件中并为它分配一个票据号。
- 回滚所有开放事务。
- 返回一个报告票据号的错误页面。

如果异常是 HTTP 异常，这被认为是预期行为（例如，一个 HTTP 重定向），并且所有数据库事务被提交，此后的行为由 HTTP 异常本身指定，HTTP 异常类不是标准的 Python 异常，它是由 web2py 定义的。

4.4 库

web2py 库作为全局对象公开给用户应用。例如（**request**、**response**、**session**、**cache**），类（**helper** 帮助对象、**validator** 验证器、**DAL** API）和函数（**T** 和 **redirect**）。

这些对象定义在以下核心文件中：

```
1. web2py.py
2. gluon/__init__.py gluon/highlight.py gluon/restricted.py gluon/streamer.py
3. gluon/admin.py gluon/html.py gluon/rewrite.py gluon/template.py
4. gluon/cache.py gluon/http.py gluon/rocket.py gluon/storage.py
5. gluon/cfs.py gluon/import_all.py gluon/sanitizer.py gluon/tools.py
6. gluon/compileapp.py gluon/languages.py gluon/serializers.py gluon/utils.py
7. gluon/contenttype.py gluon/main.py gluon/settings.py gluon/validators.py
8. gluon/dal.py gluon/myregex.py gluon/shell.py gluon/widget.py
9. gluon/decoder.py gluon/newcron.py gluon/sql.py gluon/winservice.py
10. gluon/fileutils.py gluon/portlocker.py gluon/sqlhtml.py gluon/xmlrpc.py
11. gluon/globals.py gluon/reserved_sql_keywords.py
```

web2py 附带的 tar gzip（解包解压缩）基本构建应用是：

```
1. welcome.w2p
```

它在安装时被创建，在升级时被覆盖。

第一次启动 web2py 时，创建两个新文件夹：deposit 和 applications，“welcome”应用被压缩成 “welcome.w2p” 文件，作为基本构建应用来使用，deposit 文件夹用来作为安装和卸载应用的临时存储。

web2py 的单元测试是在

```
1. gluon/tests/
```

有与各种 web 服务器连接的处理程序：

```
1.  cgihandler.py      # discouraged
2.  gaehandler.py      # for Google App Engine
3.  fcgihandler.py     # for FastCGI
4.  wsgihandler.py     # for WSGI
5.  isapiwsgihandler.py # for IIS
6.  modpythonhandler.py # deprecated
```

（“fcgihandler”调用由 Allan Saddi 开发的“gluon/contrib/gateways/fcgi.py”）和

```
anyserver.py
```

这是一个与许多不同 Web 服务器接口的脚本，第 13 章介绍。

有 3 个示例文件：

```
1. options_std.py
2. routes.example.py
3. router.example.py
```

前者是一个可选的配置文件，可以通过 -L 选项传递给 web2py.py；第二个是一个 URL 映射文件的例子，当重命名为“routes.py” 它会自动加载；第三个是一个 URL 映射的替代语法，也可以重命名（或复制到）“routes.py”。

文件

```
1. app.yaml
2. index.yaml
3. queue.yaml
```

是用来在 Google App Engine 上部署的配置文件，可以在 Deployment Recipes 章节和 Google Documentation 页面阅读更多关于它们的内容。

还有附加的库，通常由第三方开发：

Mark Pilgrim 开发的读取 RSS 和 Atom feeds 的 *feedparser*^[feedparser]：

```
1. gluon/contrib/__init__.py
2. gluon/contrib/feedparser.py
```

Trent Mick 开发的用于 wiki 标记的 *markdown2*^[markdown2]:

1. [gluon/contrib/markdown/__init__.py](#)
2. [gluon/contrib/markdown/markdown2.py](#)

markmin 标记:

1. [gluon/contrib/markmin.py](#)

Mariano Reingart 创建的生成 PDF 文档的 *pyfpdf*:

- [gluon/contrib/pyfpdf](#)

本书中没有对此介绍，对它的介绍在如下位置:

<http://code.google.com/p/pyfpdf/>

由 Mariano Reingart 创建的一个轻量级的 SOAP 服务器实现:

1. [gluon/contrib/pysimplesoap/](#)

由 Mariano Reingart 创建的一个轻量级的 JSON-RPC 客户端:

- [gluon/contrib/simplejsonrpc.py](#)

Evan Martin 开发的 *memcache*^[memcache] Python API:

- [gluon/contrib/memcache/__init__.py](#)
- [gluon/contrib/memcache/memcache.py](#)

redis_cache 是一个在 redis 数据库中存储缓存的模块:

- [gluon/contrib/redis_cache.py](#)

gql 是到 Google App Engine 的一个 DAL 端口:

1. [gluon/contrib/gql.py](#)

memdb 是在 memcache 顶部的一个 DAL 端口:

1. [gluon/contrib/memdb.py](#)

gae_memcache 是一个在 Google App Engine 上使用 memcache 的 API:

1. [gluon/contrib/gae_memcache.py](#)

pyrtf^[26] 用于生成 RTF 格式文件，由 Simon Cusack 开发，Grant Edwards 修订:

1. [gluon/contrib/pyrtf](#)
2. [gluon/contrib/pyrtf/__init__.py](#)
3. [gluon/contrib/pyrtf/Constants.py](#)
4. [gluon/contrib/pyrtf/Elements.py](#)
5. [gluon/contrib/pyrtf/PropertySets.py](#)
6. [gluon/contrib/pyrtf/README](#)
7. [gluon/contrib/pyrtf/Renderer.py](#)
8. [gluon/contrib/pyrtf/Styles.py](#)

Dalke Scientific Software 开发的 *PyRSS2Gen*^[pyrss2gen] 用于生成 RSS 订阅:

1. [gluon/contrib/rss2.py](#)

simplejson^[simplejson] 是由 Bob Ippolito 开发的用于分析和编写 JSON 对象的标准库:

1. gluon/contrib/simplejson/__init__.py
2. gluon/contrib/simplejson/decoder.py
3. gluon/contrib/simplejson/encoder.py
4. gluon/contrib/simplejson/jsonfilter.py
5. gluon/contrib/simplejson/scanner.py

Google Wallet ^[googlewallet] 提供 “pay now” 按钮，其链接谷歌作为付款处理器：

1. gluon/contrib/google_wallet.py

Stripe.com[98]提供一个简单的 API 用于接受信用卡付款：

1. gluon/contrib/stripe.py

AuthorizeNet[99]提供 API 通过 Authorize.net 网络接收信用卡付款

1. gluon/contrib/AuthorizeNet.py

Dowcommerce ^[dowcommerce] 是另一个信用卡购物车处理 API：

1. gluon/contrib/DowCommerce.py

Chris AtLee 创建的 *PAM* ^[PAM] 验证 API：

1. gluon/contrib/pam.py

一个贝叶斯分类器，为了测试目的而用虚拟数据填充数据库：

1. gluon/contrib/populate.py

当 *web2py* 作为服务运行时，允许与 Windows 任务栏交互的一个文件：

1. gluon/contrib/taskbar_widget.py

可选的 *login_methods* 和 *login_forms* 被用于身份认证：

1. gluon/contrib/login_methods/__init__.py
2. gluon/contrib/login_methods/basic_auth.py
3. gluon/contrib/login_methods/cas_auth.py
4. gluon/contrib/login_methods/dropbox_account.py
5. gluon/contrib/login_methods/email_auth.py
6. gluon/contrib/login_methods/extended_login_form.py
7. gluon/contrib/login_methods/gae_google_account.py
8. gluon/contrib/login_methods/ldap_auth.py
9. gluon/contrib/login_methods/linkedin_account.py
10. gluon/contrib/login_methods/loginza.py
11. gluon/contrib/login_methods/oauth10a_account.py
12. gluon/contrib/login_methods/oauth20_account.py
13. gluon/contrib/login_methods/openid_auth.py
14. gluon/contrib/login_methods/pam_auth.py
15. gluon/contrib/login_methods/rpx_account.py
16. gluon/contrib/login_methods/x509_auth.py

web2py 还包括一个含有如下有用脚本的文件夹

```
1.  scripts/setup-web2py-fedora.sh
2.  scripts/setup-web2py-ubuntu.sh
3.  scripts/setup-web2py-nginx-uwsgi-ubuntu.sh
4.  scripts/update-web2py.sh
5.  scripts/make_min_web2py.py
6.  ...
7.  scripts/sessions2trash.py
8.  scripts/sync_languages.py
9.  scripts/tickets2db.py
10. scripts/tickets2email.py
11. ...
12. scripts/extract_mysql_models.py
13. scripts/extract_pgsql_models.py
14. ...
15. scripts/access.wsgi
16. scripts/cpdb.py
```

前三个尤其有用，因为它们从头开始尝试完整的安装和设置一个 web2py 生产环境，其中一些在第 14 章中讨论，但它们里面都包含一个文档字符串解释用途和用法。

最后，web2py 包含建立二进制发行版需要的文件。

```
1. Makefile
2. setup_exe.py
3. setup_app.py
```

py2exe 和 py2app 分别有设置脚本，只有建立 web2py 的二进制发行版时才需要它们，你应该从来不需要运行它们。

总之，web2py 库提供以下功能：

- 将 URL 映射成函数调用。
- 通过 HTTP 处理传递和返回参数。
- 执行这些参数的验证。
- 从大多数安全问题中保护应用程序。
- 处理数据持久性（数据库、会话、缓存、cookie）。
- 为各种支持的语言执行字符串翻译。
- 以编程方式生成 HTML（例如，从数据库表）。
- 通过数据库抽象层（DAL）生成 SQL。
- 生成 Rich Text Format 格式（RTF）输出。
- 从数据库表中生成逗号分隔值（CSV）输出。
- 生成简易信息聚合（RSS）订阅。
- 为 Ajax 生成 JavaScript 对象符号（JSON）序列化字符串。
- wiki 标记（Markdown）转换为 HTML。
- 公开 XML-RPC web 服务。
- 通过流媒体上传和下载大文件。

web2py 应用包含其它文件，特别是第三方 JavaScript 库，例如 jQuery、calendar/datepicker、EditArea 和 nicEdit。文件本身都认可它们的作者。

4.5 应用程序

web2py 中开发的应用由以下几部分组成：

- 模型描述数据库表的数据表示和表之间的关系。
- 控制器描述应用逻辑和工作流。
- 视图描述应如何使用 HTML 和 JavaScript 将数据呈现给用户。
- 语言描述如何将应用中的字符串翻译成各种受支持的语言。
- 静态文件不需要处理（例如图像、CSS 样式表等）。
- ABOUT 和 README 文件是不言自明的。
- errors 存储应用生成的错误报告。
- 会话存储有关每个特定用户的信息。
- database 存储 SQLite 数据库和其它表的信息。
- cache 存储缓存的应用条目。
- modules 是其它可选的 Python 模块。
- private 文件通过控制器访问而不是由开发者直接访问。
- upload 文件由模型访问而不是由开发者直接访问（例如，由应用用户上传的文件）。
- tests 是一个目录用来存储测试脚本、装置和 mock 对象。

模型、视图、控制器、语言和静态文件的访问是通过 web 管理[design]界面，ABOUT、README 和 errors 也可以通过相应的菜单项经由管理界面访问，Sessions、cache、modules 和 private 文件可以被应用程序访问的，但不能通过管理界面访问。

一切都整齐地在组织在一个明确的目录结构中，每个已安装的 web2py 应用都复制该结构，虽然用户永远不需要直接访问文件系统：

```
1. __init__.py  ABOUT      LICENSE  models   views
2. controllers modules    private  tests    cron
3. cache       errors     upload   sessions static
```

“__init__.py”是一个空文件，这是为了让 Python（和 web2py）导入 modules 目录中的模块。

注意，admin 应用只是为服务器文件系统上的 web2py 应用提供一个 web 接口，也可以从命令行创建和开发 web2py 应用，不必使用浏览器 admin 界面，复制上面的目录结构到例如“applications/newapp/”下（或者简单的将 welcome.w2p 文件解压到新的应用目录），可以手动创建一个新的应用程序，也可以不使用 web admin 界面而从命令行创建和编辑应用文件。

4.6 API

在已被我们导入以下对象的环境中执行模型、控制器和视图：

全局对象：

```
1 request, response, session, cache
```

国际化：

```
1 T
```

导航:

```
1 redirect, HTTP
```

帮助对象:

```
1 XML, URL, BEAUTIFY
2
3 A, B, BODY, BR, CENTER, CODE, COL, COLGROUP,
4 DIV, EM, EMBED, FIELDSET, FORM, H1, H2, H3, H4, H5, H6,
5 HEAD, HR, HTML, I, IFRAME, IMG, INPUT, LABEL, LEGEND,
6 LI, LINK, OL, UL, META, OBJECT, OPTION, P, PRE,
7 SCRIPT, OPTGROUP, SELECT, SPAN, STYLE,
8 TABLE, TAG, TD, TEXTAREA, TH, THEAD, TBODY, TFOOT,
9 TITLE, TR, TT, URL, XHTML, xmlescape, embed64
10
11 CAT, MARKMIN, MENU, ON
```

表单和表

```
SQLFORM (SQLFORM.factory, SQLFORM.grid, SQLFORM.smartgrid)
```

验证器:

```
1 CLEANUP, CRYPT, IS_ALPHANUMERIC, IS_DATE_IN_RANGE, IS_DATE,
2 IS_DATETIME_IN_RANGE, IS_DATETIME, IS_DECIMAL_IN_RANGE,
3 IS_EMAIL, IS_EMPTY_OR, IS_EXPR, IS_FLOAT_IN_RANGE, IS_IMAGE,
4 IS_IN_DB, IS_IN_SET, IS_INT_IN_RANGE, IS_IPV4, IS_LENGTH,
5 IS_LIST_OF, IS_LOWER, IS_MATCH, IS_EQUAL_TO, IS_NOT_EMPTY,
6 IS_NOT_IN_DB, IS_NULL_OR, IS_SLUG, IS_STRONG, IS_TIME,
7 IS_UPLOAD_FILENAME, IS_UPPER, IS_URL
```

数据库:

```
1. DAL, Field
```

为了向后兼容 SQLDB=DAL 和 SQLField=Field, 我们鼓励使用新的语法 DAL 和 Field, 而不是旧的语法。

我们的对象和模块定义在库中, 但不会自动导入它们因为并不经常使用。

web2py 执行环境中的核心 API 实体包括 request、response、session、cache、URL、HTTP、redirect 和 T, 并在下面讨论。

几个对象和函数包括 *Auth*、*Crud* 和 *Service* 被定义在“gluon/tools.py”中, 需要时它们需要被导入:

```
1. from gluon.tools import Auth, Crud, Service
```

4.6.1 从 Python 模块访问 API

你的模型或控制器可以导入 Python 模块, 而这些可能需要使用一些 web2py 的 API, 它们这样做的方式是导入:

```
from gluon import *
```


事实上，任何 Python 模块即使不由 web2py 应用导入，只要 web2py 在 sys.path 中，都可以导入 web2py 的 API。

但有一点需要注意，web2py 中定义了一些全局对象（请求 request、响应 response、会话 session、缓存 cache、T），当一个 HTTP 请求存在（或伪造）时，其才能存在。因此，只有当它们被应用调用时模块才可以访问它们，出于这个原因，它们被放置到一个叫做 current 的容器中，这是一个线程局部对象。例如，

创建一个模块“myapp/modules/test.py”，其中包含：

```
from gluon import *
def ip(): return current.request.client
```

现在，在“myapp”的控制器中可作如下操作：

```
import test
def index():
    return "Your ip is " + test.ip()
```

注意以下几点：

- import test 首先在当前应用的模块文件夹中寻找模块，之后在 sys.path 列出的文件夹中寻找。因此，应用级模块总是优先于 Python 模块，这允许不同应用附带它们不同版本的模块，而不会发生冲突。
- 不同的用户可以同时调用相同的 index 动作，调用模块中的函数，而且不会有任何冲突，因为 current.request 是处在不同线程中的不同对象，请慎重仔细，不要在模块的函数或类（即顶层）之外访问 current.request。
- import test 是从 applications.appname.modules 导入 test 的快捷方式，使用较长的语法，可以从其它应用导入模块。

为了与普通 Python 行为的一致性，默认情况下，进行更改时 web2py 不会重新加载模块。然而，这是可以改变的，要开启自动重载模块功能，如下使用 track_changes 函数（通常在一个模型文件中，在任何导入之前）：

```
1. from gluon.custom_import import track_changes; track_changes(True)
```

从现在起，每次导入一个模块，导入器都将检查 Python 源文件(.py)是否已经改变，如果改变了，模块将被重载，这适用于所有 Python 模块，甚至是 web2py 外的模块。该模式是全局性的，适用于所有应用，无论使用何种模式，总是重新加载模型、控制器和视图的更改。使用以 False 为参数的相同函数，关闭该模式，要了解实际的跟踪状态，使用 is_tracking_changes() 函数，其也来自于 gluon.custom_import。

导入 current 的模块可以访问：

- current.request
- current.response
- current.session
- current.cache
- current.T

以及应用选择存储在 current 中的其它任何的变量，例如一个模型可以做


```
auth = Auth(db)
from gluon import current
current.auth = auth
```

现在导入的所有模块都可以访问

- `current.auth`

`current` 和 `import` 建立了一个强大的机制，为你的应用构建可扩展的和可重用的模块。

存在一个主要的警告。考虑从 `gluon` 导入 `current`，使用 `current.request` 和其他任何线程本地对象都是正确的，但不应该将它们分配给模块中的全局变量，如

```
1 request = current.request # WRONG! DANGER!
```

也不能把它分配给类属性

```
1 class MyClass:
2     request = current.request # WRONG! DANGER!
```

这是因为线程本地对象必须在运行时提取，而全局变量只在模型第一次导入时定义一次。

4.7 request 请求对象

`request` 对象是一个无处不在的被称为 `gluon.storage` 的 `web2py` 类的实例，`Storage` 扩展 Python 的 `dict` 类，它基本上是一本字典，但项目的值也可以作为属性访问：

```
1 request.vars
```

和下面的是一样的：

```
1 request['vars']
```

与字典不同，如果属性（或 key）不存在，不会引发异常。相反，它会返回 `None`。

有时创建自己的 `Storage`（存储）对象是有用的，可以如下操作：

```
1 from gluon.storage import Storage
2 my_storage = Storage() # empty storage object
3 my_other_storage = Storage(dict(a=1, b=2)) # convert dictionary to Storage.
```

`request` 有如下条目/属性，其中有一些也是 `Storage` 类的实例：

- `request.cookies`: `Cookie.SimpleCookie()` 对象包含 HTTP 请求传递的 cookie，它就像一个 cookie 字典，每个 cookie 就是一个 `Morsel` 对象。
- `request.env`: `Storage` 对象包含传递到控制器的环境变量，包括来自 HTTP 请求的 HTTP 标头变量和 WSGI 参数，便于记忆，环境变量都转换成小写，点转换成下划线。
- `request.application`: 请求的应用名称（从 `request.env.path_info` 解析）。
- `request.controller`: 请求的控制器名称（从 `request.env.path_info` 解析）。
- `request.function`: 请求的函数名称（从 `request.env.path_info` 解析）。
- `request.extension`: 请求的动作扩展。默认是“html”，如果控制器函数返回一个字典且没有指定视图，这用来确定呈现字典的视图文件的扩展名（从 `request.env.path_info` 解析）。
- `request.folder`: 应用目录，例如如果应用是“welcome”，`request.folder` 被设置到绝对路径“/path/to/welcome”，在你的程序中，应该始终使用这个变量和 `os.path.join` 函数来构建需要访问的文件的路径，尽管 `web2py` 始终使用绝对路径，一个好的规则是从不明确改变当前工作文件夹（不管它是什

么)，因为这不是一个线程安全做法。

- `request.now`: 一个 `datetime.datetime` 对象存储当前请求的日期时间。
- `request.utcnow`: 一个 `datetime.datetime` 对象存储当前请求的 UTC 日期时间。
- `request.args`: 控制器函数名称之后的 URL 路径组件列表，等同于

`request.env.path_info.split('/')[3:]`

- `request.vars`: 一个 `gluon.storage.Storage` 对象包含 HTTP GET 和 HTTP POST 查询变量。
- `request.get_vars`: 一个 `gluon.storage.Storage` 对象只包含 HTTP GET 查询变量。
- `request.post_vars`: 一个 `gluon.storage.Storage` 对象只包含 HTTP POST 查询变量。
- `request.client`: 当客户端的 ip 地址被确定，如果存在的话，由

`request.env.http_x_forwarded_for` 确定或否则的话由 `request.env.remote_addr` 确定。虽然这是有用的，但不应该被信任，因为 `http_x_forwarded_for` 可以被欺骗。

- `request.is_local`: 客户端是 `localhost` 为真，否则为假，如果代理支持

`http_x_forwarded_for`，应该能通过代理来工作。

- `request.is_https`: 如果请求使用 HTTPS 协议，则为真，否则为假。
- `request.body`: 一个只读文件流，其中包含 HTTP 请求的主体，它被自动解析得到

`request.post_vars` 然后 `rewind`，并能被 `request.body.read()` 阅读。

- 如果通过 Ajax 请求调用函数，则 `request.ajax` 为真。
- `request.cid` 是 Ajax 请求（如有）生成的组件 id，在第 12 章可以阅读到有关组件的更多内容。
- `request.restful` 是一个新的和非常有用的装饰器 decorator，可用于分离

GET/POST/PUSH/DELETE 请求来改变 web2py 动作的默认行为，一些细节将在第 10 章讨论。

● `request.user_agent()` 从客户端解析 `user_agent` 字段并以字典的形式返回信息，检测移动设备是有用的，它使用 Ross Peoples 创建的“`gluon/contrib/user_agent_parser.py`”，要知道它做什么，尝试将下面的代码嵌入在一个视图中：

```
1 {{=BEAUTIFY(request.user_agent())}}
```

- `request.wsgi` 是一个 hook（钩子），允许从内部动作调用第三方 WSGI 应用。

后者包括：

- `request.wsgi.env`
- `request.wsgi.start_response`
- `request.wsgi.middleware`

它们的用法在本章结尾讨论。

作为一个例子，下面调用一个典型的系统：

```
http://127.0.0.1:8000/examples/default/status/x/y/z?p=1&q=2
```

产生下面的 `request`（请求）对象：

variable	value
request.application	examples
request.controller	default
request.function	index
request.extension	html
request.view	status
request.folder	applications/examples/
request.args	['x', 'y', 'z']
request.vars	<Storage {'p': 1, 'q': 2}>
request.get_vars	<Storage {'p': 1, 'q': 2}>
request.post_vars	<Storage {}>
request.is_local	False
request.is_https	False
request.ajax	False
request.cid	None
request.wsgi	hook
request.env.content_length	0
request.env.content_type	
request.env.http_accept	text/xml,text/html;
request.env.http_accept_encoding	gzip, deflate
request.env.http_accept_language	en
request.env.http_cookie	session_id_examples=127.0.0.1.119725
request.env.http_host	127.0.0.1:8000
request.env.http_max_forwards	10
request.env.http_referer	http://web2py.com/
request.env.http_user_agent	Mozilla/5.0
request.env.http_via	1.1 web2py.com
request.env.http_x_forwarded_for	76.224.34.5
request.env.http_x_forwarded_host	web2py.com
request.env.http_x_forwarded_server	127.0.0.1
request.env.path_info	/examples/simple_examples/status
request.env.query_string	remote_addr:127.0.0.1
request.env.request_method	GET
request.env.script_name	
request.env.server_name	127.0.0.1
request.env.server_port	8000
request.env.server_protocol	HTTP/1.1
request.env.web2py_path	/Users/mdipierro/web2py
request.env.web2py_version	Version 1.99.1
request.env.web2py_runtime_gae	(optional, defined only if GAE detected)
request.env.wsgi_errors	<open file, mode 'w' at >
request.env.wsgi_input	
request.env.wsgi_multipart	False
request.env.wsgi_multithread	True
request.env.wsgi_run_once	False
request.env.wsgi_url_scheme	http
request.env.wsgi_version	10

实际定义了哪些环境变量取决于 web 服务器。这里，我们假设内置了 Rocket wsgi 服务器，使用 Apache Web 服务器时，变量集没有太大的不同。

request.env.http_*变量从请求 HTTP 报头解析。

request.env.web2py_*变量不是从 web 服务器环境中解析的，而是在你的应用知道 web2py 的位置和版本，以及它是否运行在 Google App Engine 上的情况下（因为特定的优化可能是必要的），由 web2py 创建。

还需注意变量 request.env.wsgi_*, 对 wsgi 适配器来说，它们是被特定的。

4.8 response 响应对象

response 是 Storage 类的另一个实例。它包含以下：

response.body: 一个 StringIO 对象，web2py 向其中写入输出页面的主体，切勿更改此变量。

response.cookies: 类似于 request.cookies，但后者包含从客户端向服务器发送

cookie, 前者包含由服务器发送到客户端的 cookie, 会话 cookie 自动处理。

`response.download(request, db)`: 是一种方法, 该方法用于实现控制器功能, 其功能允许下载已上传文件, `request.download` 期望在 `request.args` 中的最后一个参数是编码文件名 (即, upload 上传时间产生的并存储在 upload 字段的文件名), 它从编码文件名中提取上传字段名、表名以及原始文件名, `response.download` 有两个可选参数: `chunk_size` 以字节为单位设置分块数据流的大小 (默认为 64k), `attachments` 确定下载的文件是否应被视为附件 (默认为 True)。注意, `response.download` 是专门用于与 db 上传字段相关的下载文件, 对于其它类型的文件下载和流媒体, 使用 `response.stream` (见下文)。另外, 需要注意, 没有必要使用 `response.download` 访问上传到 /static 文件夹的文件——静态文件 (通常应该) 可以通过 URL (例如, /app/static/files/myfile.pdf) 直接被访问。

`response.files`: 页面需要的 CSS 和 JS 文件的列表, 在标准 "layout.html" 的顶部, 它们将通过包含的 "web2py_ajax.html" 被自动链接, 要包括新的 CSS 和 JS 文件, 只需将其追加到该列表中, 它将处理重复, 顺序是重要的。

`response.include_files()` 生成 html 头标签来包含所有 `response.files` (用在 "views/web2py_ajax.html" 中)。

`response.flash`: 可选参数, 该参数也许被包含在视图中, 通常用于通知用户一些事发生了。

`response.headers`: 一个 HTTP 响应标头的 dict。默认情况下, web2py 设置一些标头, 包括 "Content-Length", "Content-Type" 和 "X-Powered-By" (设置等同于 web2py), Web2py 还会设置 "Cache-Control", "Expires" 和 "Pragma" 标头以防止客户端缓存, 除了静态文件请求, 会为它启用客户端缓存。web2py 设置的标头可以被覆盖或删除, 并且可以添加新的标头 (例如, `response.headers['Cache-Control'] = 'private'`)。

`response.menu`: 可选参数, 可以包含在视图中, 通常用来向视图传递一个导航菜单栏, 可以被 MENU 帮助对象呈现。

`response.meta`: 一个 storage 对象 (类似一个字典) 其中包含可选 meta 元信息如 `response.meta.author`、`.description` 和/或 `.keywords`, 每个 meta 元变量的内容自动放置在适当的 META 标签中, 通过 "views/web2py_ajax.html" 中的代码实现, 它默认包含在 "views/layout.html" 中。

`response.include_meta()` 生成一个字符串, 其中包含所有序列化的 `response.meta` 标头 (用在 "views/web2py_ajax.html" 中)。

`response.postprocessing`: 是一个函数列表, 默认为空, 在输出由视图呈现之前, 这些函数被用于过滤在一个动作的输出时的响应对象, 它可以用来实现支持其它模板语言。

`response.render(view, vars)`: 一种用于控制器内显式调用视图的方法, `view` 是一个可选的参数, 它是视图文件的名称, `vars` 是一个传递给视图的命名值的字典。

`response.session_file`: 包含会话的文件流。

`response.session_file_name`: 保存会话的文件的名称。

`response.session_id`: 当前会话的 id, 它被自动确定。切勿更改此变量。

`response.session_id_name`: 此应用会话 cookie 的名称。切勿更改此变量。

`response.status`: 传递给响应的 HTTP 状态代码整数, 默认是 200 (OK)。

`response.stream(file, chunk_size, request=request)`: 当控制器返回它, web2py 以 `chunk_size` 大小的块流传输文件内容返回给客户端, `request` 参数需要在 HTTP 标头中使用组块的开始。如上所述, 应该使用 `response.download` 检索通过上传字段存储的文件,

`response.stream` 可以使用在其它情况，如返回一个临时文件或由控制器创建的 `StringIO` 对象。与 `response.download` 不同，`response.stream` 不自动设置 `Content-Disposition` 标头，所以可能需要手工设置（例如，指定作为附件下载，并提供一个文件名）。但是，它会自动设置 `Content-Type` 标头（根据文件扩展名）。

`response.subtitle`: 可选的参数，该参数可以被包括在视图中，它应包含页面副标题。

`response.title`: 可选的参数，该参数可以被包括在视图中，它应包含页面标题并且应该由标头中的 HTML 标题 TAG 呈现。

`response.toolbar`: 一个函数，允许你嵌入一个工具栏到调试页的表格 `{{=response.toolbar()}}` 中，该工具栏显示每个查询的请求、响应、会话变量和数据库访问时间。

`response._vars`: 仅能在视图中访问此变量，而不能在动作中，它包含由动作返回给视图的值。

`response.optimize_css`: 如果可以设置为 `"concat,minify,inline"` 来连接、缩小和内联 web2py 包含的 CSS 文件。

`response.optimize_js`: 如果可以设置为 `"concat,minify,inline"` 来连接、缩小和内联 web2py 包含的 CSS 文件。

`response.view`: 呈现页面的视图模板的名称。默认设置为:

```
1 "%s/%s.%s" % (request.controller, request.function, request.extension)
```

或者，如果上述文件不能定位，设置为

```
1 "generic.%s" % (request.extension)
```

改变这个变量的值来修改与一个特定动作相关联的视图文件。

`response.xmlrpc(request, methods)`: 控制器返回时，此函数通过 XML-RPC `[xmlrpc]` 公开该方法。此函数被弃用了，是因为有一个更好的机制，将在第 10 章中描述。

`response.write(text)`: 一种将文本写入到输出页面主体的方法。

`response.js` 可以包含 Javascript 代码，该代码将被执行，当且仅当接收到的响应被 web2py 组件接收，在第 12 章中讨论。

由于 `response` 是一个 `gluon.storage.Storage` 对象，它可以用来存储你也许想传递给该视图其它属性，虽然没有技术的限制，我们的建议是只存储要被变整体布局中的所有页面呈现的变量（`"layout.html"`）。

无论如何，我们强烈建议坚持如下列出的变量:

```
1 response.title
2 response.subtitle
3 response.flash
4 response.menu
5 response.meta.author
6 response.meta.description
7 response.meta.keywords
8 response.meta.*
```

因为这会让你用其它布局文件替换 web2py 附带的标准 `"layout.html"` 文件变得更容易，它们使用相同变量集。

老版本的 web2py 使用 `response.author` 而不是 `response.meta.author`，其它 meta 属性也是相似的。

4.9 session 会话对象

session 是 Storage 类的另一个实例。无论什么被存储到 session 中，例如：

```
1 session.myvariable = "hello"
```

可以在稍后的时间被检索：

```
1 a = session.myvariable
```

只要代码由相同用户在同一会话中执行（倘若用户没有删除会话 cookie 并且会话未到期），因为 session 是 Storage 对象，试图访问尚未设置的属性/键不会引发异常，而是返回 None。

session 对象有 3 个重要方法。其中一个 is forget：

```
1 session.forget(response)
```

它告诉 web2py 不保存会话，这应该被用在控制器中，其动作经常被调用并且不需要跟踪用户活动，session.forget() 防止会话文件的写入，不管它是否被修改。另外，session.forget(response) 解锁和关闭会话文件，很少需要调用此方法，因为会话未被改变的时候不会被保存。但是，如果页面同时进行多个 Ajax 请求，对于通过 Ajax 调用 session.forget(response)（假设动作不需要该会话）这一动作来说，是一个好主意。否则，每个 Ajax 动作将不得不等待前一个完成（并解锁会话文件）然后再继续，这会减慢页面加载。请注意，存储在数据库中时会话不会被锁定。

另一个方法是：

```
1 session.secure()
```

它告诉 web2py 将会话 cookie 设置成一个安全的 cookie，如果应用程序通过 https，应该对此设置，通过设置会话 cookie 以确保其安全，服务器要求浏览器不发送会话 cookie 到服务器，除非通过 https 连接。

第三个方法是 connect：

```
1 session.connect(request, response, db, masterapp=None)
```

其中 db 是开放式数据库连接（如同由 DAL 返回的）名称，它告诉 web2py 要在数据库而不是在文件系统中存储会话，session.connect 必须跟在 db=DAL(...) 之后，但在任何其它要求会话的逻辑之前，例如设立 Auth。

web2py 创建一个表：

```
1 db.define_table('web2py_session',
2     Field('locked', 'boolean', default=False),
3     Field('client_ip'),
4     Field('created_datetime', 'datetime', default=now),
5     Field('modified_datetime', 'datetime'),
6     Field('unique_key'),
7     Field('session_data', 'text'))
```

并在 session_data 字段存储 cPickle 会话。

在运行 request.application 应用时，默认情况下，选项 masterapp=None 告诉 web2py 尝试检索已有会话。

如果想让两个或多个应用共享会话，将 masterapp 设置成主应用的名称。

可以在任何时候检查应用的状态，只需打印 request、session 和 response 系统变量。做到这一点的方法之一是创建一个专用的行动：

```
1 def status():
2     return dict(request=request, session=session, response=response)
```

4.9.1 单独的会话

如果你在文件系统中存储很多会话，文件系统访问可能会成为一个瓶颈。一种解决方案如下：

```
1 session.connect(request, response, separate=True)
```

通过设置 `separate=True`，web2py 不会在“sessions/”文件夹中存储会话，而是在“sessions/”文件夹的子文件夹中存储，并将自动创建子文件夹，具有相同前缀的会话将处在同一子文件夹中。同样，请注意，在任何可能需要该会话的逻辑之前，必须调用上述代码。

4.10 cache 缓存对象

cache 是一个全局对象，在 web2py 执行环境中也可用。它有两个属性：

- `cache.ram`：在主内存中的应用缓存。
- `cache.disk`：在磁盘上的应用缓存。

cache 是可调用的，这使得它被用来作为缓存动作和视图的装饰器 decorator。

下面的例子缓存 RAM 中的 `time.ctime()` 函数：

```
1 def cache_in_ram():
2 import time
3 t = cache.ram('time', lambda: time.ctime(), time_expire=5)
4 return dict(time=t, link=A('click me', _href=request.url))
```

`lambda: time.ctime()` 的输出在 RAM 中被缓存 5s，字符串 'time' 被用作缓存关键字。

下面的例子在磁盘上缓存 `time.ctime()` 函数：

```
1 def cache_on_disk():
2 import time
3 t = cache.disk('time', lambda: time.ctime(), time_expire=5)
4 return dict(time=t, link=A('click me', _href=request.url))
```

`lambda` 输出：`time.ctime()` 被缓存在磁盘上（使用 `shelve` 模块）持续 5s。

注意，`cache.ram` 和 `cache.disk` 的第 2 个参数必须是函数或可调用的对象，如果想缓存现有对象而不是函数的输出，只需通过一个 `lambda` 函数将它返回：

```
1 cache.ram('myobject', lambda: myobject, time_expire=60*60*24)
```

下一个例子缓存 `time.ctime()` 函数到 RAM 和磁盘：

```
1 def cache_in_ram_and_disk():
2 import time
3 t = cache.ram('time', lambda: cache.disk('time',
4 lambda: time.ctime(), time_expire=5),
5 time_expire=5)
6 return dict(time=t, link=A('click me', _href=request.url))
```

`lambda` 输出：`time.ctime()` 被缓存在磁盘上（使用 `shelve` 模型），之后缓存在 RAM，持续 5 秒，web2py 先在 RAM 中寻找，如果没有再到磁盘上寻找，如果它不在 RAM 中也不在磁盘上，`lambda: time.ctime()` 被执行并更新缓存，这种技术在多进程环境中是非常有用的。两次不必是相同的。

下面的例子是在 RAM 中缓存控制器函数的输出（但不是视图）：

```

1 @cache(request.env.path_info, time_expire=5, cache_model=cache.ram)
2 def cache_controller_in_ram():
3     import time
4     t = time.ctime()
5     return dict(time=t, link=A('click me', _href=request.url))

```

cache_controller_in_ram 返回的字典被缓存在 RAM 中，持续 5 秒。注意，数据库选择的结果必须先序列化后再被缓存，更好的方法是直接使用 select 方法的 cache 参数缓存数据库选择。

下面的例子是在磁盘上缓存控制器函数的输出（但不是视图）：

```

1 @cache(request.env.path_info, time_expire=5, cache_model=cache.disk)
2 def cache_controller_on_disk():
3     import time
4     t = time.ctime()
5     return dict(time=t, link=A('click to reload',
6 _href=request.url))

```

cache_controller_on_disk 返回的字典缓存在磁盘上，持续 5 秒。牢记，web2py 不能缓存包含 un-pickle 对象的字典。

也可以缓存视图，关键是要在控制器函数中呈现视图，使控制器返回一个字符串。这是通过返回 response.render(d) 完成的，其中 d 是我们打算传递给视图的字典，下面的例子是 RAM 中缓存控制器函数的输出（包括呈现的视图）：

```

1 @cache(request.env.path_info, time_expire=5, cache_model=cache.ram)
2 def cache_controller_and_view():
3     import time
4     t = time.ctime()
5     d = dict(time=t, link=A('click to reload', _href=request.url))
6     return response.render(d)

```

response.render(d) 作为一个字符串返回呈现的视图，现在它已经被缓存了 5 秒，这是最好的也最快的缓存方法。

注意，time_expire 用来比较当前时间和所请求对象在 cache 中上一次保存的时间，它不影响未来的请求。这允许在请求对象时动态设置 time_expire，而不是在保存对象时固定它。例如：

```

1 message = cache.ram('message', lambda: 'Hello', time_expire=5)

```

现在，假设如下调用在上述调用 10 秒之后出现：

```

1 message = cache.ram('message', lambda: 'Goodbye', time_expire=20)

```

因为在第 2 个调用中 time_expire 设置成 20 秒，并且距消息第一次被保存只过去了 10 秒，值“Hello”将从缓存中恢复，它不会被更新成“Goodbye”，第一个调用中 5 秒的 time_expire 不会影响第 2 个调用。

设置 time_expire=0（或者一个负数）强制更新缓存项目（因为距上次保存所经过的时间总是>0），设置 time_expire=None 强制恢复缓存值，不管距上次保存所经过的时间（如果 time_expire 始终是 None，缓存项目将有效地永不过期）。

如下，你可以清除一个或多个缓存变量


```
1 cache.ram.clear(regex='...')
```

这里 regex 是一个正则表达式匹配你想从缓存中移除的所有键。如下，还可以清除单个项目：

```
1 cache.ram(key, None)
```

这里 key 是缓存项目的键。

也可以定义其它缓存机制例如 memcache。Memcache 通过 gluon.contrib.memcache 是可用的，并会在第 14 章详细讨论。

缓存时要小心，缓存通常是在应用层而不是用户层。如果需要，例如缓存特定用户内容，选择一个包含用户 id 的键。

4.11 URL 统一资源定位器

URL 函数是 web2py 中最重要的函数之一。它为动作和静态文件生成内部 URL 路径。

这里是一个例子：

```
1 URL('f')
```

它被映射成

```
1 /[application]/[controller]/f
```

注意，URL 函数的输出取决于当前应用的名称、调用控制器、以及其它参数，web2py 支持 URL 映射及 URL 逆映射，URL 映射允许重定义外部 URL 的格式，如果使用 URL 函数来生成所有的内部 URL，对 URL 映射的增加或改变将防止破坏 web2py 应用中的链接。

可以传递附加参数到 URL 函数，即 URL 路径中的附加项 (args) 和 URL 查询变量 (vars)：

```
1 URL('f', args=['x', 'y'], vars=dict(z='t'))
```

被映射成

```
1 /[application]/[controller]/f/x/y?z=t
```

args 属性被自动解析、解码并最终由 web2py 存储在 request.args 中。类似的，vars 被解析、解码之后存储在 request.vars 中，并且 vars 提供基本机制，web2py 通过该机制与客户端浏览器交换信息。

如果 args 仅包含一个变量，没有必要在列表中传递它。

还可以用 URL 生成到其它控制器和其它应用中的动作的 URL：

```
1 URL('a', 'c', 'f', args=['x', 'y'], vars=dict(z='t'))
```

被映射成：

```
1 /a/c/f/x/y?z=t
```

另外，也可以使用命名参数指定应用、控制器和函数：

```
1 URL(a='a', c='c', f='f')
```

如果应用名 a 找不到，假定为当前应用。

```
1 URL('c', 'f')
```

如果控制器名找不到，假定是当前的。

```
1 URL('f')
```

也可以传递函数本身而不是传递控制器函数的名称。

```
1 URL(f)
```

基于上述原因，你应该总是使用的 URL 函数来为你的应用程序生成静态文件的 URL，静态文件存储在应用的 static 子文件夹中（这就是使用管理界面上传后它们去的地方），web2py 提供一个虚拟的 'static' 控制器，它的工作就是从 static 子文件夹中检索文件，确定它们的内容类型，并将文件流传输到客户端。下面的例子为静态文件 "image.png" 生成 URL：

```
1 URL('static', 'image.png')
```

被映射成

```
1 /[application]/static/image.png
```

如果静态文件位于 static 文件夹的子文件夹中，作为文件名的一部分，可以包含子文件夹。例如为了生成：

```
1/[application]/static/images/icons/arrow.png
```

可以使用：

```
1 URL('static', 'images/icons/arrow.png')
```

不需要编码/转义的 args 和 vars 参数，这将自动完成。

默认情况下，对于当前请求的相应扩展（可以在 request.extension 中找到）被附加到函数，除非 request.extension 默认为 html。这可以重写，通过明确地包含一个扩展，作为函数名 URL(f='name.ext') 的一部分或带有扩展参数：

```
1 URL(..., extension='css')
```

可以明确地抑制当前扩展：

```
1 URL(..., extension=False)
```

4.11.1 绝对 url

默认情况下，URL 生成相对的 URL。但是，也可以生成绝对 URL，通过指定 scheme 和 host 参数（这是有用的，例如在 email 中插入 URL 时）：

```
1 URL(..., scheme='http', host='www.mysite.com')
```

可以自动包含当前请求的方案和主机只需将参数设为 TRUE。

```
1 URL(..., scheme=True, host=True)
```

如果需要的话，URL 函数还接受一个 port 参数来指定服务器端口。

4.11.2 数字签名的 url

当生成 URL，可以选择数字签名它，这将附加一个能被服务器确认的 _signature GET 变量，有两种实现方法。

可以将以下参数传递给 URL 函数：

- `hmac_key`: 签署 URL 的密钥（一个字符串）
- `salt`: 一个可选字符串在签署之前 `salt` 加密数据
- `hash_vars`: 来自 URL 查询字符串的变量名可选列表（即 GET 变量）要包含在签名中，也可以将它设置成 `TURE`，以包含全部变量；或 `FALSE`，以不包含任何变量。

下面是一个用法的例子：

```
1 KEY = 'mykey'
2
3 def one():
4     return dict(link=URL('two', vars=dict(a=123), hmac_key=KEY))
5
6 def two():
7     if not URL.verify(hmac_key=KEY): raise HTTP(403)
8     # do something
9     return locals()
```

这使得只能通过数字签名的 URL 才能访问 action two，数字签名 URL 看起来是这样的：

```
'/welcome/default/two?a=123&_signature=4981bc70e13866bb60e52a09073560ae822224e9'
```

注意，数字签名通过 `URL.verify` 函数验证，`URL.verify` 还需要上面描述的 `hmac_key`、`salt` 和 `hash_vars` 参数，并且当数字签名被创建以验证 URL 时，它们的值必须匹配传递给 URL 函数的值。

第二个更为复杂但更常见的使用数字签名的例子是与 Auth 相关联，最好通过例子来解释：

```
1 @auth.requires_login()
2 def one():
3     return dict(link=URL('two', vars=dict(a=123), user_signature=True))
4
5 @auth.requires_signature()
6 def two():
7     # do something
8     return locals()
```

在这种情况下 `hmac_key` 自动生成并在该会话中共享，这使得行动 2 代表行动 1 任何访问控制，如果该链接被生产并签名，它是有效的；否则它无效。如果该链接是被另一个用户盗取，链接将是无效的。

这是很好的做法，总是数字签名 Ajax 回调，如果使用 web2py 负载功能 `LOAD` 函数，它有一个 `user_signature` 参数也可用于这一目的。

```
{{LOAD('default', 'two', vars=dict(a=123), ajax=True, user_signature=True)}}
```

4.12 HTTP 和重定向 redirect

web2py 只定义了一个新异常叫做 `HTTP`，该异常可以使用如下命令在任何一个模型、控制器或视图中引发：

```
1 raise HTTP(400, "my message")
```

它导致控制流从用户代码跳出，返回到 web2py，并返回 HTTP 响应如下：

```
1 HTTP/1.1 400 BAD REQUEST
2 Date: Sat, 05 Jul 2008 19:36:22 GMT
3 Server: Rocket WSGI Server
```

```
4 Content-Type: text/html
5 Via: 1.1 127.0.0.1:8000
6 Connection: close
7 Transfer-Encoding: chunked
8
9 my message
```

第一个 HTTP 参数是 HTTP 的状态代码，第二个参数是字符串，返回响应的主体，附加可选命名的参数用来构建响应 HTTP 的报头。例如：

```
1 raise HTTP(400, 'my message', test='hello')
```

生成：

```
1 HTTP/1.1 400 BAD REQUEST
2 Date: Sat, 05 Jul 2008 19:36:22 GMT
3 Server: Rocket WSGI Server
4 Content-Type: text/html
5 Via: 1.1 127.0.0.1:8000
6 Connection: close
7 Transfer-Encoding: chunked
8 test: hello
9
10 my message
```

如果不想提交开放数据库的事务，在引发异常前回滚。

任何非 HTTP 导致的异常，web2py 回滚到公开数据库事务，日志错误记录，发布票据给访问者，返回到标准错误页面。

这意味着只有 HTTP 能用于交叉页控制流，其它异常必须被应用捕获，否则会被 web2py 发票据。

命令：

```
1 redirect('http://www.web2py.com')
```

是下面的快捷方式：

```
1 raise HTTP(303,
2 'You are being redirected <a href="%s">here</a>' % location,
3 Location='http://www.web2py.com')
```

HTTP 初始对象方法的命名参数直接翻译为 HTTP 报头指令，这种情况，重定向目标位置。Redirect 用可选的第二参数，是重定向的 HTTP 状态代码（默认 303）。暂时重定向修改该数字为 307 或永久重定向修改为 301。

最常见使用重定向的方法是重定向到同个应用的其它页面或（可选）传递参数：

```
1 redirect(URL('index', args=(1,2,3), vars=dict(a='b')))
```

4.13 T 和 国际化

对象 T 是语言翻译对象，它构成 web2py 类 gluon.language.translator 单个全局实例，所有字符串常量（仅字符串常量）用 T 标识，例如：

```
1 a = T("hello world")
```

用 T 标注的字符串被 web2py 识别成需要语言翻译，当执行代码（在模型、控制器或视图中）时它们将被翻译，如果需要翻译的字符串不是常量而是变量，运行时它将被加入翻译文件（除非在 GAE 上）此后被翻译。

T 对象还能包含插值变量并支持多重等价语法：

```
1 a = T("hello %s", ('Tim',))
2 a = T("hello %(name)s", dict(name='Tim'))
3 a = T("hello %s") % ('Tim',)
4 a = T("hello %(name)s") % dict(name='Tim')
```

后者的语法被推荐，因为它使翻译更容易，第一个字符串是根据请求的语言翻译文件，名称 name 变量替换独立于语言。

连接翻译的字符串和普通字符串的翻译是可能的：

```
1 T("blah ") + name + T(" blah") # invalid!
```

但相反的，没有：

```
1 name + T(" blah") # invalid!
```

下面的代码也可以，通常最好：

```
1 T("blah %(name)s blah", dict(name='Tim'))
```

或替代语法

```
1 T("blah %(name)s blah") % dict(name='Tim')
```

在两种情况下，变量名在 “%(name)s” slot 中被取代之前翻译发生。下面的替换不应该被使用：

```
1 T("blah %(name)s blah" % dict(name='Tim'))
```

因为翻译后发生替代。

要求的语言，是由 “Accept-Language” 字段 HTTP 中的头，但这一选择可以覆盖，通过编程请求一个特定的文件覆盖，例如：

```
1 T.force('it-it')
```

读取 “languages/it-it.py” 语言文件，语言文件可以通过管理界面创建和编辑。

也可以强制用 per-string 语言：

```
1 T("Hello World", language="it-it")
```

可以通过如下命令完全关闭翻译

```
1 T.force(None)
```

通常，当视图呈现时字符串翻译懒散地被评估；因此，翻译对象 force 方法不该在视图中调用。

禁用 lazy 评估通过

```
1 T.lazy = False
```

这样，字符串立即被基于当前接受或强制语言的 T 运算符翻译。

也能对个别字符串禁用 lazy 评估：

```
1 T("Hello World", lazy=False)
```

一个常见的问题是：最初的应用是在英国，假设有一个翻译文件（例如意大利语，“it-it.py”）和客户端声明以那种顺序，它接受英文（en）和意大利（it-it），以下意外情况发生：web2py 不知道默认是写在英语（en）。因此，它倾向于将什么都翻译成意大利语（it-it），因为它只发现意大利语翻译文件。如果它没有找到“it-it.py”的文件，它将使用默认语言（en）。

解决这个问题有两种方案：为英文创建一个语言翻译，这将是多余的和不必要的，或更好，告诉 web2py 语言应该使用默认语言的字符串（字符串编码到应用程序）。这是可以做到：

```
1 T.set_current_languages('en', 'en-en')
```

它存储在 t.current_languages 语言列表，不需要翻译和强迫语言文件重载。

注意，“it”和“it-it”是不同的 web2py 角度的语言。为了支持他们两个，需要两个总是小写的翻译文件，这同样适用于所有其他语言。

当前接受的语言是储存在

```
1 T.accepted_language
```

注意 T(...) 不仅能翻译字符串也能翻译变量：

```
1 >>> a="test"
2 >>> print T(a)
```

这种情况下，词“test”被翻译，但如果没有找到并且如果文件系统是可写的，将添加它到语言文件中被翻译的词列表中。

4.14 Cookies

web2py 使用 Python cookie 模块处理 cookie。

来自浏览器的 cookie 在 request.cookies 中，由服务器发送的 cookie 在 response.cookies 中。

可以如下设置 cookie：

```
1 response.cookies['mycookie'] = 'somevalue'
2 response.cookies['mycookie']['expires'] = 24 * 3600
3 response.cookies['mycookie']['path'] = '/'
```

第二行告诉浏览器保留 cookie 24 小时，第三行告诉浏览器发送 cookie 回到当前域名的任何应用（URL 路径）。

cookie 能确保安全性：

```
1 response.cookies['mycookie']['secure'] = True
```

这告诉浏览器只通过 HTTPS 发回 cookie 而不是通过 HTTP。

cookie 能被如下得到：

```
1 if request.cookies.has_key('mycookie'):
```

```
2 value = request.cookies['mycookie'].value
```

除非会话被禁用，在底层的 web2py，设置下列 cookie 并使用它处理会话：

```
1 response.cookies[response.session_id_name] = response.session_id
2 response.cookies[response.session_id_name]['path'] = "/"
```

注意，如果单个应用包含多个子域名，而且想在那些子域名间（例如，sub1.yourdomain.com, sub2.yourdomain.com 等）共享会话，必须显式设置会话 cookie 域名如下：

```
1 if not request.env.remote_addr in ['127.0.0.1', 'localhost']:
2 response.cookies[response.session_id_name]['domain'] = ".yourdomain.com"
```

上述非常有用，例如想允许用户在各子域名间保持登录。

4.15 应用程序 *init*

当部署 web2py 时，会希望设置一个默认应用，即 URL 中存在空路径时启动的应用，如同在：

```
1 http://127.0.0.1:8000
```

按照默认，遇到空路径时，web2py 寻找名为 *init* 的应用，如果没有 *init* 应用，寻找名为 *welcome* 的应用。

通过设置 routes.py 中的 default_application，默认应用的名称可以从 *init* 改成其它名称。

```
1 default_application = "myapp"
```

注意：default_application 最早出现于 web2py 版本 1.83。

以下是设置默认应用的四种方法：

- 调用默认应用“init”
- 在 routes.py 中将默认应用设为你的应用名称
- 创建一个从“applications/init”到你的应用文件夹的符号链接
- 使用下一部分讨论的 URL 重写

4.16 URL 重写

web2py 有能力在调用控制器 action（URL 映射）之前重写进来请求的 URL 路径，相反，web2py 可以重写由 URL 函数生成的 URL 路径（反 URL 映射），这么做的一个原因是为了处理 URL 的遗留数据，另一个是为了简化路径使之变短。web2py 包含两个不同的 URL 重写系统：简单易用的 *parameter-based* 系统适用于大多数使用场合，灵活的 *pattern-based* 系统适用于更复杂的场合。为了简化 URL 重写规则，在“web2py”文件夹中创建一个名为 routes.py（routes.py 的内容取决于你选用两种重写系统中的哪一种，正如下面两部分介绍的）的新文件，两个系统不能混用。

注意，如果编辑 routes.py，必须重载它。可以用两种方法完成：通过重启 web 服务器或通过点击 admin 中的路径重载按钮，如果路径中有错误，它们不会被重载。

4.16.1 基于参数的系统

基于参数的（参数的）路由器提供了若干“canned”重写 URL 的便捷访问方法，它的能力（capacities）包括：

- ★ 忽略从外部可见 URL 的默认应用、控制器和函数名（由 URL（）函数创建）
- ★ 将域（和/或端口）映射到应用或控制器
- ★ 在 URL 中嵌入语言选择器
- ★ 移除输入的 URLs 中的固定前缀并将它加入到输出的 URLs 中。
- ★ 将根文件如/robots.txt 映射到应用静态目录

参数路由还提供更灵活的输入 URL 验证。

假设编写了一个叫做 myapp 的应用并希望将它设为默认，这样应用名就不是用户看到的 URL 中的一部分了。你的默认控制器仍然是 default，也想从用户可见的 URL 中删除其名字。下面是写入 routes.py 的代码：

```
1 routers = dict(  
2 BASE = dict(default_application='myapp'),  
3 )
```

就这些，参数路由足够智能，知道如何处理这样的 URL：

```
1 http://domain.com/myapp/default/myapp
```

或者

```
1 http://domain.com/myapp/myapp/index
```

这里普通的缩减会含义模糊。如果有两个应用 myapp 和 myapp2，会得到相同的效果，此外当处于安全的时候（几乎所有时候），myapp2 的默认控制器会从 URL 中抽取出来。

下面是另一种情况：假设你想支持基于 URLs 的语言，这里 URLs 看起来如下：

```
1 http://myapp/en/some/path
```

或（重写）

```
1 http://en/some/path
```

具体方法如下：

```
1 routers = dict(  
2 BASE = dict(default_application='myapp'),  
3 myapp = dict(languages=['en', 'it', 'jp'], default_language='en'),  
4 )
```

传入的 URL 是这样的：

```
1 http://domain.com/it/some/path
```

将被路由到/myapp/some/path，并且 request.uri_language 将被设置为'it'，这样就可以强制翻译，也可以使用特定语言的静态文件。

```
1 http://domain.com/it/static/filename
```

将被映射到：

```
1 applications/myapp/static/it/filename
```

如果该文件存在，如果不存在，那么 URL 如下：

```
1 http://domain.com/it/static/base.css
```

仍然会映射到：


```
1 applications/myapp/static/base.css
```

（因为没有 `static/it/base.css`）。

因此，现在可以有特定语言的静态文件，包括图片，如果需要的话，还支持域映射：

```
1 routers = dict(  
2     BASE = dict(  
3         domains = {  
4             'domain1.com' : 'app1',  
5             'domain2.com' : 'app2',  
6         }  
7     ),  
8 )
```

做你想做的：

```
1 routers = dict(  
2     BASE = dict(  
3         domains = {  
4             'domain.com:80' : 'app/insecure',  
5             'domain.com:443' : 'app/secure',  
6         }  
7     ),  
8 )
```

映射 `http://domain.com` 访问到命名为 `insecure` 的控制器，而 `HTTPS` 访问映射到 `secure` 控制器。或者，也可以以明显的方式将不同的端口映射到不同的应用。

欲了解更多信息，请咨询标准 `web2py` 发行版的基础文件夹中提供的 `router.example.py` 文件。

注：基于参数的系统首先出现在 `web2py` 版本 1.92.1 中。

4.16.2 基于模式的系统

尽管刚刚描述的 *parameter-based* 系统对于绝大多数应用情况来说已经足够了，可替换的 *pattern-based* 系统提供了一些对更复杂的情况来说额外的灵活性，为了使用 *pattern-based* 系统，而不是定义为作为路由参数字典的路由器，你定义了两元组 `routes_in` 和 `routes_out` 的两个列表（或元组）。每个元组包含两个元素：将要替换的模式和替换它的字符串。例如：

```
1 routes_in = (  
2     '/testme', '/examples/default/index'),  
3 )  
4 routes_out = (  
5     '/examples/default/index', '/testme'),  
6 )
```

有了这些路由，URL 如下：

```
1 http://127.0.0.1:8000/testme
```

被映射到：

```
1 http://127.0.0.1:8000/examples/default/index
```

对于访问者，所有到该页面 URL 的链接看起来像 / testme。

该模式与 Python 正则表达式有相同的语法。例如：

```
1 ('.*\.php', '/init/default/index'),
```

将所有以“.php”结尾的 URL 映射到索引页。

有时想摆脱来自应用的 URL 前缀，因为只计划公开一个应用，可以如下实现：

```
1 routes_in = (  
2 ('/(?P<any>.*)', '/init/\g<any>'),  
3 )  
4 routes_out = (  
5 ('/init/(?P<any>.*)', '/\g<any>'),  
6 )
```

还有另一种可取的语法，可以与上述正则表达式符号混合使用。它包括使用\$name 代替 (?P<name>\w+) 或 \g<name>。例如：

```
1 routes_in = (  
2 ('/$c/$f', '/init/$c/$f'),  
3 )  
4  
5 routes_out = (  
6 ('/init/$c/$f', '/$c/$f'),  
7 )
```

也消除了所有 URL 中的“/example”应用前缀。

使用\$ name 符号，且只要不使用任何正则表达式，可以自动映射 routes_in 到 routes_out，例如：

```
1 routes_in = (  
2 ('/$c/$f', '/init/$c/$f'),  
3 )  
4  
5 routes_out = [(x, y) for (y, x) in routes_in]
```

如果有多个路由，第一个匹配 URL 被执行，如果没有模式匹配，路径保持不变。

可以使用\$anything 匹配 anything (.*)，直到该行结束。

这里是一个最小的“routes.py”，其被用于处理图标和机器人的请求：

```
1 routes_in = (  
2 ('/favicon.ico', '/examples/static/favicon.ico'),  
3 ('/robots.txt', '/examples/static/robots.txt'),  
4 )  
5 routes_out = ()
```

下面是一个更复杂的例子，没有不必要的前缀而暴露了一个单一的应用“myapp”，也暴露了 admin、appadmin 和静态：

```
1 routes_in = (  
2 ('/admin/$anything', '/admin/$anything'),  
3 ('/static/$anything', '/myapp/static/$anything'),  
4 ('/appadmin/$anything', '/myapp/appadmin/$anything'),  
5 ('/favicon.ico', '/myapp/static/favicon.ico'),
```

```
6 ('/robots.txt', '/myapp/static/robots.txt'),
7 )
8 routes_out = [(x, y) for (y, x) in routes_in[:-2]]
```

路由的通用语法比我们迄今所看到的简单例子要复杂得多，这里是一个更普遍的有代表性的例子：

```
1 routes_in = (
2 ('140\191\.\d+\.\d+:https://www.web2py.com:POST /(?P<any>.*).\php',
3 '/test/default/index?vars=\g<any>'),
4 )
```

它把来自匹配正则表达式的远程 IP 的 https POST 请求映射到主机 www.web2py.com

```
1 '140\191\.\d+\.\d+'
```

请求一个匹配正则表达式的页面

```
1 '/(?P<any>.*).\php'
```

到

```
1 '/test/default/index?vars=\g<any>'
```

这里 \g<any> 被匹配正则表达式取代。

通用语法是

```
1 '[remote address]:[protocol]://[host]:[method] [path]'
```

整个表达被作为正则表达式匹配，因此 "." 应始终被转义并且根据 Python 正则表达式语法可以使用 (?P<...>...) 捕获任何匹配的子表达式。

这允许重新请求那些基于客户端 IP 地址或域名，基于请求的类型、方法和路径的路由请求，也允许 web2py 将不同的虚拟主机映射到不同的应用，任何匹配的子表达式可以用来建立目标 URL，最后作为 GET 变量传递。

所有主要 Web 服务器，如 Apache 和 lighttpd 也有能力重写 URL。在生产环境中，这可以是取代 routes.py 的一种选择，无论决定做什么，我们强烈建议不要硬编码应用内部的 URL，并使用的 URL 函数来生成它们。如果路由改变，这将使你的应用有更好的可移植性。

Application-Specific URL rewrite

应用特定的 URL 重写

当使用基于模式的系统时，在位于应用基础文件夹的特定于应用的 routes.py 文件中，应用可以设置自己的路径，通过设置在基础 routers.py 中的 routes_app 启动此功能，根据传入的 URL 决定被选中的应用名称。当这发生时，特定于应用的 routes.py 被用来取代基础的 routes.py。

routes_app 的格式与 routes_in 相同，不同的是替换模式是简单的应用名称，如果将 routes_app 应用于传入的 URL 而没有得到应用名称，或产生的特定于应用的 routes.py 没有找到，该基础 routes.py 照常使用。

注：routes_app 首次出现在 web2py 版本 1.83 中。

Default application, controller, and function 默认应用，控制器和函数

使用基于模式的系统时，默认应用、控制器和函数的名称可以通过在 `routes.py` 中设置适当的值，分别从 `init`、`default` 和 `index` 改为其它名称。

```
1 default_application = "myapp"
2 default_controller = "admin"
3 default_function = "start"
```

注：这些项目最早出现在 web2py 版本 1.83 中。

4.17 出错路由 *Routes on Error*

如果服务器上有错误，还可以使用 `routes.py` 重新将请求路由到特殊动作，在全局范围内，可以为每个应用、每个错误代码或者为每个应用和错误代码指定这个映射。下面是一个例子：

```
1 routes_onerror = [
2 ('init/400', '/init/default/login'),
3 ('init/*', '/init/static/fail.html'),
4 ('*/404', '/init/static/cantfind.html'),
5 ('*/*', '/init/error/index')
6 ]
```

对于每一个元组，第一个字符串与 "[app name]/[error code]" 相匹配，如果找到匹配，失败的请求被重新路由到匹配元组的第二个字符串中的 URL，如果错误处理 URL 不是静态文件，下面的 GET 变量将被传递到错误动作：

- code: HTTP 状态代码（例如，404、500）
- ticket: 形式为 "[app name]/[ticket number]"（或 "None" 如果没有票据）
- requested_uri: 等同于 `request.env.request_uri`
- request_url: 等同于 `request.url`

错误处理动作可以通过 `request.vars` 访问这些变量，这些变量还能用于生成错误响应，尤其是，错误动作返回原始的 HTTP 错误代码而不是默认的 200 (OK) 状态码是一个好主意，这可以通过设置 `response.status = request.vars.code` 来完成，也有可能让错误动作（或队列）发送电子邮件给管理员，包括到 admin 中票据的链接。

不匹配的错误的显示默认错误页面，默认错误页面也可以在这里定制（参见 web2py 根文件夹中的 `router.example.py` 和 `routes.example.py`）：

```
1 error_message = '<html><body><h1>%s</h1></body></html>'
2 error_message_ticket = '''<html><body><h1>Internal error</h1>
3 Ticket issued: <a href="/admin/default/ticket/%(ticket)s"
4 target="_blank">%(ticket)s</a></body></html>'''
```

当一个无效应用或函数被请求时，第一个变量包含错误消息，当票据被发出时，第二个变量也包含错误消息。

`routes_onerror` 可用于两种路由机制。

4.18 后台运行任务

在 web2py 中，每一个 HTTP 请求在它自己的线程中被服务，线程被回收以提高效率并且线程由 web 服务器管理。出于安全考虑，Web 服务器对每个请求设置超时时间，这意味着，动作不应该执行耗时太长的任务，不应该创建新的线程，不应该派生进程（可以这么做但不推荐）。

执行耗时的任务的正确方法是后台执行，没有一种完成它的方法，但在这里我们将介绍 web2py 内置的三种机制：cron、*homemade task queues* 和 *scheduler*。

我们所说的 *cron* 是 web2py 中的一个不专属于 UNIX *cron* 机制的功能，web2py *cron* 也适用于 windows，web2py *cron* 是应该选择的方法，如果在后台你需要定时任务并且这些任务需要的时间与两个调用之间的时间间隔相对较短。每个任务都在自己的进程中运行，可以同时运行多个任务，但不能控制有多少任务运行，如果不小心一个任务与自身重叠，可能会导致数据库锁定和内存使用量激增。web2py 的调度采用不同的方法，正在运行的进程数目是固定的，并且它们可以在不同的机器上运行。每个进程被称为一个工人，每个工人取一个可用任务并在预定时间之后尽快执行该任务，但不一定是在确切的时间。正在运行的进程数不能多于预定任务数，这样不会有内存使用激增，调度程序的任务可以被定义在模型中并且存储在数据库中，web2py 的调度没有实现分布式队列，因为它假定分配任务的时间与运行任务的时间相比是微不足道的，工人从数据库中取任务。

在某些情况下，自制任务队列可以是 web2py 调度的一个简单替代。

4.18.1 Cron

web2py 的 *cron* 以与平台无关的方式为应用提供在预设时间执行任务的能力。

对于每个应用，*cron* 功能由 *crontab* 文件定义：

```
1 app/cron/crontab
```

这遵循 ref. [\[cron\]](#) 中定义的语法（有一些特定于 web2py 的扩展）。

这意味着，每个应用可以有一个单独的 *cron* 配置并且可以在 web2py 中改变 *cron* 配置而不影响主机操作系统本身。

这里是一个例子：

```
1 0-59/1 * * * * root python /path/to/python/script.py
2 30 3 * * * root *applications/admin/cron/db_vacuum.py
3 */30 * * * * root **applications/admin/cron/something.py
4 @reboot root *mycontroller/myfunction
5 @hourly root *applications/admin/cron/expire_sessions.py
```

这个例子中的最后两行使用扩展规范 *cron* 语法以提供额外的 web2py 功能。

文件“*applications/admin/cron/expire_sessions.py*”实际存在并附带 *admin* 应用，它检查过期会话并删除它们，“*applications/admin/cron/crontab*”按小时执行此任务。

如果任务/脚本名字前有一个星号(*)和以.py结束,它就会在web2py的环境中被执行,这意味着您可以任意使用这些所有的控制器和模型,如果使用两个星号(**),该MODEL将不被执行。这是推荐的调用方式,因为它具有更少的开销,并避免了潜在的锁定问题。

注意,在web2py环境中执行的脚本/函数在函数结尾需要一个手动的db.commit()否则交易将被还原。在shell模式下,web2py不会生成票据或有意义的回溯,cron在该模式下运行,所以在将其设为cron任务之前一定要确保你的web2py代码运行没有错误,因为从cron运行时你将不能看到那些错误。此外,如何使用模型要小心:虽然执行发生在一个单独的进程中,为了避免页面等待可能锁存数据库的cron任务,必须考虑数据库锁存。如果在cron任务中,不需要使用数据库,使用**语法。

也可以调用控制器函数,那样的情况不需要指定路径,控制器和函数将是调用应用的。特别在意上面列出的注意事项。示例:

```
1 */30 * * * * root *mycontroller/myfunction
```

如果在crontab文件中的第一个字段指定@reboot,给定的任务将只在web2py启动时执行一次,可以使用此功能,如果要在web2py启动时预缓存、检查或者初始化应用数据。需要注意cron任务是与应用并行执行的——如果直到cron任务完成后,应用没有准备好服务请求,应该执行检查以反映这一点。示例:

```
1 @reboot * * * * root *mycontroller/myfunction
```

根据你如何调用web2py,web2py cron有四种操作模式。

- *soft cron*:所有执行模式下都可用
- *hard cron*:如果使用内置web服务器(直接使用或通过Apache mod_proxy)则可用
- *external cron*:如果你能访问系统自己的cron服务则可用
- 无cron

如果使用的是内置web服务器,默认是hard cron,在所有其他情况下,默认是soft cron;如果使用的是CGI、FastCGI或WSGI,soft cron是默认方法(但要注意,web2py提供的标准wsgihandler.py文件中默认不启用soft cron)。

在crontab指定的时间之后,你的任务将在首次调用(页面加载)给web2py上被执行;但仅在页面处理之后,所以用户不会观察到延时。显然,关于任务具体什么时候执行存在不确定性,取决于站点接收的流量,而且如果web服务器有页面加载时间设置cron任务可能被打断,如果这些限制不能接受,参见*external cron*,Soft cron是一个合理的不得已而为之,但如果web服务器允许其它cron方法,应优先选用它们。

如果你使用内置web服务器(直接使用或通过Apache mod_proxy),默认为hard cron。Hard cron在并行线程中执行,所以不像soft cron,关于运行时间或执行时间精度没有限制。

在任何情况下,默认的都不会是外部cron,但要求能访问系统cron设施,它运行在平行进程中,因此soft cron的任何限制都不适用。在WSGI或FastCGI下,这是推荐的使用cron的方式。

添加到系统 crontab 文件（通常是 / etc / crontab）中的代码行的例子：

```
1 0-59/1 * * * * web2py cd /var/www/web2py/ && python web2py.py -J -C -D 1 >> /tmp/  
cron.output 2>&1
```

如果运行的是外部 cron，确保添加 -N 命令行参数到 web2py 启动脚本或配置，这样 cron 的多种类型不会有冲突。此外对于外部 cron，一定要加上 -J（或 --cronjob，这是相同的）如上文所述，这样 web2py 知道任务是由 cron 执行。web2py 用 soft cron 和 hard cron 对该内部设置。

在一个特定进程中，在不需要任何 cron 功能的情况下，可以使用 -N 命令行参数来禁用它。注意，这可能会禁用一些维护任务（如自动清理会话文件夹），此功能的最常见的用途是：

- 已经设置了系统触发的外部 cron（最常见的是使用 WSGI 设置）。
- 要调试应用，而不需要 cron 干预动作或输出。

4. 18. 2 自制任务队列

虽然以规定的时间间隔运行任务 cron 是有用的，但它并不总是运行后台任务的最佳解决方案。为了这个目的，web2py 提供运行任何 Python 脚本的能力，它就像是在一个控制器中。

```
1 python web2py.py -S app -M -N -R applications/app/private/myscript.py -A a b c
```

这里 -S app 告诉 web2py 作为“app”运行“myscript.py”，-M 告诉 web2py 执行模型，-N 告诉 web2py 不运行 cron，并且 -A a b c 传递可选命令行参数 sys.args=['a', 'b', 'c'] 到“myscript.py”。

因为为了确保同一时间运行的实例不超过一个，这种类型的后台进程不应该通过 cron 执行（也许除非是为了 cron @reboot）。使用 cron 可能出现一个进程在 cron 迭代 1 开始，cron 迭代 2 还未完成，所以 cron 不断地启动它——因此堵塞邮件服务器。

在第 8 章中，我们将提供一个如何使用上面的方法来发送电子邮件的例子。

4. 18. 3 调度程序（实验性的）

web2py 的调度非常类似于在上一小节中介绍的任务队列，有一些不同：

- 它提供一个标准机制，用于创建和调度任务。
- 没有单个后台进程但有一组工作进程。
- 工作节点的任务可以被监控因为它们的状态以及任务状态，存储在数据库中。
- 它不需要 web2py 也能工作，但不会记录在这里。

调度不使用 cron，尽管可以使用 @reboot 启动工作节点。

在调度中，任务就是一个定义在模型中（或者在模块中并由模型导入）的函数。例如：

```
1 def task_add(a,b):  
2     return a+b
```

任务总是在控制器看到的相同环境中被调用，因此控制器可以看到定义在模型中的所有全局变量，包括数据库连接（db）。任务与控制器动作不同因为它们不与 HTTP 请求关联因此不存在 request.env。

一旦任务被定义，需要通过将下面的代码添加到模型启用调度程序：

```
1 myscheduler = Scheduler(db, dict(task_add=task_add))
```

Scheduler 类的第一个参数必须是调度要使用的与工人沟通的数据库，这可能是应用程序的 db 或其它专用 db，也许是一个由多个应用程序共享的 db。调度程序将创建所需要的表，第二个参数是一个 key:value 对 Python 字典，其中 key 是公开任务要使用的名称，value 是定义任务的函数的实际名称。

一旦任务被定义并且 Scheduler 被实例化，所有需要做的就是启动工人：

```
1 python web2py.py -K myapp
```

-K 选项将启动一个工人，-K 选项的参数是以逗号分隔的应用名称的列表。有应用将别工人服务，可以启动许多工人。

现在我们的基础设施到位：明确了任务，告诉了调度，启动了工作者（S），剩下的就是实际安排任务：

可以编程方式或通过 appadmin 预定任务。事实上，任务预定可以简单地通过在表 "scheduler_task" 中添加一个条目来完成，可以通过访问 appadmin 该表：

```
1 http://127.0.0.1:8000/scheduler/appadmin/insert/db/scheduler_task
```

此表中字段的含义是显而易见的，"args" 和 "vars" 字段是要以 JSON 格式传递给任务的值。在上面的 "task_add" 例子中，"args" 和 "vars" 的例子可以是：

```
args = [3, 4]
vars = {}
```

或

```
args = []
vars = {'a':3, 'b':4}
```

任务可以是下列状态之一：

```
1 QUEUED, RUNNING, COMPLETED, FAILED, TIMEOUT
```

一旦任务存在（"scheduler_task" 表中有记录），被排队，并已准备好（满足所有在记录中指定的条件），它可以被工人取走。只要有可用的工人，它就取走首先准备好的预定运行的任务，工人在另一个表 "scheduler_run"（也创造由调度）中创建一个条目。

表 "scheduler_run" 保存所有正在运行的任务的状态，每个记录引用一个被工人取走的任务，一个任务可以有多个运行。例如，一个预计一个小时重复 10 次的任务可能会有 10 次运行（除非其中一个出现故障或时间超过 1 小时）。

可能的运行状态是：

```
1 RUNNING, COMPLETED, FAILED, TIMEOUT
```

当排队的任务被取走，它成为一个 RUNNING 任务并且它的运行状态也是 RUNNING。如果运行完成且没有异常抛出，也没有任务超时，该运行标记为 COMPLETED，该任务是被标记为 QUEUED 或 COMPLETED，将取决于稍后的时间是否会再次运行它。任务的输出是在 JSON 中序列化并存储在运行记录中。

当正在运行的任务抛出一个异常，运行标记为 FAILED 并且任务被标记为 FAILED，回溯被存储在运行记录中。

同样，当运行超时，它被停止并标记为 TIMEOUT，任务标记为 TIMEOUT。

在任何情况下，stdout 被捕获并且被记录到运行记录中。

使用 appadmin，可以检查所有 RUNNING 的任务，COMPLETED 任务的输出，FAILED 任务的错误等。

调度程序还创建了一个被称为“scheduler_worker”的表，其存储工人的心跳和他们的状态，工人可能的状态是：

```
1 ACTIVE, INACTIVE, DISABLED
```

通过使用 appadmin 改变其状态，可以禁用工人。

通过 appadmin 能完成的都能通过插入和更新这些表中的记录以编程的方式完成。

无论如何，不应该更新与 RUNNING 任务相关的记录因为这可能会造成不可预测的行为，最好的做法是使用“insert”为任务排队。例如：

```
1 db.scheduler_task.insert(  
2 status='QUEUED',  
3 application_name='myapp',  
4 task_name='my first task',  
5 function_name='task_add',  
6 args='[]',  
7 vars="{ 'a':3, 'b':4 }",  
8 enabled=True,  
9 start_time = request.now,  
10 stop_time = request.now+datetime.timedelta(days=1),  
11 repeats = 10, # run 10 times  
12 period = 3600, # every 1h  
13 timeout = 60, # should take less than 60 seconds  
14 )
```

注意字段“times_run”、“last_run_time”和“assigned_worker_name”没有在计划的时间提供，但会由工人自动填充。

还可以检索已完成任务的输出：

```
1 completed_runs = db(db.scheduler_run.status='COMPLETED').select()
```

调度是实验性的，因为它需要更广泛的测试还因为随着越来越多的功能被添加，表的结构可能发生变化。

- 我们推荐有一个单独的模型文件用来定义任务和实例化调度程序（在任务被定义之后）。
- 我们推荐每个应用至少使用一名工人，让你有更多的控制权，虽然这不是绝对必要的。
- 如果任务是定义在一个模块（而不是一个模型）中，可能需要重新启动工人。

4.19 第三方模块

web2py 是用 Python 编写的，所以它可以导入和使用任何 Python 模块，包括第三方模块，它只需要能够找到它们。如同任何 Python 应用，模块可以安装在官方 Python 的“site-packages”目录，然后在代码里它们可以从任意地方导入。

“site-packages”目录中的模块，顾名思义是站点级封装，需要 site-packages 的应用是不可移植的，除非这些模块是单独安装的，在“site-packages”中有模块的优势是多个应用可以共享它们。让我们考虑一下，例如称为“matplotlib”的绘图软件包，可以采用 PEAK easy_install 命令从 shell 安装它：

```
l easy_install py-matplotlib
```

然后就可以使用如下代码将其导入到任何模型/控制器/视图：

```
l import matplotlib
```

web2py 源代码发布和 Windows 二进制发布包在顶级文件夹中有 site-packages，Mac 二进制发布版在如下文件夹中有 site-packages 文件夹：

web2py.app/Contents/Resources/site-packages

使用 site-packages 的问题是，难以在同一时间使用不同版本的单个模块，例如可以有两个应用，但每一个使用不同版本的同一个文件。在这个例子中，sys.path 不能被改变，因为它会影响这两个应用。

对于这种情况，web2py 提供了另一种方式来导入模块，这样的方式没有改变全局 sys.path：将它们放置在应用的“modules”文件夹中，一个附带的好处是，该模块将自动与应用一起复制和发布。

一旦模块“mymodule.py”被放置到应用的“modules/”文件夹，在 web2py 应用内，可以从任何地方导入它（无需改变 sys.path）：

```
l import mymodule
```

4.20 执行环境

虽然这里所讨论的一切工作正常，我们推荐使用在第 12 章介绍的组件来构建应用。

web2py 模型和控制器文件不是 Python 模块，因而不能使用 Python 的 import 语句导入它们。这样做的原因是，模型和控制器设计在一个预先准备好的环境中执行，该环境已预先填充了 web2py 全局对象（请求、响应、会话、缓存和 T）及帮助对象函数。这是必要的，因为 Python 是一种静态（词汇）范围内的语言，而 web2py 环境是动态创建的。web2py 提供 exec_environment 函数让你直接访问模型和控制器，exec_environment 创建了一个 web2py 执行环境，加载文件到它里面，然后返回一个包含环境的 Storage 存储对象。Storage 对象也可作为一个命名空间机制，在执行环境中执行任何 Python 文件可以使用 exec_environment 加载，exec_environment 的用途包括：

- 从其它应用访问数据（模型）。
- 从其它模型或控制器访问全局变量。

- 从其它控制器执行控制器函数。
- 加载站点范围内的辅助库。

这个例子读取来自 cas 应用 user 表中的行：

```
1 from gluon.shell import exec_environment
2 cas = exec_environment('applications/cas/models/db.py')
3 rows = cas.db().select(cas.db.user.ALL)
```

另外一个例子：假设有一个控制器“other.py”，其中包含：

```
1 def some_action():
2 return dict(remote_addr=request.env.remote_addr)
```

下面是如何从另一个控制器调用这个动作（或从 web2py 的 shell）：

```
1 from gluon.shell import exec_environment
2 other = exec_environment('applications/app/controllers/other.py', request=request)
3 result = other.some_action()
```

在第 2 行，request=request 是可选的，它的效果是将当前请求传递到“other”环境中，如果没有这个参数，该环境将包含一个新的空的（除了 request.folder）请求对象，也可以传递响应和会话对象到 exec_environment。传递请求、响应和会话对象时要小心——调用对象的修改或调用行动的编码依赖性可能会导致意料之外的副作用。

第 3 行中的函数调用不执行视图，它仅仅是返回字典，除非 response.render 由“some_action”显式调用。

最后一个警告：不要不恰当地使用 exec_environment，如果想要其它应用中动作的结果，应该实现一个 XML-RPC API（使用 web2py 实现 XML-RPC API 几乎是微不足道的），不要使用 exec_environment 作为重定向机制，使用 redirect 帮助对象。

4.21 协作

应用协作的方法有很多：

- 应用可以连接到相同数据库，从而可以共享表。数据库中的所有表由所有应用程序定义是没有必要的，但它们必须被那些使用它们的应用定义，所有使用相同表的应用，必须有一个用 migrate=False 定义的表。
- 应用可以使用 LOAD 帮助对象嵌入来自其它应用的组件（第 12 章中介绍）。
- 应用可以共享会话。
- 应用可以通过 XML-RPC 远程调用对方的动作。
- 通过文件系统，应用可以访问对方的文件（假设它们共享相同的文件系统）。
- 应用可以使用 exec_environment 本地调用对方的动作，正如上面所讨论的。

- 应用可以使用下面的语法导入其它模块：

```
1 from applications.appname.modules import mymodule
```

- 应用能导入 PYTHONPATH 搜索路径 `sys.path` 中的任何模块。

应用可以使用下面的命令加载另一个应用会话：

```
1 session.connect(request, response, masterapp='appname', db=db)
```

这里“appname”是主应用的名称，它在 cookie 中设置初始会话 ID，db 是一个到数据库的连接，该数据库包含 session 表（web2py_session），所有共享会话的应用必须使用相同的数据库存储会话。

使用如下语句，应用可以从另一个应用加载模块。

```
1 import applications.otherapp.modules.othermodule
```

4.22 日志

Python 提供了日志记录 API，web2py 提供了一种配置它的机制，以便应用使用它。

在应用中，可以创建一个记录器，例如在一个模型中：

```
1 import logging
2 logger = logging.getLogger("web2py.app.myapp")
3 logger.setLevel(logging.DEBUG)
```

可以用它来记录各种重要消息

```
1 logger.debug("Just checking that %s" % details)
2 logger.info("You ought to know that %s" % details)
3 logger.warn("Mind that %s" % details)
4 logger.error("Oops, something bad happened %s" % details)
```

日志记录是一个标准的 Python 模块，在这里介绍：

```
1 http://docs.python.org/library/logging.html
```

字符串“web2py.app.myapp”定义一个应用级记录器。

为了使它正常工作，需要一个记录器的配置文件，web2py 在 web2py 根文件夹

“logging.example.conf”中提供了一个，需要将该文件重命名为“logging.conf”并根据需要对其进行定制。

这个文件是自己记录，所以应该打开并阅读它。

要创建一个可配置的“myapp”应用的记录器，必须添加 myapp 到[loggers]的键列表：

```
1 [loggers]
2 keys=root, rocket, markdown, web2py, rewrite, app, welcome, myapp
```

而且必须添加一个[logger_myapp]部分，使用[logger_welcome]作为出发点。

```
1 [logger_myapp]
2 level=WARNING
3 qualname=web2py.app.myapp
4 handlers=consoleHandler
```

5 propagate=0

“handlers”指令指定日志记录的类型，这里它将“myapp”日志记录到控制台。

4.23 WSGI（Web 服务器网关接口）

web2py 和 WSGI 之间存在又爱又恨的关系，我们的观点是，WSGI 作为协议将 Web 服务器以可移植的方式连接到 Web 应用，我们为了这一目的而使用它。web2py 中核心是一个 WSGI 应用：gluon.main.wsgibase，一些开发人员将 WSGI 作为中间通信协议推到了极致，并将 Web 应用作为一个类似有许多层（每层是一个 WSGI 中间件，独立于整个框架开发）的洋葱来开发。web2py 内部不采用这种结构，是因为我们觉得如果用一个单一的综合层处理它们，可以更好地优化框架核心功能（处理 Cookie、会话、错误、交易、调度）的处理速度和安全性。

然而，web2py 允许以三种方式（以及它们的组合）使用第三方 WSGI 应用和中间件：

- 可以编辑文件“wsgihandler.py”并包括任何第三方 WSGI 中间件。
- 可以将第三方 WSGI 中间件连接到你的应用中的任何特定动作。
- 可以在动作中调用第三方 WSGI 应用。

唯一的限制是不能使用第三方中间件更换核心 web2py 功能。

4.23.1 外部中间件

考虑文件“wsgibase.py”：

```
1 #...
2 LOGGING = False
3 #...
4 if LOGGING:
5     application = gluon.main.appfactory(wsgiapp=gluon.main.wsgibase,
6                                         logfilename='httpserver.log',
7                                         profilerfilename=None)
8 else:
9     application = gluon.main.wsgibase
```

当 LOGGING 设置为 True 时，gluon.main.wsgibase 被中间件函数 gluon.main.appfactory 包裹，它为“httpserver.log”文件提供了日志。以类似的方式，可以添加任何第三方中间件，我们参阅了官方的 WSGI 文档了解到更多信息。

4.23.2 内部中间件

给定控制器中的任何动作（例如 index）及任何第三方中间件应用（例如 MyMiddleware，它将输出转换为大写），可以使用 web2py 的装饰器将中间件应用于动作。下面是一个例子：

```

1 class MyMiddleware:
2     """converts output to upper case"""
3     def __init__(self, app):
4         self.app = app
5     def __call__(self, environ, start_response):
6         items = self.app(environ, start_response)
7         return [item.upper() for item in items]
8
9 @request.wsgi.middleware(MyMiddleware)
10 def index():
11     return 'hello world'

```

我们不能保证所有第三方中间件都能与该机制工作。

4.23.3 调用 WSGI 应用

从 web2py 动作调用 WSGI 应用是很容易的。下面是一个例子：

```

1 def test_wsgi_app(environ, start_response):
2     """this is a test WSGI app"""
3     status = '200 OK'
4     response_headers = [('Content-type', 'text/plain'),
5                          ('Content-Length', '13')]
6     start_response(status, response_headers)
7     return ['hello world!\n']
8
9 def index():
10     """a test action that calls the previous app and escapes output"""
11     items = test_wsgi_app(request.wsgi.environ,
12                          request.wsgi.start_response)
13     for item in items:
14         response.write(item, escape=False)
15 return response.body.getvalue()

```

在这种情况下，index 动作调用 test_wsgi_app 并在返回之前转移转义返回值，注意 index 本身并不是 WSGI 应用，必须使用标准的 web2py API（例如用 response.write 写入套接字）。

第 5 章 视图

web2py使用Python模型、控制器和视图，虽然在视图中它使用略作修改的Python语法使代码更具可读性，而不对适当的Python施加任何限制。

视图的目的是在HTML文档中嵌入代码（Python），一般情况下，这带来了一些问题：

- 如何转义嵌入的代码？
- 缩进应根据Python还是HTML规则？

web2py使用`{{... }}`转义嵌入在HTML中的Python代码，使用大括号而不是尖括号的优势是，它对所有常见的HTML编辑器是透明的，这使开发人员可以使用这些编辑器创建web2py视图。

由于开发人员将Python代码嵌入到HTML中，该文件的缩进应根据HTML规则而不是Python规则。因此，在`{{... }}`标签内我们允许不缩进的Python，因为Python通常使用缩进来分隔代码块，我们需要以不同的方式来分隔它们，这就是web2py模板语言使用Python关键字`pass`的原因。

一个代码块开始与一行以冒号结束的代码并且结束于一行以`pass`开始的代码。在上下文中，代码块的结尾显而易见时，关键字`pass`是没有必要的。

下面是一个例子：

```
1 {{
2 if i == 0:
3 response.write('i is 0')
4 else:
5 response.write('i is not 0')
6 pass
7 }}
```

需要注意的是`pass`是一个Python关键字，而不是web2py关键字。一些Python编辑器，比如Emacs，使用关键字`pass`来表示块的划分并用它来重新自动缩进代码。

web2py模板语言实现的功能是一样的，当它发现类似如下的代码时：

```
1 <html><body>
2 {{for x in range(10):}} {{=x}}hello<br />{{pass}}
3 </body></html>
```

它把代码转换成一个程序：

```
1 response.write("""<html><body>""", escape=False)
2 for x in range(10):
3 response.write(x)
4 response.write("""hello<br />""", escape=False)
5 response.write("""</body></html>""", escape=False)
```

`response.write`向`response.body`中写入。

当web2py视图中出现错误时，错误报告显示了生成的视图代码，而不是开发者编写的实际的视图，通过突出显示实际执行的代码（该代码可以用HTML编辑器或浏览器的DOM检查器

来调试)帮助开发者调试代码。

还要注意的是:

```
1 {{=x}}
```

生成

```
1 response.write(x)
```

默认情况下,这样注入HTML中的变量被转义,如果x是一个XML对象,转义被忽略,即使转义设置为True。

下面是一个介绍H1帮助对象的例子:

```
1 {{=H1(i)}}
```

它被转换成:

```
1 response.write(H1(i))
```

评估完成后,H1对象和它的组成部分被递归地序列化、转义并写入到响应主体中,由H1和内部HTML生成的标签不被转义,这种机制保证了在网页上显示的所有文字——只有文字——总是得到转义,从而防止XSS漏洞。同时,代码简单且易于调试。

方法response.write(obj, escape=True)有两个参数,要写入的对象以及它是否要进行转义(默认设置为True),如果obj有xml()方法,它被调用并将结果写入到响应主体(escape参数被忽略);否则,它使用对象的__str__方法将其序列化,如果转义参数是True,对其进行转义。所有内置帮助对象(例子中是H1)知道如何通过XML()方法对自身进行序列化。

这都是透明完成的,永远不需要(而且永远不应该)显式调用response.write方法。

5.1 基本语法

web2py模板语言支持所有的Python控制结构,在这里我们为每个结构提供一些例子,可以根据通常的编程实践嵌套它们。

5.1.1 for...in

在模板中可以循环任何迭代对象:

```
1 {{items = ['a', 'b', 'c']}}
2 <ul>
3 {{for item in items:}}<li>{{=item}}</li>{{pass}}
4 </ul>
```

这会产生

```
1 <ul>
2 <li>a</li>
3 <li>b</li>
```

```
4 <li>c</li>
5 </ul>
```

这里item是任意可迭代对象，如Python列表、Python元组或Rows对象，或任何作为迭代器来实现的对象。首先序列化和转义要显示的元素。

5.1.2 while

可以使用while关键字创建循环：

```
1 {{k = 3}}
2 <ul>
3 {{while k > 0:}}<li>{{=k}} {{k = k - 1}}</li>{{pass}}
4 </ul>
```

这会产生：

```
1 <ul>
2 <li>3</li>
3 <li>2</li>
4 <li>1</li>
5 </ul>
```

5.1.3 if...elif...else

可以使用条件从句：

```
1 {{
2 import random
3 k = random.randint(0, 100)
4 }}
5 <h2>
6 {{=k}}
7 {{if k % 2:}}is odd{{else:}}is even{{pass}}
8 </h2>
```

这会产生：

```
1 <h2>
2 45 is odd
3 </h2>
```

因为else显然关闭了第一个if块，没有必要使用pass语句，并且使用将是不正确的。但是，必须使用pass明确地关闭else块。

回想一下，Python中的“else if”是写成elif正如在下面的例子中：

```
1 {{
2 import random
3 k = random.randint(0, 100)
4 }}
5 <h2>
6 {{=k}}
7 {{if k % 4 == 0:}}is divisible by 4
8 {{elif k % 2 == 0:}}is even
9 {{else:}}is odd
10 {{pass}}
```

```
11 </h2>
```

它会产生：

```
1 <h2>
2 64 is divisible by 4
3 </h2>
```

5.1.4 try...except...else...finally

也可以使用try... except语句但有一点需要注意。请看下面的例子：

```
1 {{try:}}
2 Hello {{= 1 / 0}}
3 {{except:}}
4 division by zero
5 {{else:}}
6 no division by zero
7 {{finally}}
8 <br />
9 {{pass}}
```

它将产生以下输出：

```
1 Hello
2 division by zero
3 <br />
```

这个例子说明，在异常发生之前产生的所有输出（包括异常之前的输出）在try块中呈现，“Hello”被编写是因为它在异常之前。

5.1.5 def...return

web2py模板语言允许开发人员定义和实现可以返回任何Python对象或text / html字符串的函数。在这里，我们来看两个例子：

```
1 {{def itemize1(link): return LI(A(link, _href="http://" + link))}}
2 <ul>
3 {{=itemize1('www.google.com')}}
4 </ul>
```

产生如下输出：

```
1 <ul>
2 <li><a href="http://www.google.com">www.google.com</a></li>
3 </ul>
```

函数itemize1能返回一个插入在函数调用位置的帮助对象。

现在考虑下面的代码：

```
1 {{def itemize2(link):}}
2 <li><a href="http://{{=link}}">{{=link}}</a></li>
3 {{return}}
4 <ul>
5 {{itemize2('www.google.com')}}
6 </ul>
```

它产生与上述完全相同的输出。在这种情况下，函数itemize2代表一段要在函数调用位置取代web2py标记的HTML；注意，itemize2调用前面没有'='，因为该函数不返回文本，而是直接将它写入到响应中。

有一点需要注意：视图中定义的函数必须用一个return语句终止，否则自动缩进会失败。

5.2 HTML 帮助对象

考虑下面视图中的代码：

```
1 {{=DIV('this', 'is', 'a', 'test', _id='123', _class='myclass')}}}
```

它被呈现为：

```
1 <div id="123" class="myclass">thisisatest</div>
```

DIV是一个帮助对象类，即一些可以用来编程地建立HTML，它对应于HTML的<DIV>标签。

位置参数被解释为包含在open和close标签之间的对象，以下划线开头的命名参数被解释为HTML标签的属性（不再带下划线），一些帮助对象也有不以下划线开始的命名参数，这些参数是特定于标记的。

除了一组命名参数之外，帮助对象也可以使用*符号接收一个单独的列表或元组作为自己的一套组件，并且可以使用**接收单一的字典作为属性组，例如：

```
1 {{
2 contents = ['this', 'is', 'a', 'test']
3 attributes = {'_id': '123', '_class': 'myclass'}
4 =DIV(*contents, **attributes)
5 }}
```

（产生与之前相同的输出）。

以下帮助对象集：

A, B, BEAUTIFY, BODY, BR, CAT, CENTER, CODE, COL, COLGROUP, DIV, EM, EMBED, FIELDSET, FORM, H1, H2, H3, H4, H5, H6, HEAD, HR, HTML, I, IFRAME, IMG, INPUT, LABEL, LEGEND, LI, LINK, MARKMIN, MENU, META, OBJECT, ON, OL, OPTGROUP, OPTION, P, PRE, SCRIPT, SELECT, SPAN, STYLE, TABLE, TAG, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TITLE, TR, TT, UL, URL, XHTML, XML, embed64, xmlescape可以用于构建复杂的表达式，该表达式之后被序列化成XML [51] [52]。例如：

```
1 {{=DIV(B(I("<hello ", "<world>")), _class="myclass')}}}
```

被呈现为：

```
1 <div class="myclass"><b><i>hello &lt;world>&gt;</i></b></div>
```

帮助对象也可以被序列化为字符串，与使用__str__和XML方法是等价的：

```

1 >>> print str(DIV("hello world"))
2 <div>hello world</div>
3 >>> print DIV("hello world").xml()
4 <div>hello world</div>

```

在web2py中的帮助对象机制不仅是一个无需连接字符串即可生成HTML的系统，它提供了一个文档对象模型（DOM）的服务器端表示。

可以通过位置引用帮助对象的组件，并且从组件的角度看，帮助对象的行为类似列表：

```

1 >>> a = DIV(SPAN('a', 'b'), 'c')
2 >>> print a
3 <div><span>ab</span>c</div>
4 >>> del a[1]
5 >>> a.append(B('x'))
6 >>> a[0][0] = 'y'
7 >>> print a
8 <div><span>yb</span><b>x</b></div>

```

帮助对象的属性可以通过名称引用，而且从属性的角度看，帮助对象的行为类似于字典：

```

1 >>> a = DIV(SPAN('a', 'b'), 'c')
2 >>> a['_class'] = 's'
3 >>> a[0]['_class'] = 't'
4 >>> print a
5 <div class="s"><span class="t">ab</span>c</div>

```

注意，可以通过称为a.components的列表访问完整的组件集，并且可以通过称为a.attributes的字典访问完整属性集。因此，当i是整数时a[i]相当于a.components[i]，并且当s是字符串时a[s]相当于a.attributes[s]。

注意帮助对象属性作为关键字参数传递给帮助对象。然而，在一些情况下，属性名称中包含Python标识符不允许的特殊字符（例如，连字符），因此不能被用来作为关键字参数名称。例如：

```

1 DIV('text', _data-role='collapsible')

```

不会起作用，因为"_data-role"包含连字符，这将产生Python语法错误。

在这种情况下，可以将属性作为字典传递并使用Python的**函数参数符号，它把（键：值）对字典映射成关键字参数集：

```

1 >>> print DIV('text', **{'_data-role': 'collapsible'})
2 <div data-role="collapsible">text</div>

```

也可以动态的创建特殊标签：

```

1 >>> print TAG['soap:Body']('whatever', **{'_xmlns:m': 'http://www.example.org'})
2 <soap:Body xmlns:m="http://www.example.org">whatever</soap:Body>

```

5.2.1 XML（可扩展标记语言）

XML是一种用于封装那些不应该被转义的文本的对象，文本中可以包含或不包含有效的XML。例如，它可以包含JavaScript。

在这个例子中，文本被转义：

```
1 >>> print DIV("<b>hello</b>")
2 &lt;b&gt;hello&lt;/b&gt;
```

可以通过使用XML阻止转义：

```
1 >>> print DIV(XML("<b>hello</b>"))
2 <b>hello</b>
```

有时候，想呈现存储在变量中的HTML，但该HTML可能包含不安全的标记，例如脚本：

```
1 >>> print XML('<script>alert("unsafe!")</script>')
2 <script>alert("unsafe!")</script>
```

类似未转义的可执行输入（例如，在博客评论的主体中输入）是不安全的，因为它可以被用来生成跨站点脚本（XSS）攻击该页面的其它游客。

web2py的XML帮助对象可以净化我们的文本，以防止注入并转义所有标签，除了那些明确允许的。下面是一个例子：

```
1 >>> print XML('<script>alert("unsafe!")</script>', sanitize=True)
2 &lt;script&gt;alert(&quot;unsafe!&quot;)&lt;/script&gt;
```

默认情况下，XML的构造函数认为一些标签的内容和一些属性是安全的，可以使用可选参数permitted_tags和allowed_attributes重写默认值。下面是XML帮助对象可选参数的默认值：

```
1 XML(text, sanitize=False,
2 permitted_tags=['a', 'b', 'blockquote', 'br/', 'i', 'li',
3 'ol', 'ul', 'p', 'cite', 'code', 'pre', 'img/'],
4 allowed_attributes={'a': ['href', 'title'],
5 'img': ['src', 'alt', 'blockquote': ['type']})
```

5.2.2 内置帮助对象

A

该帮助对象用于创建链接。

```
1 >>> print A('<click>', XML('<b>me</b>'),
2 _href='http://www.web2py.com')
3 <a href='http://www.web2py.com'>&lt;click&gt;<b>me</b></a>
```

除了_href，可以使用callback参数传递URL。例如在一个视图中：

```
1 {{=A('<click me', callback=URL('myaction'))}}
```

按下链接的效果将是对“myaction”的一个ajax调用，而不是重定向。在这种情况下，可以可选的指定两个参数：target 和 delete：

```
1 {{=A('<click me', callback=URL('myaction'), target="t")}}
2 <div id="t"></div>
```

并且ajax回调的响应将被存储在id等于“t”的DIV中。

```
1 <div id="b">{{=A('<click me', callback=URL('myaction'), delete='div#b')}}</div>
```


一旦响应，匹配“div#b”的最接近的标记将被删除。在这种情况下，该按钮将被删除，一个典型的应用是：

```
1 {{=A('click me', callback=URL('myaction'), delete='tr')}}}
```

在一个表中，按下按钮将要执行回调并删除表的行。

callback和delete可以结合起来。

帮助对象A带一个称为cid的特殊参数。它的工作原理如下：

```
1 {{=A('linked page', _href='http://example.com', cid='myid')}}}
2 <div id="myid"></div>
```

并且点击链接会导致内容被加载在div中，这类似于上面的语法但是更有效，因为它是专门用来刷新页面组件的。第12章我们将在组件的上下文中更详细的讨论cid的应用。

这些ajax功能需要jQuery和“static/js/web2py_ajax.js”，通过在布局头中放置{{include 'web2py_ajax.html'}}，它们会被自动包含，“views/web2py_ajax.html”定义了一些基于request的变量并且包括所有必要的js和css文件。

B

这个帮助对象使其内容加粗。

```
1 >>> print B('<hello>', XML('<i>world</i>'), _class='test', _id=0)
2 <b id="0" class="test">&lt;hello&gt;<i>world</i></b>
```

BODY

该帮助对象创建页面主体。

```
1 >>> print BR()
2 <br />
```

CAT (1.98.1 及以上)

这个帮助对象连接其它帮助对象，像TAG[”]一样。

```
1 >>> print CAT('Here is a ', A('link', _href=URL()), ', and here is some ', B('bold text'), '.')
2 Here is a <a href="/app/default/index">link</a>, and here is some <b>bold text</b>.
```

CENTER

该帮助对象使其内容居中。

```
1 >>> print CENTER('<hello>', XML('<b>world</b>'),
2 >>> _class='test', _id=0)
3 <center id="0" class="test">&lt;hello&gt;<b>world</b></center>
```

CODE

该帮助对象为Python、C、C++、HTML和web2py代码执行语法高亮显示，并且最好为代码清单使用PRE，CODE也有能力创建链接到web2py的API文档。

下面是一个高亮显示部分Python代码的例子。

```
1 >>> print CODE('print "hello"', language='python').xml()
2 <table><tr valign="top"><td style="width:40px; text-align: right;"><pre style="
```

```

3 font-size: 11px;
4 font-family: Bitstream Vera Sans Mono, monospace;
5 background-color: transparent;
6 margin: 0;
7 padding: 5px;
8 border: none;
9 background-color: #E0E0E0;
10 color: #A0A0A0;
11 ^1.</pre></td><td><pre style="
12 font-size: 11px;
13 font-family: Bitstream Vera Sans Mono, monospace;
14 background-color: transparent;
15 margin: 0;
16 padding: 5px;
17 border: none;
18 overflow: auto;
19 ^<span style="color:#185369; font-weight: bold">print </span>
20 <span style="color: #FF9966">"hello"</span></pre></td></tr>
21 </table>

```

下面是一个类似的HTML的例子

```

1 >>> print CODE(
2 >>> ' <html><body>{{=request.env.remote_add}}</body></html>',
3 >>> language='html')

```

```

1 <table>...<code>...
2 <html><body>{{=request.env.remote_add}}</body></html>
3 ...</code>...</table>

```

这是 **CODE** 帮助对象的默认参数:

```
1 CODE("print 'hello world'", language='python', link=None, counter=1, styles={})
```

language参数支持的语言的值是“python”、“html_plain”、“c”、“cpp”、“web2py”及“html”，“html”语言会把{{ and }}标签解释为“web2py”代码，而“html_plain”不会。

如果指定link的值，例如“/examples/global/vars/”，代码中的web2py API引用链接到URL链接处的文档，例如“request”将被链接到“/examples/global/vars/request”。在上述的例子中，链接由“global.py”控制器中的“vars”动作处理，“global.py”控制器作为web2py的“examples”应用的部分被发布。

counter参数用于行编号。它可以被设置为三个不同值中的任意一个，也可以是没有行号时的None，或一个指定起始号码的数值，或一个字符串，如果计数器设置为一个字符串，它被解释为提示，并且没有行号。

styles参数有点棘手。如果看一下上面生成的HTML，它包含一个有两列的表，每列都有自己的使用CSS声明为inline的样式，styles属性允许覆盖这两个CSS样式。例如：

```
1 {{=CODE(..., styles={'CODE': 'margin: 0; padding: 5px; border: none;'})}}
```

styles属性必须是一个字典，并且它允许两个可能的密钥：CODE实际代码样式和LINENUMBERS左边列的样式，其中包含行号。记住，这些样式完全替换默认样式而不是简单地添加到它们。

COL

```
1 >>> print COL('a', 'b')
2 <col>ab</col>
```

COLGROUP

```
1 >>> print COLGROUP('a', 'b')
2 <colgroup>ab</colgroup>
```

DIV

除了XML之外的所有帮助对象均来自DIV并继承它的基本方法。

```
1 >>> print DIV('hello', XML('<b>world</b>'), _class='test', _id=0)
2 <div id="0" class="test">&lt;hello&gt;<b>world</b></div>
```

EM

强调它的内容。

```
1 >>> print EM('hello', XML('<b>world</b>'), _class='test', _id=0)
2 <em id="0" class="test">&lt;hello&gt;<b>world</b></em>
```

FIELDSET

这用来创建一个输入字段连同它的标签。

```
1 >>> print FIELDSET('Height:', INPUT(_name='height', _class='test'))
2 <fieldset class="test">Height:<input name="height" /></fieldset>
```

FORM

这是最重要的帮助工具之一。其形式简单，它只是一个<form>... </form>标签，但因为帮助对象是对象并且知道其所包含的内容，它们可以处理提交的表单（例如，执行字段验证）。这将在第7章中详细讨论。

```
1 >>> print FORM(INPUT(_type='submit'), _action='', _method='post')
2 <form enctype="multipart/form-data" action="" method="post">
3 <input type="submit" /></form>
```

默认情况下，“enctype”是“multipart / form-data”。

FORM的构造函数及SQLFORM也可以接收一个特殊参数。一本字典传递时，其项目被翻译成“hidden”的INPUT字段。例如：

```
1 >>> print FORM(hidden=dict(a='b'))
2 <form enctype="multipart/form-data" action="" method="post">
3 <input value="b" type="hidden" name="a" /></form>
```

H1, H2, H3, H4, H5, H6

这些帮助对象用于段落标题和副标题：

```
1 >>> print H1('hello', XML('<b>world</b>'), _class='test', _id=0)
2 <h1 id="0" class="test">&lt;hello&gt;<b>world</b></h1>
```

HEAD

用于标记HTML页面的标题。

```
1 >>> print HEAD(TITLE('<hello>', XML('<b>world</b>')))  
2 <head><title>&lt;hello&gt;<b>world</b></title></head>
```

HTML

这个帮助对象有点不同。除了作出<html>标签以外，它还使用文档类型的字符串[54, 55, 56] 添加标签。

```
1 >>> print HTML(BODY('<hello>', XML('<b>world</b>')))  
2 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
3 "http://www.w3.org/TR/html4/loose.dtd">  
4 <html><body>&lt;hello&gt;<b>world</b></body></html>
```

HTML帮助对象还需要一些额外的具有以下默认的可选参数：

```
1 HTML(..., lang='en', doctype='transitional')
```

这里文档类型可以是'strict'、'transitional'、'frameset'、'html5' 或一个完整的文档类型字符串。

XHTML

XHTML与HTML类似，但它创建的是XHTML文档类型。

```
1 XHTML(..., lang='en', doctype='transitional', xmlns='http://www.w3.org/1999/xhtml')
```

这里文档类型可以是'strict'、'transitional'、'frameset'、'html5' 或一个完整的文档类型字符串。

HR

这个帮助对象在HTML页面中创建一条水平线

```
1 >>> print HR()  
2 <hr />
```

I

该帮助对象将其内容设为斜体。

```
1 >>> print I('<hello>', XML('<b>world</b>'), _class='test', _id=0)  
2 <i id="0" class="test">&lt;hello&gt;<b>world</b></i>
```

INPUT

创建一个<input.../>标签。输入标签不能包含其它标签，并使用/>而不是>关闭，INPUT标签有一个可选属性_type，可以设置为"text"（默认值）、"submit"、"checkbox"或"radio"。

```
1 >>> print INPUT(_name='test', _value='a')  
2 <input value="a" name="test" />
```

它也有一个可选的特殊参数，称为"value"，其与"_value"不同，后者设置输入字段的默认值，前者设置其当前值。对于一个"text"类型的输入，前者覆盖后者：

```
1 >>> print INPUT(_name='test', _value='a', value='b')  
2 <input value="b" name="test" />
```

对于单选按钮，INPUT可选择设置"checked" 属性：

```

1 >>> for v in ['a', 'b', 'c']:
2 >>> print INPUT(_type='radio', _name='test', _value=v, value='b'), v
3 <input value="a" type="radio" name="test" /> a
4 <input value="b" type="radio" checked="checked" name="test" /> b
5 <input value="c" type="radio" name="test" /> c

```

复选框是类似的：

```

1 >>> print INPUT(_type='checkbox', _name='test', _value='a', value=True)
2 <input value="a" type="checkbox" checked="checked" name="test" />
3 >>> print INPUT(_type='checkbox', _name='test', _value='a', value=False)
4 <input value="a" type="checkbox" name="test" />

```

IFRAME

这个帮助对象在当前页面中包含另一个网页，另一页面的url通过“_src”属性指定。

```

1 >>> print IFRAME(_src='http://www.web2py.com')
2 <iframe src="http://www.web2py.com"></iframe>

```

IMG

它可以用来在HTML中嵌入图像：

```

1 >>> IMG(_src='http://example.com/image.png', _alt='test')
2 

```

下面是A，IMG和包括静态图像链接的URL帮助对象的组合：

```

1 >>> A(IMG(_src=URL('static', 'logo.png'), _alt="My Logo"),
2 _href=URL('default', 'index'))
3 <a href="/myapp/default/index">
4 
5 </a>

```

LABEL

它用于为INPUT字段创建一个LABEL标签。

```

1 >>> print LABEL('hello', XML('<b>world</b>'), _class='test', _id=0)
2 <label id="0" class="test">&lt;hello&gt;<b>world</b></label>

```

LEGEND

它用于为表单中的字段创建一个legend标签。

```

1 >>> print LEGEND('Name', _for='myfield')
2 <legend for="myfield">Name</legend>

```

LI

它创建一个项目列表，并应包含在UL或OL标记中。

```

1 >>> print LI('hello', XML('<b>world</b>'), _class='test', _id=0)
2 <li id="0" class="test">&lt;hello&gt;<b>world</b></li>

```

META

用于在HTML头中创建META标签。例如：

```
1 >>> print META(_name='security', _content='high')
2 <meta name="security" content="high" />
```

MARKMIN

实现markmin的wiki语法。它根据下面例子中描述的markmin规则，将输入文本转换成输出html：

```
1 >>> print MARKMIN("this is bold or italic and this [[a link
http://web2py.com]])")
2 <p>this is <b>bold</b> or <i>italic</i> and
3 this <a href="http://web2py.com">a link</a></p>
```

web2py附带的文件中描述了markmin语法：

```
1 http://127.0.0.1:8000/examples/static/markmin.html
```

在第12章中有一些在plugin_wiki上下文中的示例，它广泛使用MARKMIN。

可以使用markmin生成HTML、LaTeX和PDF文件：

```
1 m = "Hello world [[link http://web2py.com]]"
2 from gluon.contrib.markmin.markmin2html import markmin2html
3 print markmin2html(m)
4 from gluon.contrib.markmin.markmin2latex import markmin2latex
5 print markmin2latex(m)
6 from gluon.contrib.markmin.markmin2pdf import markmin2pdf
7 print markmin2pdf(m) # requires pdflatex
```

(MARKMIN帮助对象是markmin2html的一个快捷方式)

这是一个基本的语法primer框：

SOURCE	OUTPUT
# title	title
## section	section
### subsection	subsection
bold	bold
<i>italic</i>	<i>italic</i>
<code>verbatim</code>	<code>verbatim</code>
http://google.com	http://google.com
http://...	http:...
http://...png	
http://...mp3	<audio src="http://...mp3"></audio>
http://...mp4	<video src="http://...mp4"></video>
qr:http://...	
embed:http://...	<iframe src="http://..."></iframe>
[[click me #myanchor]]	click me
$\int_a^b \sin(x)dx$	$\int_a^b \sin(x)dx$

仅包括一个没有标记的到图像视频或音频文件的链接，导致相应的图像、视频或音频文件就会被自动包括（对于音频和视频，它使用HTML<audio>和<video>标签）。

添加一个带有qr:前缀的链接，例如

```
1 qr:http://web2py.com
```

导致相应的QR码被嵌入并链接到所述的URL。

添加一个链接标题embed:的前缀，例如：

```
1 embed:http://www.youtube.com/embed/xlw8hKTJ2Co
```

导致页面被嵌入，在这个例子中嵌入了一个youtube视频。

也可以使用下面的语法嵌入图像：

```
1 [[image-description http://.../image.png right 200px]]
```

无序列表使用：

```
1 - one
2 - two
3 - three
```

有序列表使用：

```
1 + one
2 + two
3 + three
```

表使用：

```
1 -----
2 X | 0 | 0
3 0 | X | 0
4 0 | 0 | 1
5 -----
```

MARKMIN语法也支持引用文字、HTML5音频和视频标签、图像对齐、自定义CSS，并且它可以扩展：

```
1 MARKMIN("`abab`:custom", extra=dict(custom=lambda text: text.replace('a', 'c')))
```

生成

```
' cbc b'
```

自定义块用“... “:<key>分隔并且它们被函数呈现，该函数作为相应键值传递给MARKMIN的额外字典参数。注意，该函数可能需要转义输出，以防止XSS。

OBJECT

用于在HTML中嵌入对象（例如，一个flash播放器）。

```
1 >>> print OBJECT('<hello>', XML('<b>world</b>'),
2 >>> _src='http://www.web2py.com')
3 <object src="http://www.web2py.com">&lt;hello&gt;<b>world</b></object>
```

OL

它代表了有序列表，该列表应包含LI标签，不是LI 对象的OL参数会被自动包含在...</ li>标签中。

```
1 >>> print OL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <ol id="0" class="test"><li>&lt;hello&gt;</li><li><b>world</b></li></ol>
```


ON

这是为了向后兼容，它只是True的一个别名，专门用于复选框并且已经被弃用了，因为True更加Pythonic。

```
1 >>> print INPUT(_type='checkbox', _name='test', _checked=ON)
2 <input checked="checked" type="checkbox" name="test" />
```

OPTGROUP

允许在一个SELECT中组合多个选项并且它在使用CSS自定义字段时是有用的。

```
1 >>> print SELECT('a', OPTGROUP('b', 'c'))
2 <select>
3 <option value="a">a</option>
4 <optgroup>
5 <option value="b">b</option>
6 <option value="c">c</option>
7 </optgroup>
8 </select>
```

OPTION

这应该只作为SELECT/ OPTION组合的一部分被使用。

```
1 >>> print OPTION('<hello>', XML('<b>world</b>'), _value='a')
2 <option value="a">&lt;hello&gt;<b>world</b></option>
```

至于INPUT的情况，web2py区分“_value”（OPTION的值）和“value”（封闭选择的当前值）。如果它们是平等的，option被选择。

```
1 >>> print SELECT('a', 'b', value='b'):
2 <select>
3 <option value="a">a</option>
4 <option value="b" selected="selected">b</option>
5 </select>
```

P

这用于标记一个段落。

```
1 >>> print P('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <p id="0" class="test">&lt;hello&gt;<b>world</b></p>
```

PRE

生成一个<pre>...</pre>标签用于显示预格式化的文本，通常应优先为代码清单选用CODE帮助对象。

```
1 >>> print PRE('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <pre id="0" class="test">&lt;hello&gt;<b>world</b></pre>
```

SCRIPT

这是包括或链接脚本如JavaScript。为了真过时了的浏览器的利益，标签之间的内容呈现为HTML注释。

```
1 >>> print SCRIPT('alert("hello world");', _language='javascript')
2 <script language="javascript"><!--
```

```
3 alert("hello world");
4 //--></script>
```

SELECT

创建一个<select>...</select>标签，与OPTION帮助对象合用。这些不是OPTION对象的SELECT参数被自动转换为选项。

```
1 >>> print SELECT(' <hello>', XML(' <b>world</b>'), _class='test', _id=0)
2 <select id="0" class="test">
3 <option value="&lt;hello&gt;">&lt;hello&gt;</option>
4 <option value="&lt;b&gt;world&lt;/b&gt;"><b>world</b></option>
5 </select>
```

SPAN

类似DIV但用于标记内联（而不是块）内容。

```
1 >>> print SPAN(' <hello>', XML(' <b>world</b>'), _class='test', _id=0)
2 <span id="0" class="test">&lt;hello&gt;<b>world</b></span>
```

STYLE

类似于脚本，但用来包含或链接CSS代码。这里的CSS包括

```
1 >>> print STYLE(XML('body {color: white}'))
2 <style><!--
3 body { color: white }
4 //--></style>
```

这里链接：

```
1 >>> print STYLE(_src='style.css')
2 <style src="style.css"><!--
3 //--></style>
```

TABLE、TR、TD

这些标签（以及可选的THEAD、TBODY和TFOOTER帮助对象）用来建立HTML表格。

```
1 >>> print TABLE(TR(TD('a'), TD('b')), TR(TD('c'), TD('d')))
2 <table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

TR期待TD内容，不是TD对象的参数被自动转换。

```
1 >>> print TABLE(TR('a', 'b'), TR('c', 'd'))
2 <table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

使用Python的*函数的参数标记能很容易的将Python数组转换成一个HTML表，它将列表中的元素映射到位置函数的参数。

在这里，我们逐行进行：

```
1 >>> table = [['a', 'b'], ['c', 'd']]
2 >>> print TABLE(TR(*table[0]), TR(*table[1]))
3 <table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

这里我们一次完成所有行：

```
1 >>> table = [['a', 'b'], ['c', 'd']]
2 >>> print TABLE(*[TR(*rows) for rows in table])
3 <table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

TBODY

这用来标记表的主体中包含的行，而不是页眉或页脚行，它是可选的。

```
1 >>> print TBODY(TR('<hello>'), _class='test', _id=0)
2 <tbody id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tbody>
```

TEXTAREA

这个帮助对象创建了一个<textarea>...</textarea>标签。

```
1 >>> print TEXTAREA('<hello>', XML('<b>world</b>'), _class='test')
2 <textarea class="test" cols="40" rows="10">&lt;hello&gt;<b>world</b></textarea>
```

唯一需要注意的是，可选的“value”（“值”），将覆盖其内容（内部HTML）

```
1 >>> print TEXTAREA(value='<hello world>', _class='test')
2 <textarea class="test" cols="40" rows="10">&lt;hello world&gt;</textarea>
```

TFOOT

这用来标记表尾行。

```
1 >>> print TFOOT(TR(TD('<hello>')), _class='test', _id=0)
2 <tfoot id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tfoot>
```

TH

在表头中使用它来取代TD。

```
1 >>> print TH('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <th id="0" class="test">&lt;hello&gt;<b>world</b></th>
```

THEAD

用来标记表头行。

```
1 >>> print THEAD(TR(TH('<hello>')), _class='test', _id=0)
2 <thead id="0" class="test"><tr><th>&lt;hello&gt;</th></tr></thead>
```

TITLE

用于在HTML标头中标记页面标题。

```
1 >>> print TITLE('<hello>', XML('<b>world</b>'))
2 <title>&lt;hello&gt;<b>world</b></title>
```

TR

标记表中的一行，它应该呈现在表内，并包含<td>...</td>标签，不是TD的对象的TR参数是会被自动转换。

```
1 >>> print TR('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <tr id="0" class="test"><td>&lt;hello&gt;</td><td><b>world</b></td></tr>
```

TT

将文本标记为打字机（等宽）文本。

```
1 >>> print TT('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <tt id="0" class="test">&lt;hello&gt;<b>world</b></tt>
```

UL

标志着一个无序列表并应包含LI项目。如果其内容未标记为LI，UL会自动完成。

```
1 >>> print UL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <ul id="0" class="test"><li>&lt;hello&gt;</li><li><b>world</b></li></ul>
```

embed64

embed64(filename=None, file=None, data=None, extension=' image/gif')

将提供的数据（二进制）编码成base64数据；filename（文件名）：如果提供的话，在'rb'模式下打开并读取该文件；file（文件）：如果提供的话，读取该文件；data（数据）：如果提供的话，使用提供的数据。

xmlescape

xmlescape(data, quote=True)返回所提供数据的转义字符串。

```
1 >>> print xmlescape('<hello>')
2 &lt;hello&gt;
```

5.2.3 自定义帮助对象

TAG

有时需要生成自定义XML标签，web2py提供TAG，其为一个通用标签生成器。

```
1 {{=TAG.name('a', 'b', _c='d')}}
```

生成如下XML

```
1 <name c="d">ab</name>
```

参数"a"、"b"和"d"被自动转义，使用XML帮助对象来抑制这种行为，使用TAG可以生成API未提供的HTML/ XML标签，标签可以嵌套，并且使用str()序列化。等效语法是：

```
1 {{=TAG['name']('a', 'b', c='d')}}
```

如果创建的TAG对象名称为空，可以用它来把多个字符串和HTML帮助对象连接在一起，而不需要将它们嵌入到周围标签中，但这种用法已经过时了，使用CAT帮助对象来取代它。

请注意，TAG是一个对象，而且TAG.name或TAG['name']是一个函数，该函数返回一个临时的帮助对象类。

MENU

MENU帮助对象接收一个列表或元组的列表，形式为response.menu（正如第4章中描述的）并使用无序列表生成一个表示菜单的树状结构。例如：

```
1 >>> print MENU([[ 'One', False, 'link1'], [ 'Two', False, 'link2']])
2 <ul class="web2py-menu web2py-menu-vertical">
3 <li><a href="link1">One</a></li>
4 <li><a href="link2">Two</a></li>
```

```
5 </ul>
```

每个菜单项可以有第四个参数，它是一个嵌套的子菜单（依此递归类推）：

```
1 >>> print MENU([[ 'One', False, 'link1', [[ 'Two', False, 'link2']] ]])
2 <ul class="web2py-menu web2py-menu-vertical">
3 <li class="web2py-menu-expand">
4 <a href="link1">One</a>
5 <ul class="web2py-menu-vertical">
6 <li><a href="link2">Two</a></li>
7 </ul>
8 </li>
9 </ul>
```

MENU帮助对象用如下可选参数：

- `_class`:默认为“web2py-menu web2py-menu-vertical”并设置其它UL元素的类。
- `ul_class`:默认为“web2py-menu-vertical”并设置内部UL元素的类。
- `li_class`:默认为“web2py-menu-expand”并设置内部LI元素的类。

MENU用一个可选参数`mobile`。当设置为`True`时，它返回一个带有全部菜单选项的SELECT下拉菜单和一个重定向到与选中的选项相对应页面的`onchange`属性，而不是建立一个递归的UL菜单结构。这被设计成菜单表示的另一个选择，增进了在小型移动设备如手机上的可用性。

通常情况下，使用以下语法将菜单用在布局中：

```
1 {{=MENU(response.menu, mobile=request.user_agent().is_mobile)}}
```

在种方式下，能自动检测移动设备并且菜单被相应的呈现。

5.3 美化

BEAUTIFY用来建立复合对象的HTML表示，包括列表、元组和字典：

```
1 {{=BEAUTIFY({"a": ["hello", XML("world")], "b": (1, 2)})}}
```

BEAUTIFY返回一个类似XML的对XML可序列化的，带有一个漂亮的构造函数参数表示，在这种情况下，XML表示：

```
1 {"a": ["hello", XML("world")], "b": (1, 2)}
```

将呈现为：

```
1 <table>
2 <tr><td>a</td><td>:</td><td>hello<br />world</td></tr>
3 <tr><td>b</td><td>:</td><td>1<br />2</td></tr>
4 </table>
```

5.4 服务器端 DOM 和解析

5.4.1 elements 方法

DIV帮助对象以及所有派生的帮助对象提供了搜索方法element和elements。

element返回匹配指定条件的第一个子元素（如果没有匹配的，返回None）。

elements返回一个匹配指定条件的所有子元素的列表。

element和**elements**使用相同的语法来指定匹配条件，它允许三种混合和匹配的可能性：类jQuery表达式，确切属性值匹配，以及使用正则表达式匹配。

下面是一个简单的例子：

```
1 >>> a = DIV(DIV(DIV('a', _id='target', _class='abc')))  
2 >>> d = a.elements('div#target')  
3 >>> d[0][0] = 'changed'  
4 >>> print a  
5 <div><div><div id="target" class="abc">changed</div></div></div>
```

elements的未命名参数是一个字符串，其中可能包含：标签名称，之前为#的标签ID，之前是一个点的类，方括号中的属性显式值。

下面是4个通过id搜索以前的标签的等价方法：

```
1 >>> d = a.elements('#target')  
2 >>> d = a.elements('div#target')  
3 >>> d = a.elements('div[id=target]')  
4 >>> d = a.elements('div', _id='target')
```

下面是4个通过class搜索以前的标签的等价方法：

```
1 >>> d = a.elements('.abc')  
2 >>> d = a.elements('div.abc')  
3 >>> d = a.elements('div[class=abc]')  
4 >>> d = a.elements('div', _class='abc')
```

可以使用任何属性来定位元素（不只是id和class），其中包括多个属性（element函数可以带多个命名参数），但只返回第一个匹配的元素。

使用jQuery语法“div#target”可以指定由空格分隔的多个搜索条件：

```
1 >>> a = DIV(SPAN('a', _id='t1'), DIV('b', _class='c2'))  
2 >>> d = a.elements('span#t1, div#c2')
```

或等价的

```
1 >>> a = DIV(SPAN('a', _id='t1'), DIV('b', _class='c2'))  
2 >>> d = a.elements('span#t1', 'div#c2')
```

如果使用name参数指定属性值，它可以是一个字符串或正则表达式：

```
1 >>> a = DIV(SPAN('a', _id='test123'), DIV('b', _class='c2'))  
2 >>> d = a.elements('span', _id=re.compile('test\d{3}'))
```

DIV帮助对象的一个特殊命名参数（及其派生）是find，可以用它来在标签的文本内容中指定搜索值或搜索正则表达式。例如：

```
1 >>> a = DIV(SPAN('abcde'), DIV('fghij'))
2 >>> d = a.elements(find='bcd')
3 >>> print d[0]
4 <span>abcde</span>
```

或

```
1 >>> a = DIV(SPAN('abcde'), DIV('fghij'))
2 >>> d = a.elements(find=re.compile('fg\w{3}'))
3 >>> print d[0]
4 <div>fghij</div>
```

5.4.2 components 方法

下面是一个在HTML字符串中列出所有元素的例子：

```
1 html = TAG('<a>xxx</a><b>yyy</b>')
2 for item in html.components: print item
```

5.4.3 parent 方法

parent返回当前元素的父元素。

```
1 >>> a = DIV(SPAN('a'), DIV('b'))
2 >>> d = a.element('a').parent()
3 >>> d['_class'] = 'abc'
4 >>> print a
5 <div class="abc"><span>a</span><div>b</div></div>
```

5.4.4 flatten 方法

扁平化方法递归地将给定元素的子元素的内容序列化为普通文本（无标签）：

```
1 >>> a = DIV(SPAN('this', DIV('is', B('a'))), SPAN('test'))
2 >>> print a.flatten()
3 thisisatest
```

可以传递一个可选参数render给flatten，即一个使用不同的协议呈现/简化内容的函数，下面是一个将一些标签序列化成Markmin wiki语法的例子：

```
1 >>> a = DIV(H1('title'), P('example of a ', A('link', _href='#test')))
2 >>> from gluon.html import markmin_serializer
3 >>> print a.flatten(render=markmin_serializer)
4 # titles
5
```


6 example of `[[a link #test]]`

在写作本书的时候，我们提供`markmin_serializer`和`markdown_serializer`。

5.4.5 解析

TAG对象也是一个XML/ HTML解析器，它可以读取文本并将其转换成一个树状结构的帮助对象，这允许使用上面的API操作：

```
1 >>> html = '<h1>Title</h1><p>this is a <span>test</span></p>'
2 >>> parsed_html = TAG(html)
3 >>> parsed_html.element('span')[0]='TEST'
4 >>> print parsed_html
5 <h1>Title</h1><p>this is a <span>TEST</span></p>
```

5.5 页面布局

视图可以扩展并以树状结构包含其它视图。

例如，我们能想到的一个视图的“`index.html`”，扩展“`layout.html`”并包括“`body.html`”。同时，“`layout.html`”中可以包括“`header.html`”和“`footer.html`”。

树的根就是我们所说的布局视图。就像任何其它HTML模板文件，可以使用web2py管理界面编辑它，文件名“`layout.html`”仅仅是一个惯例。

这是一个最低限度的页面，它扩展“`layout.html`”视图并包括“`page.html`”视图：

```
1 {{extend 'layout.html'}}
2 <h1>Hello World</h1>
3 {{include 'page.html'}}
```

扩展的布局文件必须包含`{{include}}`指令，如下：

```
1 <html>
2 <head>
3 <title>Page Title</title>
4 </head>
5 <body>
6 {{include}}
7 </body>
8 </html>
```

调用视图时，扩展（布局）视图被加载，调用视图取代布局中的`{{include}}`指令，处理将继续递归进行，直到所有`extend`和`include`扩展都已被处理，生成的模板，之后被翻译成Python代码。注意，当一个应用是字节码编译时，编译的是这个Python代码，而不是原视图文件本身，因此一个给定视图的字节码编译版本是一个`single.pyc`文件，其中包含的Python代码不只是原视图文件的，而是它的整个扩展和包含视图的树。

extend、*include*、*block*和*super*是特殊模板指令，而不是Python命令。

`{{extend ...}}` 指令之前的任何内容或代码将在扩展视图的内容/代码之前被插入（因此被执行）。虽然这通常不用于在扩展视图的内容之前插入实际的HTML内容，它可以作为一种有用的手段，定义扩展视图可用的变量或函数。例如考虑一个视图“index.html”：

```
1 {{sidebar_enabled=True}}
2 {{extend 'layout.html'}}
3 <h1>Home Page</h1>
```

“layout.html”的节选：

```
1 {{if sidebar_enabled:}}
2 <div id="sidebar">
3 Sidebar Content
4 </div>
5 {{pass}}
```

因为“index.html”中的*sidebar_enabled* 赋值在*extend* 之前，该行被插入在“layout.html”开始之前，使*sidebar_enabled*在“layout.html”代码内处处可用（**welcome**应用使用了一个较为复杂的版本）。

还值得指出的是由控制器函数返回的变量不仅可用在函数的主视图中，而且可用在其所有扩展和包含视图中。

*extend*或*include*（即扩展或包含的视图名称）的参数可以是一个Python变量（但不能是Python表达式）。然而，这会带来一个限制——在*extend*或*include*语句中使用变量的视图不能字节码编译，正如上面提到的，字节码编译的视图包括整个扩展和包含视图的树，因此具体的扩展及包含视图必须在编译时已知，如果视图名称是变量（其值直到运行时才能确定），这是不可能的，由于字节码编译视图可以提供显著的速度提升，应尽量避免在*extend*和*include*中使用变量。

在某些情况下，在*include*中使用变量的一个替代是简单地把`{{include ...}}` 指令放在 `if ... else`块内。

```
1 {{if some_condition:}}
2 {{include 'this_view.html'}}
3 {{else:}}
4 {{include 'that_view.html'}}
5 {{pass}}
```

字节码编译上面的代码不存在任何问题，因为未涉及任何变量。但是注意，字节码编译的视图实际上包括“this_view.html”和“that_view.html”的Python代码，尽管只有其中一个视图的代码被执行，其取决于*some_condition*的值。

记住，这仅适用于*include* ——不能把`{{extend ...}}`指令放在*if...else*块内。

布局用于封装页面共性（页眉，页脚，菜单），虽然它们不是强制性的，它们将使应用更容

易编写和维护。特别是，我们建议编写利用以下可以在控制器中设置的变量的布局，使用这些熟悉的变量将有助于使布局可互换：

```
1 response.title
2 response.subtitle
3 response.meta.author
4 response.meta.keywords
5 response.meta.description
6 response.flash
7 response.menu
8 response.files
```

除了menu和files，这些都是字符串，它们的含义应该是显而易见的。

response.menu菜单是一个3元组或4元组列表。三个要素是：链接名称、一个表示链接是否处于活动状态（是否是当前链接）的布尔值和所链接页面的URL。例如：

```
1 response.menu = [('Google', False, 'http://www.google.com', []),
2 ('Index', True, URL('index'), [])]
```

第四个元组元素是一个可选的子菜单。

response.files是页面需要的CSS 和 JS文件的列表。

我们还建议使用：

```
1 {{include 'web2py_ajax.html'}}
```

在HTML头中，因为这将包括jQuery库并为特殊效果和Ajax定义一些向后兼容的JavaScript函数。“web2py_ajax.html”包括视图中的response.meta标签、jQuery基础、日历日期选择器以及所有必需的CSS和JS response.files。

5.5.1 默认页面布局

以下web2py基本构建应用welcome附带的“views/layout.html”（剥离一些可选部分），任何新应用都有类似的默认布局：

```
1 <!DOCTYPE html>
2 <head>
3     <meta charset="utf-8" />
4     <title>{{=response.title or request.application}}</title>
5
6     <!-- http://dev.w3.org/html5/markup/meta.name.html -->
7     <meta name="application-name" content="{{=request.application}}" />
8
9     <script src="{{=URL('static', 'js/modernizr.custom.js')}}" /></script>
10
11     <!-- include stylesheets -->
12     {{
13         response.files.append(URL('static', 'css/skeleton.css'))
```

```

14     response.files.append(URL('static', 'css/web2py.css'))
15     response.files.append(URL('static', 'css/superfish.css'))
16     response.files.append(URL('static', 'js/superfish.js'))
17 }}
18
19 {{include 'web2py_ajax.html'}}
20
21 <script type="text/javascript">
22 jQuery(function() { jQuery('ul.sf-menu').supersubs({minWidth:12,maxWidth:30,
23 extraWidth:3}).superfish(); });
24 </script>
25
26 {{
27 # using sidebars need to know what sidebar you want to use
28 left_sidebar_enabled = globals().get('left_sidebar_enabled',False)
29 right_sidebar_enabled = globals().get('right_sidebar_enabled',False)
30 middle_columns = {0:'sixteen',1:'twelve',2:'eight'}[
31 (left_sidebar_enabled and 1 or 0)+(right_sidebar_enabled and 1 or 0)]
32 }}
33 </head>
34 <body>
35     <div class="wrapper"><!-- for sticky footer -->
36
37         <div class="topbar">
38             <div class="container">
39                 <div class="sixteen columns">
40                     <div id="navbar">
41                         {{='auth' in globals() and auth.navbar(separators=(' ',' / ',''))}}
42                     </div>
43                     <div id="menu">
44                         {{=MENU(response.menu,
45                             _class='mobile-menu' if is_mobile else
46                             'sf-menu',
47                             mobile=request.user_agent().is_mobile)}}
48                     </div>
49                 </div>
50             </div><!-- topbar -->
51
52 <div class="flash">{{=response.flash or ''}}</div>
53
54 <div class="header">
55     <div class="container">

```

```

56     <div class="sixteen columns">
57         <h1 class="remove-bottom" style="margin-top: .5em;">
58             {{=response.title or request.application}}
59         </h1>
60         <h5>{{=response.subtitle or ''}}</h5>
61     </div>
62
63     <div class="sixteen columns">
64         <div class="statusbar">
65             {{block statusbar}}
66             <span class="breadcrumbs">{{=request.function}}</span>
67             {{end}}
68         </div>
69     </div>
70 </div>
71 </div>
72
73 <div class="main">
74     <div class="container">
75         {{if left_sidebar_enabled:}}
76         <div class="four columns left-sidebar">
77             {{block left_sidebar}}
78             <h3>Left Sidebar</h3>
79             <p></p>
80             {{end}}
81         </div>
82         {{pass}}
83
84         <div class="{{=middle_columns}} columns center">
85             {{block center}}
86             {{include}}
87             {{end}}
88         </div>
89
90         {{if right_sidebar_enabled:}}
91         <div class="four columns">
92             {{block right_sidebar}}
93             <h3>Right Sidebar</h3>
94             <p></p>
95             {{end}}
96         </div>
97         {{pass}}
98
99     </div><!-- container -->
100 </div><!-- main -->
101

```

```

102     <div class="push"></div>
103 </div><!-- wrapper -->
104
105 <div class="footer">
106     <div class="container header">
107         <div class="sixteen columns">
108             {{block footer}} <!-- this is default footer -->
109             <div class="footer-content" >
110                 {{=T('Copyright')}} &#169; 2011
111                 <div style="float: right;">
112                     <a href="http://www.web2py.com/">
113                         
115                     </a>
116                 </div>
117             </div>
118             {{end}}
119         </div>
120     </div><!-- container -->
121 </div><!-- footer -->
122
123 </body>
124 </html>

```

缺省布局有一些特征使它很容易使用和定制：

- 这是用HTML5编写的并为向后兼容性使用“modernizr” [49]库，实际的布局包括一些IE浏览器要求的额外的条件语句，为了简洁起见省略了它们。
- 它显示可以设置在模型中的response.title和response.subtitle，如果它们都没有被设置，采用应用名作为标题。
- 它包含web2py_ajax.html文件到其头部，该文件生成所有链接和脚本导入语句。
- 为了灵活布局，它使用“skeleton” [50]的一个修改版本，它能在移动设备上运行并重新排列以适合小型屏幕。
- 它为动态级联菜单使用“superfish.js”，有一个明确的脚本来激活superfish级联菜单并且如果没有必要的话可以删除它。
- {{= auth.navbar (...)}}显示欢迎当前用户以及身份验证函数链接，如登录、注销、注册、更改密码等，取决于上下文。这是一个 helper工厂并且它的输出像其它任何帮助对象一样操作，被放置在{{try:}}...{{except:pass}}中，以防auth未定义。
- {{=MENU(response.menu)}}显示菜单结构为...。

- 呈现页面时，`{{include}}`将被扩展视图的内容替换。
- 默认情况下它采用的是有条件的三列（左边和右边的侧边栏可以被扩展视图关闭）
- 它使用以下类：header、main、footer
- 它包含以下功能块：statusbar、left_sidebar、center、right_sidebar、footer。

如下视图可以打开并填写侧边栏：

```
1 {{left_sidebar_enable=True}}
2 {{extend 'layout.html'}}
3
4 This text goes in center
5
6 {{block left_sidebar}}
7 This text goes in sidebar
8 {{end}}
```

5.5.2 定制的默认布局

自定义默认布局是很容易的，因为CSS文件被记录：

- “skeleton.css”包含复位、网格布局、表单样式
- “web2py.css”包含特定的web2py样式
- “superfish.css”包含菜单样式

要改变颜色和背景图片，只需要将以下的代码追加到“web2py.css”：

```
1 body { background: url('images/background.png') repeat-x #3A3A3A; }
2 a { color: #349C01; }
3 .header h1 { color: #349C01; }
4 .header h2 { color: white; font-style: italic; font-size: 14px;}
5 .statusbar { background: #333333; border-bottom: 5px #349C01 solid; }
6 .statusbar a { color: white; }
7 .footer { border-top: 5px #349C01 solid; }
```

菜单以中性色彩方式建立，但可以改变这一点。

当然，也可以用自己的文件完全替代“layout.html”和“web2py.css”。

5.5.3 移动部署

默认layout.html的设计是移动设备友好的，但那还不够。当页面被移动设备访问时，可能需要使用不同的视图。

为了使开发台式机和移动设备更容易，web2py中包含@mobilize装饰器，这个装饰器应用于

应该有正常的视图和移动视图的动作。如下说明这一点：

```
1 from gluon.contrib.user_agent_parser import mobilize
2 @mobilize
3 def index():
4     return dict()
```

注意，在控制器中它被使用之前，装饰器一定是重要的。“index”函数被普通浏览器（台式计算机）调用时，web2py会使用“[controller]/index.html”视图呈现返回的字典，然而当移动设备调用它时，该字典将被“[controller]/index.mobile.html”呈现。注意移动视图有“mobile.html”扩展名。

另外，可以使用下面的逻辑来使所有视图友好移动：

```
1 if request.user_agent().is_mobile:
2     response.view.replace('.html', '.mobile.html')
```

创建“*.mobile.html”的任务留给了开发者，但我们强烈建议使用“jQuery Mobile”插件，这使得任务很容易完成。

5.6 视图中的函数

考虑“layout.html”：

```
1 <html>
2     <body>
3         {{include}}
4         <div class="sidebar">
5             {{if 'mysidebar' in globals():}} {{mysidebar()}} {{else:}}
6                 my default sidebar
7             {{pass}}
8         </div>
9     </body>
10 </html>
```

与此扩展视图

```
1 {{def mysidebar():}}
2 my new sidebar!!!
3 {{return}}
4 {{extend 'layout.html'}}
5 Hello World!!!
```

注意函数定义在{{extend...}} 语句之前——这会导致函数在“layout.html”执行之前被创建，所以可以在“layout.html”内的任意位置调用函数，甚至在{{include}}之前。还要注意函数包含在扩展视图中而无需=前缀。

代码生成如下输出：

```
1 <html>
2     <body>
3         Hello World!!!
```

```

4         <div class="sidebar">
5             my new sidebar!!!
6         </div>
7     </body>
8 </html>

```

注意，该函数定义在HTML中（尽管它可能还包含Python代码）所以Response.Write用来编写它的内容（函数不返回该内容）。这就是为什么布局使用{{mysidebar()}}而不是{{=mysidebar()}}调用视图，以这种方式定义的函数可以使用参数。

5.7 视图中的块

使视图更加模块化的另一种方法是通过使用{{block...}}s并且这种机制是上一节讨论的机制的一种替代。

考虑“layout.html”：

```

1 <html>
2     <body>
3         {{include}}
4         <div class="sidebar">
5             {{block mysidebar}}
6             my default sidebar
7         {{end}}
8     </div>
9 </body>
10 </html>

```

与此扩展视图

```

1 {{extend 'layout.html'}}
2 Hello World!!!
3 {{block mysidebar}}
4 my new sidebar!!!
5 {{end}}

```

它生成的输出如下：

```

1 <html>
2     <body>
3         Hello World!!!
4         <div class="sidebar">
5             my new sidebar!!!
6         </div>
7     </body>
8 </html>

```

可以有許多塊，並且如果被擴展視圖中有塊，但擴展視圖中沒有，被擴展視圖的內容被使用。

此外，注意到其不像函数，没有必要在`{{extend ...}}`之前定义块——即使定义在`extend`之后，可以用它们来在被扩展视图的任意位置创建替换。

在块内，可以使用表达式`{{super}}`来包含父内容。例如，如果我们将上述被扩展视图替换为：

```
1 {{extend 'layout.html'}}
2 Hello World!!!
3 {{block mysidebar}}
4 {{super}}
5 my new sidebar!!!
6 {{end}}
```

我们得到：

```
1 <html>
2     <body>
3         Hello World!!!
4         <div class="sidebar">
5             my default sidebar
6             my new sidebar!!!
7         </div>
8     </body>
9 </html>
```

第 6 章 数据库抽象层

6.1 依赖性 dependencies

web2py提供作为数据库抽象层（DAL）的API（应用程序编程接口），把Python对象映射成为数据库对象，比如：查询、表和记录对象等。DAL使用指定语言实时动态地为数据库后端生成SQL语句，以便你不必写SQL代码或者学习不同的SQL语言（SQL是通用的术语），并且应用程序可以在不同类型的数据库之间移植。在撰写本书的时候，支持的数据库是SQLite（Python支持，web2py也一样）、PostgreSQL、MySQL、Oracle、MSSQL、FireBird、DB2、Informix、Ingres 和 Google App Engine（SQL 和 NoSQL）（部分支持）。事实上我们能支持更多的数据库，请查阅web2py网站和邮件列表获得更多最新的改进，Google NoSQL将作为一个特别的案例在第13章中加以讨论。

web2py的Windows二进制发行版直接用于SQLite和MySQL，其Mac二进制发行版直接用于SQLite，要使用其他任意一个数据库后端，从源码发行版开始运行并为所需要的后端安装合适的驱动。

一旦合适的驱动安装以后，从源文件启动web2py，它会找到驱动。以下是驱动列表：

database	driver (source)
SQLite	sqlite3 or pysqlite2 or zxJDBC [58] (on Jython)
PostgreSQL	psycopg2 [59] or zxJDBC [58] (on Jython)
MySQL	pymysql [60] or MySQLdb [61]
Oracle	cx_Oracle [62]
MSSQL	pyodbc [63]
FireBird	kinterbasdb [64]
DB2	pyodbc [63]
Informix	informixdb [65]
Ingres	ingresdbi [66]

（pymysql 随着web2py）web2py 定义了以下的类来构成DAL：

DAL 代表一个数据库连接。例如：

```
1 db = DAL('sqlite://storage.db')
```

Table代表了数据库的表。不能直接实例化表对象，而是用DAL.define_table实例化它。

```
1 db.define_table('mytable', Field('myfield'))
```

Table对象最重要的一些方法是：`.insert`、`.truncate`、`.drop` 和 `.import_from_csv_file`。

Field是数据库的一个字段。它能被实例化并作为参数传递给db.define_table。

DAL Rows 是一个数据库select（查询）方法返回的对象。它可以被认为是Row对象若干行的列表：

```
1 rows = db(db.mytable.myfield!=None).select()
```

Row对象包含字段数值。

```
1 for row in rows:
```

```
2     print row.myfield
```

Query是代表SQL “ where ” 条件的一个对象。

```
1 myquery = (db.mytable.myfield != None) | (db.mytable.myfield > 'A')
```

Set是表示记录集合的对象。它最重要的方法有：`count`、`select`、`update`和`delete`。例如：

```
1 myset = db(myquery)
2 rows = myset.select()
3 myset.update(myfield='somevalue')
4 myset.delete()
```

Expression有点像`orderby`或`groupby`的表达式。`Field`类从`Expression`继承，这里有一个例子。

```
1 myorder = db.mytable.myfield.upper() | db.mytable.id
2 db().select(db.table.ALL, orderby=myorder)
```

6.2 连接字符串

与数据库的连接是通过创建一个 DAL 对象实例来建立的。

```
1 >>> db = DAL('sqlite://storage.db', pool_size=0)
```

`db` 不是一个关键词，它是一个存储连接对象 DAL 的局部变量，也可以给它取个另外不同的名字。DAL 构造函数需要单个连接字符串的参数，连接字符串是仅有的依赖特定后端数据库的 web2py 代码。下表中是支持一些特定类型后端数据库连接字符串的例子（所有情况下，我们假定数据库在本地主机上运行采用默认端口并命名为 `test`）：

SQLite	sqlite://storage.db
MySQL	mysql://username:password@localhost/test
PostgreSQL	postgres://username:password@localhost/test
MSSQL	mssql://username:password@localhost/test
FireBird	firebird://username:password@localhost/test
Oracle	oracle://username/password@test
DB2	db2://username:password@test
Ingres	ingres://username:password@localhost/test
Informix	informix://username:password@test
Google App Engine/SQL	google:sql
Google App Engine/NoSQL	google:datastore

注意 SQLite 数据库只由一个文件构成，如果不存在，它会被创建，每次被访问的时候该文件会被锁定。就 MySQL, PostgreSQL, MSSQL, FireBird, Oracle, DB2, Ingres 和 Informix 情况，数据库“test”需在 web2py 外创建，一旦连接建立，web2py 会适当地创建、修改和删除表。

也可以把连接字符串设置为 `None`，这种情况下，DAL 将不连接到任何一个后端数据库，但是 API 能被测试所访问。这类例子将在第七章中讨论。

6.2.1 连接池

DAL 构造函数的第二个参数是 `pool_size`，它的默认值是 0。

因为，为每个请求建立一个新的数据库连接是会相当慢的，因此 web2py 对连接池采用了一种机制，即一旦一个连接建立了，页面被服务了并且事务处理完成，连接不被关闭而是转到连接池里。当下一个 http 请求到达，web2py 尝试从连接池里获得连接并为新的事务处理所用，如果在连接池中没有可用连接，新的连接会被建立。

`pool_size` 参数在 SQLite 和 Google App Engine 中忽略。

池中的连接在线程中依次共享，在这个意义上，可以被两个不同但不是并发的线程使用，每个web2py进程仅有一个连接池。

当web2py启动，连接池总是空的。连接池可增长至pool_size的值与当前最大并发请求数间的最小值，这就意味着如果pool_size=10而我们的服务器接收的并发请求数从未超过5次，那么实际的连接池大小就不会超过5，如果pool_size=0那么连接池没有被使用。

连接池机制在SQLite中忽略，因为它不能产生任何效益。

6.2.2 连接失败

如果 web2py 连接数据库失败，它等待 1 秒后最多尝试 5 次后宣布失败。在连接池情形下，可能存在池中连接保持打开但一段时间不用而被数据库终端关闭的情况，得益于重试功能，web2py 可以再建立这些丢弃的连接。

当使用连接池一个连接被使用以后，重新放回池中然后再循环。可能存在当池中连接空闲但连接被数据库服务器关闭，这可能是由于故障或超时造成。这种情况出现时，web2py 进行检测并重新建立连接。

6.2.3 复制的数据库 replicated database

DAL (...) 第一个参数可以是 URI (统一资源标识) 的列表。这种情况下，web2py 尝试连接它们中每一个，这样做的主要意图是应对多数据库服务器和在多服务器之间的负荷分担，下面是一个典型应用示例：

```
1 db = DAL(['mysql://...1','mysql://...2','mysql://...3'])
```

这个例子中 DAL 尝试连接第一个数据库，如果失败了，它会连接第二个、第三个，这同样可以应用于数据库负荷分担的主从配置。我们会在第 13 章就扩展性进行更多的讨论。

6.3 保留关键词

还有另外一个参数可以传递给 DAL 的构造函数，其用来检查表名和字段名是否违反目标后端数据库的 SQL 保留关键字。

这个参数是 check_reserved，它的默认值是 None。

这是一个包含数据库后端适配器名字的字符串列表。

适配器名字与在 DAL 中使用的连接字符串一样，因此如果你想检查是否与 PostgreSQL 和 MSSQL 冲突，那么你的连接字符串可以采用下面形式：

```
1 db = DAL('sqlite://storage.db',
2 check_reserved=['postgres', 'mssql'])
```

DAL 会采用与列表同样的顺序检查关键字。

会有两个额外的选项 “all” (全部) 和 “common” (通常)，如果你指定 all，它会检查是否与所有知道的 SQL 关键字冲突；如果你指定 common，它仅仅检查通常的 SQL 关键字，比如 SELECT、INSERT 和 UPDATE 等等。

为支持后台数据库，也可指定是否检查与非保留字冲突，这种情形可以追加 nonreserved 在名字后。例如：

```
1 check_reserved=['postgres', 'postgres_nonreserved']
```

下面列出的数据库后端支持保留字检查。

PostgreSQL	postgres(_nonreserved)
MySQL	mysql
FireBird	firebird(_nonreserved)
MSSQL	mssql
Oracle	oracle

6.4 DAL、Table 和 Field

要理解掌握 DAL API 最好的方法是亲自尝试每一个函数。

这可以通过 web2py shell 交互来完成，但是最终 DAL 代码要写入模块和控制器。

以创建一个连接开始。为了更好的效果，可以用 SQLite 数据库。

当更换后台引擎，所讨论的并不会有任何的改变。

```
1 >>> db = DAL('sqlite://storage.db')
```

现在数据库连接上了，并且连接存在全局变量 db 中。

任何时候都可以得到连接字符串。

```
1 >>> print db._uri
```

```
2 sqlite://storage.db
```

还有数据库的名字

```
1 >>> print db._dbname
```

```
2 sqlite
```

连接字符串被称之为 _uri，因为它是 Uniform Resource Identifier 的缩写。

DAL 允许与同一个数据库的多个连接或者与不同数据库连接，甚至是不同类型的多个数据库。目前，我们假定是单个数据库，这情形是最常见的。

DAL 最重要的方法是 define_table。

```
1 >>> db.define_table('person', Field('name'))
```

它定义、存储并返回一个叫 “person” 的 **Table** 对象包含字段（列）“name”，这个对象也可以通过 db.person 访问，因此无需抓住返回值。

不要声明字段名 “id”，因为这个已被 web2py 创建。

默认情况下，每个表格都有一个 “id” 字段。它是一个自增整数字段（从 1 开始），用来交叉引用并使每个记录独一无二，因此 id 是主键（primary key）。（备注：id 从 1 开始是后端指定的，例如这在 Google App Engine NoSQL 就不适用。）

可以选择定义一个字段类型 type='id'，web2py 会用这个字段作为自增 id 字段，不推荐这样使用，除非访问传统数据库表，也可以使用不同的主键但会有一些限制，这些会在传统数据库和键表章节讨论。

6.5 记录表示

指定记录表示格式虽是可选但我们推荐。

```
1 >>> db.define_table('person', Field('name'), format='% (name)s')
```

或

```
1 >>> db.define_table('person', Field('name'), format='% (name)s %(id)s')
```

或者更为复杂的可以采用函数

```
1 >>> db.define_table('person', Field('name'),
2     format=lambda r: r.name or 'anonymous')
```

使用格式属性有两个目的：

- 为了在 select /option 中采用下拉框表示参考记录
- 为引用到这张表的所有字段设置 db.othertable.person.represent 属性，这意味着

SQLTABLE (SQL 的表) 不会以 id 显示引用, 而是采用首选的表示格式。

以下是一个 Field 字段构造函数的默认值。

```
1 Field(name, 'string', length=None, default=None,
2       required=False, requires='<default>',
3       ondelete='CASCADE', notnull=False, unique=False,
4       uploadfield=True, widget=None, label=None, comment=None,
5       writable=True, readable=True, update=None, authorize=None,
6       autodelete=False, represent=None, compute=None,
7       uploadfolder=os.path.join(request.folder, 'uploads'),
8       uploadseparate=None)
```

并非所有都与每个字段有关, “length” (长度) 只与 “string” (字符串) 类型的字段有关, “uploadfield” 和 “authorize” 只与 “upload” 类型的字段有关。 “ondelete” 只与 “reference” 和 “upload” 类型的字段有关。

- length 指定了 “string” 、 “password” 或 “upload” 字段的最大长度。如果 length 不指定被使用的默认值, 但默认值不能够保证后向兼容, 为了避免不想要的升级迁移, 我们推荐一定要指定 string、 password 或 upload 字段的长度。
- default 给字段设定默认值。默认值可以在执行插入操作的时, 如果没有一个明确指定的参数值时使用, 也可以用来预填充表单, 这些表单从使用 SQLFORM 的表编译得到。注意, 不单是固定的值, 默认值可以是函数 (包括 lambda 函数), 能够为字段返回合适类型的值, 那种情形下, 一旦每条记录插入就调用一次函数, 即便是多条记录插入到单个事务也一样。
- required 告知 DAL, 如果字段没有明确指定值, 那么插入这张表是不允许的。
- requires 是一个验证器或是一个验证器列表。它不被 DAL 使用, 但被 SQLFORM 使用, 给定类型的默认验证器在下表中列出:

field type	default field validators
string	IS_LENGTH(length) default length is 512
text	IS_LENGTH(65536)
blob	None
boolean	None
integer	IS_INT_IN_RANGE(-1e100, 1e100)
double	IS_FLOAT_IN_RANGE(-1e100, 1e100)
decimal(n,m)	IS_DECIMAL_IN_RANGE(-1e100, 1e100)
date	IS_DATE()
time	IS_TIME()
datetime	IS_DATETIME()
password	None
upload	None
reference <table>	IS_IN_DB(db,table.field,format)
list:string	None
list:integer	None
list:reference <table>	IS_IN_DB(db,table.field,format,multiple=True)

十进制字段类型要求并返回 Decimal (十进制) 对象的值, 在 Python decimal 十进制模块中定义, SQLite 不处理十进制类型, 因此我们在内部把它处理为 double 类型, (n, m) 分别是数字个数的总数和十进制小数点以后的数字的个数。

- list: 特殊字段, 它们被设计来使用 NoSQL 某些非规范功能 (Google App Engine NoSQL 情况下, ListProperty 和 StringListProperty 字段类型), back-port 所有其它能够支持的相关的数据库。在关系数据库列表上作为一个文本字段形式存储, 条目用符号|分开, 每个字符串中的分隔符|转义为||。它们在各自的章节讨论。

注意, requires=... 在表单级是强制的, required=True 在 DAL (插入) 级是强制的, 而 notnull、 unique 和 ondelete 在数据库级是强制的, 虽然有时它们看起来冗余, 但用 DAL 编程时保持它们的区别是很重要的。

- `ondelete` 翻译为 “ON DELETE” SQL 语句。默认时它的值是 “CASCADE”（层叠），它告知数据库当删除一条记录时要删除引用到这条记录的所有记录。要关闭这项功能，把 `ondelete` 设置为 “NO ACTION” 或 “SET NULL”。
- `notnull=True` 翻译为 “NOT NULL” SQL 语句。它能防止数据库的字段插入空的值。
- `unique=True` 翻译为 “UNIQUE” SQL 语句，它确保这个字段的值在表内是独特的，在数据库级是强制的。
- `uploadfield` 仅用在 “upload” 类型字段。一个 “upload” 类型字段存储了一个在其他地方保存的文件的名字，默认在文件系统的应用程序 “uploads/” 文件夹下。如果 `uploadfield` 被设置，那么文件存在同名表所在的区域，`uploadfield` 的值是该区域的名字。稍后这将在 SQLFORM 相关内容中更加详细地讨论。
- `uploadfolder` 默认指应用程序 “uploads/” 文件夹。如果设定为不同的路径，文件会上传至不同的文件夹。例如
`uploadfolder=os.path.join(request.folder, static/temp)` 文件将会被上传至 `web2py/applications/myapp/static/temp` 文件夹。
- `uploadseparate` 如果设置为 `True` 上载文件到 `uploadfolder` 文件夹的不同子文件夹。这样可以优化以避免太多的文件在同样的文件夹/子文件夹中，警告：不能在不中断系统的情况下把 `uploadseparate` 的值从 `True` 改为 `False`，`web2py` 或采用分离的子文件夹或不采用。在文件被上载后，修改的动作会阻止 `web2py` 获得那些文件，如果这种情况发生，可以移动文件并修复问题，但不在这里说明。
- `widget` 必须是可用的 `widget` 对象之一，包括客户 `widget`，例如：SQLFORM.widgets.string.widget，一些可用的 `widget` 会在后面讨论。每个字段类型都有一个默认的 `widget`。
- `label` 是一个字符串（或者是能被序列化为字符串的内容），包含为这个字段在自动生成表单中所用的标识。
- `comment` 是一个字符串（或者是能被序列化为字符串的内容），包含一个与这个字段相关的评价，会显示在在自动生成的表单输入区域的右边。
- `writable` 如果一个字段是可写属性，它能在自动生成中创建和修改表单中被编辑。
- `readable` 如果一个字段是可读属性，它指在只读的表单中可见，如果一个字段既不可读也不可写，它不会在创建和修改表单的时候显示。
- `update` 包含了当记录被修改时这个字段的默认值。
- `compute` 是一个可选函数。如果一条记录被插入或者修改，`compute` 函数将被执行，并且这个字段会填充函数执行结果，记录作为一个 `dict` 传递给 `compute` 函数，`dict` 不含当前值或任何其它 `compute` 字段。
- `authorize` 用于需要访问控制相关的字段，仅 “upload” 字段，这会在鉴权和授权内容部分中更详细的讨论。
- `autodelete` 决定当记录所参考的文件被删除时，相应的上传文件是否被删除，仅为 “upload” 字段。
- `represent` 可以是 `None` 或是一个函数，这个函数得到字段值并返回字段值的替换表示，例如：

```

1 db.mytable.name.represent = lambda name,row: name.capitalize()
2 db.mytable.other_id.represent = lambda id,row: row.myfield
3 db.mytable.some_uploadfield.represent = lambda value,row: \
4     A('get it', _href=URL('download', args=value))

```

- `Blob` 字段也是特殊的。默认情况下，二进制数据在被存储到确切的数据库字段前是 `base64` 编码的，当取出的时候被解码，虽然这会产生比 `blob` 字段本身所需存储空间多占用 25% 的副作用，但它又两个优势。平均来说它减少了 `web2py` 和数据库服务器之间数据通信量，而且它使得通信独立于 `back-end-specific escaping`

conventions. 传统。

大多数字段和表的属性可以在它们被定义以后进行修改。

```
1 db.define_table('person', Field('name', default=''), format='% (name)s')
2 db.person._format = '% (name)s/% (id)s'
3 db.person.name.default = 'anonymous'
```

（注意表的属性通常用下划线做前缀以避免可能与字段名冲突。）

列出给定的数据库连接定义的表：

```
1 >>> print db.tables
2 ['person']
```

列出给定表中已经定义的字段：

```
1 >>> print db.person.fields
2 ['id', 'name']
```

查询表的类型：

```
1 >>> print type(db.person)
2 <class 'gluon.sql.Table'>
```

通过 DAL 连接应用访问表：

```
1 >>> print type(db['person'])
2 <class 'gluon.sql.Table'>
```

类似地可以使用多种等价的办法通过名字访问字段：

```
1 >>> print type(db.person.name)
2 <class 'gluon.sql.Field'>
3 >>> print type(db.person['name'])
4 <class 'gluon.sql.Field'>
5 >>> print type(db['person']['name'])
6 <class 'gluon.sql.Field'>
```

给定字段，访问定义时设置的属性：

```
1 >>> print db.person.name.type
2 string
3 >>> print db.person.name.unique
4 False
5 >>> print db.person.name.notnull
6 False
7 >>> print db.person.name.length
8 32
```

包括它的父表、表名和父连接：

```
1 >>> db.person.name._table == db.person
2 True
3 >>> db.person.name._tablename == 'person'
4 True
5 >>> db.person.name._db == db
6 True
```

字段也有方法。其中一些用于生成查询，我们将在后面看到，字段对象的一个特殊方法是validate，被称之为字段的验证器。

```
1 print db.person.name.validate('John')
```

它返回一个元组 (value, error) (值、错误)，如果输入通过验证器，error的值是None。

6.6 迁移

`define_table` 函数检查是否有相应的表存在。如果不存在，它生成 SQL 创建表并执行 SQL；如果表存在但与已定义的不同，它生成 SQL 来重构表并执行；如果一个字段改变类型但没有更名，它会进行数据转换（如果你不想这样，你需要定义表两次。第一次，让 web2py 删除该字段并清除它；第二次，添加新定义的由 web2py 创建的字段）；如果表存在并且与当前的定义匹配，就不做任何处理。所有的情况下，该函数创建 `db.person` 对象表示表。

我们定义这样的举动为“migration”（迁移），web2py 对所有的迁移和迁移尝试记录日志在“`databases/sql.log`”文件中。

`define_table` 的第一个参数总是表名。其它没有名字的参数是字段（Field），该函数最后有一个可选的名为“migrate”的参数，这个参数必须用名字显式引用：

```
1 >>> db.define_table('person', Field('name'), migrate='person.table')
```

迁移的返回值是文件名（在“databases”应用程序文件夹中），web2py 为该表存储了内部迁移的信息。这些文件非常重要，绝不能移走除非整个数据库删除了，这种情形下，“.table”文件需手动删除，默认的 migrate 的值是 True，这让 web2py 通过连接字符串的散列产生文件名。如果 migrate 设置为 False，迁移是不能执行的，web2py 假定表以数据存储存在，它包含（至少）`define_table` 列出的字段。最佳实践是给迁移表一个清晰的名字。

在同一个应用中两张表有相同的迁移名字的可能性不大。

DAL 类也用“migrate”参数，它决定了迁移调用 `define_table` 的默认参数值。例如，

```
1 >>> db = DAL('sqlite://storage.db', migrate=False)
```

设定迁移默认参数值为 False，无论任何时候，调用 `db.define_table` 无迁移参数。

对所有的表在连接的时候迁移可以被禁用。

```
1 db = DAL(..., migrate_enabled=False)
```

当两个应用共享一个数据库时，推荐使用这样的操作。两个应用中的一个执行迁移，另一个禁用迁移。

6.7 修复损坏迁移

迁移时存在两个常见的问题，有办法恢复它们。

一个是 SQLite 特有的，SQLite 不强制列的类型也不能删除列。这意味着如果字符串类型的类删除了，但它并没有被真正地删除；如果添加了一个不同类型的列（比如日期型），日期型的列却包含字符串结束（实际中是无用数据），web2py 不会报错，是因为它不知道数据库里面的是什么，直到它检索记录失败。

当查询记录 web2py 在 `gluon.sql.parse` 函数中返回出错，这是问题的症结：由于上述问题导致的列数据损坏。

解决问题的方法是更新表的所有记录并且用 None 更新列中有问题的值。

另一个问题更常见也是 MySQL 的典型问题，MySQL 在一个事务中不允许多于一张 ALTER TABLE（重构表）。这意味着 web2py 必须把复杂的事务分解成比较小的（一张 ALTER TABLE 一次），且一次提交一张，这使得复杂的事务能被提交处理，也造成了部分失败，让 web2py 陷入毁坏的状态。为什么一个事务的部分会失败呢？因为，举例来说，web2py 试图转换数据，它牵扯到重构一张表，把字符串列转换为时期类型，但数据没能够转换。web2py 会发生什么呢？它根本不清楚存在数据库中表的结构到底是什么样的。

解决的办法在于禁止所有表的迁移并允许假迁移。

```
1 db.define_table(..., migrate=False, fake_migrate=True)
```

这会根据表的定义重建关于表的 web2py 元数据，尝试多种表的定义看看哪一种符合（迁移失败之前的和迁移失败之后的）。一旦成功，删除 fake_migrate=True 属性。

在尝试修复迁移问题之前，复制 “applications/yourapp/databases/*.table” 文件是谨慎的做法。

对所有的表，迁移问题也可一次就修复了。

```
1 db = DAL(..., fake_migrate_all=True)
```

但是如果失败了，不会有助于缩小问题范围。

6.8 插入方法 insert

给定一张表，像这样插入记录

```
1 >>> db.person.insert(name="Alex")
2 1
3 >>> db.person.insert(name="Bob")
4 2
```

插入操作返回每个插入记录的唯一 id。

截断表格，例如删除所有记录复位 id 计数。

```
1 >>> db.person.truncate()
```

现在，如果再次插入一条记录，计数器又从 1 开始（这是后台特定的但不适用于 GoogleNoSQL）：

```
1 >>> db.person.insert(name="Alex")
2 1
```

web2py 同时提供了 bulk_insert 块插入的方法。

```
1 >>> db.person.bulk_insert([{'name': 'Alex'}, {'name': 'John'}, {'name': 'Tim'}])
2 [3, 4, 5]
```

它需要列出要插入记录的字段并且一次完成多条记录的插入，它返回所插入记录的 ID 值。

对于所支持的关系型数据库，使用这个函数与采用循环单条记录插入相比没有什么优势，但对于 Google App Engine NoSQL，它会有较大的速度优势。

6.9 提交和回滚 commit and rollback

除非你执行了 commit（提交）命令，否则 creat（创建）、drop（删除）、insert（插入）、truncate（截断）、delete（删除）和 update（更新）操作都不会被执行。

```
1 >>> db.commit()
```

为了检验，让我们插入一条新记录：

```
1 >>> db.person.insert(name="Bob")
2 2
```

执行回滚，例如从上次提交后忽略所有的操作：

```
1 >>> db.rollback()
```

如果现在再次插入，计数器再次被设置为 2，因为上次的插入做了回滚操作。

```
1 >>> db.person.insert(name="Bob")
2 2
```


模块、视图和控制器中的代码被封装，在web2py代码中看起来是像这样：

```
1 try:
2     execute models, controller function and view
3 except:
4     rollback all connections
5     log the traceback
6     send a ticket to the visitor
7 else:
8     commit all connections
9     save cookies, sessions and return the page
```

在 web2py 中无需显式调用 commit 和 rollback，除非想要更粒度的控制。

6.10 原始的 sql

6.10.1 定时查询 Timing queries

所有的查询被web2py自动定时，变量db._timings是一个元组列表，每个元组包含原始的SQL查询，其传递给数据库驱动以及以秒为单位执行所需的时间。这个参数能用工具条在视图中显示。

```
1 {{=response.toolbar()}}
```

6.10.2 executesql 方法

DAL允许显式发布SQL语句。

```
1 >>> print db.executesql('SELECT * FROM person;')
2 [(1, u'Massimo'), (2, u'Massimo')]
```

这种情况下，DAL对返回值不做语法分析或者变换，格式取决于特定的数据库驱动程序，这种查询用法正常但不需要，在索引中更普遍。executesql 有两个可选参数：placeholders 和 as_dict； placeholders 是可选的能被替换的值序列，或者如果DB驱动支持，在SQL中一个字典键匹配的占位符。

如果 as_dict 设置为 True，DB 驱动返回的结果指针会被转换成字典键序列，用 db 对象字段名，对正常的查询用 as_dict=True 返回与用.as_list() 的结果是一样的。

```
1 [{field1: value1, field2: value2}, {field1: value1b, field2: value2b}]
```

6.10.3 _lastsql 文件

不论 SQL 采用 executesql 手动执行还是通过 DAL 生成 SQL 代码，总能在 db._lastsql 中找到 SQL 的代码，这对调试程序非常有用：

```
1 >>> rows = db().select(db.person.ALL)
2 >>> print db._lastsql
3 SELECT person.id, person.name FROM person;
```

web2py 生成查询时从不用*符，web2py 查询字段时总是明确的。

6.11 删除 drop

最后，删除表并且所有的数据将丢失：

```
1 >>> db.person.drop()
```

6.12 索引 Indexes

目前 DAL API 不提供命令生成表的索引，但这可以用 `executesql` 命令实现，这是因为索引的存在会使迁移复杂化，最好进行显式地处理，索引对那些用在循环查询的字段是需要的，

下面是一个在 SQLite 中用 SQL 生成索引的例子：

```
1 >>> db = DAL('sqlite://storage.db')
2 >>> db.define_table('person', Field('name'))
3 >>> db.executesql('CREATE INDEX IF NOT EXISTS myidx ON person (name);')
```

其它数据库术语也有相同的语法，但不直接支持 IF NOT EXISTS 的选项。

6.13 传统数据库和键表

web2py 在某些条件下可以连接到传统数据库。

最简单的方法是当下面条件满足：

- 每张表必须有唯一的自增的整型字段“id”。
- 记录必须通过“id”字段显式引用。

当访问一张现存的表，例如在当前应用中不能通过 web2py 生成，总是设置 `migrate=False`。

如果传统表有自增整型字段，但是不叫“id”，web2py 仍能访问它，但是表的定义必须显式包含 `Field('...', 'id')` 这里...是那个自增整型字段的名称。

```
1 db.define_table('account',
2 Field('accnum', 'integer'),
3 Field('acctype'),
4 Field('accdesc'),
5 primarykey=[ 'accnum', 'acctype' ],
6 migrate=False)
```

- primarykey 是一组构成主键的字段列表。
- 所有的主键字段即便没有指定都有 NOT NULL 设置。
- 对只能引用的键表而言就是其它的键表。
- 引用字段必须使用 `tablename.fieldname` 格式。
- `update_record` 函数对键表 Rows 对象不可用。

注意目前这仅对 DB2、MS-SQL、Ingres 和 Informix 有用，其它数据库可以简单添加。

在撰写本书的时候，我们不能保证 primarykey 属性能用于每个现存的传统表及每个支持的后端数据库。为了简单化，我们推荐如果可能，就创建一个带自增的 id 字段的数据库视图。

6.14 分布式事务

在写本书的时候，这个功能仅能被 PostgreSQL、MySQL 和 Firebird 支持，因为它们公

开两个阶段提交的 API。

假定到不同 PostgreSQL 数据库的两个（或更多）连接，例如：

```
1 db_a = DAL('postgres://...')
2 db_b = DAL('postgres://...')
```

在模块或控制器中，用下面语句同时提交它们：

```
1 DAL.distributed_transaction_commit(db_a, db_b)
```

失败了，这个函数回滚并引发异常。

控制器中，当动作返回，如果有两个确切的连接而又没有调用上面的函数，web2py 分别提交这两个事务，这意味着有可能一个事务处理成功而另外一个失败了，分布式事务能够防止这样的情况发生。

6.15 手册上传

考虑下面的模型：

```
1 >>> db.define_table('myfile', Field('image', 'upload'))
```

正常情况 insert 通过 SQLFORM 或者 crud 表单（是 SQLFORM）自动地处理，但偶尔已经有相关文件系统的文件并想编程上传。这能用下面的方式完成：

```
1 >>> stream = open(filename, 'rb')
2 >>> db.myfile.insert(image=db.myfile.image.store(stream, filename))
```

上传 field 对象的 store 方法采用文件 stream 和文件名参数，它用文件名确定文件扩展名（类型），为该文件创建新的临时文件（依据 web2py 上传机制），把文件内容装载在新的临时文件（在上传文件夹内，特别说明的除外）。它返回新的临时文件名，存储在 db.myfile 表的 image 字段。

与.store 相反的是.retrieve：

```
1 >>> row = db(db.myfile).select().first()
2 >>> (filename, stream) = db.myfile.image.retrieve(row.image)
3 >>> import shutil
4 >>> shutil.copyfileobj(stream, open(filename, 'wb'))
```

6.16 Query（查询）、Set（集合）和 Rows 对象

让我们再思考下前面的表定义（删除），并插入三条记录：

```
1 >>> db.define_table('person', Field('name'))
2 >>> db.person.insert(name="Alex")
3 1
4 >>> db.person.insert(name="Bob")
5 2
6 >>> db.person.insert(name="Carl")
7 3
```

用变量存放表。例如用变量 person，执行：

```
1 >>> person = db.person
```

可以用变量存储字段，比如 name。举个例子，执行：

```
1 >>> name = person.name
```

甚至还能生成 query（用运算符==、!=、<、>、<=、>=、like 和 belongs），用变量 q 存储 query，如下所示：

```
1 >>> q = name=='Alex'
```

当查询调用 db 时，你定义了记录集。可用变量 s 存储，像这样：

```
1 >>> s = db(q)
```

注意迄今还没有执行数据库查询操作，DAL+Query 简单定义了 db 中符合查询的记录集，web2py 从涉及查询的表（或多张表）来决定，事实上，无需具体说明。

6.17 查询 select

给定一个集合 s，用 select（查询）命令得到记录：

```
1 >>> rows = s.select()
```

它返回 gluon.aql.Rows 类的迭代对象，它的成员是 Row 对象，gluon.aql.Row 对象用起来像字典，它们的成员也能作为属性访问，比如 gluon.storage.Storage。前者与后者有区别是因为它的值是只读的。

Rows 对象允许对 select 结果循环，为每行打印所选字段的值：

```
1 >>> for row in rows:
2     print row.id, row.name
3 1 Alex
```

用一个语句完成所有步骤：

```
1 >>> for row in db(db.person.name=='Alex').select():
2     print row.name
3 Alex
```

select 命令能带参数，所有未命名的参数被解释为你想要取到的字段名。例如显式地得到字段 id 和字段 name。

```
1 >>> for row in db().select(db.person.id, db.person.name):
2     print row.name
3 Alex
4 Bob
5 Carl
```

表的属性 ALL 允许指定所有字段：

```
1 >>> for row in db().select(db.person.ALL):
2     print row.name
3 Alex
4 Bob
5 Carl
```

注意：没有查询字符串传递给 db。web2py 明白如果没有附加信息，想要 person 表的所有字段，那么就是想要 person 表的所有记录。

一个等同可以替换的语法如下：

```
1 >>> for row in db(db.person.id > 0).select():
2     print row.name
3 Alex
4 Bob
5 Carl
```

web2py 知悉如果没有额外的信息，就是索要 person 表所有记录（id>0），那么想要 person 表的所有字段。

给定一行

```
1 row = rows[0]
```

用多条等价的表达来得到值：

```
1 >>> row.name
2 Alex
3 >>> row['name']
4 Alex
5 >>> row('person.name')
6 Alex
```

当查询表达式而非列的时候，后者的语法特别方便。我们会在后面展示。

执行

```
1 rows.compact = False
```

禁用 compact（紧凑）表示法

```
1 row[i].name
```

相反，启用不紧凑表示：

```
1 row[i].person.name
```

这是特殊的，也不是极少数情况下才需要的。

6.17.1 快捷方式 Shortcuts

DAL 支持各种各样代码简化的捷径，特别是：

```
1 myrecord = db.mytable[id]
```

返回存在的给定 id 的记录，如果 id 不存在，返回 None。上面的语句等价于

```
1 myrecord = db(db.mytable.id==id).select().first()
```

通过 id 删除记录：

```
1 del db.mytable[id]
```

等价于

```
1 db(db.mytable.id==id).delete()
```

如果存在的话，用给定的 id 删除记录。

插入记录：

```
1 db.mytable[0] = dict(myfield='somevalue')
```

等价于

```
1 db.mytable.insert(myfield='somevalue')
```

它用右手边字典指定的字段值创建新记录。

更新记录：

```
1 db.mytable[id] = dict(myfield='somevalue')
```

它等价于：

```
db(db.mytable.id==id).update(myfield='somevalue')
```

它用右手边字典所指定的字段值更新现存的记录。

6.17.2 得到一个 Row 对象

然而，另一种方便的语法如下：

```
1 record = db.mytable(id)
2 record = db.mytable(db.mytable.id==id)
3 record = db.mytable(id,myfield='somevalue')
```

显然与 `db.mytable[id]` 相同，但上面的语法更灵活和安全。它首先检查 `id` 是否是整型（或 `str(id)` 函数结果是否是整型），如果不是返回 `None`（它绝不会引发异常），允许指定满足的条件的多个记录，如果条件不满足，也返回 `None`。

6.17.3 递归查询 Recursive selects

思考前面定义的 `person` 表，还有一张引用 `person` 表的新表 `dog`：

```
1 >>> db.define_table('dog', Field('name'), Field('owner', db.person))
```

这张表的简单查询：

```
1 >>> dogs = db(db.dog).select()
```

它等价于

```
1 >>> dogs = db(db.dog._id>0).select()
```

此处 `._id` 是对该表主键的引用。正常情况 `db.dog._id` 和 `db.dog.id` 是一样的，本书大部分都这么假定。

对 `dog` 表的 `Row`，不仅能够从查询的表（`dog`）得到，还能从链接表（递归地）得到：

```
1 >>> for dog in dogs: print dog.name, dog.owner.name
```

`dog.owner.name` 要求在 `dog` 表中数据库查询命令，因此它不高效。尽管当访问个人记录这简单而且实用，但是我们建议联合而不是递归查询。

通过对 `person` 表引用的 `dogs` 表，你能后向完成。

```
1 person = db.person(id)
2 for dog in person.dog.select(orderby=db.dog.name):
3 print person.name, 'owns', dog.name
```

这里最后表达式 `person.dog` 是下面的快捷表示

```
1 db(db.dog.owner==person.id)
```

即被当前 `person` 引用的 `dogs` Set 集对象。如果引用表格对被引用表多次引用，该语法出问题，这种情况下，需要更清楚并且使用完全查询。

6.17.4 视图中序列化 Rows 对象

给出下面包含查询的动作：

```
1 def index()
2 return dict(rows = db(query).select())
```

用下面的语法，查询结果在视图中显示：

```
1 {{extend 'layout.html'}}
2 <h1>Records</h1>
3 {{=rows}}
```

这等于：

```
1 {{extend 'layout.html'}}
2 <h1>Records</h1>
3 {{=SQLTABLE(rows)}}
```

SQLTABLE 把 rows 对象变成包含列名标头的 HTML 表，每条记录一个对象。Rows 对象交替标示为 even 类和 odd 类。在后台，Rows 对象首先转换成 SQLTABLE 对象（不与 Table 混淆），然后序列化，从数据库提出的值也被与 field 相关的验证器格式化，然后转义（注意：在视图中这样用 db，不会被视为好的 MVC 实践）。

但是，显式调用 SQLTABLE 可能有时也很方便。

SQLTABLE 构造函数采用下面可选参数：

- linkto URL 或用来链接引用字段的动作（默认 None）
- upload URL 或允许对已上传文件下载的动作（默认 None）
- headers 把字段名映射成用做标头的标签的字典（默认 {}），也可以是指令，当前我们支持 headers='fieldname:capitalize'。
- truncate 表中长值截断的字符位数（默认 16）
- columns 以列的方式显示 fieldname 列表（tablename.fieldname 格式），那些未列出的将不显示（默认 all）
- **attributes 通常帮助对象属性会传递给大部分外部 TABLE 对象。

这有一个例子：

```
1 {{extend 'layout.html'}}
2 <h1>Records</h1>
3 {{=SQLTABLE(rows,
4 headers='fieldname:capitalize',
5 truncate=100,
6 upload=URL('download'))
7 }}
```

SQLTABLE 有用，但需要更多的时候是有类型的。SQLFORM.grid 是 SQLTABLE 的扩展，能创建具有搜索特性和分页的表，它能打开详细记录、创建、编辑、删除记录，SQLFORM.smartgrid 更一般化，不仅允许上述所有功能还能创建访问引用记录的按钮。

使用 SQLFORM.grid 的例子如下：

```
1 def index():
2 return dict(grid=SQLFORM.grid(query))
```

和相应的视图：

```
1 {{extend 'layout.html'}}
2 {{=grid}}
```

SQLFORM.grid 和 SQLFORM.smartgrid 是比 SQLTABLE 更好的选择，因为虽然它们高层级又有所限制，但功能更强。这将在第 8 章作更为详细的解释。

6.17.5 排序、分组、限制、区别 orderby, groupby, limitby, distinct

select 命令采用五个可选参数：orderby、groupby、limitby、left 和 cache，这里我们讨论前三个。

得到名字字母顺序排序的记录：

```
1 >>> for row in db().select(
2 db.person.ALL, orderby=db.person.name):
3 print row.name
4 Alex
5 Bob
```

```
6 Carl
```

得到名字按逆序排序的记录（注意波浪线~）：

```
1 >>> for row in db().select(  
2 db.person.ALL, orderby=~db.person.name):  
3 print row.name  
4 Carl  
5 Bob  
6 Alex
```

让得到的记录以随机顺序显示：

```
1 >>> for row in db().select(  
2 db.person.ALL, orderby='<random>'):  
3 print row.name  
4 Carl  
5 Alex  
6 Bob
```

orderby='<random>' 的用法不被Google NoSQL支持。然而在这种情况下，同样还有其它很多，内置插件还不够，可以采用import：

```
1 import random  
2 rows=db(...).select().sort(lambda row: random.random())
```

根据多字段进行记录排序，多个字段用“|”符号串联：

```
1 >>> for row in db().select(  
2 db.person.ALL, orderby=db.person.name|db.person.id):  
3 print row.name  
4 Carl  
5 Bob  
6 Alex
```

同时用 groupby 和 orderby，把指定字段值相同的记录分组（这是后端指定的，不是对Google NoSQL的）：

```
1 >>> for row in db().select(  
2 db.person.ALL,  
3 orderby=db.person.name, groupby=db.person.name):  
4 print row.name  
5 Alex  
6 Bob  
7 Carl
```

让参数 distinct=True，指定那些仅想查询的不同记录，与用所有指定字段分组是同样效果，除了它不需要排序之外。当用 distinct，很重要的是不要查询所有字段，特别是不要用 id 字段，否则所有的记录总是不同的。

这有一个例子：

```
1 >>> for row in db().select(db.person.name, distinct=True):  
2 print row.name  
3 Alex  
4 Bob  
5 Carl
```

注意 distinct 参数也能做为表达式，例如：

```
1 >>> for row in db().select(db.person.name, distinct=db.person.name):
```

```
2 print row.name
3 Alex
4 Bob
5 Carl
```

用 `limitby`, 查询记录的子集 (这种情况, 头两个从 0 开始):

```
1 >>> for row in db().select(db.person.ALL, limitby=(0, 2)):
2 print row.name
3 Alex
4 Bob
```

6.17.6 逻辑运算符

查询可以用二进制与 AND 运算符 “&” 连接:

```
1 >>> rows = db((db.person.name=='Alex') & (db.person.id>3)).select()
2 >>> for row in rows: print row.id, row.name
3 4 Alex
```

二进制或 OR 运算符 “|”:

```
1 >>> rows = db((db.person.name=='Alex') | (db.person.id>3)).select()
2 >>> for row in rows: print row.id, row.name
3 1 Alex
```

取反查询 (或者子查询), 用二进制运算符 “!=”:

```
1 >>> rows = db((db.person.name!='Alex') | (db.person.id>3)).select()
2 >>> for row in rows: print row.id, row.name
3 2 Bob
4 3 Carl
```

或用 “~” 一元运算符显式取反:

```
1 >>> rows = db(~db.person.name=='Alex') | (db.person.id>3)).select()
2 >>> for row in rows: print row.id, row.name
3 2 Bob
4 3 Carl
```

因为 Python 限制超载 “and” 和 “or” 运算符, 这些不能用在形成查询语句。因而, 必须使用二进制运算符。

也有可能用合适的逻辑运算符来构建查询:

```
1 >>> query = db.person.name!='Alex'
2 >>> query &= db.person.id>3
3 >>> query |= db.person.name=='John'
```

6.17.7 count, isempty, delete 和 update 方法

对集合中的记录计数:

```
1 >>> print db(db.person.id > 0).count()
2 3
```

注意 `count` 采用可选的 `distinct` 参数, 默认 `False`, 它的作用与 `select` 命令中的 `distinct` 参数非常想象。

有时需要检查表是否为空, 比计数更有效的方法是用 `isempty` 方法:

```
1 >>> print db(db.person.id > 0).isempty()
2 False
```


或等同于：

```
1 >>> print db(db.person).isempty()
2 False
```

delete（删除）集合中的记录：

```
1 >>> db(db.person.id > 3).delete()
```

把参数名传递给相应的需要修改的字段，update（更新修改）集合中的所有记录：

```
1 >>> db(db.person.id > 3).update(name='Ken')
```

6.17.8 表达式

赋给更新语句的参数值可以是表达式。例如考虑下面这个模型：

```
1 >>> db.define_table('person',
2     Field('name'),
3     Field('visits', 'integer', default=0))
4 >>> db(db.person.name == 'Massimo').update(
5 visits = db.person.visits + 1)
```

用在查询语句中的参数值也可以是表达式：

```
1 >>> db.define_table('person',
2 Field('name'),
3 Field('visits', 'integer', default=0),
4 Field('clicks', 'integer', default=0))
5 >>> db(db.person.visits == db.person.clicks + 1).delete()
```

6.17.9 update_record 方法

we2py 允许使用 update_record 更新已经在内存中的单个记录：

```
1 >>> row = db(db.person.id==2).select().first()
2 >>> row.update_record(name='Curt')
```

update_record 不应该与下面混淆

```
1 >>> row.update(name='Curt')
```

因为对单个 row 对象，方法 update 更新 row 对象，而不是数据库的记录，如同在 update_record 情况（更新数据库记录而不是 row 对象）。

修改 row 对象的属性（有一个）也是可能的，无参调用 update_record（）保存修改：

```
1 >>> row = db(db.person.id > 2).select().first()
2 >>> row.name = 'Curt'
3 >>> row.update_record() # saves above change
```

6.17.10 first 和 last 方法

给出一个包含记录的 Rows 对象：

```
1 >>> rows = db(query).select()
2 >>> first_row = rows.first()
```

```
3 >>> last_row = rows.last()
```

等价于：

```
1 >>> first_row = rows[0] if len(rows)>0 else None
2 >>> last_row = rows[-1] if len(rows)>0 else None
```

6.17.11 as_dict 和 as_list 方法

使用 as_dict () 方法能把 Row 对象序列化为规则字典，用 as_list () 方法能把 Rows 对象序列化为字典列表。以下是一些例子：

```
1 >>> rows = db(query).select()
2 >>> rows_list = rows.as_list()
3 >>> first_row_dict = rows.first().as_dict()
```

这些方法很方便地把 Rows 传递给通用视图，或存储 Rows 在会话中（因为 Rows 对象自身不能被序列化，它有一个打开 DB 连接的引用）：

```
1 >>> rows = db(query).select()
2 >>> session.rows = rows # not allowed!
3 >>> session.rows = rows.as_list() # allowed!
```

6.17.12 find、exclude 和 sort 对象

当需要执行两个查询，其中一个包含上次查询的子集。这种情况，再访问数据库是无意义的，find、exclude 和 sort 对象允许操作 Rows 对象，不必访问数据库生成另外一个。更具体：

- find 返回经一个条件过滤的新的 Rows，源中保持不变。
- exclude 返回经一个条件过滤的新的 Rows，并在源 Rows 中删除。
- sort 返回由一个条件排序的新的 Rows，源中保持不变。

这些方法用单个参数、函数作用于每一行。

下面是用法示例：

```
1 >>> db.define_table('person',Field('name'))
2 >>> db.person.insert(name='John')
3 >>> db.person.insert(name='Max')
4 >>> db.person.insert(name='Alex')
5 >>> rows = db(db.person).select()
6 >>> for row in rows.find(lambda row: row.name[0]=='M'):
7     print row.name
8 Max
9 >>> print len(rows)
10 3
11 >>> for row in rows.exclude(lambda row: row.name[0]=='M'):
12     print row.name
13 Alex
14 >>> print len(rows)
15 2
16 >>> for row in rows.sort(lambda row: row.name):
17     print row.name
18 Alex
19 John
```

它们可以组合为：

```
1 >>> rows = db(db.person).select()
2 >>> rows = rows.find(
3 lambda row: 'x' in row.name).sort(
4 lambda row: row.name)
5 >>> for row in rows:
6 print row.name
7 Alex
8 Max
```

6.18 其它方法

6.18.1 update_or_insert 方法

有时执行插入操作，仅当没有同值的记录才被插入。能这样做

```
1 db.define_table('person',Field('name'),Field('birthplace'))
2 db.person.update_or_insert(name='John',birthplace='Chicago')
```

这条记录只有没其它生于 Chicago 叫 John 的用户才能被插入。

指定那个值作为关键值，来决定记录是否存在。例如：

```
1 db.person.update_or_insert(db.person.name=='John',
2 name='John',birthplace='Chicago')
```

如果存在 John 记录它的出生地会被更新，否则新的记录会被创建。

6.18.2 validate_and_insert 和 validate_and_update 方法

函数

```
1 ret = db.mytable.validate_and_insert(field='value')
```

作用与下面非常相似

```
1 id = db.mytable.insert(field='value')
```

除了那些在执行插入前为字段调用验证器，如果验证不通过则安全退出，如果验证器通不过，出错信息记录在 `ret.error`；如果通过，新记录的 `id` 在 `ret.id` 中。请注意正常情况下，验证被表单处理逻辑执行，因此这个函数几乎不需要。

同样地

```
1 ret = db(query).validate_and_update(field='value')
```

作用与下面语句一样

```
1 num = db(query).update(field='value')
```

除了那些在执行更新前为字段调用验证器，注意仅当插入涉及到单个表时才用，更新记录数在 `res.updated` 中且错误将会是 `ret.errors`。

6.18.3 smart_query 方法（实验性的）

有时需要使用自然语言解析查询命令，例如

```
1 name contain m and age greater than 18
```

DAL 提供了解析这类查询命令的一种方法：

```
1 search = 'name contain m and age greater than 18'
2 rows = db.smart_query([db.person], search).select()
```

第一个参数必须是允许被搜索的表或字段列表，如果搜索字符串无效，导致 `RuntimeError`，这项功能能用来构建 RESTful 接口（参看第 10 章），它被 `SQLFORM.grid` 和 `SQLFORM.smartgrid` 在内部使用。

在智能查询搜索字符串中，字段仅能被字段名标识，或被 `tablename.fieldname` 标识，如果包含空格，字符串可以用双引号分隔。

6.19 计算字段

DAL field 有 `compute` 属性，这必须是采用 Row 对象的函数（或 `lambda`），并为 field 返回值。当一条新纪录被修改，包括插入和更新，如果对于 field 的值没有被提供，web2py 尝试用 `compute` 函数从其它字段值计算。这是一个例子：

```
1 >>> db.define_table('item',
2   Field('unit_price', 'double'),
3   Field('quantity', 'integer'),
4   Field('total_price',
5     compute=lambda r: r['unit_price']*r['quantity']))
6 >>> r = db.item.insert(unit_price=1.99, quantity=5)
7 >>> print r.total_price
8 9.95
```

注意计算得到的值存在 db 中，当检索它不被计算，如同在后面描述的虚拟字段情况一样。计算字段的两种典型应用如下：

- wiki 应用，存储已处理的输入 wiki 文本为 HTML，避免每次请求的预处理
- 为了搜索，为字段计算标准值，用来搜索。

6.20 虚拟字段

6.20.1 旧风格虚拟字段

虚拟字段也是计算字段（上一小节），但它区别于那些，是因为它是 `virtual`。在这个意义上，它不存储在 db 中，当每次从数据库中提取记录时，它就计算，它们用来简化用户代码而不用增加存储，但不能用于搜索。

为了定义一个或多个虚拟字段，需要定义容器类，实例化它并链接到表或查询。例如，考虑下面的表：

```
1 >>> db.define_table('item',
2   Field('unit_price', 'double'),
3   Field('quantity', 'integer'),
```

`total_price` 虚拟字段能被定义如下：

```
1 >>> class MyVirtualFields(object):
2   def total_price(self):
3   return self.item.unit_price*self.item.quantity
4 >>> db.item.virtualfields.append(MyVirtualFields())
```

注意这个类的每个方法采用的单个参数（self），都是一个新的虚拟字段，self 指查询的每个 row（行）。Field 值在 self.item.unit_price 全路径引用，表链接到虚拟字段，通过追加类的实例到表的 virtualfields 属性。

虚拟字段能访问递归字段

```
1 >>> db.define_table('item',
2 Field('unit_price', 'double'))
3 >>> db.define_table('order_item',
4 Field('item', db.item),
5 Field('quantity', 'integer'))
6 >>> class MyVirtualFields(object):
7 def total_price(self):
8 return self.order_item.item.unit_price \
9 * self.order_item.quantity
10 >>> db.order_item.virtualfields.append(MyVirtualFields())
```

注意递归字段访问 self.order_item.unit_price，self 是循环记录。

这也能对 JOIN 结果做：

```
1 >>> db.define_table('item',
2 Field('unit_price', 'double'))
3 >>> db.define_table('order_item',
4 Field('item', db.item),
5 Field('quantity', 'integer'))
6 >>> rows = db(db.order_item.item==db.item.id).select()
7 >>> class MyVirtualFields(object):
8 def total_price(self):
9 return self.item.unit_price \
10 * self.order_item.quantity
11 >>> rows.setvirtualfields(order_item=MyVirtualFields())
12 >>> for row in rows: print row.order_item.total_price
```

注意这种情况下语法的不同之处。虚拟字段访问了 self.item.unit_price 和 self.order_item.quantity，属于联合查询，使用 rows 对象的 setvirtualfields 方法，虚拟字段连接到表的行，这个方法采用任意个命名的参数，用来设置多个虚拟字段，在多个类中定义，并把它们连接到多张表：

```
1 >>> class MyVirtualFields1(object):
2 def discounted_unit_price(self):
3 return self.item.unit_price*0.90
4 >>> class MyVirtualFields2(object):
5 def total_price(self):
6 return self.item.unit_price \
7 * self.order_item.quantity
8 def discounted_total_price(self):
9 return self.item.discounted_unit_price \
10 * self.order_item.quantity
11 >>> rows.setvirtualfields(
12 item=MyVirtualFields1(),
13 order_item=MyVirtualFields2())
14 >>> for row in rows:
15 print row.order_item.discounted_total_price
```

虚拟字段可能是 lazy，所有需要做的是返回函数，调用函数访问它：

```
1 >>> db.define_table('item',
2 Field('unit_price', 'double'),
3 Field('quantity', 'integer'),
4 >>> class MyVirtualFields(object):
```

```

5 def lazy_total_price(self):
6 def lazy(self=self):
7 return self.item.unit_price \
8 * self.item.quantity
9 return lazy
10 >>> db.item.virtualfields.append(MyVirtualFields())
11 >>> for item in db(db.item).select():
12 print item.lazy_total_price()

```

或用 lambda 函数缩写：

```

1 >>> class MyVirtualFields(object):
2 def lazy_total_price(self):
3 return lambda self=self: self.item.unit_price \
4 * self.item.quantity

```

6.20.2 新风格的虚拟字段（实验性的）

web2py 提供一种新的更简单的方法定义虚拟字段和 lazy 虚拟字段，这部分标记为实验是因为有可能在此描述的 API 会有一些变化。

这里我们考虑前面章节中相同的例子，特别考虑下面的模型：

```

1 >>> db.define_table('item',
2 Field('unit_price','double'),
3 Field('quantity','integer'),

```

如下定义一个 total_price 虚拟字段：

```
1 >>> db.item.total_price = Field.Virtual(lambda row: row.unit_price*row.quantity)
```

即简单定义一个新字段 total_price 为 Field.Virtual。构造函数的唯一参数是函数，该函数用 row 参数并返回已计算值。

当记录被选中时，上面定义的一个虚拟字段会被自动计算：

```
1 >>> for row in db(db.item).select(): print row.total_price
```

也能定义 lazy 虚拟字段，当调用时，按需计算。例如：

```

1 >>> db.item.total_price = Field.Lazy(lambda row, discount=0.0: \
2 row.unit_price*row.quantity*(1.0-discount/10

```

这种情况下，row.total_price 不是值而是函数，除隐式的 row 以外，函数与它传递给 lazy 构造函数是同样的参数。（对 rows 对象来说，把它视为 self）

上面例子中 lazy 字段允许计算每个 item 的总价格：

```
1 >>> for row in db(db.item).select(): print row.total_price()
```

它也允许传递可选 discount 百分比（15%）：

```
1 >>> for row in db(db.item).select(): print row.total_price(15)
```

记住虚拟字段没有和其它字段相同的属性（default、readable 和 requires 等），它们不出现在 db.table.fields 列表，并且在表（TABLE）和网格中默认不可见（SQLFORM.grid、SQLFORM.smartgrid）。

6.21 一对多关系

为说明采用 web2py DAL 如何实现一对多关系，我们在这定义另外一张引用到 person 表的 dog 表：

```
1 >>> db.define_table('person',
2   Field('name'),
3   format='%(name)s')
4 >>> db.define_table('dog',
5   Field('name'),
6   Field('owner', db.person),
7   format='%(name)s')
```

表dog有两个字段，表的name和owner。当字段类型是其它表时，它打算通过它的id，引用到其它表的字段。事实上，可以显示实际类型值，并得到：

```
1 >>> print db.dog.owner.type
2 reference person
```

现在，插入三条dog记录，Alex两条和Bob一条：

```
1 >>> db.dog.insert(name='Skipper', owner=1)
2 1
3 >>> db.dog.insert(name='Snoopy', owner=1)
4 2
5 >>> db.dog.insert(name='Puppy', owner=2)
6 3
```

象对其它任何表一样进行查询：

```
1 >>> for row in db(db.dog.owner==1).select():
2   print row.name
3 Skipper
4 Snoopy
```

因为dog引用 person，一个人可以有很多条狗，因此，现在person表的记录得到新的属性dog，它是定义人所拥有的狗集合。这允许对所有人进行循环，很轻易得到他们的狗：

```
1 >>> for person in db().select(db.person.ALL):
2   print person.name
3   for dog in person.dog.select():
4     print ' ', dog.name
5 Alex
6 Skipper
7 Snoopy
8 Bob
9 Puppy
10 Carl
```

6.21.1 内联 Inner joins

另一种方式是通过使用一个连接实现类似的结果，具体是 INNER JOIN（内联），当查询链接到两张或多张表时，web2py 自动和透明地执行连接，如同下面的例子：

```
1 >>> rows = db(db.person.id==db.dog.owner).select()
2 >>> for row in rows:
3   print row.person.name, 'has', row.dog.name
4 Alex has Skipper
```



```
5 Alex has Snoopy
6 Bob has Puppy
```

观察 web2py 联合，因此现在 rows 对象包含两条记录，每张表一条，链接在一起。因为两条记录可能有名字相冲突的字段，当从行（对象）取出字段值时，需要指定这张表，意味着当执行前：

```
1 row.name
```

显然无论是人名还是狗名，在联合查询结果中，必须更清晰地说：

```
1 row.person.name
```

或：

```
1 row.dog.name
```

有 INNER JOIN（内联）的替代语法：

```
1 >>> rows = db(db.person).select(join=db.dog.on(db.person.id==db.dog.owner))
2 >>> for row in rows:
3     print row.person.name, 'has', row.dog.name
4 Alex has Skipper
5 Alex has Snoopy
6 Bob has Puppy
```

当输出一样时，在这两种情况下生成的 SQL 是不同的，当同一张表联合两次和别名时，后者语法消除可能的奇异：

```
1 >>> db.define_table('dog',
2     Field('name'),
3     Field('owner1', db.person),
4     Field('owner2', db.person))
5 >>> rows = db(db.person).select(
6     join=[db.person.with_alias('owner1').on(db.person.id==db.dog.owner1).
7     db.person.with_alias('owner2').on(db.person.id==db.dog.owner2)])
```

join 值是 db.table.on(...) 列表联合。

6.21.2 左外联 Left outer join

注意到 Carl 不出现在上面的列表中，因为他没有狗，如果你打算查询人（不论他有没有狗）和他的狗（如果他们有），那需要执行 LEFT OUTER JOIN，用选择命令的参数“left”完成。下面是例子：

```
1 >>> rows=db().select(
2     db.person.ALL, db.dog.ALL,
3     left=db.dog.on(db.person.id==db.dog.owner))
4 >>> for row in rows:
5     print row.person.name, 'has', row.dog.name
6 Alex has Skipper
7 Alex has Snoopy
8 Bob has Puppy
9 Carl has None
```

此处：

```
1 left = db.dog.on(...)
```

做 left join（左联）查询。db.dog.on 参数是进行 join 所需要的条件（与上面内联相同），

在左联，对哪个字段查询必须是清晰的。

6.21. 3 分组和计数 Grouping and counting

当执行联合时，有时需要根据一定的标准分类 rows 对象并进行计数，例如统计每个人拥有的狗的数量。Web2py 也允许这么做。首先，需要 count 运算符；其次，你想通过 owner 联合 person 表和 dog 表；最后，你想查询所有 rows (person+dog)，按照 person 分组它们，当分组时进行统计：

```
1 >>> count = db.person.id.count()
2 >>> for row in db(db.person.id==db.dog.owner).select(
3 db.person.name, count, groupby=db.person.name):
4 print row.person.name, row[count]
5 Alex 2
6 Bob 1
```

注意 count 运算符（内嵌的）用做一个字段。这里仅有的问题是如何取回信息，每行明确包含一个人以及数字，但是 count 不是 person 字段也不是一张表。那么，它在哪儿呢？它在存储对象代表记录，有关键词等价于查询表达式本身。

6.22 多对多关系

在之前的例子，我们允许一只狗有一个主人，而一个人可以有多只狗。如果狗 Skipper 为 Alex 和 Curt 所有会怎样？这需要多对多关系，它通过中间表实现，通过所有关系表链接人和狗。

下面是如何做到：

```
1 >>> db.define_table('person',
2 Field('name'))
3 >>> db.define_table('dog',
4 Field('name'))
5 >>> db.define_table('ownership',
6 Field('person', db.person),
7 Field('dog', db.dog))
```

现存的所有关系现被重新写：

```
1 >>> db.ownership.insert(person=1, dog=1) # Alex owns Skipper
2 >>> db.ownership.insert(person=1, dog=2) # Alex owns Snoopy
3 >>> db.ownership.insert(person=2, dog=3) # Bob owns Puppy
```

现在，添加新的关系，Curt 共同拥有 Skipper：

```
1 >>> db.ownership.insert(person=3, dog=1) # Curt owns Skipper too
```

因为现在在表之间有三重关系，定义新的集合执行操作是很方便的：

```
1 >>> persons_and_dogs = db(
2 (db.person.id==db.ownership.person) \
3 & (db.dog.id==db.ownership.dog))
```

现在，从新 Set（集）很容易查询所有人及他们的狗：

```
1 >>> for row in persons_and_dogs.select():
2 print row.person.name, row.dog.name
3 Alex Skipper
```

```
4 Alex Snoopy
5 Bob Puppy
6 Curt Skipper
```

类似地，查找 Alex 拥有的所有的狗：

```
1 >>> for row in persons_and_dogs(db.person.name=='Alex').select():
2     print row.dog.name
3 Skipper
4 Snoopy
```

和 Skipper 的所有的主人：

```
1 >>> for row in persons_and_dogs(db.dog.name=='Skipper').select():
2     print row.person.name
3 Alex
4 Curt
```

轻量级替换多对多关系是标签，Tagging在IS_IN_DB验证器内容部分讨论，Tagging甚至能工作在数据库后端，像Google App Engine NoSQL，不支持JOIN。

6.23 多对多、list:<type>和容器

web2py 提供下列特殊字段类型：

```
1 list:string
2 list:integer
3 list:reference <table>
```

它们分别包含字符串列表、整型列表和引用列表。

就Google App Engine NoSQL list而言：string被映射为 StringListProperty，其它两个映射为ListProperty（整型）；就关系数据库而言，它们都映射为文本字段，包含用|分隔的项目列表。例如[1, 2, 3]映射为|1|2|3|。

对于字符串列表，字符有转义的，其中任何|都被替换为||。无论如何，这是内部表示，对用户是透明的。

可以用 list: string，例如用下面的方式：

```
1 >>> db.define_table('product',
2     Field('name'),
3     Field('colors', 'list:string'))
4 >>> db.product.colors.requires=IS_IN_SET(('red', 'blue', 'green'))
5 >>> db.product.insert(name='Toy Car', colors=['red', 'green'])
6 >>> products = db(db.product.colors.contains('red')).select()
7 >>> for item in products:
8     print item.name, item.colors
9 Toy Car ['red', 'green']
```

list: integer 用相同的方式工作，但项目必须是整型。

通常，该要求在表单层是被强制的，不是在 insert 层。

对 list: <type> 字段，contains(value) 运算符映射为非 trivial 查询，该检查包含值的列表。Contains 运算符也为被规则字符串和文本字段使用，它映射为 LIKE '%value'。

list: reference 和 contains(value) 运算符特别有用，非规范化多对多关系。下面是例子：

```
1 >>> db.define_table('tag', Field('name'), format='%(name)s')
2 >>> db.define_table('product',
```

```

3 Field('name'),
4 Field('tags', 'list:reference tag'))
5 >>> a = db.tag.insert(name='red')
6 >>> b = db.tag.insert(name='green')
7 >>> c = db.tag.insert(name='blue')
8 >>> db.product.insert(name='Toy Car', tags=[a, b, c])
9 >>> products = db(db.product.tags.contains(b)).select()
10 >>> for item in products:
11     print item.name, item.tags
12 Toy Car [1, 2, 3]
13 >>> for item in products:
14     print item.name, db.product.tags.represent(item.tags)
15 Toy Car red, green, blue

```

注意 list: reference tag 字段有默认的约束

```
1 requires = IS_IN_DB(db, 'tag.id', db.tag._format, multiple=True)
```

在表单中产生 SELECT/OPTION 多个下拉框。

也要注意这个字段默认 represent 属性，它代表了一个作为格式化引用的以逗号分隔列表来表示引用列表，这被用在读表单和 SQLTABLES 中。

list:reference 有默认验证器和默认表示，但 list:integer 和 list:string 没有。因此，如果想在表单中使用它们，则需要 IS_IN_SET 或 IS_IN_DB 验证器。

6.24 其它运算符

Web2py 有其它运算符，该运算符提供 API 访问相同 SQL 运算符。让我们定义另一张表”log” 存储安全事件，它们的 event_time（事件时间）和 severity（严重程度），在那儿 severity 是个整型数。

```

1 >>> db.define_table('log', Field('event'),
2 Field('event_time', 'datetime'),
3 Field('severity', 'integer'))

```

如前所述，插入一些事件、“port scan”（端口扫描）、“xss injection”（xss 注入）和“unauthorized login”（未经授权登录），对于这个事例，可以用不同的级别登录相同的event_time（分别1、2、3）。

```

1 >>> import datetime
2 >>> now = datetime.datetime.now()
3 >>> print db.log.insert(
4 event='port scan', event_time=now, severity=1)
5 1
6 >>> print db.log.insert(
7 event='xss injection', event_time=now, severity=2)
8 2
9 >>> print db.log.insert(
10 event='unauthorized login', event_time=now, severity=3)
11 3

```

6.24.1 like, startswith, contains, upper 和 lower 运算符

Field有like运算符，可以用来匹配字符串：

```
1 >>> for row in db(db.log.event.like('port%')).select():
```

```
2 print row.event
3 port scan
```

“port %”表示字符串以“port”开始，百分符号“%”是通配符，代表任何字符序列，Web2py也提供一些快捷：

```
1 db.mytable.myfield.startswith('value')
2 db.mytable.myfield.contains('value')
```

分别与下面语句等价：

```
1 db.mytable.myfield.like('value%')
2 db.mytable.myfield.like('%value%')
```

注意contains容器对list:<type>字段有特殊的意义，已在之前章节讨论过。

contains容器方法也能传递值列表，可选布尔参数，all表示搜索包含所有值的记录：

```
1 db.mytable.myfield.contains(['value1','value2'], all=True)
```

或从该列表的任何值

```
1 db.mytable.myfield.contains(['value1','value2'], all=False)
```

upper和lower方法允许把字段的值转换为大写或小写，也能用like运算符连接它们：

```
1 >>> for row in db(db.log.event.upper().like('PORT')).select():
2 print row.event
3 port scan
```

6.24.2 year, month, day, hour, minutes 和 seconds 运算符

Date 和 datetime 字段有 day、month 和 year 方法，datetime 和 time 字段有 hour、minutes 和 seconds 方法。下面是例子：

```
1 >>> for row in db(db.log.event_time.year()==2009).select():
2 print row.event
3 port scan
4 xss injection
5 unauthorized login
```

6.24.3 belongs 运算符

SQL IN 运算符通过 belongs 方法实现，当字段值属于指定集合（元组列表），返回真：

```
1 >>> for row in db(db.log.severity.belongs((1, 2))).select():
2 print row.event
3 port scan
4 xss injection
```

DAL 也允许作为 belongs 运算符参数似的嵌套查询，唯一要注意的是，嵌套查询需要_select 不是 select，仅有一个字段显式查询，就是定义集合的那个。

```
1 >>> bad_days = db(db.log.severity==3)._select(db.log.event_time)
2 >>> for row in db(db.log.event_time.belongs(bad_days)).select():
3 print row.event
```

```
4 port scan
5 xss injection
6 unauthorized login
```

6.24.4 sum, min, max 和 len 运算符

之前，已用过 count 运算符统计记录数。相似地，可以用 sum 运算符来累积（求和）记录组中指定字段值。在 count 情况下，求和的结果通过存储对象得到：

```
1 >>> sum = db.log.severity.sum()
2 >>> print db().select(sum).first()[sum]
3 6
```

可以用 min 和 max 运算符得到查询记录中的最小和最大值。

```
1 >>> max = db.log.severity.max()
2 >>> print db().select(max).first()[max]
3 3
```

.len() 计算 string 字符串、text 文本或 boolean 布尔字段的长度。

表达式可以组合形成更复杂的表达式。例如，我们正在计算日志中所有严重字符串的长度总和，增加一个：

```
1 >>> sum = (db.log.severity.len()+1).sum()
2 >>> print db().select(sum).first()[sum]
```

6.24.5 子字符串 Substrings

可以构建表达式引用子字符串。例如我们分类 dogs，它的名字用三个相同的字符开始，并且每类中查询一个：

```
1 db(db.dog).select(dictinct = db.dog.name[:3])
```

6.24.6 coalesce 默认值 和 coalesce_zero

有时候当需要从数据库中取出值，如果记录的值设置为 NULL，也需要默认值。在 SQL 中有一个关键词、COALESCE 来实现。Web2py 有等效的 coalesce 方法：

```
1 >>> db.define_table('sysuser',Field('username'),Field('fullname'))
2 >>> db.sysuser.insert(username='max',fullname='Max Power')
3 >>> db.sysuser.insert(username='tim',fullname=None)
4 print db(db.sysuser).select(db.sysuser.fullname.coalesce(db.sysuser.username))
5 "COALESCE(sysuser.fullname,sysuser.username)"
6 Max Power
7 tim
```

其它时候，需要计算数学表达式，但一些字段值设置为 None，而它应该为 0，Coalesce_zero 用来完成查询中默认转换 None 为 0。

6.25 生成原始 sql

有时需要生成 SQL 但不执行。用 web2py 很容易做到，因为每个执行数据库 IO 命令，有等同的不执行命令，很容易地返回应该执行的 SQL。这些命令有像函数一样的相同的名字和语法，但他们以下划线开头：

以下是 `_insert`

```
1 >>> print db.person._insert(name='Alex')
2 INSERT INTO person(name) VALUES ('Alex');
```

以下是 `_count`

```
1 >>> print db(db.person.name=='Alex')._count()
2 SELECT count(*) FROM person WHERE person.name='Alex';
```

以下是 `_select`

```
1 >>> print db(db.person.name=='Alex')._select()
2 SELECT person.id, person.name FROM person WHERE person.name='Alex';
```

以下是 `_delete`

```
1 >>> print db(db.person.name=='Alex')._delete()
2 DELETE FROM person WHERE person.name='Alex';
```

最后是 `_update`

```
1 >>> print db(db.person.name=='Alex')._update()
2 UPDATE person SET WHERE person.name='Alex';
```

而且总能用 `db.lastsql` 返回最近 SQL 代码，无论它是用 `executesql` 手动执行还是 DAL 生成 SQL。

6.26 导出和导入数据

6.26.1 CSV（一张表一次）

当 DAL Rows 对象被转换为字符串，它自动序列为 CSV 格式：

```
1 >>> rows = db(db.person.id==db.dog.owner).select()
2 >>> print rows
3 person.id, person.name, dog.id, dog.name, dog.owner
4 1, Alex, 1, Skipper, 1
5 1, Alex, 2, Snoopy, 1
6 2, Bob, 3, Puppy, 2
```

用 CSV 格式序列单个表，并把它存在文件 “test.csv”：

```
1 >>> open('test.csv', 'w').write(str(db(db.person.id).select()))
```

可以容易地读回：

```
1 >>> db.person.import_from_csv_file(open('test.csv', 'r'))
```

当导入时，web2py 在 CSV 头中查找字段名。这个例子它找到两列：“person.id” 和 “person.name”，忽略 “person.” 前缀，忽略 “id” 字段；然后所有记录追加，并指定新的 id。这些操作可以通过 appadmin web 接口执行。

6.26.2 CSV（所有表一次）

web2py 中，可以用两个命令 back/restore（备份/恢复）整个数据库：

导出：

```
1 >>> db.export_to_csv_file(open('somefile.csv', 'wb'))
```

导入：

```
1 >>> db.import_from_csv_file(open('somefile.csv', 'rb'))
```

这种机制即便是导入数据库与导出数据库不同也能够使用。数据以 CSV 文件存储在“somefile.csv”，每张表以表示表名的那行开始执行，另起行字段名：

```
1 TABLE tablename
2 field1, field2, field3, ...
```

两张表用\r\n\r\n分隔，文件以下面这行结束

```
1 END
```

如果不存储在数据库中，文件不包括上传的文件。任何情况，轻而易举分开压缩“uploads”文件夹。

在导入时，如果它不是空的，新记录被附加到数据库。一般而言，新导入记录不会与源记录（保存的）有相同的记录 id，但 web2py 会存储引用，因此即使 id 值改变了它们也不会破损。

如果表包含字段“uuid”，这个字段被用来标识重复，而且如果导入记录与已有记录有同样的“uuid”，之前的记录会更新。

6.26.3 CSV 和远程数据库同步

考虑下面的模型：

```
1 db = DAL('sqlite:memory:')
2 db.define_table('person',
3 Field('name'),
4 format='% (name)s')
5 db.define_table('dog',
6 Field('owner', db.person),
7 Field('name'),
8 format='% (name)s')
9
10 if not db(db.person).count():
11 id = db.person.insert(name="Massimo")
12 db.dog.insert(owner=id, name="Snoopy")
```

每条记录都用 ID 标识，通过 ID 引用，如果有数据库的两个拷贝，且被 distinct web2py 安装使用，ID 仅在每个数据库中是唯一的，而不是在数据库之间，当从不同数据库融合记录时，这是一个问题。

为了使在数据库之间的记录也是唯一标识的，它们必须：

- 有唯一 id (UUID)
- 有 event_time（如果多拷贝是为了区分出那个是最近的）

- 引用 UUID 而不是 id

无需修改 web2py 而得到。下面是如何做：

1. 改变上面的模型如下：

```
1 db.define_table('person',
2 Field('uuid', length=64, default=lambda: str(uuid.uuid4()))),
3 Field('modified_on', 'datetime', default=now),
4 Field('name'),
5 format='% (name)s')
6
7 db.define_table('dog',
8 Field('uuid', length=64, default=lambda: str(uuid.uuid4()))),
9 Field('modified_on', 'datetime', default=now),
10 Field('owner', length=64),
11 Field('name'),
12 format='% (name)s')
13
14 db.dog.owner.requires = IS_IN_DB(db, 'person.uuid', '% (name)s')
15
16 if not db(db.person.id).count():
17 id = uuid.uuid4()
18 db.person.insert(name="Massimo", uuid=id)
19 db.dog.insert(owner=id, name="Snoopy")
```

注意，在上表定义中，两个 'UUID' 字段默认值是 lambda 函数，返回 UUID（转换为字符串）。每条记录插入时 lambda 函数就被调用一次，为了确保每条记录有唯一 UUID，即使单个事务中插入多条记录也是一样。

2. 创建控制动作导出数据库：

```
1 def export():
2 s = StringIO.StringIO()
3 db.export_to_csv_file(s)
4 response.headers['Content-Type'] = 'text/csv'
5 return s.getvalue()
```

3. 创建控制动作导入保存的其它数据库和同步记录中的拷贝：

```
1 def import_and_sync():
2 form = FORM(INPUT(_type='file', _name='data'), INPUT(_type='submit'))
3 if form.process(session=None).accepted:
4 db.import_from_csv_file(form.vars.data.file, unique=False)
5 # for every table
6 for table in db.tables:
7 # for every uuid, delete all but the latest
8 items = db(db[table]).select(db[table].id,
9 db[table].uuid,
10 orderby=db[table].modified_on,
11 groupby=db[table].uuid)
12 for item in items:
13 db((db[table].uuid==item.uuid)&\
14 (db[table].id!=item.id)).delete()
15 return dict(form=form)
```

注意 session=None 禁用 CSRF 保护，因为这个 URL 设想是从外部访问。

4. 手动创建索引，为了搜索更快使用 uuid。

注意步骤 2 和 3 适用每个数据库模型，它们不是专门为这个例子的。

换个方法，可以用 XML-RPC 导出/导入文件。

如果记录引用上传文件，也需要导出/导入上传文件夹的内容。注意哪些文件已被 UUID 标识了，因此你不必担心名字冲突和引用。

6.26.4 HTML 与 XML（一张表一次）

DAL Rows 对象也有 xml 方法（同帮助对象），序列化到 XML/HTML：

```
1 >>> rows = db(db.person.id > 0).select()
2 >>> print rows.xml()
3 <table>
4 <thead>
5 <tr>
6 <th>person.id</th>
7 <th>person.name</th>
8 <th>dog.id</th>
9 <th>dog.name</th>
10 <th>dog.owner</th>
11 </tr>
12 </thead>
13 <tbody>
14 <tr class="even">
15 <td>1</td>
16 <td>Alex</td>
17 <td>1</td>
18 <td>Skipper</td>
19 <td>1</td>
20 </tr>
21 ...
22 </tbody>
23 </table>
```

如果需要序列化 DAL Rows，使用其它任何定制标签的 XML 格式，也可用通用的 TAG 帮助对象轻易实现，注意：

```
1 >>> rows = db(db.person.id > 0).select()
2 >>> print TAG.result(*[TAG.row(*[TAG.field(r[f], _name=f) \
3 for f in db.person.fields]) for r in rows])
4 <result>
5 <row>
6 <field name="id">1</field>
7 <field name="name">Alex</field>
8 </row>
9 ...
10 </result>
```

6.26.5 数据表示

export_to_csv_file 函数接受关键词参数 represent，当为 True 时，导出数据时它会用列 represent 函数而不是原始数据。

函数接受关键词参数 `colnames`，包含想要导出的列的名字列表，默认对所有列。

`export_to_csv_file` 和 `import_from_csv_file` 接受关键词参数，告诉 csv 解析器保存/载入文件格式：

- `delimiter`：分隔值的分隔符（默认 ‘,’）
- `quotechar`：用来分隔字符串的字符（默认双引号）
- `quoting`：quote 系统（默认 `csv.QUOTE_MINIMAL`）

下面是应用举例：

```
1 >>> import csv
2 >>> db.export_to_csv_file(open('/tmp/test.txt', 'w'),
3 delimiter=',',
4 quotechar='"',
5 quoting=csv.QUOTE_NONNUMERIC)
```

这会呈现相似的

```
1 "hello"|35|"this is the text description"|"2009-03-03"
```

更多信息咨询 Python 官方文档 [67]

6.27 缓存查询

查询方法也采用一个缓存参数，默认为 `None`。为缓存之目的，它应该被设置为元组，第一个元素是缓存模型（`cache.ram`、`cache.disk` 等），第二个元素是用秒计的终止时间。

下面的例子，你看到控制器，在定义 `db.log` 表之前，其表缓存一个查询。实际查询从后端数据库得到数据，经常不会每 60 秒超过一次，存储结果在 `cache.ram` 里，下一次调用这个控制器发生在自上次数据库 I/O 60 秒内，它只从 `cache.ram` 里得到之前的数据。

```
1 def cache_db_select():
2     logs = db().select(db.log.ALL, cache=(cache.ram, 60))
3     return dict(logs=logs)
```

查询结果是复杂的 un-pickleable 对象，它们不能存在会话里，除了在这解释的以外，也不能用其它任何方式缓存。

6.28 自引用和别名

用引用到自己的字段定义表是可能的，但通常的符号会失败。下面的代码出错，因为它在定义之前用到变量 `db.person`。

```
1 db.define_table('person',
2 Field('name'),
3 Field('father_id', db.person),
4 Field('mother_id', db.person))
```

解决方法在于用替代符号

```
1 db.define_table('person',
2 Field('name'),
3 Field('father_id', 'reference person'),
4 Field('mother_id', 'reference person'))
```

事实上，db.tablename 和 “reference tablename” 是等价的字段类型。

如果表引用自己，那么不使用 SQL “AS” 关键词去执行联合 JOIN 查询一个人以及他的父母是不可能的实现的。这在 web2py 中用 with_alias 实现，下面是例子：

```
1 >>> Father = db.person.with_alias('father')
2 >>> Mother = db.person.with_alias('mother')
3 >>> db.person.insert(name='Massimo')
4 1
5 >>> db.person.insert(name='Claudia')
6 2
7 >>> db.person.insert(name='Marco', father_id=1, mother_id=2)
8 3
9 >>> rows = db().select(db.person.name, Father.name, Mother.name,
10 left=(Father.on(Father.id==db.person.father_id),
11 Mother.on(Mother.id==db.person.mother_id)))
12 >>> for row in rows:
13 print row.person.name, row.father.name, row.mother.name
14 Massimo None None
15 Claudia None None
16 Marco Massimo Claudia
```

注意，我们选择区分下面参数：

- “father_id”：字段名用在表 “person”；
- “father”：对于上述字段引用的表，我们想使用别名与数据库通信。
- “Father”：web2py 用的变量，引用那个别名。

差别很小，把它们三个用成一样的名字也不会出错：

```
1 db.define_table('person',
2 Field('name'),
3 Field('father', 'reference person'),
4 Field('mother', 'reference person'))
5 >>> father = db.person.with_alias('father')
6 >>> mother = db.person.with_alias('mother')
7 >>> db.person.insert(name='Massimo')
8 1
9 >>> db.person.insert(name='Claudia')
10 2
11 >>> db.person.insert(name='Marco', father=1, mother=2)
12 3
13 >>> rows = db().select(db.person.name, father.name, mother.name,
14 left=(father.on(father.id==db.person.father),
15 mother.on(mother.id==db.person.mother)))
16 >>> for row in rows:
17 print row.person.name, row.father.name, row.mother.name
18 Massimo None None
19 Claudia None None
20 Marco Massimo Claudia
```

但是为了构建正确的查询，清楚区别很重要。

6.29 高级特性

6.29.1 表继承 Table inheritance

可以创建包含另外表的所有字段的表，它足以把表传递并替代 `define_table` 的一个字段。
例如：

```
1 db.define_table('person', Field('name'))
2 db.define_table('doctor', db.person, Field('specialization'))
```

为了能在多个其它地方使用，也可以定义一个不存储在数据库里的虚拟表，例如：

```
1 signature = db.Table(db, 'signature',
2 Field('created_on', 'datetime', default=request.now),
3 Field('created_by', db.auth_user, default=auth.user_id),
4 Field('updated_on', 'datetime', update=request.now),
5 Field('updated_by', db.auth_user, update=auth.user_id))
6
7 db.define_table('payment', Field('amount', 'double'), signature)
```

这个例子假设标准 web2py 认证已启用。

注意如果使用 Auth，则 web2py 已创建了这样的表：

```
1 auth = Auth(db)
2 db.define_table('payment', Field('amount', 'double'), auth.signature)
```

当使用表继承，如果想继承的表也继承验证器，要确认在定义继承表之前就定义父表验证器。

6.29.2 公共 fields 和 multi-tenancy 多分租

`db.common.fields` 是属于所有表的字段列表，这列表还可以包含表，它被理解为来自表的所有字段。例如，偶尔发现需要为所有的表添加签名除了 `auth` 表。这种情况下，调用 `db.define_tables()` 后，在定义另外表之前插入

```
1 db._common_fields.append(auth.signature)
```

`request_tenant` 字段是特殊的，这个字段不存在，但却可以被创建，并把它添加到任意的表（或所有的表中）

```
1 db._common_fields.append(Field('request_tenant',
2 default=request.env.http_host, writable=False))
```

对每张叫做 `db._request_tenant` 字段的表，所有查询的记录会被自动过滤：

```
1 db.table.request_tenant == db.table.request_tenant.default
```

对于每条插入记录，这个字段设置为默认值。上面的例子中我们已经选择：

```
1 default = request.env.http_host
```

即我们选择询问我们的应用来过滤全部查询的所有表：

```
1 db.table.request_tenant == request.env.http_host
```

这个简单技巧允许我们把任何应用变换为多分租应用，也就是即使我们运行一个应用实例和我们用到单个数据库，如果应用被两个或多个域访问（例子中，域名从 `request.env.http_host` 得到），访问者看到依赖域的不同数据。想想在不同的域名下运行多个 Web stores，有一个应用程序和一个数据库。

可以用如下语句关闭多分租过滤:

```
1 rows = db(query, ignore_common_filters=True).select()
```

6.29.3 通用过滤器 Common filters

通用过滤器是上面 multi-tenant 思想的一般化, 它提供了一种简单方法去防止重复同样的查询, 考虑下表这个例子:

```
1 db.define_table('blog_post',
2 Field('subject'),
3 Field('post_text', 'text'),
4 Field('is_public', 'boolean'),
5 common_filter = lambda query: db.blog_post.is_public==True
6 )
```

任何表中的查询、删除和更新, 仅包括公开的博客 posts, 属性也能在控制器中修改:

```
1 db.blog_post._common_filter = lambda query: db.blog_post.is_public == True
```

它有两种服务类型, 一种是在blog post搜索, 避免重复“db.blog_post.is_public==True”, 另一种是安全增强, 防止忘记禁止浏览非公开的posts。

一种情况下, 你确实想要被通用过滤器滤掉的项目 (例如, 允许 admin 看非公开 posts), 可以删除过滤器:

```
1 db.blog_post._common_filter = None
```

或忽略它:

```
1 db(query, ignore_common_filters=True).select(...)
```

6.29.4 定制 Field 类型 (实验性的)

能定义新的或自定义字段类型。例如我们思考下面的例子, 一个字段包含压缩形式的二进制数据:

```
1 from gluon.dal import SQLCustomType
2 import zlib
3
4 compressed = SQLCustomType(
5 type='text',
6 native='text',
7 encoder=(lambda x: zlib.compress(x or '')),
8 decoder=(lambda x: zlib.decompress(x))
9 )
10
11 db.define_table('example', Field('data', type=compressed))
```

SQLCustomType是字段类型工厂, 它的type参数必须是标准web2py类型之一, 它告诉web2py在web2py层级如何处理字段值。只要关注数据库, native是字段名字, 允许的名字取决于数据库引擎, encoder可选变换功能, 当数据存储时应用, decoder是可选的反变换功能。

这个特征标记为实验阶段。实践中, 它在web2py使用了很长一段时间, 它确实有用但会造成代码不可移植, 例如当native type (原先类型) 是数据库专用的时。它不能用在Google

App Engine NoSQL。

6.29.5 不定义表使用 DAL

只要通过下面的语句， DAL 就能被 Python 程序使用：

```
1 from gluon import DAL, Field
2 db = DAL('sqlite://storage.sqlite', folder='path/to/app/databases')
```

即导入 DAL 和 Field，连接和指定包含 .table 文件的文件夹（应用/数据库文件夹）。

为访问数据和它的属性，我们仍需要用 db.define_tables() 定义我们将要访问的所有表。

如果我们只需要访问数据而不是 web2py 表的属性，我们无需再定义表，而是简单要求 web2py 从 .table 元数据文件读取必要信息：

```
1 from gluon import DAL, Field
2 db = DAL('sqlite://storage.sqlite', folder='path/to/app/databases',
3 auto_import=True))
```

这允许我们可以而无需再定义它，就访问任何 db.table。

6.29.6 从一个 db 复制数据到另外一个

假设你正在使用下面数据库的情况：

```
1 db = DAL('sqlite://storage.sqlite')
```

希望用不同的连接字符串移到另外一个数据库：

```
1 db = DAL('postgres://username:password@hocalhost/mydb')
```

在切换之前，你想为新的数据库移动数据并重建所有的元数据，我们假定新数据库存在，但也假定它是空的。

web2py 为你提供完成这个工作的脚本：

```
1 cd web2py
2 python scripts/cpdb.py \
3 -f applications/app/databases \
4 -y 'sqlite://storage.sqlite' \
5 -Y 'postgres://username:password@hocalhost/mydb'
```

运行脚本后，可以在模型中轻松切换连接字符串，所有的都要能够即开即用。新数据就在那儿。

脚本提供了各种各样的命令行选项，其允许你将数据从一个应用移动到另一个，移动所有的表或一些表，清除表中的数据。更多信息尝试：

```
1 python scripts/cpdb.py -h
```

6.29.7 新的 DAL 和适配器的注意事项

数据库抽象层的源代码在 2010 年完全重写，在它保持后向兼容的同时，重写令它更模块化并且更加易于扩展。这里我们解释其主要逻辑。

文件“gluon/dal.py”，除其它外，定义下列类。

```
1 ConnectionPool
2 BaseAdapter extends ConnectionPool
3 Row
```

```
4 DAL
5 Reference
6 Table
7 Expression
8 Field
9 Query
10 Set
11 Rows
```

除 **BaseAdapter** 外，它们的使用，在之前章节已解释了，当 Table 和 Set 对象的方法与数据库通信时，它们委托适配器方法产生 SQL 任务或函数调用。

例如：

```
1 db.mytable.insert(myfield='myvalue')
```

调用

```
1 Table.insert(myfield='myvalue')
```

委托适配器返回：

```
1 db._adapter.insert(db.mytable, db.mytable._listify(dict(myfield='myvalue')))
```

这里 db.mytable._listify 方法把参数字典转换为(field, value) 列表，并调用适配器的 insert 方法，db._adapter 或多或少完成下面：

```
1 query = db._adapter._insert(db.mytable, list_of_fields)
2 db._adapter.execute(query)
```

此处，第一行创建查询，第二行执行。

BaseAdapter 为所有适配器定义接口。

在写本书的时候，“gluon/dal.py” 包含下面适配器：

```
1 SQLiteAdapter extends BaseAdapter
2 JDBCSQLiteAdapter extends SQLiteAdapter
3 MySQLAdapter extends BaseAdapter
4 PostgreSQLAdapter extends BaseAdapter
5 JDBCPostgreSQLAdapter extends PostgreSQLAdapter
6 OracleAdapter extends BaseAdapter
7 MSSQLAdapter extends BaseAdapter
8 MSSQL2Adapter extends MSSQLAdapter
9 FireBirdAdapter extends BaseAdapter
10 FireBirdEmbeddedAdapter extends FireBirdAdapter
11 InformixAdapter extends BaseAdapter
12 DB2Adapter extends BaseAdapter
13 IngresAdapter extends BaseAdapter
14 IngresUnicodeAdapter extends IngresAdapter
15 GoogleSQLAdapter extends MySQLAdapter
16 NoSQLAdapter extends BaseAdapter
17 GoogleDatastoreAdapter extends NoSQLAdapter
18 CubridAdapter extends MySQLAdapter (experimental)
19 TeradataAdapter extends DB2Adapter (experimental)
20 SAPDBAdapter extends BaseAdapter (experimental)
21 CouchDBAdapter extends NoSQLAdapter (experimental)
22 MongoDBAdapter extends NoSQLAdapter (experimental)
```

覆盖了 **BaseAdapter** 的行为。

每个适配器多少有下面的结构：

```
1 class MySQLAdapter(BaseAdapter):
2
3 # specify a driver to use
4 driver = globals().get('pymysql', None)
```

```

5
6 # map web2py types into database types
7 types = {
8 'boolean': 'CHAR(1)',
9 'string': 'VARCHAR(%(length)s)',
10 'text': 'LONGTEXT',
11 ...
12 }
13
14 # connect to the database using driver
15 def __init__(self, db, uri, pool_size=0, folder=None, db_codec='UTF-8',
16 credential_decoder=lambda x:x, driver_args={},
17 adapter_args={}):
18 # parse uri string and store parameters in driver_args
19 ...
20 # define a connection function
21 def connect(driver_args=driver_args):
22 return self.driver.connect(**driver_args)
23 # place it in the pool
24 self.pool_connection(connect)
25 # set optional parameters (after connection)
26 self.execute('SET FOREIGN_KEY_CHECKS=1;')
27 self.execute("SET sql_mode='NO_BACKSLASH_ESCAPES' ;")
28
29 # override BaseAdapter methods as needed
30 def lastrowid(self, table):
31 self.execute('select last_insert_id();'

```

看着各种各样例子所示的适配器，很容易写出新的。

当 db 实例被创建：

```
1 db = DAL('mysql://...')
```

uri 字符串前缀定义了适配器，映射也在“gluon/dal.py”下面字典定义：

```

1 ADAPTERS = {
2 'sqlite': SQLiteAdapter,
3 'sqlite:memory': SQLiteAdapter,
4 'mysql': MySQLAdapter,
5 'postgres': PostgreSQLAdapter,
6 'oracle': OracleAdapter,
7 'mssql': MSSQLAdapter,
8 'mssql2': MSSQL2Adapter,
9 'db2': DB2Adapter,
10 'teradata': TeradataAdapter,
11 'informix': InformixAdapter,
12 'firebird': FireBirdAdapter,
13 'firebird_embedded': FireBirdAdapter,
14 'ingres': IngresAdapter,
15 'ingresu': IngresUnicodeAdapter,
16 'sapdb': SAPDBAdapter,

```

```

17 'cubrid': CubridAdapter,
18 'jdbc:sqlite': JDBCSQLiteAdapter,
19 'jdbc:sqlite:memory': JDBCSQLiteAdapter,
20 'jdbc:postgres': JDBCPostgreSQLAdapter,
21 'gae': GoogleDatastoreAdapter, # discouraged, for backward compatibility
22 'google:datastore': GoogleDatastoreAdapter,
23 'google:sql': GoogleSQLAdapter,
24 'couchdb': CouchDBAdapter,
25 'mongodb': MongoDBAdapter,
26 }

```

uri字符串被适配器自己解析得更清晰。

对于任何适配器，都可以用不同的驱动替换：

```

1 from gluon.dal import MySQLAdapter
2 MySQLAdapter.driver = mysqldb

```

指定可选驱动参数和适配器参数：

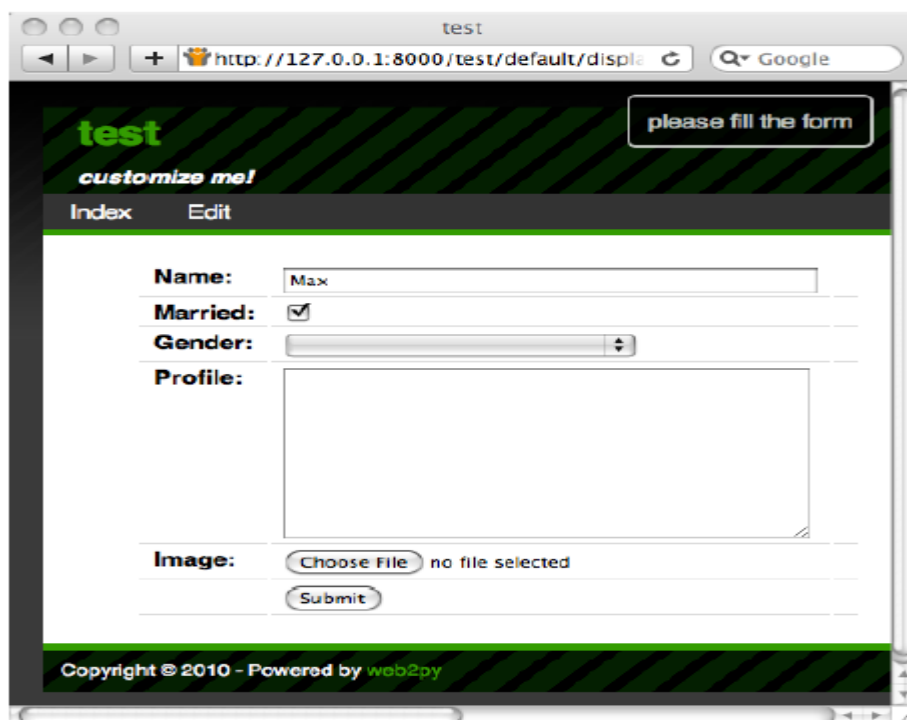
```

1 db =DAL(..., driver_args={}, adapter_args=())

```

6.29.8 Gotchas 陷阱

SQLite不支持删除和修改列，这意味着web2py迁移工作会变为下列情况，如果从表中删除一个字段，列仍然在数据库中但对web2py不可见；如果决定再安置列，web2py将会再创建它但不成功。这种情况，必须设置fake_migrate=True，以使元数据重建而不必再添加列，同样原因，SQLite也不清楚列类型的任何改变，如果在string字段插入数字，它被存为字符



串；如果在后面改变模型，用类型integer替换类型string，SQLite会继续保持数字为字符串，当试图提取数据的时候会引发问题。

MySQL在单个事务范围不支持多ALTER TABLE，这意味着任何迁移操作会分解为多个commit。如果出现失败，它造成迁移中断（web2py元数据不再与真实数据库表结构保持同

步），虽不走运，但能够避免（一次迁移一张表）或后验更正（恢复web2py模型到在数据库相应的表结构，设置fake_migrate=True；元数据被重建后，设置fake_migrate=False，再次迁移表格。）

Google SQL 和MySQL一样存在相同的问题，且更多些。特别是表元数据本身必须存在数据库的表中，不被web2py迁移，这是因为Google App Engine有只读文件系统。Google中web2py迁移：上述SQL结合MySQL问题会导致元数据损坏。这会再一次被阻止（我迁移表一次，然后设置migrate=False，元数据表不能再被访问，或它能后验修复（使用Google dashboard访问数据库，从叫做 web2py_filesystem的表删除任何损坏的条目）

MSSQL不支持SQL OFFSET关键词。因此数据库并不能分页，当执行limitby=(a,b) web2py会得到最初的b（数量）行，而丢弃最初的a（数量），与其它数据库引擎比，这会导致相当大的开销。

Oracle也不支持分页。它既不支持OFFSET也不支持LIMIT关键词，web2py实现分页但把db(...) select(limitby=(a,b))翻译成复杂的三层嵌套查询（**Oracle**官方文档建议），简单查询有效，但可能因涉及别名字段和联合查询而中断。

MSSQL在ONDELETE CASCADE表中，存在循环引用的问题。这是MSSQL漏洞，可以对所有引用字段设置ondelete属性为“NO ACTION”克服它，也能在定义表前对所有的设置一次。

```
1 db = DAL('mssql://...')
2 for key in ['reference', 'reference FK']:
3     db._adapter.types[key]=db._adapter.types[key].replace(
4     '%(on_delete_action)s', 'NO ACTION')
```

MSSQL也有参数传递给DISTINCT关键词的问题，因此它这样工作：

```
1 db(query).select(distinct=True)
```

而不是：

```
1 db(query).select(distinct=db.mytable.myfield)
```

Google NoSQL（数据存储）不允许联合、左联、聚类、表达式，多于一张表的OR，like运算及在“text”字段中搜索。事务受限制，不会被web2py自动提供（需要用Google API run_in_transaction，可以在Google App Engine在线文档中查找到），Google也限制每次能检索到的记录数（写书时是1000条），Google数据存储记录ID是整型，但它们不连续，当SQL“list:string”类型映射到“text”类型，在Google数据存储，它被映射为ListStringProperty。相同地，“list:integer”和“list:reference”被映射为“ListProperty”，这使得在Google NoSQL比在SQL数据库更高效地搜索这些字段类型内容。

第7章 表单和验证器 Forms and validators

在 web2py 中有四种不同的方法创建表单：

- FORM 就 HTML 帮助对象，提供低级别实现，FORM 对象能序列化为 HTML，并知道它包含的字段，FORM 对象知道怎样验证所提交表单的值。
- SQLFORM 提供创建高级别的 API，从现存的数据库表 create（创建）、update（更改）和 delete（删除）表单。
- FORM.factory 是 SQLFORM 顶端的抽象层，即使没有数据库存在也能利用表单生成特性。它从表的描述生成与 SQLFORM 非常相似的表单，无需创建数据库表。
- CRUD 方法，这在功能上等同于 SQLFORM，也基于 SQLFORM，但提供更紧凑的符号。

所有这些表单是自知的，如果输入不能通过验证，它们能自己修改并增加出错消息，有关验证的变量和验证生成的出错提示，表单能被查询。

使用帮助对象，任意的 HTML 代码，都能被插入表单或从表单提出。

FORM 和 SQLFORM 是帮助对象，能用像 DIV 相似的方法一样操作，例如可以设置表单风格：

```
1 form = SQLFORM(..)
2 form['_style'] = 'border:1px solid black'
```

7.1 表单 Forms

考虑以下具有 “default.py” 控制器的 test 应用的例子：

```
1 def display_form():
2 return dict()
```

关联 “default/display_form.html” 视图：

```
1 {{extend 'layout.html'}}
2 <h2>Input form</h2>
3 <form enctype="multipart/form-data"
4 action="{{=URL()}}" method="post">
5 Your name:
6 <input name="name" />
7 <input type="submit" />
8 </form>
9 <h2>Submitted variables</h2>
10 {{=BEAUTIFY(request.vars)}}
```

这是要求用户名的正则 HTML 表单，当填写表单时，按了提交按钮，表单自己提交，变量 request.vars.name 和它的值显示在底部。

可以用帮助对象生成同样的表单，这能在视图或动作中完成；因为 web2py 在动作中处理表单，在动作中定义表单都是可以的。

下面是新的控制器：

```
1 def display_form():
```

```
2 form=FORM('Your name:', INPUT(_name='name'), INPUT(_type='submit'))
3 return dict(form=form)
```

关联“default/display_form.html”视图：

```
1 {{extend 'layout.html'}}
2 <h2>Input form</h2>
3 {{=form}}
4 <h2>Submitted variables</h2>
5 {{=BEAUTIFY(request.vars)}}
```

迄今，代码与之前的是相同的，但表单由语句`{{=form}}`生成，序列化FORM对象。现在通过增加表单验证和处理，我们增加了一级复杂程度。

改变控制器如下：

```
1 def display_form():
2 form=FORM('Your name:',
3 INPUT(_name='name', requires=IS_NOT_EMPTY()),
4 INPUT(_type='submit'))
5 if form.accepts(request, session):
6 response.flash = 'form accepted'
7 elif form.errors:
8 response.flash = 'form has errors'
9 else:
10 response.flash = 'please fill the form'
11 return dict(form=form)
```

关联“default/display_form.html”视图：

```
1 {{extend 'layout.html'}}
2 <h2>Input form</h2>
3 {{=form}}
4 <h2>Submitted variables</h2>
5 {{=BEAUTIFY(request.vars)}}
6 <h2>Accepted variables</h2>
7 {{=BEAUTIFY(form.vars)}}
8 <h2>Errors in form</h2>
9 {{=BEAUTIFY(form.errors)}}
```

注意：

- 动作中，我们为输入字段“name”增加`requires=IS_NOT_EMPTY()`验证器。
- 动作中，我们增加对`form.accepts(...)`调用。
- 视图中，我们显示`form.vars`和`form.errors`，以及表单和`request.vars`。

所有工作通过form对象方法`accepts`来完成，它根据声明的要求（验证器表达）过滤`request.vars`，`accepts`方法存储那些通过验证的变量到`form.vars`，如果字段值不满足要求，失败的验证器返回错误，错误存放在`form.errors`，`form.vars`和`form.errors`都是`glunon.storage`对象，Storage对象与`request.vars`类似，前者包含通过验证的值，例如：

```
1 form.vars.name = "Max"
```

后者包含错误，例如：

```
1 form.errors.name = "Cannot be empty!"
```

`accepts`方法完整描述如下：

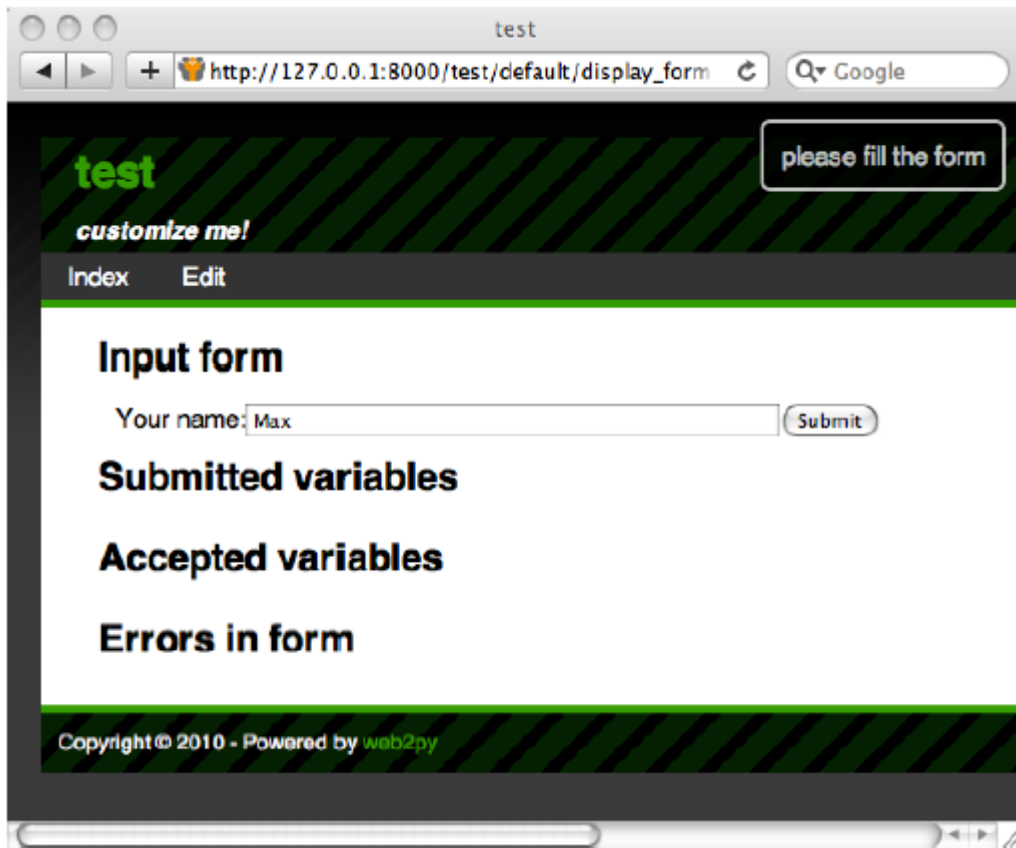
```
1 form.accepts(vars, session=None, formname='default',
2 keepvalues=False, onvalidation=None,
3 dbio=True, hideerror=False):
```


可选参数的意义将在下一小节阐述。

第一个参数可以是`request.vars`、`request.get_vars`、`request.post_vars`或简单`request`，后者等同与接受像输入的`request.post_vars`一样。

如果表单被接受，`accepts`函数返回`True`，否者返回`False`。如果出错或当它没有提交（例如，第一时间它只是显示），表单不被接受。

下面是这个网页第一次显示看起来的样子：



无效输入提交时它看起来像：

The screenshot shows a web browser window titled "test" with the URL "http://127.0.0.1:8000/test/default/display_form". The page has a green header with the text "test" and "customize me!". Below the header is a navigation bar with "Index" and "Edit" links. The main content area is titled "Input form" and contains a text input field labeled "Your name:". A red error message "enter a value" is displayed below the input field. A "Submit" button is located below the error message. Below the "Input form" section are two sections: "Submitted variables" and "Accepted variables", both showing "name :". At the bottom is a section titled "Errors in form". A message box in the top right corner says "form has errors".

下图是合法输入提交的页面：

The screenshot shows the same web browser window after a successful submission. The "form accepted" message box is now in the top right corner. The "Submitted variables" section now displays "name : Max". The "Accepted variables" section also displays "name : Max". The "Input form" section still shows the "Your name:" input field and the "Submit" button, but the red error message is gone. The "Errors in form" section is still present at the bottom.

7.1.1 process 和 validate 方法

下面语句的快捷方式:

```
1 form.accepts(request.post_vars, session,...)
```

是

```
1 form.process(...).accepted
```

后者不需要request和session参数（虽然可以选择指定它们），它与accepts不同，因为它自己返回表单。在内部，process调用accepts并传递参数给它，accepts返回的参数存储在form.accepted中。

process函数采用一些accepts没有用的额外的参数：

- message_onsuccess
- onsuccess: 如果等于‘flash’（默认）并且表单被接受了，它会闪现上面的message_onsuccess

```
1 -  
    message_onfailure
```

```
1 -  
onfailure: 如果等于‘flash’（默认）并且表单验证失败，它会闪现上面的  
message_onfailure
```

- next: 表单接受之后用户重定向

onsuccess和onfailure能做象lambda表单一样的函数：do_something(form)。

```
1 form.validate(...)
```

是下面的快捷形式

```
1 form.process(...,dbio=False).accepted
```

7.1.2 隐藏字段Hidden fields

当上面的表单对象用`{=form}`序列化，因为之前accepts方法的调用，它现在看起来如下：

```
1 <form enctype="multipart/form-data" action="" method="post">  
2 your name:  
3 <input name="name" />  
4 <input type="submit" />  
5 <input value="783531473471" type="hidden" name="_formkey" />  
6 <input value="default" type="hidden" name="_formname" />  
7 </form>
```

注意出现两个隐藏字段：“_formkey”和“_formname”，它们的出现是调用accepts触发的，它们起到两个不同但很重要的作用：

- 隐藏字段“_formkey”是一次性令牌，web2py用来阻止表单重复提交。当表单序列化并存在session中，这个字段值生成；当表单提交，这个值必须匹配，否则accepts返回False不报错，好像表单根本没有被提交，这是因为web2py不能决定表单是否被正确提交。
- 隐藏字段“_formname” web2py生成作为表单的名字，但名字可能被覆盖，这个字段是需要的，允许包含和处理多个表单的页，web2py用名字区别所提交的表单。
- 可选隐藏字段指定为FORM(...,hiddent=dict(...))。

如果上面的表单用空的“name”字段提交，表单不能通过验证，当表单再次列化，它呈现如

下:

```
1 <form enctype="multipart/form-data" action="" method="post">
2 your name:
3 <input value="" name="name" />
4 <div class="error">cannot be empty!</div>
5 <input type="submit" />
6 <input value="783531473471" type="hidden" name="_formkey" />
7 <input value="default" type="hidden" name="_formname" />
8 </form>
```

注意，在序列化的表单中出现的“error”类DIV。web2py在表单中插入这个错误消息告知访问者不通过验证的字段，accepts方法就提交而言，决定表单提交，检查“name”字段是否空，是否需要，最终从验证器插入错误消息到表单。

基本“layout.html”视图期望用来处理“error”类DIV，默认布局采用jQuery效果显示和向下滑动错误并用红色背景。查看11章获取更多详细信息。

7.1.3 keepvalues 参数

可选参数 keepvalues，在表单被接受和没有重定向时，告诉 web2py 做什么，因此同一个表单再次被显示。默认，表单清空，如果 keepvalues 设置为 True，表单用之前插入的值预处理，当有表单设想用来重复插入多条相似的记录时，这很有效用，如果 dbio 参数设置为 False，web2py 接受表单后不执行任何 DB 插入/更改操作；如果 hideerror 设置为 True，表单又有错误，表单呈现时，这不会被显示（这取决于你从 form.errors 显示它们，不论怎样）。Onvalidation 参数下面介绍。

7.1.4 onvalidation 参数

onvalidation 参数是 None 或是函数，函数用表单参数而无返回值，这个函数验证后立即（如果验证通过）被调用并传递表单，在其它任何发生之前，这个函数的目的是多方面的。例如它可以对表单执行附加检查，最后给表单添加出错，也可以用来计算一些基于其它字段值的字段的值，它能用来在记录创建/更改之前，触发一些动作（如发送电子邮件）。

下面是例子：

```
1 db.define_table('numbers',
2 Field('a', 'integer'),
3 Field('b', 'integer'),
4 Field('c', 'integer', readable=False, writable=False))
5
6 def my_form_processing(form):
7 c = form.vars.a * form.vars.b
8 if c < 0:
9 form.errors.b = 'a*b cannot be negative'
10 else:
11 form.vars.c = c
12
13 def insert_numbers():
14 form = SQLFORM(db.numbers)
15 if form.process(onvalidation=my_form_processing).accepted:
16 session.flash = 'record inserted'
17 redirect(URL())
18 return dict(form=form)
```

7.1.5 检测记录修改

当填写表单修改记录时，另外一个用户同时编辑这一条记录，很小可能性事件。因此，当我们保存记录时，我们想检查可能的冲突。如下实现：

```
1 db.define_table('dog',Field('name'))
2
3 def edit_dog():
4     dog = db.dog(request.args(0)) or redirect(URL('error'))
5     form=SQLFORM(db.dog,dog)
6     form.process(detect_record_change=True)
7     if form.record_changed:
8         # do something
9     elif form.accepted:
10        # do something else
11    else:
12        # do nothing
13    return dict(form=form)
```

7.1.6 表单和重定向

最通常使用表单的方法是通过自提交，因为所提交字段变量被生成表单同样的工作处理，一旦表单接受了，再次显示当前的页面是不寻常的（这里我们所做的仅是简单保持），更常见的是重定向，访问者可到“next”页面。

下面是新例子的控制器：

```
1 def display_form():
2     form = FORM('Your name:',
3     INPUT(_name='name', requires=IS_NOT_EMPTY()),
4     INPUT(_type='submit'))
5     if form.process().accepted:
6         session.flash = 'form accepted'
7         redirect(URL('next'))
8     elif form.errors:
9         response.flash = 'form has errors'
10    else:
11        response.flash = 'please fill the form'
12    return dict(form=form)
13
14 def next():
15    return dict()
```

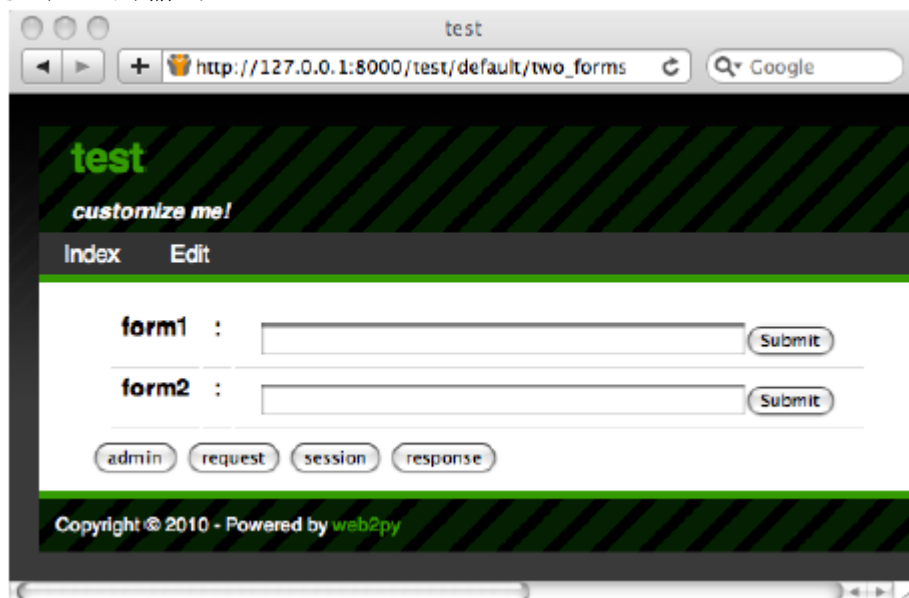
为在下一页面设置 flash 而不是当前页面，必须使用 session.flash 而不是 response.flash，重定向后，web2py 把前者移到后者，注意使用 session.flash 要求不用 session.forget()。

7.1.7 多表单每页面

这节内容应用到 FORM 和 SQLFORM 对象，多表单每页面是可能的，但必须允许 web2py 区分它们，如果这些通过 SQLFORM 从不同表得到，那么 web2py 自动给它们不同的名字，否则需要清晰地给它们不同的表单名字。下面是例子：

```
1 def two_forms():
2     form1 = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
3                 INPUT(_type='submit'))
4     form2 = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
5                 INPUT(_type='submit'))
6     if form1.process(formname='form_one').accepted:
7         response.flash = 'form one accepted'
8     if form2.process(formname='form_two').accepted:
9         response.flash = 'form two accepted'
10    return dict(form1=form1, form2=form2)
```

下面是它产生的输出：



当访问者提交空 form1，仅 form1 显示错误；如果访问者提交空 form2，仅 form2 显示错误消息。

7.1.8 共享表单

这节内容应用到 FORM 和 SQLFORM 对象。这里讨论的是可能性不是推荐，因而用表单自提交总是好的实践，但有时没有选择，因为发送表单的动作和接收表单的动作属不同的应用，能够生成表单提交给不同动作，这通过在 FORM 或 SQLFORM 对象属性中指定处理动作 URL 实现。例如：

```
1 form = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
2             INPUT(_type='submit'), _action=URL('page_two'))
3
4 def page_one():
5     return dict(form=form)
```

```

6
7 def page_two():
8 if form.process(session=None, formname=None).accepted:
9 response.flash = 'form accepted'
10 else:
11 response.flash = 'there was an error in the form'
12 return dict()

```

注意，因为“page_one”和“page_two”使用同样的 form，我们仅需定义一次，为了不重复我们自己，在所有动作之外替换它。控制器开头的公共部分代码每次在给调用动作控制前被执行，因“page_one”不调用 process（不调用 accepts），表单没有名字和键，必须在 process 传递 session=None 和设置 formname=None，或者当“page_two”接收它，表单不验证。

7.2 SQLFORM

现在，通过用模型文件提供应用，我们来到下一阶段：

```

1 db = DAL('sqlite://storage.sqlite')
2 db.define_table('person', Field('name', requires=IS_NOT_EMPTY()))

```

修改控制器如下：

```

1 def display_form():
2 form = SQLFORM(db.person)
3 if form.process().accepted:
4 response.flash = 'form accepted'
5 elif form.errors:
6 response.flash = 'form has errors'
7 else:
8 response.flash = 'please fill out the form'
9 return dict(form=form)

```

视图无需修改。

新控制器中，无需构建 FORM，因为 SQLFORM 构造函数从定义在模型中的表 db.person 创建了一个。这个新表单，序列化呈现如下：

```

1 <form enctype="multipart/form-data" action="" method="post">
2 <table>
3 <tr id="person_name__row">
4 <td><label id="person_name__label"
5 for="person_name">Your name: </label></td>
6 <td><input type="text" class="string"
7 name="name" value="" id="person_name" /></td>
8 <td></td>
9 </tr>
10 <tr id="submit_record__row">
11 <td></td>
12 <td><input value="Submit" type="submit" /></td>
13 <td></td>
14 </tr>
15 </table>
16 <input value="9038845529" type="hidden" name="_formkey" />
17 <input value="person" type="hidden" name="_formname" />
18 </form>

```


自动生成的表单比之前低级别表单要复杂得多。首先，它有若干行的表，每行有三列。第一列包含字段标签（从 db.person 决定的），第二列包含输入字段（最终错误消息），和第三列是可选的，因此空（它能用 SQLFORM 构造函数中的字段填充）。

表单中的所有标签有从表和字段名继承的名字，这允许用 CSS 和 JavaScript 方便表单定制，这种能力在 11 章更详细讨论。

更重要的是，现在 accepts 方法为你做了更大量的工作，正如前面的例子，它执行输入验证，额外地，如果输入通过验证，它也执行数据库插入新的记录和存储新记录唯一“id”在 form.vars.id。

SQLFORM 对象，通过存储上传文件到“upload”文件夹，也自动处理“upload”字段（在重新安全地命名之后以避免冲突，并防止目录遍历攻击），存储它们的名字（它们的新名字）到数据库合适字段。在表单被处理后，新的文件名在 form.vars.fieldname（即在 form.vars.fieldname 里替换了 cgi.FieldStorage 对象），因而恰好上传后就可以容易地引用新名。

SQLFORM 用复选框显示“boolean”值，文本框显示“text”值，值要在有限范围或下拉框，“upload”字段有允许用户下载已上传文件的链接。它隐藏了“blob”字段，因为它们被认为是经过不同处理，正如后面讨论那样。

例如，考虑下面的模型：

```
1 db.define_table('person',
2 Field('name', requires=IS_NOT_EMPTY()),
3 Field('married', 'boolean'),
4 Field('gender', requires=IS_IN_SET(['Male', 'Female', 'Other'])),
5 Field('profile', 'text'),
6 Field('image', 'upload'))
```

这种情况，SQLFORM(db.person) 生成如下所示的表单：

The screenshot shows a web browser window with the address bar displaying 'http://127.0.0.1:8000/test/default/display'. The page has a dark green header with the word 'test' in green and 'customize me!' in white. A button labeled 'please fill the form' is in the top right. Below the header is a navigation bar with 'Index' and 'Edit' links. The main form area contains the following fields:

- Name:** A text input field containing 'Max'.
- Married:** A checkbox that is checked.
- Gender:** A dropdown menu with a downward arrow.
- Profile:** A large text area.
- Image:** A section with a 'Choose File' button and the text 'no file selected'.

At the bottom of the form is a 'Submit' button. The footer of the page says 'Copyright © 2010 - Powered by web2py'.

SQLFORM构造函数允许各种定制，如仅显示字段子集，修改标签，添加值到可选第三列，或创建UPDATE和DELETE表单，相对于INSERT表单像当前这个，SQLFORM是web2py里单个最大的节省时间的对象，SQLFORM类定义在“gluon/sqlhtml.py”中，它能够通过超越它的xml方法容易地扩展，方法序列化对象，修改它的输出。

SQLFORM构造函数的特征如下：

```
1 SQLFORM(table, record = None,
2 deletable = False, linkto = None,
3 upload = None, fields = None, labels = None,
4 col3 = {}, submit_button = 'Submit',
5 delete_label = 'Check to delete:',
6 showid = True, readonly = False,
7 comments = True, keepopts = [],
8 ignore_rw = False, record_id = None,
9 formstyle = 'table3cols',
10 buttons = ['submit'], separator = ': ',
11 **attributes)
```

- 可选第二参数为指定的记录（参看下一节），把INSERT表单转换为UPDATE表单。
- 如果deletable设置为True，UPDATE表单显示“Check to delete”复选框，这个字段的标签值通过delete_label参数设置。
- submit_button设置提交按钮的值。
- id_label设置记录“id”标签。
- 如果showid设置为False，记录“id”不显示。
- Fields是你想显示的可选字段名列表。如果列表提供，仅列表中的字段显示，例如：

```
1 fields = ['name']
```

- labels是字段标签字典。字典键是字段名，相应的值是作为标签显示的，如果标签没有提高，web2py从字段名继承标签（它大写字段名，把下划线替换为空格）。例如：

```
1 labels = {'name': 'Your Full Name:'}
```

- col3是第三列值的字典。例如：

```
1 col3 = {'name': A('what is this?',
2 _href='http://www.google.com/search?q=define:name')}
```

- linkto 和 upload 对用户定义控制器可选URL，允许表单处理引用字段，这在本章后详细讨论。
- readonly，如果设置为True，只读显示表单
- comments，如果设置为False，不显示col3评论
- ignore_rw，正常情况，对于create/update表单，标示writable=True的字段才显示，对与只读表单，标示readable=True的字段才显示。设定ignore_rw=True导致哪些限制被忽略，所有字段都显示，这大多数用在appadmin接口为每个表单显示所有字段，其超过模型指示。
- formstyle决定在html序列化表单所使用的风格。它可以是“table3cols”（默认）、“table2cols”（一行为标签和评论，一行做输出）、“ul”（生成无序的输入字段列表）以及“divs”（表示使用css友好divs的表单，随意定制），Formstyle也能是函数带(record_id, field_label, field_widget, filed_comment)属性的参数，返回TR()对象。
- 是INPUTs 或 TAG.BUTTONs（虽然技术上可以是任何帮助对象的组合）列表，添加到DIV，提交按钮所在。
- separator设置字符串区分表单标签和表单输入字段。
- 可选attributes是以下划线开头的参数，你想传递它给呈现SQLFORM对象的FORM tag。

例如：

```
1 _action = '.'  
2 _method = 'POST'
```

有一个特别属性。当字典传递时，它的项目翻译为“hidden” INPUT字段（参看第5章FORM帮助对象的例子）。

```
1 form = SQLFORM(..., hidden=...)
```

导致隐藏字段在提交时不多不少地传递，form.accepts(...)不打算读取接收到的隐藏字段并把它们移到form.vars，原因是考虑安全。隐藏字段可能被篡改，因此需要明确地把隐藏字段从请求移到表单：

```
1 form.vars.a = request.vars.a  
2 form = SQLFORM(..., hidden=dict(a='b'))
```

7.2.1 SQLFORM 和 insert/update/delete

表单接受时，SQLFORM创建一条新db记录，假设form=SQLFORM(db.test)，那么最后创建的记录的id能在myform.vars.id访问到。

如果把记录作为可选第二参数传递到SQLFORM构造函数，表单成为该记录UPDATE表单，这意味着当提交表单，现有记录更新，没有新记录插入。如果设置参数deletable=True，UPDATE表单显示“check to delete”复选框，如果检查，记录被删除。

如果表单提交并且删除复选框选定，属性form.deleted设置为True。

修改之前例子的控制器，以便当我们传递附加整型参数到URL路径，如下：

```
1 /test/default/display_form/2
```

如果有相应id的记录，SQLFORM为记录生成UPDATE/DELETE表单：

```
1 def display_form():  
2 record = db.person(request.args(0)) or redirect(URL('index'))  
3 form = SQLFORM(db.person, record)  
4 if form.process().accepted:  
5 response.flash = 'form accepted'  
6 elif form.errors:  
7 response.flash = 'form has errors'  
8 return dict(form=form)
```

第二行找到记录，第三行产生UPDATE/DELETE表单，第4行做所有相应的表单处理。

更新表单与创建表单非常相似，除了更新表单是用当前记录预填充的而且它预览图像。默认deletable = True，更新表单显示“delete record”选项。

Edit表单也包含隐藏INPUT字段name="id"，用来识别记录。这个id为了更多的安全考虑也存储在服务器侧，如果访问者篡改了这个字段的值，UPDATE不执行，web2py产生语法错，

“user is tampering with form”。

当字段用writable=False标示，字段不在创建表单中显示，在更新表单只读显示。如果字段用writable=False和readable=False标示，那么字段根本不显示，即使是在更新表单。

如下创建表单：

```
1 form = SQLFORM(..., ignore_rw=True)
```

忽略readable和writable属性，总是显示所有字段，appadmin中表单默认忽略它们。

如下创建表单：

```
1 form = SQLFORM(table, record_id, readonly=True)
```

总是只读模式显示所有字段，它们不能被接受。

7.2.2 HTML 中的 SQLFORM

有时你想用SQLFORM从它的生成和处理获益，但需要在HTML中定制表单，不可以用SQLFORM对象参数获得，因此不得不用HTML设计表单。

现在，编辑之前的控制器并添加新动作：

```
1 def display_manual_form():
2 form = SQLFORM(db.person)
3 if form.process(session=None, formname='test').accepted:
4 response.flash = 'form accepted'
5 elif form.errors:
6 response.flash = 'form has errors'
7 else:
8 response.flash = 'please fill the form'
9 # Note: no form instance is passed to the view
10 return dict()
```

插入表单在相关联的“default/display_manual_form.html”视图：

```
1 {{extend 'layout.html'}}
2 <form>
3 <ul>
4 <li>Your name is <input name="name" /></li>
5 </ul>
6 <input type="submit" />
7 <input type="hidden" name="_formname" value="test" />
8 </form>
```

注意动作不返回表单，因为它不需要传递到视图。视图包含在HTML中手动创建的表单，表单包含隐藏字段“_formname”，必须是同样formname，指定动作中accepts的参数，Web2py用表单名，如果在同一页面有多个表单，决定那一个是提交的；如果页面包含单个表单，可以设置formname=None并在视图中忽略隐藏字段。

form.accepts 在response.vars中查找数据，与数据库表db.person字段匹配，这些字段在HTML中以格式<input name="field_name_goes_here" />被声明。

注意给定例子中，表单变量作为参数传递给URL，如果不需要，POST协议必须指定。

更要注意，如果上传字段指定，表单需要设定允许。下面展示两个选项：

```
1 <form enctype="multipart/form-data" method="post">
```

7.2.3 SQLFORM 和上传

“upload”字段类型是特殊的。它们以type="file" INPUT字段呈现，除非另有指定，上传文件用缓冲流式传送，用一个自动指定的安全名字存储在应用的“upload”文件夹，这个文件的名字保存到上传类型的字段。

作为例子，考虑下面的模型：

```
1 db.define_table('person',
2 Field('name', requires=IS_NOT_EMPTY()),
3 Field('image', 'upload'))
```

可以用同样的控制动作“display_form”显示上例。

当插入新的记录，表单允许浏览文件。例如选择jpg图像，如下文件上传并存储：

```
1 applications/test/uploads/person.image.XXXXX.jpg
```

“XXXXXX”是web2py为文件指定的随机标识符。

注意，默认情况下，上传文件的源文件名是b16编码，并为文件创建新文件名，这个名字通过默认“download”动作得到，用来设定到源文件名内容配置头。

仅它的扩展被保留，这是个安全性要求，因为文件名包含特殊字符，可能允许访问者执行目录遍历攻击或其它恶意操作。

新文件名存储在form.vars.image。

当用UPDATE表单编辑记录，显示到现存上传文件的链接是很好的，web2py提供了方法实现。如果通过上传参数传递URL给SQLFORM构造函数，web2py在那个URL使用动作下载文件。考虑下面的动作：

```
1 def display_form():
2 record = db.person(request.args(0)) or redirect(URL('index'))
3 form = SQLFORM(db.person, record, deletable=True,
4 upload=URL('download'))
5 if form.process().accepted:
6 response.flash = 'form accepted'
7 elif form.errors:
8 response.flash = 'form has errors'
9 return dict(form=form)
10
11 def download():
12 return response.download(request, db)
```

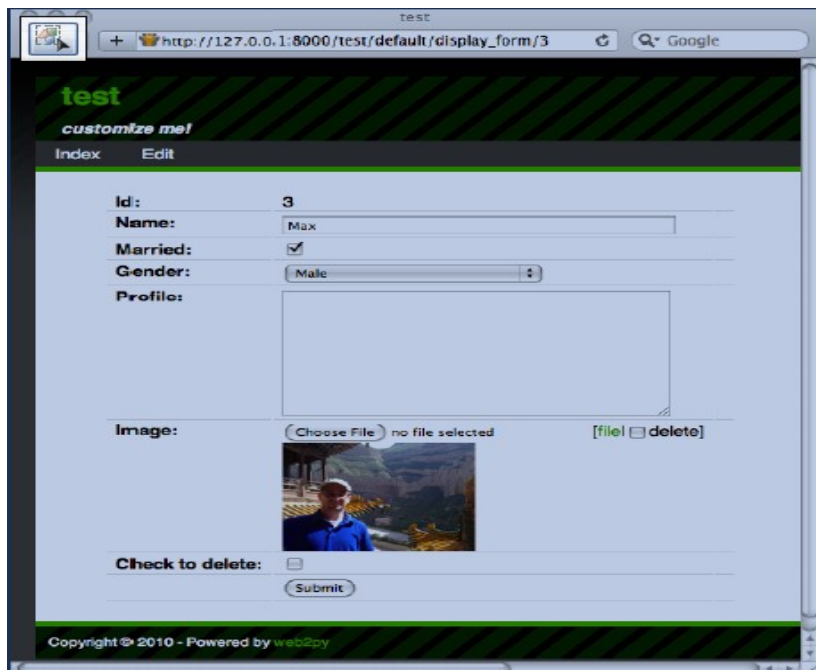
现在，在URL插入新的记录：

```
1 http://127.0.0.1:8000/test/default/display_form
```

通过访问如下，上传图像，提交表单，编辑先创建的记录：

```
1 http://127.0.0.1:8000/test/default/display_form/3
```

（这里我们假定最后记录id=3）。表单将显示图像如下的预览：



这个表单，当序列化时，生成如下HTML：

```
1 <td><label id="person_image__label" for="person_image">Image: </label></td>
2 <td><div><input type="file" id="person_image" class="upload" name="image"
3 /><a href="/test/default/download/person.image.0246683463831.jpg">file</a>|
4 <input type="checkbox" name="image__delete" />delete</div></td></td></tr>
5 <tr id="delete_record__row"><td><label id="delete_record__label" for="delete_record
6 >Check to delete:</label></td><td><input type="checkbox" id="delete_record"
7 class="delete" name="delete_this_record" /></td>
```

包含允许下载上传文件的链接，从数据库记录删除文件的复选框，这样存储NULL在“image”字段。



为什么这种机制显露？为什么需要写下载函数？因为你想在下载函数中强化一些授权机制。参看第9章的例子。

正常上传文件存储在“app/uploads”，但可以指定另外位置：

```
1 Field('image', 'upload', uploadfolder='...')
```

大多数操作系统，当许多文件在同一个文件夹，访问文件系统会变慢。如果打算上传超过1000个文件，可以要求web2py在子文件夹中组织上传：

```
1 Field('image', 'upload', uploadseparate=True)
```

7.2.4 存储源文件名

Web2py自动存储源文件名在新UUID文件名，当文件被下载时检索它，一旦下载成功，源文件名存储在HTTP响应内容处置报头，它无需编程地全部透明完成。有时你想把源文件名存储在数据库字段中，这种情况，需要修改模型并增加字段来存储它：

```
1 db.define_table('person',
2 Field('name', requires=IS_NOT_EMPTY()),
3 Field('image_filename'),
4 Field('image', 'upload'))
```


然后需要修改控制器处理它：

```
1 def display_form():
2     record = db.person(request.args(0)) or redirect(URL('index'))
3     url = URL('download')
4     form = SQLFORM(db.person, record, deletable=True,
5     upload=url, fields=['name', 'image'])
6     if request.vars.image!=None:
7         form.vars.image_filename = request.vars.image.filename
8     if form.process().accepted:
9         response.flash = 'form accepted'
10    elif form.errors:
11        response.flash = 'form has errors'
12    return dict(form=form)
```

注意SQLFORM不显示“image_filename”字段。“display_form”动作把request.vars.image的文件名移到form.vars.image_filename，以便被accepts处理，并存储在数据库。下载函数，在服务文件前，在数据库中检查源文件名，在内容处置头中使用它。

7.2.5 autodelete 属性（自动删除）

SQLFORM，删除记录，不删除记录引用到的物理上传文件，原因是Web2py不知道该文件是否被使用/链接其它表或被用于其它目的，如果你知道当相应记录被删除时，删除实际文件是安全的，可作如下操作：

```
1 db.define_table('image',
2 Field('name', requires=IS_NOT_EMPTY()),
3 Field('file', 'upload', autodelete=True))
```

Autodelete 属性默认是 False。当设置为 True，当记录被删除，确定文件也被删除。

7.2.6 链接到引用记录

现在，考虑两张表被引用字段链接的情况。例如：

```
1 db.define_table('person',
2 Field('name', requires=IS_NOT_EMPTY()))
3 db.define_table('dog',
4 Field('owner', db.person),
5 Field('name', requires=IS_NOT_EMPTY()))
6 db.dog.owner.requires = IS_IN_DB(db, db.person.id, '%(name)s')
```

一个人有几条狗，每条狗归属于一主人，主人是人。

狗主人要求用 ‘%(name)s’ 来引用合法 db.person.id。

让我们用这个应用的 appadmin 接口添加一些人和他们的狗。

当编辑已有的人，appadmin UPDATE 表单显示到页面的链接，列出属于这个人的狗，这个动作能用 SQLFORM 的参数 linkto 来复制，Linkto 指向从 SQLFORM 接收查询字符串新动作 URL，并列出相应记录。下面是例子：

```
1 def display_form():
2     record = db.person(request.args(0)) or redirect(URL('index'))
3     url = URL('download')
4     link = URL('list_records', args='db')
5     form = SQLFORM(db.person, record, deletable=True,
6     upload=url, linkto=link)
```

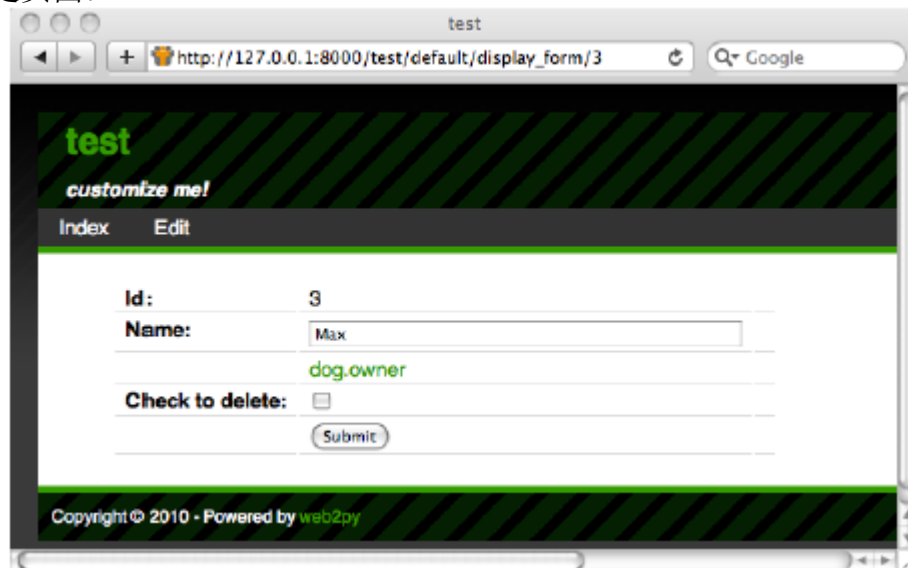


```

7 if form.process().accepted:
8     response.flash = 'form accepted'
9 elif form.errors:
10    response.flash = 'form has errors'
11    return dict(form=form)

```

如下是页面：



有一个叫做“dog.owner”的链接。这个链接的名字能通过 SQLFORM 的参数 labels 修改，例如：

```

1 labels = {'dog.owner': "This person's dogs"}

```

如果点击链接指向到：

```

1 /test/default/list_records/dog?query=dog.owner%3D5

```

“list_records”是指定的动作，用 request.args(0) 设置引用表名字以及用 request.vars.query 设置 SQL 查询字符串。

URL 中的查询字符串包含值“dog.owner=5”，适当的 url 编码（当 URL 解析时，web2py 自动解码）。

可以轻易地实现很一般的“list_records”动作如下：

```

1 def list_records():
2     table = request.args(0)
3     query = request.vars.query
4     records = db(query).select(db[table].ALL)
5     return dict(records=records)

```

用关联“default/list_records.html”视图：

```

1 {{extend 'layout.html'}}
2 {{=records}}

```

当查询返回一组记录并序列化在视图中，一开始它转化为 SQLTABLE 对象（不与 Table 一样），然后序列化到 HTML 表，此处每个字段相对应表的一列。

7.2.7 预填充表单

使用下面语法总能预填充表单：

```

1 form.vars.name = 'fieldvalue'

```

如上语句必须在表单声明后和在表单被接受前插入，无论字段（例子中的“name”）是否在表单中显式可见。

7.2.8 添加额外表单元素到 SQLFORM

有时，希望在创建后添加额外的项目到表单。例如可能希望添加复选框用来确认用户同意网站的条款和条件：

```
1 form = SQLFORM(db.yourtable)
2 my_extra_element = TR(LABEL('I agree to the terms and conditions'),
3     INPUT(_name='agree', value=True, _type='checkbox'))
4 form[0].insert(-1, my_extra_element)
```

变量 my_extra_element 要与表风格适应。这个例子，假定默认 formstyle='table3cols'。

提交后，form.vars.agree 包含复选框的状态，例如可以用在 onvalidation 动作。

7.2.9 无数据库 IO 的 SQLFORM

有时，当你想用 SQLFORM 从数据库表生成表单时，也相应地想验证一个提交表单，但却不想使用在数据库中的任何自动 INSERT/UPDATE/DELETE 命令。例如下面一个例子，当一个字段的值要从其它输入字段的值计算，另一种情况是，需要对输入数据执行附加验证，其并不能通过标准验证器获取。

很轻松地被分解：

```
1 form = SQLFORM(db.person)
2 if form.process().accepted:
3     response.flash = 'record inserted'
```

成：

```
1 form = SQLFORM(db.person)
2 if form.validate():
3     ### deal with uploads explicitly
4     form.vars.id = db.person.insert(**dict(form.vars))
5     response.flash = 'record inserted'
```

同样为 UPDATE/DELETE 表单，可被分解：

```
1 form = SQLFORM(db.person, record)
2 if form.process().accepted:
3     response.flash = 'record updated'
```

成：

```
1 form = SQLFORM(db.person, record)
2 if form.validate():
3     if form.deleted:
4         db(db.person.id==record.id).delete()
5     else:
6         record.update_record(**dict(form.vars))
7     response.flash = 'record updated'
```

在表包括“upload”类型字段（“fieldname”）、process(dbio=False) 和 validate()

处理上传文件储存，好像process(dbio=True)默认行为一样。

Web2py对上传文件分配的名字能被找到：

```
1 form.vars.fieldname
```

7.3 SQLFORM.factory

有些情况是当你想生成表单，你有数据库表但不想用数据库表，简单地想利用 SQLFORM 能力生成 CSS 友好的漂亮表单，并可能执行文件上传和重命名。

这可以通过form_factory完成。下面是例子，你生成表单，执行验证，上传一个文件，在 session 存储所有东西：

```
1 def form_from_factory():
2 form = SQLFORM.factory(
3 Field('your_name', requires=IS_NOT_EMPTY()),
4 Field('your_image', 'upload'))
5 if form.process().accepted:
6 response.flash = 'form accepted'
7 session.your_name = form.vars.your_name
8 session.filename = form.vars.your_image
9 elif form.errors:
10 response.flash = 'form has errors'
11 return dict(form=form)
```

下面是“default/form_from_factory.html”视图：

```
1 {{extend 'layout.html'}}
2 {{=form}}
```

需要使用下划线而不是空格做字段标签，或显式传递labels字典到from_factory，就像你为 SQLFORM 要做的一样，默认SQLFORM.factory使用所生成的html “id” 属性生成表单，好像表单是从叫做“no_table”的表生成。要改变这个虚拟表名，使用给制造厂的table_name属性。

```
1 form = SQLFORM.factory(..., table_name='other_dummy_name')
```

如果要放置两个制造厂生成的表单在同一张而避免CSS冲突，修改table_name是必须的。

7.3.1 单个表单多表 One form for multiple tables

经常遇见有两个表（例如‘client’和‘address’），其被引用链接在一起，你想创建单个允许掺入关于client和它的默认地址信息的表单。下面是如何实现：模型：

```
1 db.define_table('client',
2 Field('name'))
3 db.define_table('address',
4 Field('client', db.client, writable=False, readable=False),
5 Field('street'), Field('city'))
```

控制器：

```
1 def register():
2 form=SQLFORM.factory(db.client, db.address)
```

```

3 if form.process().accepted:
4 id = db.client.insert(**db.client._filter_fields(form.vars))
5 form.vars.client=id
6 id = db.address.insert(**db.address._filter_fields(form.vars))
7 response.flash='Thanks for filling the form'
8 return dict(form=form)

```

注意SQLFORM.factory（它从表的公用字段产生one表单，也继承它们的校验器），表单接受它执行的两次插入，一些数据在一张表而一些在另一张表。
这仅当表没有相同的字段名。

7.4 CRUD

最近添加到web2py的是Create/Read/Update/Delete (CRUD) API，在SQLFORM上面。CRUD创建SQLFORM，但它简化了编码，应为它包含了表单创建、表单处理、警告和重定向，所有功能到一个函数。

第一件要注意的事，便是CRUD区别于目前我们用到的其它web2py API，因为它没有披露，必须导入，也必须链接到指定数据库。例如：

```

1 from gluon.tools import Crud
2 crud = Crud(db)

```

上面定义的crud对象提供了下列API：

- crud.tables() 返回在数据库定义的表的列表。
- crud.create(db.tablename) 给表tablename 返回一个创建的表单。
- crud.read(db.tablename, id) 给tablename 和记录id 返回一个只读表单。
- crud.update(db.tablename, id) 给tablename 和记录id 返回更新表单。
- crud.delete(db.tablename, id) 删除记录。
- crud.select(db.tablename, query) 返回一个从表查询到的记录列表。
- crud.search(db.tablename) 返回元组（表单，记录），表单是搜索表单，记录是基于提交的搜索表单的记录列表。
- crud() 返回上面基于request.args() 的其中之一。

例如，下面动作：

```

1 def data(): return dict(form=crud())

```

将显露下面的URL：

```

1 http://.../[app]/[controller]/data/tables
2 http://.../[app]/[controller]/data/create/[tablename]
3 http://.../[app]/[controller]/data/read/[tablename]/[id]
4 http://.../[app]/[controller]/data/update/[tablename]/[id]
5 http://.../[app]/[controller]/data/delete/[tablename]/[id]
6 http://.../[app]/[controller]/data/select/[tablename]
7 http://.../[app]/[controller]/data/search/[tablename]

```

而，下面的动作：

```

1 def create_tablename():
2 return dict(form=crud.create(db.tablename))

```

会展露创建方法：

```

1 http://.../[app]/[controller]/create_tablename

```

而下面的动作：

```
1 def update_tablename():
2 return dict(form=crud.update(db.tablename, request.args(0)))
```

只展示 update 方法

```
1 http://.../[app]/[controller]/update_tablename/[id]
```

等等。

CRUD 行为可以用两种方法定制：设定 crud 对象的一些属性或者传递额外参数到各自方法。

7.4.1 设置 Settings

下面是当前 CRUD 属性的完整列表、它们的默认值和意思：

对所有 crud 表单强制认证：

```
1 crud.settings.auth = auth
```

用法在第 9 章讲解。

指定控制器，定义返回 crud 对象的数据函数：

```
1 crud.settings.controller = 'default'
```

指定 URL，成功“create”记录后重定向：

```
1 crud.settings.create_next = URL('index')
```

指定 URL，成功“update”记录后重定向：

```
1 crud.settings.update_next = URL('index')
```

指定 URL，成功“delete”记录后重定向：

```
1 crud.settings.delete_next = URL('index')
```

指定 URL 用来链接上传文件：

```
1 crud.settings.download_url = URL('download')
```

为 crud.create 表单，指定额外函数在标准校验过程完成后执行：

```
1 crud.settings.create_onvalidation = StorageList()
```

StorageList 作为 Storage 对象是相同的，它们定义在文件“gluon/storage.py”，但默认[] 作为与 None 相反。它允许下面的语法：

```
1 crud.settings.create_onvalidation.mytablename.append(lambda form:...)
```

为 crud.update 表单，指定额外函数在标准校验过程完成后执行：

```
1 crud.settings.update_onvalidation = StorageList()
```

crud.create 表单完成后，指定额外函数执行：

```
1 crud.settings.create_onaccept = StorageList()
```

crud.update 表单完成后，指定额外函数执行：

```
1 crud.settings.update_onaccept = StorageList()
```

crud.update 表单完成后，如果记录删除，指定额外函数执行：

```
1 crud.settings.update_ondelate = StorageList()
```

crud.delete 表单完成后，指定额外函数执行：

```
1 crud.settings.delete_onaccept = StorageList()
```

要决定“update”表单是否有“delete”按钮：

```
1 crud.settings.update_deletable = True
```

要决定“update”表单是否显示已编辑记录的 id：

```
1 crud.settings.showid = False
```

要决定，表单成功提交之后，是否保存之前插入值或重置到默认：

```
1 crud.settings.keepvalues = False
```

Crud 总是检测正在编辑的记录，介于表单显示和提交的时间区间，是否被第三方修改，这

个动作等同于：

```
1 form.process(detect_record_change=True)
```

它被如下设置：

```
1 crud.settings.detect_record_change = True
```

它能够通过设置变量为False 改变/禁用。

可以用下面语句改变表单风格

```
1 crud.settings.formstyle = 'table3cols' or 'table2cols' or 'divs' or 'ul'
```

可以在所有 crud 表单中设置分隔符：

```
1 crud.settings.label_separator = ':'
```

可以用下面语句给表单添加验证码，用同样约定解释身份认证：

```
1 crud.settings.create_captcha = None
```

```
2 crud.settings.update_captcha = None
```

```
3 crud.settings.captcha = None
```

7.4.2 消息 Messages

下面是定制消息列表：

```
1 crud.messages.submit_button = 'Submit'
```

为 create 和 update 表单设定 “submit” 按钮文本。

```
1 crud.messages.delete_label = 'Check to delete:'
```

在 “update” 表单中设定 “delete” 按钮标签。

```
1 crud.messages.record_created = 'Record Created'
```

就成功记录创建，设定 flash 消息。

```
1 crud.messages.record_updated = 'Record Updated'
```

就成功记录更新，设定 flash 消息。

```
1 crud.messages.record_deleted = 'Record Deleted'
```

就成功记录删除，设定 flash 消息。

```
1 crud.messages.update_log = 'Record %(id)s updated'
```

就成功记录更新，设定日志消息。

```
1 crud.messages.create_log = 'Record %(id)s created'
```

就成功记录创建，设定日志消息。

```
1 crud.messages.read_log = 'Record %(id)s read'
```

就成功记录访问，设定日志消息。

```
1 crud.messages.delete_log = 'Record %(id)s deleted'
```

就成功记录删除，设定日志消息。

注意crud.messages属于类gluon.storage.Message，它与gluon.storage.Storage很像，但它可以无需T运算符自动地翻译它的值。

日志消息当且仅当CRUD连接到Auth（认证）使用，第9章讨论。事件日志记录在Auth表

“auth_events”。

7.4.3 方法 Methods

CRUD 方法行为，每次调用基础上，也能定制。

下面是它们的详尽描述：

```
1 crud.tables()
2 crud.create(table, next, onvalidation, onaccept, log, message)
3 crud.read(table, record)
4 crud.update(table, record, next, onvalidation, onaccept, ondelete, log, message,
    deletable)
5 crud.delete(table, record_id, next, message)
6 crud.select(table, query, fields, orderby, limitby, headers, **attr)
7 crud.search(table, query, queries, query_labels, fields, field_labels, zero,
    showall, chkall)
```

- table 是方法应该起作用的 DAL 表或表名。
- record 和 record_id 是方法应该作用的记录 id。
- next 是成功后重定向的 URL。如果 URL 包含子串 “[id]”，当前创建/更新记录的 id 会被替换。
- onvalidation 有同 SQLFORM(..., onvalidation) 一样的功能。
- onaccept 是表单提交接受后，并作用于调用的函数，但在重定向以前。
- log 是日志消息，CRUD 内的日志消息是 form.vars 字典中形如 “%(id)s” 的变量。
- message 表单接受的 flash 消息。
- 当记录通过 “update” 表单删除，ondelete 代替 onaccept 被调用。
- deletable 决定 “update” 表单是否该有删除选项。
- query 是用来查询记录的查询命令。
- fields 是查询字段的列表。
- orderby 决定被查询记录排序（参看第6章）。
- limitby 决定查询记录要显示的范围（参看第6章）。
- headers 是表头名字的字典。
- queries 列表，像 [‘equals’, ‘not equal’, ‘contains’] 包含搜索表单中允许的方法。
- query_labels 字典像 query_labels=dict(equals=‘Equals’) 给搜索方法名字。
- fields 在搜索 widget 中被列出的字段列表。
- field_labels 字典，把字段名映射到标签。
- zero 默认 “choose one” 用作默认选项在搜索 widget 下拉。
- showall 如果你想在第一次调用，每次查询 rows 被返回（1.98.2 版后添加），设定为 True。
- chkall 设定为 True 用于打开搜索表单中所有复选框（1.98.2 版后添加）。

下面是单个控制器函数的用法例子：

```
1 # assuming db.define_table('person', Field('name'))
2 def people():
3     form = crud.create(db.person, next=URL('index'),
4     message=T("record created"))
5     persons = crud.select(db.person, fields=['name'],
6     headers={'person.name': 'Name'})
7     return dict(form=form, persons=persons)
```


下面是一个非常通用的控制函数，搜索、创建和编辑那些位于从表名被传递 `request.args(0)` 的表中的任何记录：

```
1 def manage():
2     table=db[request.args(0)]
3     form = crud.update(table,request.args(1))
4     table.id.represent = lambda id, row: \
5     A('edit:', id, _href=URL(args=(request.args(0), id)))
6     search, rows = crud.search(table)
7     return dict(form=form, search=search, rows=rows)
```

注意 `table.id.represent=...` 行告诉web2py改变id字段标识，显示链接而不是页面本身，以 `request.args(1)` 传递 `id`，其把创建页转换为更新页。

7.4.4 记录版本 Record versioning

SQLFORM 和 CRUD提供了一个数据库记录版本的设施：

如果你有表 (`db.mytable`) 需要全部版本历史，你只需做如下：

```
1 form = SQLFORM(db.mytable, myrecord).process(onsuccess=auth.archive)
1 form = crud.update(db.mytable, myrecord, onaccept=auth.archive)
```

当更新时，`auth.archive`定义新表叫做**`db.mytable_archive`**（名字从它引用到表继承），在创建文件夹表中，它存储记录的备份（如同它更新前），包含对当前记录的引用。

因为记录实际更新（只有它之前的状态存档），引用不会损坏。

所有都在后台完成，如果希望访问存档的表，需要在模型中定义它：

```
1 db.define_table('mytable_archive',
2     Field('current_record', db.mytable),
3     db.mytable)
```

注意表扩展 `db.mytable`（包括它的所有字段），并添加一个引用到 `current_record`。

`auth.archive`没有时间戳存储记录，除非源表有时间戳字段，例如：

```
1 db.define_table('mytable',
2     Field('created_on', 'datetime',
3     default=request.now, update=request.now, writable=False),
4     Field('created_by', auth.user,
5     default=auth.user_id, update=auth.user_id, writable=False),
```

这些字段没什么特别的，可以给它们任何你想叫的名字，记录存档之前，它们被填充，并用记录的每个拷贝存档，文档属性表的名字或参考字段名可以像如下修改：

```
1 db.define_table('myhistory',
2     Field('parent_record', db.mytable),
3     db.mytable)
4 # ...
5 form = SQLFORM(db.mytable, myrecord)
6 form.process(onsuccess = lambda form:auth.archive(form,
7     archive_table=db.myhistory,
8     current_record='parent_record'))
```

7.5 定制表单 Custom forms

如果表单用SQLFORM、SQLFORM.factory或CRUD创建，有多个办法把它嵌入允许多种程度定制的视图。考虑下面模型例子：

```
1 db.define_table('image',
2 Field('name'),
3 Field('file', 'upload'))
```

和上传动作

```
1 def upload_image():
2 return dict(form=crud.create(db.image))
```

为upload_image嵌入表单到视图最简单的方法是

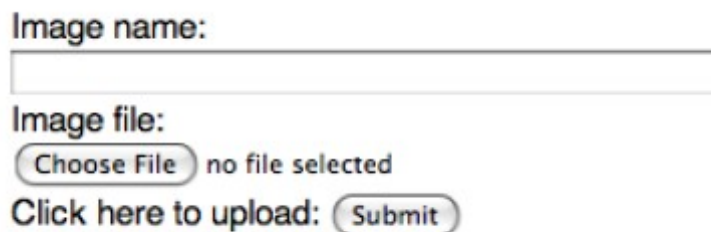
```
1 {{=form}}
```

这个结果是标准的表布局，如果想用不同的布局，可以分解表单为组件

```
1 {{=form.custom.begin}}
2 Image name: <div>{{=form.custom.widget.name}}</div>
3 Image file: <div>{{=form.custom.widget.file}}</div>
4 Click here to upload: {{=form.custom.submit}}
5 {{=form.custom.end}}
```

对字段form.custom.widget[fieldname]被序列化到合适的widget，如果表单提交并包含错误，它们像通常一样被附加在下面的widget。

上面的例子表单显示为下面的图像。



注意可用下面的方法得到相似的结果：

```
1 crud.settings.formstyle='table2cols'
```

无需使用定制表单。其它可能的 formstyles 是“table3col”（默认）、“divs”和“ul”。

如果你不想用 web2py 序列化的 widget，可以用 HTML 替换它们。有些变量可用来完成这个：

- form.custom.label[fieldname] 包含给字段的标签。
- form.custom.comment[fieldname] 包含给字段的评论。
- form.custom.dspval[fieldname] 表单类型和字段类型相关显示字段表示。
- form.custom.inpval[fieldname] 表单类型和字段类型相关值被用在字段编码。

遵循下面规定是重要的。

7.5.1 CSS 规定 CSS conventions

由 SQLFORM、SQLFORM.factory 或 CRUD 生成的表单中的标签，遵循严格的 CSS 命名习惯，可以用来更深入地定制表单。

给定表“mytable”和“string”类型字段“myfield”，默认被下面语句呈现

```
1 SQLFORM.widgets.string.widget
```

看起来象下面这样：

```
1 <input type="text" name="myfield" id="mytable_myfield"
2 class="string" />
```

注意：

- INPUT标签类与字段类型是一样的。这对jQuery代码在“web2py_ajax.html”中工作是非常重要的。要确保只能有数字在“integer”和“double”字段，“time”、“date”和“datetime”字段显示弹出日历/日期拾取calendar/datepicker。
- id是类的名字加上字段名，用下划线连接。这允许你唯一地引用到字段，通过，例如jQuery('#mytable_myfield')操作字段的stylesheet或绑定关联字段事件的动作（focus、blur、keyup等）。
- name是字段名。

7.5.2 隐藏错误 Hide errors

偶尔，你想在除了默认的一些地方禁用自动错误放置和显示表单错误消息，这能简单地被实现：

- FORM 或 SQLFORM 情况，传递hideerror=True给accepts的方法。
- CRUD 情况，设置crud.settings.hideerror=True。

想修改视图来显示错误（因为它们不再自动显示）。

下面的例子，错误显示在表单上面而不在表单里。

```
1 {{if form.errors:}}
2 Your submitted form contains the following errors:
3 <ul>
4 {{for fieldname in form.errors:}}
5 <li>{{=fieldname}} error: {{=form.errors[fieldname]}}</li>
6 {{pass}}
7 </ul>
8 {{form.errors.clear()}}
9 {{pass}}
10 {{=form}}
```

错误会象下面图像展示的那样显示：

Your submitted form contains the following errors:

- name error: enter a value

Name:

File: no file selected

这个机制适用于定制表单。


```
4 error_message='must be YYYY-MM-DD!')
```

有关%的完全描述，直接看 IS_DATETIME 验证器。

IS_DATETIME

这个验证器检查字段值包含合法的指定格式的日期时间，用翻译运算符指定格式是好的实践，以便不同的地方支持不同的格式。

```
1 requires = IS_DATETIME(format=T('%Y-%m-%d %H:%M:%S'),
2 error_message='must be YYYY-MM-DD HH:MM:SS!')
```

下列符号能用作格式字符串（下面显示符号和例子字符串）：

```
1 %Y '1963'
2 %y '63'
3 %d '28'
4 %m '08'
5 %b 'Aug'
6 %B 'August'
7 %H '14'
8 %I '02'
9 %p 'PM'
10 %M '30'
11 %S '59'
```

IS_DATETIME_IN_RANGE

作用与之前的验证器非常像，但允许指定范围：

```
1 requires = IS_DATETIME_IN_RANGE(format=T('%Y-%m-%d %H:%M:%S'),
2 minimum=datetime.datetime(2008, 1, 1, 10, 30),
3 maximum=datetime.datetime(2009, 12, 31, 11, 45),
4 error_message='must be YYYY-MM-DD HH:MM:SS!')
```

有关%的完全描述，直接查看 IS_DATETIME 验证器。

IS_DECIMAL_IN_RANGE

```
1 INPUT(_type='text', _name='name', requires=IS_DECIMAL_IN_RANGE(0, 10, dot='.'))
```

它把输入转化为Python十进制，如果十进制数不在指定范围内生成错误，与Python Decimal 算术做比较。

最小和最大限制可以是 None，意味着没有下限或上限。

dot 参数可选，允许全局化这些字符，其被用来分隔十进制数。

IS_EMAIL

检查看起来像电子邮件地址的字段值，不尝试发送邮件证实。

```
1 requires = IS_EMAIL(error_message='invalid email!')
```

IS_EQUAL_TO

检查验证的值是否等于给定值（可能是变量）：

```
1 requires = IS_EQUAL_TO(request.vars.password,
2 error_message='passwords do not match')
```

IS_EXPR

就一个变量值而言，它的第一个参数是包含逻辑表达式的字符串，如果表达式的值为 True，它验证字段值。例如：

```
1 requires = IS_EXPR('int(value)%3==0',
2 error_message='not divisible by 3')
```

应首先检查值是否是整型，以防异常发生。

```
1 requires = [IS_INT_IN_RANGE(0, 100), IS_EXPR('value%3==0')]
```

IS_FLOAT_IN_RANGE

检查字段值是否是有限范围内的浮点数，下面的例子 $0 \leq \text{value} \leq 100$ ：

```
1 requires = IS_FLOAT_IN_RANGE(0, 100, dot=".",
2 error_message='too small or too large!')
```

dot 参数可选，允许全局化这些字符，其被用来分隔十进制数。

IS_INT_IN_RANGE

检查字段值是否是有限范围整型数，下面 $0 \leq \text{value} < 100$ 例子：

```
1 requires = IS_INT_IN_RANGE(0, 100,
2 error_message='too small or too large!')
```

IS_IN_SET

检查字段值是否在集合内：

```
1 requires = IS_IN_SET(['a', 'b', 'c'], zero=T('choose one'),
2 error_message='must be a or b or c')
```

这个zero参数是可选的，它决定了默认文本选项，一个选项不会被IS_IN_SET验证器自己接受。如果不想一个“choose one”选项，设置zero=None。

zero选项在修改版中介绍引入（1.67.1），它不破坏后向兼容性，这个意义上它不影响应用，但之后它改变它们的行为，之前没有zero选项。

集合的元素必须总是字符串，除了这个验证器以IS_INT_IN_RANGE开始（值转换为整型）或IS_FLOAT_IN_RANGE（值转换为浮点型）。例如：

```
1 requires = [IS_INT_IN_RANGE(0, 8), IS_IN_SET([2, 3, 5, 7],
2 error_message='must be prime and less than 10')]
```

也可以用字典或元组列表让下拉列表更具有描述性：

```
1 ### Dictionary example:
2 requires = IS_IN_SET({'A': 'Apple', 'B': 'Banana', 'C': 'Cherry'}, zero=None)
3 ### List of tuples example:
4 requires = IS_IN_SET([('A', 'Apple'), ('B', 'Banana'), ('C', 'Cherry')])
```

IS_IN_SET 和 Tagging

IS_IN_SET验证器有可选属性multiple=False，如果设置为True，多个值可存在一个字段中，字段类型必须是list:integer 或 list:string。创建和更新表单的多应用是自动处理的，但对DAL是透明的，我们强烈建议用jQuery multiselect插件呈现多字段。

注意当multiple=True，IS_IN_SET会接受0或更多值，即当没有任何被选择时，它接受字段，Multiple也可以是表单(a, b)元组，a和b是分别是能被查询的最小和（互斥）最大项目数。

IS_LENGTH

检查字段值长度以满足给定边界，适用于文本和文件输入。

它的参数是：

maxsize: 最大允许长度/大小（默认255）

minsize: 最小允许长度/大小

例如：检查文本字符串是否少于33个字符：

```
1 INPUT(_type='text', _name='name', requires=IS_LENGTH(32))
```

检查密码字符串是否不超过5个字符：

```
1 INPUT(_type='password', _name='name', requires=IS_LENGTH(minsize=6))
```

检查上传文件大小是否在1KB和1MB之间：

```
1 INPUT(_type='file', _name='name', requires=IS_LENGTH(1048576, 1024))
```

除了文件之外，对所有字段类型，检查值的长度。在文件情况中，值是cookie.FieldStorage，因此它验证文件数据的长度，这是直觉期望的行为。

IS_LIST_OF

它不是合适的验证器，被用来允许字段验证返回多个值，用在那些罕见的情况，当表单包含多个同名字段或多个选择框（复选框），它仅有的参数是另一个验证器，所做的是应用其它验证器到列表中的每个元素。例如，下面的表达式检查列表中的每个项是否是0-10范围内的整数：

```
1 requires = IS_LIST_OF(IS_INT_IN_RANGE(0, 10))
```

它从不返回错误，不包含错误消息，内部验证器控制错误生成。

IS_LOWER

这个验证器从不返回错误，它就把值转换为小写。

```
1 requires = IS_LOWER()
```

IS_MATCH

这个验证器用来匹配值与正则表达式，如果不匹配返回错误。下面是验证US邮政编码用法的例子：

```
1 requires = IS_MATCH('^\d{5}(-\d{4})?$',  
2 error_message='not a zip code')
```

下面是验证IPv4地址的例子（注意：IS_IPV4验证器更适合这个目的）：

```
1 requires = IS_MATCH('^\d{1,3}(\.\d{1,3}){3}$',  
2 error_message='not an IP address')
```

下面是验证US电话号码的用法例子：

```
1 requires = IS_MATCH('^1?((-)\d{3}-?/(\d{3})\d{3}-?\d{4})$',  
2 error_message='not a phone number')
```

有关Python正则表达式更多信息，参考Python官方文档。

IS_MATCH采用默认为False的可选参数strict，当设置为True，它只匹配字段串开头：

```
1 >>> IS_MATCH('a')('ba')  
2 ('ba', <lazyT 'invalid expression'>) # no pass  
3 >>> IS_MATCH('a', strict=False)('ab')  
4 ('a', None) # pass!
```

IS_MATCH采用另一个默认为False的可选参数search，当设置为True，它用regex方法search而不是match方法验证字符串。

IS_NOT_EMPTY

这个验证器检查字段值的内容，而不是空字符串。

```
1 requires = IS_NOT_EMPTY(error_message='cannot be empty!')
```

IS_TIME

这个验证器检查字段值是否包含指定格式的合法时间。

```
1 requires = IS_TIME(error_message='must be HH:MM:SS!')
```

IS_URL

如果以下任何为true，拒绝URL字符串：

- 字符串为空或None
- 字符串使用了在URL中不允许的字符
- 字符串破坏任何HTTP语法规则
- 方案指定的URL（如果指定了）不是‘http’或‘https’
- 顶级域名（如果主机名字指定了）不存在（这些规则基于RFC2616[68]）

这个函数仅检查URL的语法。例如，它不检查URL指向真实的文档，亦或它仅为语法上的，

在简写的URL的情况下（例如，‘google.ca’），这个函数自动在URL前添加‘http://’。

如果参数mode=‘generic’被使用，那这个函数功能的行为改变了。如果下面任何为真，它拒绝URL字符串：

- 字符串为空或None
- 字符串使用了在URL中不允许的字符
- 方案指定的URL（如果指定了）不是合法的
（这些规则基于RFC2396[69]）

允许的列表用许可的allowed_schemes参数定制，如果从列表剔除了None，则缩写URLs（缺少方案比如‘http’）被拒绝。

默认预先添加方案用prepend_scheme定制，如果设置prepend_scheme为None，那么添加方案无效，URL要求预添加的解析仍被接受，但它的返回值不会被修改。

IS_URL兼容RFC3490[70]国际域名命名（IDN）标准，因此URL可以是常规字符串或unicode字符串，如果URL域名组成部分（比如，google.ca）包含非US-ASCII字母，那域名会转换成Punycode（RFC3492[71]定义）。IS_URL有点超越了标准，其允许非US-ASCII字符在路径被表示以及URL组成的查询，这些非US-ASCII字符将采用编码。例如，空格编码为‘%20’，unicode字符用十六进制码0x4e86会编成‘% 4e86’。

例如：

```
1 requires = IS_URL())
2 requires = IS_URL(mode='generic')
3 requires = IS_URL(allowed_schemes=['https'])
4 requires = IS_URL(prepend_scheme='https')
5 requires = IS_URL(mode='generic',
6 allowed_schemes=['ftps', 'https'],
7 prepend_scheme='https')
```

IS_SLUG

```
1 requires = IS_SLUG(maxlen=80, check=False, error_message='must be slug')
```

如果check设置为True，它检查验证的值是否是slug（只允许字母数字字符和不重复破折号）。

如果check设置为False（默认），它把输入值转换为slug。

IS_STRONG

强制字段复杂性要求（通常口令密码字段）

例如：

```
1 requires = IS_STRONG(min=10, special=2, upper=2)
```

这里

- min是值的最小长度
- special是要求的特殊符号的最小个数，特殊符号是下列中的任何一个!@#%&*() {} [] - +
- upper是大写字母的最小个数

IS_IMAGE

这个验证器检查通过文件输入上传的文件是否用所选图像格式中的一种保存，有给定限制范围的尺寸（宽和高）。

它不检查文件最大尺寸（用IS_LENGTH实现），如果没有数据上传，它返回验证失败，支持文件格式BMP、GIF、JPEG、PNG，并且它不要求Python图像库。

代码部分取自参看文献[72]

它采用下面参数：

- extensions: 迭代器包含那些许可图像文件小写扩展
 - maxsize: 迭代器包含图像的最大宽度和高度
 - minsize: 迭代器包含图像的最小宽度和高度
- 用 (-1, -1) 为最小尺寸旁路图像尺寸检查。

下面是一些例子：

- 检查上传文件是否是支持的图像格式：

```
l requires = IS_IMAGE()
```

- 检查上传文件是否是JPEG或PNG

```
l requires = IS_IMAGE(extensions=('jpeg', 'png'))
```

- 检查上传文件PNG最大尺寸是否是200*200像素：

```
l requires = IS_IMAGE(extensions=('png'), maxsize=(200, 200))
```

注意：对于包含requires = IS_IMAGE()的表，显示编辑表单，delete复选框不出现，因为删除文件将导致验证失败。要显示delete复选框用下面的验证：

```
l requires = IS_EMPTY_OR(IS_IMAGE())
```

IS_UPLOAD_FILENAME

该验证器检查通过文件输入上传的文件名和扩展名是否符合给定标准。

它根本不保证文件类型，如果没有数据上传，返回验证失败。

它的参数是：

- filename: 文件名 (.前) 正则表达式
- extension: 扩展名 (.后) 正则表达式
- lastdot: 点用做文件名/扩展名的分隔符，True指明最后的点（例如，“file.tar.gz” 分解为 “file.tar” + “gz”），如果False则表明第一个点（“file.tar.gz” 分解为 “file” + “tar.gz”）
- case: 0意味着保持不变，1意味着把字符串转换为小写（默认），2意味把字符串转换为大写。

如果没有点存在，扩展名检查将针对空字符串开展，文件名检查将针对整个字符串值。

例如：

检查文件是否有pdf扩展名（不区分大小写）：

```
l requires = IS_UPLOAD_FILENAME(extension='pdf')
```

检查文件是否有扩展名tar.gz，文件名用backup开头：

```
l requires = IS_UPLOAD_FILENAME(filename='backup.*', extension='tar.gz', lastdot=False)
```

检查文件是否没有扩展名并且名字不匹配README（区分大小写）：

IS_IPV4

该验证器检查字段值是否是十进制形式IPv4地址，在一定范围可以设置强制地址。

IPv4正则表达式来自参考文献[73]，它的参数是：

- minip 最低允许的地址，接受：str，例如192.168.0.1；数字迭代，例如 [192, 168, .0, 1]；整型，例如3232235521。
- maxip最高允许的地址，与上面相同

三个例子的值都相同，因为地址用下面函数变换成整型进行包含检查：

```
l number = 16777216 * IP[0] + 65536 * IP[1] + 256 * IP[2] + IP[3]
```

例如：

检查IPv4地址的合法性：

```
l requires = IS_IPV4()
```

检查私有网络IPv4地址的合法性：

```
l requires = IS_IPV4(minip='192.168.0.1', maxip='192.168.255.255')
```

IS_LOWER

该验证器从不返回错误，它把值转换为小写字母。

```
1 requires = IS_LOWER()
```

IS_UPPER

该验证器从不返回错误，它把参数值转换为大写字母。

```
1 requires = IS_UPPER()
```

IS_NULL_OR

过时，是下面描述的IS_EMPTY_OR的别名。

IS_EMPTY_OR

有时，你需要允许在字段的空值及其它要求，例如字段是日期但它也能是空。

IS_EMPTY_OR验证器允许：

```
1 requires = IS_EMPTY_OR(IS_DATE())
```

CLEANUP

这是个过滤器，从不失效，仅清除所有十进制ASCII代码不在[10, 13, 32-127] 的字符。

```
1 requires = CLEANUP()
```

CRYPT

这也是个过滤器，它在输入端执行安全哈希并被用来防止输入的密码明文传递给数据库。

```
1 requires = CRYPT()
```

如果密钥没有指定，用MD5算法；如果密钥指定为CRYPT，则用HMAC算法；用HMAC密钥包含前缀确定算法，例如SHA512：

```
1 requires = CRYPT(key='sha512:thisisthekey')
```

这是推荐的语法，密钥必须是关联使用数据库唯一的字符串，密钥不能再修改，如果你丢失了之前哈希密钥，值变得无用。

CRYPT验证器哈希散列输入，这使它有些特殊，如果要验证密码字段，请在它散列之前，可以使用验证器列表中的CRYPT，但必须确定它是列表的最后，以便最后调用。例如：

```
1 requires = [IS_STRONG(), CRYPT(key='sha512:thisisthekey')]
```

CRYPT有min_length参数，默认0。

7.6.2 数据库验证器 Database validators

IS_NOT_IN_DB

考虑下面的例子：

```
1 db.define_table('person', Field('name'))
```

```
2 db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

它要求，当插入新的 person 记录时，他/她的名字并未已存在数据库，db，字段 person.name 中，与其它验证器，这个要求在表单处理级是强制的，不是数据库级。这意味着，有很小的概率，即如果两个访问者试图同时插入相同的 person.name 记录，导致了竞争情况，两个记录都被接受了。因此，安全起见告知数据库这个字段应该有唯一的值。

```
1 db.define_table('person', Field('name', unique=True))
```

```
2 db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

现在如果竞争情况出现，数据库产生OperationalError，两条插入记录之一被拒绝。

IS_NOT_IN_DB的第一个参数是数据库连接或Set（集合）。在后者情况中，你仅需检查被Set定义的set对象。

下面的代码，例如不允许同名的两个人彼此在10天之内注册：

```
1 import datetime
2 now = datetime.datetime.today()
3 db.define_table('person',
4 Field('name'),
5 Field('registration_stamp', 'datetime', default=now))
6 recent = db(db.person.registration_stamp>now-datetime.timedelta(10))
7 db.person.name.requires = IS_NOT_IN_DB(recent, 'person.name')
```

IS_IN_DB

考虑下面的表和要求：

```
1 db.define_table('person', Field('name', unique=True))
2 db.define_table('dog', Field('name'), Field('owner', db.person))
3 db.dog.owner.requires = IS_IN_DB(db, 'person.id', '%(name)s',
4 zero=T('choose one'))
```

它在dog INSERT/UPDATE/DELETE表单层级强制实施，要求dog.owner在数据库db的字段person.id有合法id，因为这个dog.owner字段验证器呈现为下拉框。该验证器的第三个参数是字符串，描述下拉框里的元素，例子中想看到person%(name) s而不是person%(id)s，%(...) s被每条记录在圆括号内的字段值替换。

这个zero 选项工作很像是为IS_IN_SET 验证器。

该校验器的zero参数可能是数据库连接或DAL Set，如同S_NOT_IN_DB，当想要限制下拉框记录数时，这对于例子会有用的，这个例子中，当每次控制器调用时，我们用控制器中的IS_IN_DB动态限制记录数：

```
1 def index():
2 (...)
3 query = (db.table.field == 'xyz') #in practice 'xyz' would be a variable
4 db.table.field.requires=IS_IN_DB(db(query),...)
5 form=SQLFORM(...)
6 if form.process().accepted: ...
7 (...)
```

如果想字段验证，但又不想用下拉框，必须把验证器放在列表：

```
1 db.dog.owner.requires = [IS_IN_DB(db, 'person.id', '%(name)s')]
```

偶尔，你想用下拉框（而又不想用上面的列表语法），想用附加的验证器。为了这个目的，IS_IN_DB验证器用额外参数_and，指向其它验证器列表，如果应用了验证值传递给IS_IN_DB进行验证。例如验证所有数据库中狗主人不在下面的子集：

```
1 subset=db(db.person.id>100)
2 db.dog.owner.requires = IS_IN_DB(db, 'person.id', '%(name)s',
3 _and=IS_NOT_IN_DB(subset, 'person.id'))
```

S_IN_DB也用cache参数，作用像select的cache参数。

IS_IN_DB和 Tagging

S_IN_DB验证器有可选属性multiple=False，如果设置为True，多个值能存在一个字段，这个字段必须是list:reference类型，已在第6章讨论，Tagging的清晰例子在那里讨论了。Multiple引用在创建和更新表单中自动处理，但对DAL是透明的，我们强烈建议用jQuery多查询插件呈现多字段。

7.6.3 自定义验证器 Custom validators

所有的验证器遵从下面的原型：

```
1 class sample_validator:
2     def __init__(self, *a, error_message='error'):
3         self.a = a
4         self.e = error_message
5     def __call__(self, value):
6         if validate(value):
7             return (parsed(value), None)
8         return (value, self.e)
9     def formatter(self, value):
10        return format(value)
```

即当调用验证一个值时，校验器返回元组 (x, y)，如果 y 是 None，那么传递给校验器的值和 x 包含解析值。例如，当验证器要求值是整数，x 转换为整型（值），如果值没有通过验证，那么 x 包含输入值而 y 包含错误消息解释验证失败，这个错误消息用来报告表单中的不合法的错误。

验证器也包含 formatter 方法，它必须执行与 __call__ 调用正好相反的转换。例如，考虑 IS_DATE 源代码：

```
1 class IS_DATE(object):
2     def __init__(self, format='%Y-%m-%d', error_message='must be YYYY-MM-DD!'):
3         self.format = format
4         self.error_message = error_message
5     def __call__(self, value):
6         try:
7             y, m, d, hh, mm, ss, t0, t1, t2 = time.strptime(value, str(self.format))
8             value = datetime.date(y, m, d)
9             return (value, None)
10        except:
11            return (value, self.error_message)
12    def formatter(self, value):
13        return value.strftime(str(self.format))
```

一旦成功，__call__ 方法从表单读取日期字符串，并用构造函数指定的格式字符串把它转换成 datetime.date 对象，formatter 对象采用 datetime.date 对象，并使用同样的格式把它转换为字符串表示，在表单 formatter 自动调用，但可以显式调用它，把对象转换为适当的表示。例如：

```
1 >>> db = DAL()
2 >>> db.define_table('atable',
3     Field('birth', 'date', requires=IS_DATE('%m/%d/%Y')))
4 >>> id = db.atable.insert(birth=datetime.date(2008, 1, 1))
5 >>> row = db.atable[id]
6 >>> print db.atable.formatter(row.birth)
7 01/01/2008
```

当需要多个验证器的时候（存储在列表），它们顺序运行，通过输出作为下一个输入，当其中一个验证器失败，链式运行终止。

反之，当我们调用字段 formatter 方法，相关验证器的 formatter 也链接起来，但以相反的顺序。

注意替代定制验证器，也可用 `form.accepts(...)`，`form.process(...)` 和 `form.validate(...)` 的 `onvalidate` 参数。

7.6.4 依赖的验证器 Validators with dependencies

通常在所有模型中，验证器一次性被全部设置。

偶尔，你需要验证字段，而验证器依赖于另外字段的值，这能以各种方法实现，能在模型中或控制器中实现。

例如，如下页生成注册表单，要求用户名和密码两次，没有字段可以为空，两次密码必须一致：

```
1 def index():
2     form = SQLFORM.factory(
3         Field('username', requires=IS_NOT_EMPTY()),
4         Field('password', requires=IS_NOT_EMPTY()),
5         Field('password_again',
6             requires=IS_EQUAL_TO(request.vars.password)))
7     if form.process().accepted:
8         pass # or take some action
9     return dict(form=form)
```

同样的机制能用在FORM 和 SQLFORM对象。

7.7 Widgets 小工具

下面是可用 web2py 小工具的列表：

```
1 SQLFORM.widgets.string.widget
2 SQLFORM.widgets.text.widget
3 SQLFORM.widgets.password.widget
4 SQLFORM.widgets.integer.widget
5 SQLFORM.widgets.double.widget
6 SQLFORM.widgets.time.widget
7 SQLFORM.widgets.date.widget
8 SQLFORM.widgets.datetime.widget
9 SQLFORM.widgets.upload.widget
10 SQLFORM.widgets.boolean.widget
11 SQLFORM.widgets.options.widget
12 SQLFORM.widgets.multiple.widget
13 SQLFORM.widgets.radio.widget
14 SQLFORM.widgets.checkboxes.widget
15 SQLFORM.widgets.autocomplete
```

前十个对相应字段类型是默认的，当字段的要求是IS_IN_SET 和 IS_IN_DB并且 multiple=False（默认行为），“options”小工具被使用；当字段的要求是IS_IN_SET 和 IS_IN_DB并且multiple=True，“multiple”小工具被使用；“radio”和“checkboxes”默认从不使用，但能手动设置；autocomplete自动完成工具很特殊的并在它的章节讨论。例如，我们有用文本域表示的“string”字段：

```
1 Field('comment', 'string', widget=SQLFORM.widgets.text.widget)
```

小工具能指定到字段posteriori：

```
1 db.mytable.myfield.widget = SQLFORM.widgets.string.widget
```

有时，小工具用附加参数，需要指定它们的值，这种情况，可以使用 lambda

```
1 db.mytable.myfield.widget = lambda field,value: \
```



```
2 SQLFORM.widgets.string.widget(field,value,_style='color:blue')
```

Widgets是帮助对象制造厂，它们的头两个参数总是field和value，其它参数包括正常帮助对象属性，如_style和_class等，一些widgets也用特殊参数。特别是SQLFORM.widgets.radio和SQLFORM.widgets.checkboxes采用style参数（不要与_style混淆），能设置为“table”、“ul”或“divs”并以所包含表单的formstyle匹配。

可以创建新的widgets或是扩展现有widgets。

SQLFORM.widgets[type]是类，SQLFORM.widgets[type].widget是相应类的静态成员函数，每个widget函数有两个参数：字段对象和字段对象当前值，它返回widget的表示，作为例子，字符串widget能被如下重编码：

```
1 def my_string_widget(field, value):
2     return INPUT(_name=field.name,
3                  _id="%s_%s" % (field._tablename, field.name),
4                  _class=field.type,
5                  _value=value,
6                  requires=field.requires)
7
8 Field('comment', 'string', widget=my_string_widget)
```

id和类的值必须遵守这一章后面描述的习惯，Widget包含自己的校验器，但好的实践是关联字段“requires”属性的校验器，并使widget从那儿得到。

7.7.1 自动完成widget Autocomplete widget

对autocomplete widget有两种可能的用法：从列表自动完成一个字段或自动完成引用字段（自动完成的字符串是引用的表示，用id实现）。

第一种情况容易：

```
1 db.define_table('category',Field('name'))
2 db.define_table('product',Field('name'),Field('category'))
3 db.product.category.widget = SQLFORM.widgets.autocomplete(
4     request, db.category.name, limitby=(0,10), min_length=2)
```

limitby指示widget一次显示不超过10条建议，min_length指示widget执行Ajax回调得到建议，仅当用户在搜索框敲入至少两个字符。

第二种情况复杂地多：

```
1 db.define_table('category',Field('name'))
2 db.define_table('product',Field('name'),Field('category'))
3 db.product.category.widget = SQLFORM.widgets.autocomplete(
4     request, db.category.name, id_field=db.category.id)
```

这种情况下，id_field的值告诉widget，即使自动完成的值是db.category.name，存储的值也是相应的db.category.id。可选参数是orderby，指示widget如何对建议排序（默认字母顺序），

Widget通过Ajax工作。Ajax在哪回调？这个widget有些神奇，回调是widget对象自己的方法。它如何显露？web2py任何代码生成响应引发HTTP异常，Widget用下面的方式利用这种可能性：widget给Ajax发送调用，对同一个URL，第一个位置生成widget，把特殊令牌放在request.vars。widget再次实例化，它发现令牌，并引发HTTP异常以响应请求，所有这些在后台完成，对编程者隐藏。

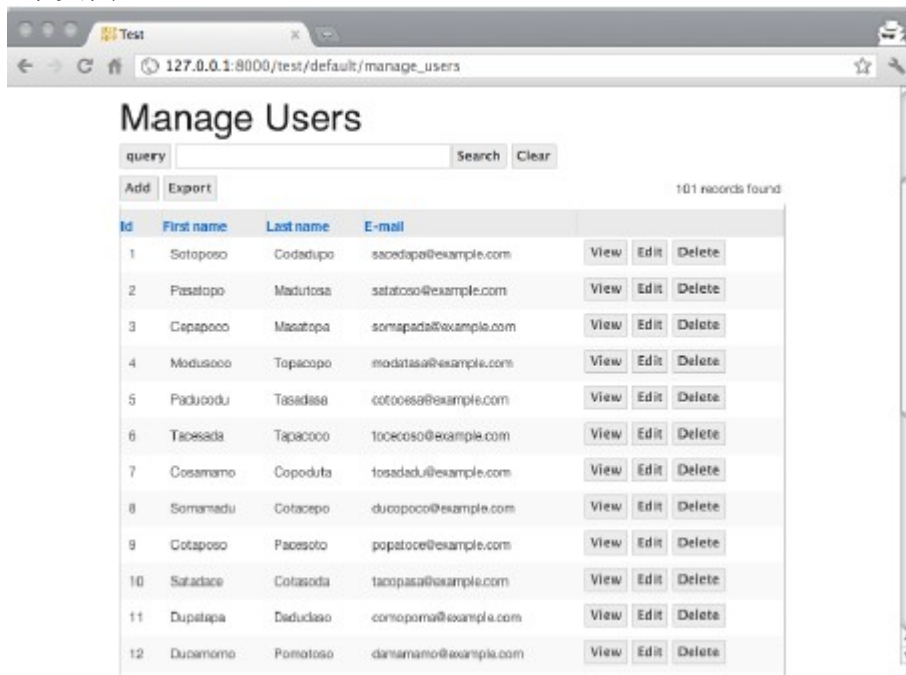
7.8 SQLFORM.grid 和 SQLFORM.smartgrid (实验性的)

这两个高级工具创建复杂的 CRUD 控制，它们提供分页，以及从单个工具，浏览、搜索、排序、创建、更新和删除记录的能力。

两个中最简单的是SQLFORM.grid。下面是使用的例子：

```
1 @auth.requires_login()
2 def manage_users():
3     grid = SQLFORM.grid(db.auth_user)
4     return locals()
```

产生如下页面：



SQLFORM.grid的第一个参数是表或查询，网格工具提供匹配查询的记录访问。

在我们开始网格工具参数长列表的参数之前，我们需要理解它如何工作，工具查看request.args决定做什么（浏览、搜索、创建、更新和删除等），每个由工具所创建的按钮链接同样的函数（上面情况的manage_users），但传递不同的request.args。默认情况，网格生成所有的URL是数字签名并被验证的，这意味着不登录，不能执行某些动作（创建、更新、删除）。这些限制能被如下解除：

```
1 def manage_users():
2     grid = SQLFORM.grid(db.auth_user, user_signature=False)
3     return locals()
```

但我们不推荐使用。

因为网格工作的方式是，每人只能有一个网格每个控制器函数，除非通过LOAD作为组件嵌入。

因为包含网格的函数能自己操作命令行参数，网格需要知道哪个参数该被网格处理哪个不该，例如，下面是允许管理任何表的代码例子：

```
1 @auth.requires_login()
2 def manage():
3     table = request.args(0)
4     if not table in db.tables(): redirect(URL('error'))
5     grid = SQLFORM.grid(db[table], args=request.args[:1])
```

```
6 return locals()
```

grid 的 args 参数指定哪个 request.args 该被一起传递，并被工具忽略，在我们的例子中，request.args[:1]是我们想要管理的表的名字，它被 manage 函数处理，而不是工具。

grid 的完整网格用法描述如下：

```
1 SQLFORM.grid(query,
2 fields=None,
3 field_id=None,
4 left=None,
5 headers={},
6 orderby=None,
7 searchable=True,
8 sortable=True,
9 deletable=True,
10 editable=True,
11 details=True,
12 create=True,
13 csv=True,
14 paginate=20,
15 selectable=None,
16 links=None,
17 upload = '<default>',
18 args=[],
19 user_signature = True,
20 maxtextlengths={},
21 maxtextlength=20,
22 onvalidation=None,
23 oncreate=None,
24 onupdate=None,
25 ondelete=None,
26 sorter_icons=('[^]', '[v]'),
27 ui = 'web2py',
28 showbuttontext=True,
29 search_widget='default',
30 _class="web2py_grid",
31 formname='web2py_grid',
32 ignore_rw = False,
33 formstyle = 'table3cols'):
```

- fields 是从数据库得到的字段列表，它也用来决定哪些字段在网格视图中显示。
- field_id 必须是被用作 ID 表的字段，例如 db.mytable.id。
- headers 是字典，映射 'tablename.fieldname' 到相应头标签。
- left 是可选左联表达式用来构建...select(left=...)。
- orderby 为行 (row) 做默认排序。
- searchable、sortable、deletable、details、create 分别决定是否能 search、sort、delete、view details 和 create 新记录。
- csv 如果设置为 true，允许以 CSV 格式下载 grid。
- paginate 设置每页最大的行数。
- links 用来显示新列，链接到其它页面，Links 参数必须是 dict 字典列表 (header=' name' ,body=lambda row:A(...))，header 是新列的头、body 是带 row 参数并返回值的一个函数。例子中，值是 A() 帮助对象。
- 在 grid 视图中，对于每个字段值，maxtextlength 设置显示的文本最大长度。对每个字段，使用 maxtextlengths，这个值能覆盖， 'tablename.fieldname' 的字典：

length。

- onvalidation、oncreate、onupdate 和 ondelete 是回调函数，除了ondelete，所有函数用表单对象作为输入。
- sorter_icons 是两个字符串列表（或帮助对象），用来表示每个字段升和降序排序的选项。
- ui设置为‘web2py’，将生成web2py友好的类名；设置为jquery-ui，将生成jquery UI友好的类名，但它也可以是其自身的一组类名集，其对于各种grid组件来说：

```
1 ui = dict(widget='',
2 header='',
3 content='',
4 default='',
5 cornerall='',
6 cornertop='',
7 cornerbottom='',
8 button='button',
9 buttontext='buttontext button',
10 buttonadd='icon plus',
11 buttonback='icon leftarrow',
12 buttonexport='icon downarrow',
13 buttondelete='icon trash',
14 buttonedit='icon pen',
15 buttontable='icon rightarrow',
16 buttonview='icon magnifier')
```

- search_widget允许超越默认搜索工具，我们推荐读者参考“gluon/sqlhtml.py”中的源代码获得细节。
- showbutton允许关闭所有按钮。
- _class是为grid容器的类。
- formname、ignore_rw 和 formstyle传递给SQLFORM对象，被创建/更新表单grid使用。deletable、editable 和 details通常是布尔值，但也可以是函数，用row对象参数，决定是否显示相应按钮。

SQLFORM.smartgrid看起来很像grid，实际上它包含网格，但它被设计用作输入而不是查询那么仅仅一张表，以及浏览表并查询引用表。

例如，考虑下表结构：

```
1 db.define_table('parent',Field('name'))
2 db.define_table('child',Field('name'),Field('parent','reference parent'))
```

用SQLFORM.grid，可以列出所有的父parent：

```
1 SQLFORM.grid(db.parent)
```

所有的子：

```
1 SQLFORM.grid(db.child)
```

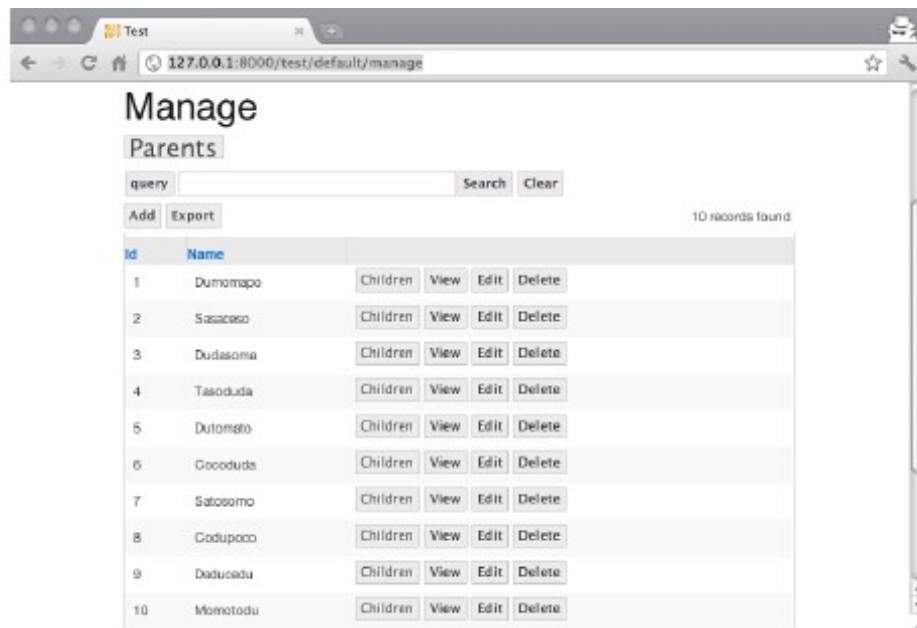
一张列出所有的父与子的表：

```
1 SQLFORM.grid(db.parent, left=db.child.on(db.child.parent=db.parent.id))
```

用SQLFORM.smartgrid，可以把所有数据放在一个工具内派生这些表：

```
1 @auth.requires_login():
2 def manage():
3     grid = SQLFORM.smartgrid(db.parent, linked_tables=['child'])
4     return locals()
```

看起来如下：



注意额外的“children”链接。可用规则的grid创建额外的links，但它们指向不同的动作，用smartgrid，它们被同一个工具自动创建和处理。

也注意，当点击给定父类的“children”链接，只能得到那个父类的子的列表（这是显然的），也注意如果试图添加新的子，父类的值为新的子自动设置为查询父类（显示在关联到工具的breadcrumbs），这个字段的值能被覆盖，我们能通过设置它只读来防止：

```
1 @auth.requires_login():
2 def manage():
3     db.child.parent.writable = False
4     grid = SQLFORM.smartgrid(db.parent, linked_tables=['child'])
5     return locals()
```

如果linked_tables参数没有指定，所有引用表自动链接。尽管这样，为了避免意外公开数据，我们推荐显示列出要链接的表格。

下面的代码为系统中的表创建非常有力的管理界面：

```
1 @auth.requires_membership('managers'):
2 def manage():
3     table = request.args(0) or 'auth_user'
4     if not table in db.tables(): redirect(URL('error'))
5     grid = SQLFORM.smartgrid(db[table], args=request.args[:1])
6     return locals()
```

smartgrid与grid用同样的参数，更多注意事项：

- 第一个参数是表，不是查询。
- 有额外参数约束，即‘tablename’字典：query被用于进一步限制访问显示在‘tablename’ grid的记录。
- 有额外参数linked_tables，通过smartgrid访问的表的表名的列表。
- 所有的参数除了table、args、linked_tables和user_signatures，都能是解释如下的字典。

考虑之前的grid：

```
1 grid = SQLFORM.smartgrid(db.parent, linked_tables=['child'])
```

它允许访问db.parent和db.child，除了为导航控制，对每个表，smarttable就是一个grid。这意味着，这种情况下，smartgrid能给父创建grid，给子创建grid，我们也许想传递不同的参数集给这些grid，例如不同的searchable参数集。

而对grid，我们传递布尔值：

```
1 grid = SQLFORM.grid(db.parent, searchable=True)
```

对smartgrid，我们传递布尔值字典：

```
1 grid = SQLFORM.smartgrid(db.parent, linked_tables=['child'],  
2 searchable= dict(parent=True, child=False))
```

这样，我们使父可搜索，但父的子不可搜索（也不该有那么多搜索工具）。

*grid*和*smartgrid*小工具就停留在这，但它们标识为实验阶段，因为它们返回实际的html布局以及能传递给他们的确切参数集，当新功能加入，可能会改变。

*grid*和*smartgrid*象*crud*一样不会自动实现访问控制，但能用*auth*通过显式许可检查集成：

```
1 grid = SQLFORM.grid(db.auth_user,  
2 editable = auth.has_membership('managers'),  
3 deletable = auth.has_membership('managers'))
```

或

```
1 grid = SQLFORM.grid(db.auth_user,  
2 editable = auth.has_permission('edit', 'auth_user'),  
3 deletable = auth.has_permission('delete', 'auth_user'))
```

*smartgrid*是web2py仅有的小工具，显示表名，它需要单数和复数。例如，父可以有单个“Child”或许多“Children”。因此表对象需要知道它自己的单数和复数名，web2py一般猜测它们，但可以显式设置：

```
1 db.define_table('child', ..., singular="Child", plural="Children")
```

或者用

```
1 db.define_table('child', ...)  
2 db.child._singular = "Child"  
3 db.child._plural = "Children"
```

它们也该用T运算符国际化。

然后复数和单数值被*smartgrid*用来为头和链接提供正确的名字。

第 8 章 电子邮件和短信系统

8.1 设置电子邮件

Web2py 提供 `gluon.tools.Mail` 类使得用 web2py 发送电子邮件变得简单，可用下面语句定义邮件服务器：

```
1 from gluon.tools import Mail
2 mail = Mail()
3 mail.settings.server = 'smtp.example.com:25'
4 mail.settings.sender = 'you@example.com'
5 mail.settings.login = 'username:password'
```

注意，如果应用使用 Auth（下一章讨论），Auth 对象包括它自己的邮件对象都在 `auth.settings.mailer`，因此可以用它，如下面所示：

```
1 mail = auth.settings.mailer
2 mail.settings.server = 'smtp.example.com:25'
3 mail.settings.sender = 'you@example.com'
4 mail.settings.login = 'username:password'
```

需要用 SMTP 服务器合适的参数替换 `mail.settings`，如果 SMTP 服务器不需要认证，设置 `mail.settings.login=False`。

为了调试的目的，可以设置

```
1 mail.settings.server = 'logging'
```

电子邮件不发送而是记录在控制台。

8.1.1 给 Google App Engine 配置电子邮件

为了能从 Google App Engine 账户发送电子邮件：

```
1 mail.settings.server = 'gae'
```

撰写本书的时候，web2py 不支持附件和 Google App Engine 加密电子邮件。

8.1.2 x509 和 PGP 加密

能用下面的设置发送 x509（SMIME）加密邮件：

```
1 mail.settings.cipher_type = 'x509'
2 mail.settings.sign = True
3 mail.settings.sign_passphrase = 'your passphrase'
4 mail.settings.encrypt = True
5 mail.settings.x509_sign_keyfile = 'filename.key'
6 mail.settings.x509_sign_certfile = 'filename.cert'
7 mail.settings.x509_crypt_certfiles = 'filename.cert'
```

能用下面的设置发送 PGP 加密邮件：

```
1 from gpgme import pgp
2 mail.settings.cipher_type = 'gpg'
3 mail.settings.sign = True
4 mail.settings.sign_passphrase = 'your passphrase'
5 mail.settings.encrypt = True
```

后者需要 python-pyme 包。

8.2 发送电子邮件

一旦 mail 定义了，它能用来发送邮件通过：

```
1 mail.send(to=['somebody@example.com'],
2 subject='hello',
3 # If reply_to is omitted, then mail.settings.sender is used
4 reply_to='us@example.com',
5 message='hi there')
```

如果成功发送了电子邮件，它返回True，否则返回False。mail.send()完整参数列表如下：

```
1 send(self, to, subject=None, message=None, attachments=1,
2 cc=1, bcc=1, reply_to=1, encoding='utf-8', headers={})
```

注意，to、cc和 bcc，每个都采用一个电子邮件地址列表。

headers是头的字典，在发送电子邮件之前提炼标头，例如：

```
1 headers = {'Return-Path' : 'bounces@example.org'}
```

以下是展示mail.send()使用的一些附加的例子。

8.2.1 简单文本电子邮件

```
1 mail.send('you@example.com',
2 'Message subject',
3 'Plain text body of the message')
```

8.2.2 HTML 电子邮件

HTML 电子邮件

```
1 mail.send('you@example.com',
2 'Message subject',
3 '<html>html body</html>')
```

如果电子邮件主体用<html>开始并以</html>结束，它作为HTML电子邮件发送。

8.2.3 文本和 HTML 混合电子邮件

电子邮件消息可以是元组（文本、html）：

```
1 mail.send('you@example.com',
2 'Message subject',
3 ('Plain text body', '<html>html body</html>'))
```


8.2.4 抄送和密送 cc and bcc emails

```
1 mail.send('you@example.com',
2 'Message subject',
3 'Plain text body',
4 cc=['other1@example.com', 'other2@example.com'],
5 bcc=['other3@example.com', 'other4@example.com'])
```

8.2.5 附件 Attachments

```
1 mail.send('you@example.com',
2 'Message subject',
3 '<html></html>',
4 attachments = Mail.Attachment('/path/to/photo.jpg', content_id='photo'))
```

8.2.6 多附件 Multiple attachments

```
1 mail.send('you@example.com',
2 'Message subject',
3 'Message body',
4 attachments = [Mail.Attachment('/path/to/first.file'),
5 Mail.Attachment('/path/to/second.file')])
```

8.3 发送短信

从 web2py 应用发送短消息需要第三方服务，传递消息到接收者。通常，这不是免费服务，而且国家与国家不同，我们尝试了很多的服务几乎都不成功，电话公司封堵源自这些服务的电子邮件，因为它们最终用做垃圾邮件源。

好点的方法是用电话公司自己来转发 SMS，每个电话公司有电子邮件地址，其唯一关联每个移动电话号码，因此 SMS 消息能作为电子邮件发送到电话号码。web2py 提供模块帮助这个处理：

```
1 from gluon.contrib.sms_utils import SMSCODES, sms_email
2 email = sms_email('1 (111) 111-1111', 'T-Mobile USA (tmail)')
3 mail.send(to=email, subject='test', message='test')
```

SMSCODES 是字典，映射主要电话公司名字到电子邮件地址前缀，sms_email 函数用电话号码（作为字符串）和电话公司的名字，并返回电话的电子邮件地址。

8.4 用模板系统生成消息

能用模板系统生成邮件。例如，考虑如下数据库表：

```
1 db.define_table('person', Field('name'))
```

打算发送给数据库里每个人，下面的是存储在视图文件 “message.html” 中的消息：

```
1 Dear {{=person.name}},
2 You have won the second prize, a set of steak knives.
```

可以用下面的方法实现：

```
1 for person in db(db.person).select():
2     context = dict(person=person)
3     message = response.render('message.html', context)
4     mail.send(to=['who@example.com'],
5               subject='None',
6               message=message)
```

大部分工作在如下语句中完成：

```
1 response.render('message.html', context)
```

用在字典“context”中的定义的变量呈现了视图“message.html”，而且它用呈现的电子邮件文本返回字符串，context 是字典包含对模板文件可视的变量。

如果消息用<html>开始并以</html>结束，这个电子邮件是HTML电子邮件。

注意，如果想在 HTML 电子邮件中包含回到你的网站链接，可以用 URL 函数。然而，默认情况是 URL 函数生成相对 URL，其将来自电子邮件不能用，要生成绝对的 URL，需要指定 URL 函数的 scheme 和 host 参数。例如：

```
1 <a href="{=URL(..., scheme=True, host=True)}" ">Click here</a>
```

或者

```
1 <a href="{=URL(..., scheme='http', host='www.site.com')}" ">Click here</a>
```

用在生成电子邮件文本的同样的机制，也用来生成 SMS 消息或其它任何基于模板类型消息。

8.5 用后台任务发送消息

发送电子邮件消息的操作可占用几秒，因为需要登录和通信可能的远端的 SMTP 服务器，为避免用户等待发送操作完成，有时希望通过后台任务让邮件排队稍后发送，正如第 4 章描述的，这可以通过设置自制任务队列或用 web2py 调度器实现。下面我们提供了一个使用自制任务队列的例子。

首先，在我们应用模型文件里，我们设置数据库模型存储我们的电子邮件队列：

```
1 db.define_table('queue',
2     Field('status'),
3     Field('email'),
4     Field('subject'),
5     Field('message'))
```

然后，从控制器，我们能排队要发送的消息：

```
1 db.queue.insert(status='pending',
2     email='you@example.com',
3     subject='test',
4     message='test')
```

接下来，我们需要后台处理脚本读取队列并发送邮件：

```
1 # in file /app/private/mail_queue.py
2 import time
3 while True:
4     rows = db(db.queue.status=='pending').select()
5     for row in rows:
6         if mail.send(to=row.email,
7             subject=row.subject,
```

```
8 message=row.message):
9 row.update_record(status='sent')
10 else:
11 row.update_record(status='failed')
12 db.commit()
13 time.sleep(60) # check every minute
```

最后，如第四章描述的，我们需要运行mail_queue.py脚本，好像它在我们应用控制器中：

```
1 python web2py.py -S app -M -N -R applications/app/private/mail_queue.py
```

-S app告诉web2py以“app”运行“mail_queue.py”，-M告诉web2py执行模型，-N告诉web2py不要运行cron。

这里我们假定在“mail_queue.py”中引用的mail对象，在我们应用的模型文件中被定义，因此是在“mail_queue.py”脚本中可用，因为-M选项。也要注意，为了对其它并发的处理不锁定数据库，尽可能快地提交任何改变是重要的。

如第四章记述的，这种后台进程的类型不能通过cron（可能cron@reboot除外）执行，因为需要确认在同一个时间仅有一个实例在运行。

注意，通过后台处理发送电子邮件的一个缺点是：它使得在电子邮件失败的情况给用户反馈困难，如果电子邮件从控制器动作直接发送，你能捕捉到任何错误，并立即返回错误消息给用户，可是用后台进程，电子邮件异步发送，在控制器动作已经返回了它的响应以后，那么它告知用户失败变得更加复杂。

第9章 访问控制 Access Control

Web2py 包含强大的并且可定制的基于角色的访问控制机制（RBAC）。

下面是来自维基百科（Wikipedia）的定义：

“基于角色的访问控制（RBAC）是一种给授权用户限制系统访问的方法，它是新的替代强制访问控制（MAC）和自主访问控制（DAC）的方案，RBAC有时被认为基于角色的安全。

RBAC是一种策略中性和灵活的访问控制技术，足够强大到来模拟DAC和MAC。相反，如果角色图限制为树而不是部分排序集，MAC能模拟RBAC。

RBAC开发出来之前，MAC和DAC被视为仅知的访问控制模型：如果一个模型不是MAC，被视为DAC模型，反之亦然。1990s后期研究表明RBAC不属于任何一类。

在一个组织中，角色为各种工作职能而创造。执行某些操作的权限派发给特定角色，成员（或其它系统用户）被指派了特定角色，通过那些角色指派得到执行特定系统功能的权限，与基于内容的访问控制（CBAC）不同，RBAC不看消息内容（比如连接的源）。

因为用户不被直接地分配权限，而仅通过他们的角色（或角色）获得，个别用户权利的管理变成简单地给用户指派合适角色的问题，这简化了公共操作，比如添加用户或改变用户部门。

RBAC不用于访问控制表（ACLs），用在传统自主访问控制系统，该系统用组织中的意思分配权限给特定操作而不是低级别数据对象。例如，访问控制表可以用来准许或取消对特定系统文件写访问，但它没有规定那个文件如何被修改。

”

实现RBAC的web2py类叫做Auth。

Auth需要（定义）下面的表：

- `auth_user` 存储用户名字、电子邮件地址、密码和状态（注册等候、接受、阻止）
- `auth_group` 为用户在多对多结构中存储组或角色。默认情况，每个用户在它自己的组中，但一个用户可以在多个组，并且每个组可以包含多个用户，组用角色和描述定义。
- `auth_membership` 在多对多结构中链接用户和组。
- `auth_permission` 链接组合权限，权限用名字或可选表和记录来定义，例如某个组的成员有“update”权限对指定表的指定记录。
- `auth_event` 日志在其它表中改变，通过RBAC控制的对象的CRUD成功访问。

原理上，没有对角色名字和权限名字的限制。开发者可以创建它们固定组织中的角色和权限，一旦它们被创建，web2py提供API检查是否用户登录了，用户是否是所给出组的成员，或用户是否是任意一个组的用户，并得到了需要的权限。web2py也提装饰器来限制访问基于登录、成员资格和权限的任意功能，web2py也理解一些特定的权限，即那些具有与CRUD方法有相应的名字（create, read, update, delete）创建、读取、更新、删除），且无需使用装饰器而强制自动使用它们。

本章，我们将逐个讨论RBAC的不同部分。

9.1 认证 Authentication

为了使用 RBAC，用户需要被识别，这意味着他们需要注册（或被注册）并登录。

Auth提供多种登录方法，默认的一种是由基于本地auth_user表识别用户。另外，能针对第三方认证系统登录用户，能对供应商单点登录，例如Google、PAM、LDAP、Facebook、LinkedIn、Dropbox、OpenID、OAuth等。

要开始使用**Auth**，至少需要在模型文件中有下面的代码，也由web2py “welcome” 应用提供，并假定有db链接对象：

```
1 from gluon.tools import Auth
2 auth = Auth(db, hmac_key=Auth.get_or_create_key())
3 auth.define_tables()
```

db.auth_user表的password字段默认是CRYPT验证器，需要hmac_key，Auth.get_or_create_key()是函数，从在应用文件夹中的文件“private/auth.key”读取hmac key，如果文件不存在，它创建一个随机的hmac_key，如果多个apps共享同一个auth数据库，确认它们也使用同一个hmac_key。

默认情况，web2py用电子邮件登录；如果不是，而想用用户名登录，设置auth.define_tables(username=True)。

如果多个 app 共享同一个 auth 数据库，你也许想禁止迁移：
auth.define_tables(migrate=False)。

要公开**Auth**，也需要在控制器中（例如在“default.py”）的如下函数。

```
1 def user(): return dict(form=auth())
```

auth对象和用户动作也已定义在基本构建应用中。

web2py也包括例子视图“welcome/views/default/user.html”，其合适地呈现这个函数，看起来如下：

```
1 {{extend 'layout.html'}}
2 <h2>{{T( request.args(0).replace('_', ' ').capitalize() )}}</h2>
3 <div id="web2py_user_form">
4   {{=form}}
5   {{if request.args(0)=='login':}}
6   {{if not 'register' in auth.settings.actions_disabled:}}
7   <br/><a href="{{=URL(args='register')}}">register</a>
8   {{pass}}
9   {{if not 'request_reset_password' in auth.settings.actions_disabled:}}
10  <br/>
11  <a href="{{=URL(args='request_reset_password')}}">lost password</a>
12  {{pass}}
13  {{pass}}
14 </div>
```

注意，这个函数简单显示form，因此它可以用常用定制表单语法定制，仅需注意的是表单用

form=auth()显示依赖于request.args(0);因此,如果用定制登录表单替换默认auth()登录表单,需要下面视图中的if语句:

```
1 {{if request.args(0)=='login':}}...custom login form...{{pass}}
```

上面的控制器公开了许多动作:

```
1 http://.../[app]/default/user/register
2 http://.../[app]/default/user/login
3 http://.../[app]/default/user/logout
4 http://.../[app]/default/user/profile
5 http://.../[app]/default/user/change_password
6 http://.../[app]/default/user/verify_email
7 http://.../[app]/default/user/retrieve_username
8 http://.../[app]/default/user/request_reset_password
9 http://.../[app]/default/user/reset_password
10 http://.../[app]/default/user/impersonate
11 http://.../[app]/default/user/groups
12 http://.../[app]/default/user/not_authorized
```

- register 允许用户注册,集成CAPTCHA,但是默认禁用。
- login 允许注册用户登录(如果注册确认了或不需要确认,如果批准或无需批准,且如果它没有被锁定)。
- logout 做你期望做的,也作为其他方法,日志事件并用来触发一些事件。
- profile 允许用户编辑它们的简介,即auth_user表的内容,注意该表没有固定的结构并能被定制。
- change_password 允许用户以失败安全模式修改他们的密码。
- verify_email 如果电子邮件确认打开,那么访问者,一旦注册,就接收邮件链接确认它们的电子邮件信息,且链接指向这个动作。
- retrieve_username 默认情况,Auth使用电子邮件和密码登录,但它是可选的,且它使用用户名而不是电子邮件,后面这种情况,如果用户忘记他/她的用户名, retrieve_username方法允许用户输入电子邮件地址并通过电子邮件得到用户名。
- request_reset_password 允许忘记他们密码的用户请求新密码,他们会得到确认电子邮件指向reset_password。
- impersonate 允许一个用户“impersonate”另一用户,这对于调试和支持目的是重要的, request.args[0]是那个被模拟的用户的id,如果登录用户 has_permission('impersonate'、db.auth_user、user_id),这才会被允许。
- groups 列出当前登录用户是成员的组。
- not_authorized 显示错误消息,当访问者试图做那些他/她未被授权的事情。
- navbar 是帮助对象生成任务栏,其有login/register/等链接。

默认情况,它们全都公开,但也可能限制访问这些动作的其中一部分。

所有上面的方法都能被扩展或Auth子类替代。

所有上面的方法能用于分离动作。例如:

```
1 def mylogin(): return dict(form=auth.login())
2 def myregister(): return dict(form=auth.register())
3 def myprofile(): return dict(form=auth.profile())
4 ...
```

要限制访问函数,只需登录为访问者,如下面例子装饰函数:

```
1 @auth.requires_login()
```

```
2 def hello():
3 return dict(message='hello %(first_name)s' % auth.user)
```

任何函数都能被装饰，不仅仅是公开的动作，当然，这仍然是一个简单的访问控制的例子，更复杂地例子后面会讨论。

auth.user包含当前登录用户db.auth_user记录的副本或反之None，也有auth.user_id与auth.user.id（即当前登录用户的id）一样或None。

9.1.1 注册限制 Restrictions on registration

如果你想允许访问者能注册但不能登录，直到注册被管理员批准。

```
1 auth.settings.registration_requires_approval = True
```

你可以通过appadmin界面批准注册，在表auth_user中查找，等候注册有registration_key字段，设置为“pending”，当该字段设置为空，注册通过。

通过appadmin界面，也可以阻止用户登录，在表auth_user定位用户，设置registration_key为“blocked”，“blocked”用户不允许登录，注意这会阻止访问者登录，但它不会强制已经登录的访问者退出，如果愿意也可用“disabled”替换“blocked”，完全相同的行动。

可以用下面的语句完全阻止访问“register”页面。

```
1 auth.settings.actions_disabled.append('register')
```

如果允许用户注册，并且注册后自动登录，但仍想发送电子邮件证实以便他们退出后不能再登录，除非他们完成了电子邮件中的指示，可以如下实现：

```
1 auth.settings.registration_requires_approval = True
2 auth.settings.login_after_registration = True
```

其它Auth方法也能用同样的方法限制。

9.1.2 集成OpenID, Facebook 等

可以用web2py基于角色的访问控制，也可用其它服务认证，比如OpenID、Facebook、LinkedIn、Google、Dropbox、MySpace和Flickr等，最简单的方法是使用Janrain Engage（以前 RPX）(Janrain.com)。

Dropbox作为特例在第14章讨论，因为它不仅允许登录，它还为登录的用户提供存储服务。

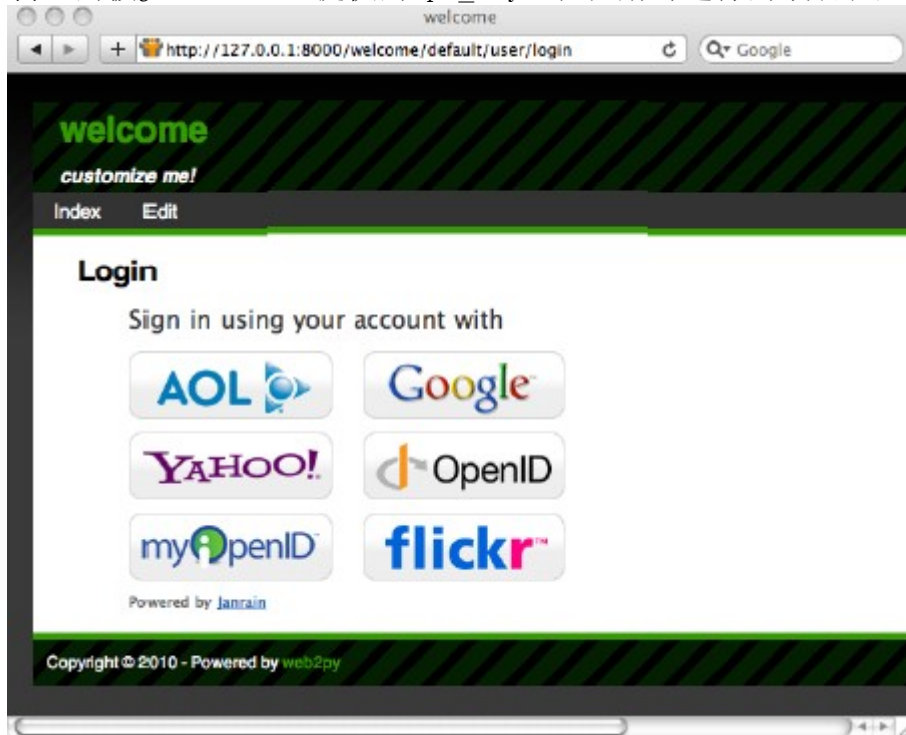
Janrain Engage是提供中间件认证的服务，可以用Janrain.com注册、登记域名（你应用的名字）以及设置将使用的URL，并提供一个API给你。

现在，编辑你的web2py应用模型，在auth对象定义之后，把下面几行内容放置：

```
1 from gluon.contrib.login_methods.rpx_account import RPXAccount
2 auth.settings.actions_disabled=['register', 'change_password',
request_reset_password']
3 auth.settings.login_form = RPXAccount(request,
4 api_key='...',
5 domain='...',
6 url = "http://localhost:8000/%s/default/user/login" % request.application)
```

第一行导入新登录方法，第二行禁用本地注册，第三行要求web2py用PBX登录方法，你必须

插入你自己的被Janrain.com提供的api_key，在注册时选择的域名和登录页面的外部url。



当新用户第一次登录时，web2py创建新db.auth_user关联该用户，它用registration_id字段为用户存储唯一的id。大多数认证方法也提供用户名、电子邮件、姓和名字但这不保证哪些字段提供，其取决于用户选择的登录方法，如果同一个用户用不同的认证机制登录两次（例如，一次用OpenID，另外一次用Facebook），Janrain可能不认为他/她是同一个用户，并发给不同的registration_id。

可以定制存储在db.auth_user的数据和Janrain提供的数据之间的映射。

```
1 auth.settings.login_form.mappings.Facebook = lambda profile:\n2 dict(registration_id = profile["identifier"],\n3 username = profile["preferredUsername"],\n4 email = profile["email"],\n5 first_name = profile["name"]["givenName"],\n6 last_name = profile["name"]["familyName"])
```

字典的键是db.auth_user的字段，值是Janrain提供配置文件对象的数据项，后者的详细说明参见在线Janrain文档。

Janrain也保留有关你的用户登录的统计。

该登录表单完全集成web2py基于角色的访问控制，你可以创建组，生成组用户成员，分配权限和阻止用户等。

Janrain的免费基本服务允许上限为2500唯一注册用户每年登录，提供更多用户需要升级到它们的付费服务等级。

如果选择不用Janrain而想用不同的登录方法（LDAP、PAM、Google、OpenID、OAuth/Facebook、LinkedIn等），你也可以这样做。实现的API在本章后面描述。

9.1.3 CAPTCHA 和 reCAPTCHA

为防止你网站的垃圾邮件和机器人注册，你可能需要注册CAPTCHA，Web2py支持即开即用的

reCAPTCHA[74]，这是因为reCAPTCHA很好的设计、免费、可接入（它能读取给访问者的词）、易安装且不需要安装第三方库。

下面是使用reCAPTCHA需要做的：

- 注册reCAPTCHA[74]并获得（PUBLIC_KEY，PRIVATE_KEY）账号对，这是两个字符串。
- 在auth对象被定义后，追加下面的代码到你的模型：

```
1 from gluon.tools import Recaptcha
2 auth.settings.captcha = Recaptcha(request,
3 'PUBLIC_KEY', 'PRIVATE_KEY')
```

如果你访问‘localhost’或‘127.0.0.1’ web网站，reCAPTCHA可能不工作，因为它仅对公众可见的web网站注册才工作。

Recaptcha构造函数采用下面可选参数：

```
1 Recaptcha(..., use_ssl=True, error_message='invalid', label='Verify:', options='')
```

注意默认use_ssl=False。

options可以是配置字符串，例如options="theme:' white' ,lang:' fr' "。

更多细节：reCAPTCHA[?]和定制[?]

如果不想使用reCAPTCHA以及查看在“gluon/tools.py”中的Recaptcha类的定义，因为对使用其它CAPTCHA系统来说，它是容易的。

注意Recaptcha就是扩展了DIV的帮助对象，它生成虚拟字段验证使用reCaptcha服务，因此它能以任何形式使用，包括用在定义FORMs：

```
1 form = FORM(INPUT(...), Recaptcha(...), INPUT(_type='submit'))
```

通过注入，你可以在所有SQLFORM类型中使用它：

```
1 form = SQLFORM(...) or SQLFORM.factory(...)
2 form.element('table').insert(-1, TR('', Recaptcha(...), ''))
```

9.1.4 定制 Auth

调用

```
1 auth.define_tables()
```

定义所有还未定义的Auth表，这意味着如果你想这么做，你可以定义你自己的auth_user表。

有一些方法定制auth，最简单的方法是添加额外字段：

```
1 # after auth = Auth(db)
2 auth.settings.extra_fields['auth_user'] = [
3 Field('address'),
4 Field('city'),
5 Field('zip'),
6 Field('phone')]
7 # before auth.define_tables(username=True)
```

你不但可以给表“auth_user”，也可以给其它表“auth_”声明额外字段，用extra_fields是推荐的方法，因为它不会打破任何内部机制。

另外一个方法（给专家的！）在于自己定义你的auth表。

在auth.define_tables()之前定义表，它不使用默认而是定义的，如何实现它如下：

```
1 # after auth = Auth(db)
2 db.define_table(
3 auth.settings.table_user_name,
4 Field('first_name', length=128, default=''),
5 Field('last_name', length=128, default=''),
6 Field('email', length=128, default='', unique=True), # required
7 Field('password', 'password', length=512, # required
8 readable=False, label='Password'),
9 Field('address'),
10 Field('city'),
11 Field('zip'),
12 Field('phone'),
13 Field('registration_key', length=512, # required
14 writable=False, readable=False, default=''),
15 Field('reset_password_key', length=512, # required
16 writable=False, readable=False, default=''),
17 Field('registration_id', length=512, # required
18 writable=False, readable=False, default=''))
19
20 # do not forget validators
21 custom_auth_table = db[auth.settings.table_user_name] # get the custom_auth_table
22 custom_auth_table.first_name.requires = \
23 IS_NOT_EMPTY(error_message=auth.messages.is_empty)
24 custom_auth_table.last_name.requires = \
25 IS_NOT_EMPTY(error_message=auth.messages.is_empty)
26 custom_auth_table.password.requires = [IS_STRONG(), CRYPT()]
27 custom_auth_table.email.requires = [
28 IS_EMAIL(error_message=auth.messages.invalid_email),
29 IS_NOT_IN_DB(db, custom_auth_table.email)]
30
31 auth.settings.table_user = custom_auth_table # tell auth to use custom_auth_table
32
33 # before auth.define_tables()
```

你可以添加你想要的任何字段，并改变验证器但不能删除这个例子中标示为“required”的字段。

使“password”、“registration_key”、“reset_password_key”和“registration_id”字段 readable=False 和 writable=False，因为访问者不被允许随意更改它们。

如果增加字段“username”，它会用来替换“email”登录，如果你要做，还需要增加验证器：

```
1 auth_table.username.requires = IS_NOT_IN_DB(db, auth_table.username)
```

9.1.5 重命名 Auth 表

Auth 表的实际名字存储在：

```
1 auth.settings.table_user_name = 'auth_user'
2 auth.settings.table_group_name = 'auth_group'
3 auth.settings.table_membership_name = 'auth_membership'
4 auth.settings.table_permission_name = 'auth_permission'
```

```
5 auth.settings.table_event_name = 'auth_event'
```

在 auth 对象定义后并在 Auth 表定义前，表的名字可以重新指定上面变量得到修改。例如：

```
1 auth = Auth(db)
2 auth.settings.table_user_name = 'person'
3 #...
4 auth.define_tables()
```

实际的表可以被引用，独立地用它们的实际名字：

```
1 auth.settings.table_user
2 auth.settings.table_group
3 auth.settings.table_membership
4 auth.settings.table_permission
5 auth.settings.table_event
```

9.1.6 其它登录方法和登录表单

Auth 提供多登录方法以及创建新登录方法的钩子，每种支持的登陆方法对应文件夹中的文件。

```
1 gluon/contrib/login_methods/
```

对于每种登录的方法，参阅在文件中的文档，下面是一些例子。

首先，我们需要区别两种登录方法：

- 用 web2py 登录表单的登录方法（虽然凭证在 web2py 外确认），例如 LDAP。
- 需要外部单点登录表单的登录方法（例子是Google and Facebook）。

后一种情况，web2py从不得到登录凭证，仅是服务提供商发布的登录令牌，令牌存储在 db.auth_user.registration_id。

让我们考虑第一种情况的例子：

Basic

有认证服务，例如在 url

```
1 https://basic.example.com
```

接受基本访问控制。那意味服务器接受带有表单头的 HTTP 请求：

```
1 GET /index.html HTTP/1.0
2 Host: basic.example.com
3 Authorization: Basic QWxhZGRpbjpvGVuIHNlc2FtZQ==
```

后面的字符串是用户名：密码字符串base64的编码。如果用户授权了，服务响应200OK，否则400、401、 402、403 或404响应。

想用标准 Auth 登录表单输入用户名和密码，并对这样的服务确认凭据，所有你需要做的就是添加下面的代码到你的应用：

```
1 from gluon.contrib.login_methods.basic_auth import basic_auth
2 auth.settings.login_methods.append(
3 basic_auth('https://basic.example.com'))
```

注意auth.settings.login_methods是认证方法的列表，依次运行，它默认设置为：

```
1 auth.settings.login_methods = [auth]
```

当另外的方法追加了，例如basic_auth，Auth首先尝试基于auth_user内容登录访问者，

当这失败，它尝试列表中的下一个方法，如果方法成功登录访问者，且如果 `auth.settings.login_methods[0]==auth`，**Auth**采取下面的动作：

- 如果用户不存在于`auth_user`，新用户被创建并且用户名/电子邮件地址和密码被存储。
- 如果用户不存在于`auth_user`，但新接受的密码与存储的老的密码不匹配，老密码用新的替换（注意密码总存储散列的除非特别说明）。

如果你不希望存储新密码到`auth_user`，那么改变登录方法的顺序就足够了，或从列表删除`auth`。例如：

```
1 from gluon.contrib.login_methods.basic_auth import basic_auth
2 auth.settings.login_methods = \
3 [basic_auth('https://basic.example.com')]
```

同样适用于其它登录方法描述如下。

SMTP and Gmail

可以用远程SMTP服务器确认登录凭据，比如**Gmail**，即如果他们提供的电子邮件和密码是访问**Gmail SMTP**服务器（`smtp.gmail.com:587`）的有效凭据，接受用户登录。所有需要的代码如下：

```
1 from gluon.contrib.login_methods.email_auth import email_auth
2 auth.settings.login_methods.append(
3 email_auth("smtp.gmail.com:587", "@gmail.com"))
```

`email_auth`第一个参数是**SMTP**服务器的地址：端口，第二个参数是电子邮件域名。这对任何需要TLS认证的**SMTP**服务器都有用。

PAM

用可插拔认证模块（PAM）的认证与前面的情况相同，它允许webpy用操作系统账户认证用户：

```
1 from gluon.contrib.login_methods.pam_auth import pam_auth
2 auth.settings.login_methods.append(pam_auth())
```

LDAP

用LDAP认证与前面的情况非常相似。

要使用LDAP登录MS活动目录：

```
1 from gluon.contrib.login_methods.ldap_auth import ldap_auth
2 auth.settings.login_methods.append(ldap_auth(mode='ad',
3 server='my.domain.controller',
4 base_dn='ou=Users, dc=domain, dc=com'))
```

要使用LDAP登录Lotus Notes 和 Domino：

```
1 auth.settings.login_methods.append(ldap_auth(mode='domino',
2 server='my.domino.server'))
```

要使用LDAP登录OpenLDAP（用CN）：

```
1 auth.settings.login_methods.append(ldap_auth(mode='cn',
2 server='my.ldap.server', base_dn='ou=Users, dc=domain, dc=com'))
```

Google App Engine

当在Google App Engine运行时，用Google认证要求跳过web2py登录表单，重定向到Google页面，成功就返回，因为行为与前面的例子不同，所以API有点不同。

```
1 from gluon.contrib.login_methods.gae_google_login import GaeGoogleAccount
```

```
2 auth.settings.login_form = GaeGoogleAccount()
```

OpenID

前面我们讨论了集成Janrain（有OpenID支持），那是最简单的方法使用OpenID。但有时，你不想依赖第三方的服务，想直接从消费者（你的应用）访问OpenID供应商。

下面是例子：

```
1 from gluon.contrib.login_methods.openid_auth import OpenIDAuth
2 auth.settings.login_form = OpenIDAuth(auth)
```

OpenIDAuth需要python-openid模块单独地安装。

在底层，这种登录方法定义下面的表：

```
1 db.define_table('alt_logins',
2 Field('username', length=512, default=''),
3 Field('type', length=128, default='openid', readable=False),
4 Field('user', self.table_user, readable=False))
```

给每个用户存储openid用户名，如果你想给当前登录用户显示openid：

```
1 {{=auth.settings.login_form.list_user_openids()}}
```

OAuth2.0 and Facebook

前面我们讨论了集成Janrain（有Facebook支持），但有时，你不想依赖第三方的服务，想直接访问OAuth2.0供应商；例如Facebook，下面是如何实现：

```
1 from gluon.contrib.login_methods.oauth20_account import OAuthAccount
2 auth.settings.login_form=OAuthAccount(YOUR_CLIENT_ID, YOUR_CLIENT_SECRET)
```

如果你想用Facebook OAuth2.0登录到指定Facebook应用访问它的API而不是你自己的应用，情况会变得更复杂，下面是访问Facebook Graph API的例子。

首先，你必须安装Facebook Python SDK。

其次，在你的模型里需要下面的代码：

```
1 # import required modules
2 from facebook import GraphAPI
3 from gluon.contrib.login_methods.oauth20_account import OAuthAccount
4 # extend the OAuthAccount class
5 class FaceBookAccount(OAuthAccount):
6     """OAuth impl for Facebook"""
7     AUTH_URL="https://graph.facebook.com/oauth/authorize"
8     TOKEN_URL="https://graph.facebook.com/oauth/access_token"
9     def __init__(self, g):
10         OAuthAccount.__init__(self, g,
11                               YOUR_CLIENT_ID,
12                               YOUR_CLIENT_SECRET,
13                               self.AUTH_URL,
14                               self.TOKEN_URL)
15         self.graph = None
16 # override function that fetches user info
17 def get_user(self):
18     """Returns the user using the Graph API"""
19     if not self.accessToken():
20         return None
21     if not self.graph:
22         self.graph = GraphAPI((self.accessToken()))
23     try:
24         user = self.graph.get_object("me")
25     return dict(first_name = user['first_name'],
```



```

26 last_name = user['last_name'],
27 username = user['id'])
28 except GraphAPIError:
29 self.session.token = None
30 self.graph = None
31 return None
32 # use the above class to build a new login form
33 auth.settings.login_form=FaceBookAccount()

```

LinkedIn

前面我们讨论了集成Janrain（有LinkedIn支持），那是使用OAuth的最简单的方法，但有时，你不想依赖第三方的服务，或者想直接访问LinkedIn以获得比Janrain所提供更多的信息。

下面是例子：

```

1 from gluon.contrib.login_methods.linkedin_account import LinkedInAccount
2 auth.settings.login_form=LinkedInAccount(request, KEY, SECRET, RETURN_URL)

```

LinkedInAccount需要单独安装的“python-linkedin”模块。

X509

可以通过传递X509证书给页面来登录，你的凭据会从证书中提取，这要求从下面安装M2Crypto：

```

1 http://chandlerproject.org/bin/view/Projects/MeTooCrypto

```

一旦你安装好了M2Crypto，就可以做：

```

1 from gluon.contrib.login_methods.x509_auth import X509Account
2 auth.settings.actions_disabled=['register', 'change_password', '
request_reset_password']
3 auth.settings.login_form = X509Account()

```

现在你可以传递X509证书在web2py中认证，如何实现依赖于浏览器的，但你可能更倾向于给web服务使用证书。这种情况，可以用比如cURL尝试你的认证：

```

1 curl -d "firstName=John&lastName=Smith" -G -v --key private.key \
2 --cert server.crt https://example/app/default/user/profile

```

用Rocket即开即用（web2py内建web服务器），但如果是用不同的web服务器，你可能需要在web服务器侧一些额外的配置工作。特别是，需要告诉你的web服务器，证书在本地主机的哪里，并且它需要确认来自客户的证书。如何实现依赖于web服务器，因而此处省略。

Multiple login forms

一些登录方法修改登录表单，一些不修改，当它们做这个的时候，不能共存的。但通过提供在同一页面的多登录表单可以实现一些共存，Web2py提供了一种方法去实现。下面是混合常用登录（auth）和RPX登录（janrain.com）的例子：

```

1 from gluon.contrib.login_methods.extended_login_form import ExtendedLoginForm
2 other_form = RPXAccount(request, api_key='...', domain='...', url='...')
3 auth.settings.login_form = ExtendedLoginForm(request,
4 auth, other_form, signals=['token'])

```

如果信号设置了并且请求中的参数匹配任何信号，它将返回other_form.login_form调用，other_form能处理一些特定情况，例如，other_form.login_form内的OpenID多步登录。

否则，它会用other_form一起呈现正常登录表单。

9.2 Mail 和 Auth 类

能用以下定义 mail 对象（电子邮件对象）：

```
1 from gluon.tools import Mail
2 mail = Mail()
3 mail.settings.server = 'smtp.example.com:25'
4 mail.settings.sender = 'you@example.com'
5 mail.settings.login = 'username:password'
```

或简单地使用 auth 提供的邮件对象：

```
1 mail = auth.settings.mailer
2 mail.settings.server = 'smtp.example.com:25'
3 mail.settings.sender = 'you@example.com'
4 mail.settings.login = 'username:password'
```

你需要用合适的参数为SMTP服务器替换mail.settings，如果SMTP服务器不需要认证，设置mail.settings.login=False。

你可以在第8章中读到更多有关电子邮件API和电子邮件配置的内容，这里我们把讨论限定在Mail和Auth间的交互。

Auth中，默认情况电子邮件确认是禁止的，要启用电子邮件，在模型中auth定义的地方追加下面几行：

```
1 auth.settings.registration_requires_verification = False
2 auth.settings.registration_requires_approval = False
3 auth.settings.reset_password_requires_verification = True
4 auth.messages.verify_email = 'Click on the link http://' + \
5 request.env.http_host + \
6 URL(r=request, c='default', f='user', args=['verify_email']) + \
7 '/%(key)s to verify your email'
8 auth.messages.reset_password = 'Click on the link http://' + \
9 request.env.http_host + \
10 URL(r=request, c='default', f='user', args=['reset_password']) + \
11 '/%(key)s to reset your password'
```

需要替换auth.messages.verify_email字符串

```
1 'Click on the link ...'
```

在auth.messages.verify_email，用动作verify_email的合适完整的URL，这是必须的，因为web2py可能安装在代理，它不能绝对肯定地确定它自己的公众URL。

9.3 授权 Authorization

一旦新用户注册，新的组会被创建去包含该用户，该新用户角色传统上是

“user_[id]”，[id]是新创建用户id的id。能用下面的语句禁用创建组：

```
1 auth.settings.create_user_groups = False
```

但是我们建议不要这样做。

在组内用户有成员资格，每个组用名字/角色标识。组有权限，用户所具有的权限是因为他们隶属组。

可以创建组，通过appadmin授予成员资格和权限或用下面的方法编程授予：

```
1 auth.add_group('role', 'description')
```

返回新创建组的id。

```
1 auth.del_group(group_id)
```

用group_id删除组。

```
1 auth.del_group(auth.id_group('user_7'))
```

用角色 “user_7” 删除组，即组唯一关联到用户编号7。

```
1 auth.user_group(user_id)
```

返回唯一关联到的由user_id标识的用户组id。

```
1 auth.add_membership(group_id, user_id)
```

给组group_id的user_id成员资格，如果user_id未被指定，那么web2py假定当前登录的用户。

```
1 auth.del_membership(group_id, user_id)
```

剥夺组group_id的user_id成员资格，如果user_id未被指定，那么web2py假定当前登录的用户。

```
1 auth.has_membership(group_id, user_id, role)
```

检查user_id是否具有组group_id的成员资格或组是否有指定的角色，仅group_id或role应该传递给函数，不是两个一起，如果user_id没有指定，那么web2py假定当前登录的用户。

```
1 auth.add_permission(group_id, 'name', 'object', record_id)
```

将对象 “object” （也是用户定义）中 “name” （用户定义的）权限给组group_id成员，如果 “object” 是表名，则权限指整个表，通过设定record_id为0值；或通过设定record_id比0大的值，来实现权限参考特定记录。当就表给权限，通常用集合（ ' create' , ' read' , ' update' , ' delete' , ' select' ）做权限名，因为这些权限能被CRUD理解并执行。

如果group_id是0，web2py使用唯一与当前登录用户相关的组。

也可以用auth.id_group(role=“...”)给出角色名字得到组的id。

```
1 auth.del_permission(group_id, 'name', 'object', record_id)
```

剥夺权限。

```
1 auth.has_permission('name', 'object', record_id, user_id)
```

检查用user_id标识的用户是否有所请求权限的组的成员资格。

```
1 rows = db(auth.accessible_query('read', db.mytable, user_id))\
2 .select(db.mytable.ALL)
```

返回表 “mytable” 的用户user_id有 “read” 权限的所有行，如果user_id没有指定，则web2py假定当前登录用户，accessible_query(...)能与其它查询组合一起，使更复杂的accessible_query(...)是唯一要求JOIN的Auth方法，因此它不适用于Google App Engine。

假设下面的定义：

```
1 >>> from gluon.tools import Auth
```

```
2 >>> auth = Auth(db)
```

```
3 >>> auth.define_tables()
4 >>> secrets = db.define_table('document', Field('body'))
5 >>> james_bond = db.auth_user.insert(first_name='James',
6 last_name='Bond')
```

有例子:

```
1 >>> doc_id = db.document.insert(body = 'top secret')
2 >>> agents = auth.add_group(role = 'Secret Agent')
3 >>> auth.add_membership(agents, james_bond)
4 >>> auth.add_permission(agents, 'read', secrets)
5 >>> print auth.has_permission('read', secrets, doc_id, james_bond)
6 True
7 >>> print auth.has_permission('update', secrets, doc_id, james_bond)
8 False
```

9.3.1 装饰器 Decorators

最常见检查权限的途径不是显式调用上面的方法，而是装饰函数以便相关登录用户的权限被检查。下面是例子:

```
1 def function_one():
2     return 'this is a public function'
3
4 @auth.requires_login()
5 def function_two():
6     return 'this requires login'
7
8 @auth.requires_membership('agents')
9 def function_three():
10    return 'you are a secret agent'
11
12 @auth.requires_permission('read', secrets)
13 def function_four():
14    return 'you can read secret documents'
15
16 @auth.requires_permission('delete', 'any file')
17 def function_five():
18    import os
19    for file in os.listdir('./'):
20        os.unlink(file)
21    return 'all files deleted'
22
23 @auth.requires(auth.user_id==1 or request.client=='127.0.0.1', requires_login=True)
24 def function_six():
25    return 'you can read secret documents'
26
27 @auth.requires_permission('add', 'number')
28 def add(a, b):
29    return a + b
30
31 def function_seven():
32    return add(3, 4)
```

@auth.requires (condition) 的条件参数是可以调用的，@auth.requires也用可选参数

`requires_login`，默认是`True`；如果设置为`False`，在条件值是`true/false`前，它不需要登录，条件可以是布尔值或值为布尔的函数。

注意那些权限访问所有函数，除了第一个是受权限限制，访问者可能有也可能没有权限。

如果用户没有登录，权限不能被检查，用户重定向到登录页面然后回到要求权限的页面。

9.3.2 组合要求 `Combining requirements`

偶尔，需要组合要求，这能通过一般 `requires` 装饰器实现，用单个参数 `true` 或 `false` 条件。例如，给代理访问权，但是仅在星期四：

```
1 @auth.requires(auth.has_membership(group_id=agents) \
2 and request.now.weekday()==1)
3 def function_seven():
4 return 'Hello agent, it must be Tuesday!'
```

或等价于：

```
1 @auth.requires(auth.has_membership(role='Secret Agent') \
2 and request.now.weekday()==1)
3 def function_seven():
4 return 'Hello agent, it must be Tuesday!'
```

9.3.3 授权和 CRUD `Authorization and CRUD`

使用装饰器和/或显式检查提供实现访问控制的方法。

另一种实现访问控制方法总是使用CRUD（与SQLFORM相反）访问数据库，强制数据库表和记录访问控制。用下面的语句，通过链接Auth和CRUD实现：

```
1 crud.settings.auth = auth
```

这能防止访问者使用任何CRUD函数，除非访问者登录而且有权限。例如，允许访问者粘贴评论，但仅能更新他们自己的评论（假设`crud`、`auth`和`db.comment`已定义）：

```
1 def give_create_permission(form):
2 group_id = auth.id_group('user_%s' % auth.user.id)
3 auth.add_permission(group_id, 'read', db.comment)
4 auth.add_permission(group_id, 'create', db.comment)
5 auth.add_permission(group_id, 'select', db.comment)
6
7 def give_update_permission(form):
8 comment_id = form.vars.id
9 group_id = auth.id_group('user_%s' % auth.user.id)
10 auth.add_permission(group_id, 'update', db.comment, comment_id)
11 auth.add_permission(group_id, 'delete', db.comment, comment_id)
12
13 auth.settings.register_onaccept = give_create_permission
14 crud.settings.auth = auth
15
16 def post_comment():
```

```

17 form = crud.create(db.comment, onaccept=give_update_permission)
18 comments = db(db.comment).select()
19 return dict(form=form, comments=comments)
20
21 def update_comment():
22 form = crud.update(db.comment, request.args(0))
23 return dict(form=form)

```

可以查询指定字段（那些你没有‘read’权限的）：

```

1 def post_comment():
2 form = crud.create(db.comment, onaccept=give_update_permission)
3 query = auth.accessible_query('read', db.comment, auth.user.id)
4 comments = db(query).select(db.comment.ALL)
5 return dict(form=form, comments=comments)

```

权限命名用下面的代码实现：

```
1 crud.settings.auth = auth
```

是“read”、“create”、“update”、“delete”、“select”、“impersonate”。

9.3.4 授权和下载 Authorization and downloads

使用装饰器及 crud.settings.auth 不授权文件通过通常的下载函数下载

```
1 def download(): return response.download(request, db)
```

如果想要这么做，必须显式声明当下载时，哪一个“upload”字段包含需要访问控制的文件。例如：

```

1 db.define_table('dog',
2 Field('small_image', 'upload'),
3 Field('large_image', 'upload'))
4
5 db.dog.large_image.authorization = lambda record: \
6 auth.is_logged_in() and \
7 auth.has_permission('read', db.dog, record.id, auth.user.id)

```

上传文件属性 authorization 可以是 None（默认）或是函数，决定用户是否登录并有当前记录的‘read’权限。这个例子，没有限制下载“small_image”字段链接的图像，但我们需要访问控制被“large_image”字段链接的图像。

9.3.5 访问控制和基本认证

有时需要公开那些装饰器要求作为服务访问控制的动作，即从程序或脚本调用它们，能使用认证检查授权。

Auth 允许通过 Basic 认证登录：

```
1 auth.settings.allow_basic_login = False
```

如此设置，如下动作：

```
1 @auth.requires_login()
```

```
2 def give_me_time():
3 import time
4 return time.ctime()
```

能被调用，例如从 shell 命令：

```
1 wget --user=[username] --password=[password]
2 http://.../[app]/[controller]/give_me_time
```

Basic 登录经常是给服务的唯一选择（下一章描述），默认禁用。

9.3.6 手动认证 Manual Authentication

有时候，你想实现你自己的逻辑，采用“manual”用户登录。这能通过调用函数实现：

```
1 user = auth.login_bare(username, password)
```

如果用户存在并且密码有效，login_bare 返回用户，否则它返回 None，如果“auth_user”表没有“username”字段，那么 username 是电子邮件。

9.3.7 设置和消息 Settings and messages

这是所有能为 Auth 定制的参数列表

下面语句指向 gluon.tools.Mail 对象，允许 auth 发送电子邮件：

```
1 auth.settings.mailer = None
```

下面必须是定义 user 动作的控制器名称：

```
1 auth.settings.controller = 'default'
```

下面是非常重要的设置：

```
1 auth.settings.hmac_key = None
```

它必须设置为类似“sha512:a-pass-phrase”，为 auth_user 表的“password”字段，它将被传递到 CRYPT 验证器。它是算法并通过字符串散列密码。

默认情况，auth 要求最小密码长度为 4，也可以改变：

```
1 auth.settings.password_min_length = 4
```

要禁用动作，追加它的名字如下所示列表：

```
1 auth.settings.actions_disabled = []
```

例如：

```
1 auth.settings.actions_disabled.append('register')
```

将禁用注册。

如果你想接收电子邮件确认注册，设置为 True：

```
1 auth.settings.registration_requires_verification = False
```

注册后要自动登录用户，即使他们没有完成电子邮件确认程序，设置下面为 True：

```
1 auth.settings.login_after_registration = False
```

如果是新注册者，在能登录前必须等待通过，设置为 True:

```
1 auth.settings.registration_requires_approval = False
```

批准由设置 registration_key==构成，通过 appadmin 或编程实现。

如果你不想每个新用户生成新组，设置下面为 False:

```
1 auth.settings.create_user_groups = True
```

下面的设置决定了替代登录方法和登录表单，如前面讨论的:

```
1 auth.settings.login_methods = [auth]
```

```
2 auth.settings.login_form = auth
```

你想允许 basic 登录吗?

```
1 auth.settings.allows_basic_login = False
```

下面是 login 动作的 URL:

```
1 auth.settings.login_url = URL('user', args='login')
```

如果用户试图访问注册页面，但他已经登录，他会被重定向到这个 URL:

```
1 auth.settings.logged_url = URL('user', args='profile')
```

这必须指向下载动作的 URL，简介 (profile) 包含图像的情况:

```
1 auth.settings.download_url = URL('download')
```

在大量的可能 auth 动作之后（在没有参考的情况下），这些必须指向你想进行重定向到你的用户到 URL:

```
1 auth.settings.login_next = URL('index')
2 auth.settings.logout_next = URL('index')
3 auth.settings.profile_next = URL('index')
4 auth.settings.register_next = URL('user', args='login')
5 auth.settings.retrieve_username_next = URL('index')
6 auth.settings.retrieve_password_next = URL('index')
7 auth.settings.change_password_next = URL('index')
8 auth.settings.request_reset_password_next = URL('user', args='login')
9 auth.settings.reset_password_next = URL('user', args='login')
10 auth.settings.verify_email_next = URL('user', args='login')
```

如果访问者没有登录，调用需要认证的函数，用户重定向到 auth.settings.login_url，默认 URL (' default' , ' user/login')。可以通过如下重定义替换这个动作:

```
1 auth.settings.on_failed_authentication = lambda url: redirect(url)
```

这是为重定向调用的函数，传递给这个函数的参数 url' 是为登录页面的 url。

如果访问者没有权限访问给定的函数，访问者重定向到如下定义的 URL

```
1 auth.settings.on_failed_authorization = \
2 URL('user', args='on_failed_authorization')
```

你能改变这个变量并重定向用户到其它地方。

通常，on_failed_authorization 是 URL，但它也可以是返回 URL 的函数，并且它在授权失败会被调用。

这些是回调的列表，应该在任何数据库 IO 前，单验证每个相应动作表后执行：

```
1 auth.settings.login_onvalidation = []
2 auth.settings.register_onvalidation = []
3 auth.settings.profile_onvalidation = []
4 auth.settings.retrieve_password_onvalidation = []
5 auth.settings.reset_password_onvalidation = []
```

每个回调必须是函数，用 form 对象参数，它能在数据库 IO 执行前，修改表单对象属性。

下面是在数据库 IO 执行后且在重定向前应该执行的回调列表：

```
1 auth.settings.login_onaccept = []
2 auth.settings.register_onaccept = []
3 auth.settings.profile_onaccept = []
4 auth.settings.verify_email_onaccept = []
```

下面是例子：

```
1 auth.settings.register_onaccept.append(lambda form:\
2 mail.send(to='you@example.com', subject='new user',
3 message="new user email is %s'%form.vars.email))
```

可以给任何 auth 动作启用验证码：

```
1 auth.settings.captcha = None
2 auth.settings.login_captcha = None
3 auth.settings.register_captcha = None
4 auth.settings.retrieve_username_captcha = None
5 auth.settings.retrieve_password_captcha = None
```

如果.captcha设置指向gluon.tools.Recaptcha，所有表单，相应选项（象.login_captcha）设置为 None，都有验证码，而那些相应选项设置为False的则没有。如果相反，.captcha设置为None，只有相应选项设置为gluon.tools.Recaptcha对象的表单有验证码，其它就没有。

下面是登录会话终止时间：

```
1 auth.settings.expiration = 3600 # seconds
```

可以改变密码字段的名字（例如，在 Firebird 中“password”是关键字，不能用作字段名）：

```
1 auth.settings.password_field = 'password'
```

通常登录表单试图验证电子邮件。这能通过修改设置禁用：

```
1 auth.settings.login_email_validate = True
```

你想在编辑简介页面中显示记录 id 吗？

```
1 auth.settings.showid = False
```

对客户表单，你可能想在表单中禁用自动错误提示：

```
1 auth.settings.hideerror = False
```

对客户表单，也可以改变风格：

```
1 auth.settings.formstyle = 'table3cols'
```

（可以是“table2cols”、“divs”和“ul”）

可以给 auth 生成的表单设定分隔符：

```
1 auth.settings.label_separator = ':'
```

默认情况，登录表单提供选项，通过“rememberme”选项扩展登录。终止时间可以修改或通过下面设置禁止这个选项：

```
1 auth.settings.long_expiration = 3600*24*30 # one month
```

```
2 auth.settings.remember_me_form = True
```

也可以定制下面的消息，它们的使用和内容应当是显而易见的：

```
1 auth.messages.submit_button = 'Submit'
2 auth.messages.verify_password = 'Verify Password'
3 auth.messages.delete_label = 'Check to delete:'
4 auth.messages.function_disabled = 'Function disabled'
5 auth.messages.access_denied = 'Insufficient privileges'
6 auth.messages.registration_verifying = 'Registration needs verification'
7 auth.messages.registration_pending = 'Registration is pending approval'
8 auth.messages.login_disabled = 'Login disabled by administrator'
9 auth.messages.logged_in = 'Logged in'
10 auth.messages.email_sent = 'Email sent'
11 auth.messages.unable_to_send_email = 'Unable to send email'
12 auth.messages.email_verified = 'Email verified'
13 auth.messages.logged_out = 'Logged out'
14 auth.messages.registration_successful = 'Registration successful'
15 auth.messages.invalid_email = 'Invalid email'
16 auth.messages.unable_send_email = 'Unable to send email'
17 auth.messages.invalid_login = 'Invalid login'
18 auth.messages.invalid_user = 'Invalid user'
19 auth.messages.is_empty = "Cannot be empty"
20 auth.messages.mismatched_password = "Password fields don't match"
21 auth.messages.verify_email = ...
22 auth.messages.verify_email_subject = 'Password verify'
23 auth.messages.username_sent = 'Your username was emailed to you'
24 auth.messages.new_password_sent = 'A new password was emailed to you'
25 auth.messages.password_changed = 'Password changed'
26 auth.messages.retrieve_username = 'Your username is: %(username)s'
27 auth.messages.retrieve_username_subject = 'Username retrieve'
28 auth.messages.retrieve_password = 'Your password is: %(password)s'
29 auth.messages.retrieve_password_subject = 'Password retrieve'
30 auth.messages.reset_password = ...
31 auth.messages.reset_password_subject = 'Password reset'
32 auth.messages.invalid_reset_password = 'Invalid reset password'
33 auth.messages.profile_updated = 'Profile updated'
34 auth.messages.new_password = 'New password'
35 auth.messages.old_password = 'Old password'
36 auth.messages.group_description = \
37 'Group uniquely assigned to user %(id)s'
38 auth.messages.register_log = 'User %(id)s Registered'
39 auth.messages.login_log = 'User %(id)s Logged-in'
40 auth.messages.logout_log = 'User %(id)s Logged-out'
41 auth.messages.profile_log = 'User %(id)s Profile updated'
42 auth.messages.verify_email_log = 'User %(id)s Verification email sent'
```

```

43 auth.messages.retrieve_username_log = 'User %(id)s Username retrieved'
44 auth.messages.retrieve_password_log = 'User %(id)s Password retrieved'
45 auth.messages.reset_password_log = 'User %(id)s Password reset'
46 auth.messages.change_password_log = 'User %(id)s Password changed'
47 auth.messages.add_group_log = 'Group %(group_id)s created'
48 auth.messages.del_group_log = 'Group %(group_id)s deleted'
49 auth.messages.add_membership_log = None
50 auth.messages.del_membership_log = None
51 auth.messages.has_membership_log = None
52 auth.messages.add_permission_log = None
53 auth.messages.del_permission_log = None
54 auth.messages.has_permission_log = None
55 auth.messages.label_first_name = 'First name'
56 auth.messages.label_last_name = 'Last name'
57 auth.messages.label_username = 'Username'
58 auth.messages.label_email = 'E-mail'
59 auth.messages.label_password = 'Password'
60 auth.messages.label_registration_key = 'Registration key'
61 auth.messages.label_reset_password_key = 'Reset Password key'
62 auth.messages.label_registration_id = 'Registration identifier'
63 auth.messages.label_role = 'Role'
64 auth.messages.label_description = 'Description'
65 auth.messages.label_user_id = 'User ID'
66 auth.messages.label_group_id = 'Group ID'
67 auth.messages.label_name = 'Name'
68 auth.messages.label_table_name = 'Table name'
69 auth.messages.label_record_id = 'Record ID'
70 auth.messages.label_time_stamp = 'Timestamp'
71 auth.messages.label_client_ip = 'Client IP'
72 auth.messages.label_origin = 'Origin'
73 auth.messages.label_remember_me = "Remember me (for 30 days)"

```

add|del|has成员资格日志允许使用 “%(user_id)s” 和 “(group_id)s”，add|del|has权限允许使用 “%(user_id)s”、“”%(name)s”、“%(table_name)s” 和 “%(record_id)s”。

9.4 集中认证服务 *Central Authentication Service*

Web2py 提供对第三方认证的支持和单点登录。下面我们讨论集中认证服务（CAS），它是工业标准，客户端和服务端都构建到 web2py 内。

CAS是公开的分布式认证的协议，它的工作原理如下：当访问者到达我们的web网站，我们的应用在会话中检查用户是否已经注册（例如通过session.token对象），如果用户没有经过认证，控制器从CAS应用重定向用户，用户可以登录、注册和管理他的凭据（姓名、电子邮件和密码），如果用户注册，他接到电子邮件，注册没有完成，直到他响应了电子邮件，一旦用户成功注册并登录，CAS应用重定向用户到我们的应用并发给密钥，我们的应用使用密钥，通过到后台CAS服务器的HTTP请求得到用户的凭据。

使用这样的机制，多个应用可以通过单个CAS服务器实现单点登录，提供认证的服务器称之为服务供应商，寻找认证访问者的应用程序叫做服务消费者。

CAS与OpenID很相似，有一个主要的不同，即在OpenID情况下，访问者选择服务供应商，在CAS情形，我们的应用做决定，使CAS更安全。

运行web2py的 CAS供应商，较易拷贝基本构建应用。

事实上，任何web2py应用公开了动作

```
1 # in provider app
2 def user(): return dict(form=auth())
```

是CAS2.0提供商，URL它的服务能被访问。

```
1 http://.../provider/default/user/cas/login
2 http://.../provider/default/user/cas/validate
3 http://.../provider/default/user/cas/logout
```

（我们假设应用叫做“provider”）

你能通过简单委托认证给提供商，从其它任何web应用（消费者）访问这个服务：

```
1 # in consumer app
2 auth = Auth(db, cas_provider = 'http://127.0.0.1:8000/provider/default/user/cas')
```

当你访问登录url消费者应用，它会重定向到你到提供商应用，执行认证并重定向回消费者。

注册、退出登录、修改密码获取密码的所有处理，在供应商应用完成，关于登录用户的条目会被创建在消费者侧，以便你添加额外的字段和本地配置，得益于CAS2.0，供应商的所有字段可读，在消费者auth_user表有相应字段，会自动拷贝。

Auth(..., cas_provider='...')与第三方供应商工作，并支持CAS1.0和2.0，版本自动检测。默认情况，它从基（上面的cas_provider url）构建供应商的URL，通过追加：

```
:
1 /login
2 /validate
3 /logout
```

这能在消费者和供应商中修改

```
1 # in consumer or provider app (must match)
2 auth.settings.cas_actions['login']='login'
3 auth.settings.cas_actions['validate']='validate'
4 auth.settings.cas_actions['logout']='logout'
```

如果想从不同的域名连接到web2py CAS供应商，你必须添加允许域名来启用它们：

```
1 # in provider app
2 auth.settings.cas_domains.append('example.com')
```

9.4.1 用 web2py 授权非 web2py 应用

这是可能的但依赖于web服务器，这里我们假定两个应用运行在同一个web服务器：具有mod_wsgi的 Apache，两个应用中的一个 web2py采用通过 Auth提供访问控制的应用，另外一个可能是CGI脚本、PHP程序或其它任意的，当一个客户请求访问后者，我们想指导web服务器请求到前一个应用的权限。

首先，我们需要修改web2py并添加下面的控制器：

```
1 def check_access():
2 return 'true' if auth.is_logged_in() else 'false'
```

如果用户登录了它则返回true，否则返回false，现在我们后台启动web2py进程：

```
1 nohup python web2py.py -a '' -p 8002
```

端口8002是必须的，不需要启动admin，因而没有admin密码。

然后，我们需要编辑Apache配置文件（例如“/etc/apache2/sites-available/default”）并指导apache，以便当非web2p程序调用时，它应该调用上面的check动作，仅当它返回true，它才能执行并响应请求，否则拒绝访问。

因为web2py和非web2py应用运行在同一个域名，如果用户登录到web2py应用，web2py会话cookie传递给Apache，即使当另外一个应用被请求，会允许凭证确认。

要实现这个，我们需要脚本，“web2py/scripts/access.wsgi”能实现之，该脚本就在web2py里，所有需要我们做的是，告诉apache调用这个脚本，应用的URL需要访问控制，和脚本的位置：

```
1 <VirtualHost *:80>
2 WSGIDaemonProcess web2py user=www-data group=www-data
3 WSGIProcessGroup web2py
4 WSGIScriptAlias / /home/www-data/web2py/wsgihandler.py
5
6 AliasMatch ^myapp/path/needng/authentication/myfile /path/to/myfile
7 <Directory /path/to/>
8 WSGIAccessScript /path/to/web2py/scripts/access.wsgi
9 </Directory>
10 </VirtualHost>
```

这里“m^ yapp/path/needng/authentication/myfile”是正则表达式，应该匹配输入请求“/path/to/”是web2py文件夹的绝对位置。

“access.wsgi”脚本包含下面的行：

```
1 URL_CHECK_ACCESS = 'http://127.0.0.1:8002/(app)s/default/check_access'
```

它指向我们请求的web2py应用，但你可以编辑它，指向特定的应用，运行在除8002端口的其它端口。

你也可以修改check_access()动作，使它的逻辑更复杂，这个动作能得到URL，原始地用环境变量请求

```
1 request.env.request_uri
```

你可以实现更复杂的规则：

```
1 def check_access():
2 if not auth.is_logged_in():
3 return 'false'
4 elif not user_has_access(request.env.request_uri):
5 return 'false'
6 else:
7 return 'true'
```

第 10 章 服务 Services

W3C 定义一个 web 服务为“一个设计的软件系统来支持基于网络的互操作的机器与机器的交互”。这是一个宽泛的定义，它包含相当大量的协议，协议不是用来人机通信，而是机器与机器的通信，比如 XML、JSON、RSS 等。

本章我们讨论如何用 web2py 实现 web 服务。如果感兴趣消费第三方服务（Twitter、Dropbox 等）的例子，应该查看第 9 章和第 14 章，web2py 提供即开即用，对许多协议的支持，包括 XML、JSON、RSS、CSV、XMLRPC、JSONRPC、AMFRPC 和 SOAP，web2py 也能够扩展支持更多的协议。

那些协议中的每一个都有多种方法支持，我们区分它们为：

- 以给定格式呈现函数输出（例如 XML、JSON、RSS、CSV）
- 远程过程调用（例如 XMLRPC、JSONRPC、AMFRPC）

10.1 呈现字典

10.1.1 HTML、XML 和 JSON

考虑下面的动作：

```
1 def count():
2 session.counter = (session.counter or 0) + 1
3 return dict(counter=session.counter, now=request.now)
```

这个动作返回计数，即当访问者重载页面时计数加 1，并且返回当前页面请求的时间戳。

通常这个页面被请求，通过：

```
1 http://127.0.0.1:8000/app/default/count
```

以 HTML 呈现。无需写哪怕一行代码，通过添加 URL 的扩展，可以要求 web2py 用不同的协议呈现这个页面：

```
1 http://127.0.0.1:8000/app/default/count.html
2 http://127.0.0.1:8000/app/default/count.xml
3 http://127.0.0.1:8000/app/default/count.json
```

动作返回的字典会分别用 HTML、XML 和 JSON 呈现。

下面是 XML 输出：

```
1 <document>
2 <counter>3</counter>
3 <now>2009-08-01 13:00:00</now>
4 </document>
```

下面是 JSON 输出：

```
1 { 'counter':3, 'now':'2009-08-01 13:00:00' }
```

注意日期、时间和日期时间对象用 ISO 格式以字符串呈现。这不是 JSON 标准的部分，而是 web2py 的规定。

10.1.2 通用视图 Generic views

例如，当 “.xml” 扩展名被调用，web2py 查找一个叫 “default/count.xml” 的模板文件，如果没有找到它，则查找叫 “generic.xml” 的模板，文件 “generic.html”、“generic.xml” 和 “genericjson” 都由当前基本构建应用提供，其它扩展能容易地被用户定义。

为安全起见，通用视图只允许在本地主机上访问，要让远程客户能访问，你需要设置 `response.generic_patterns`。

假设你正在使用基本应用的拷贝，且在模型/db.py 中编辑下面的代码行

- 限制仅本地主机访问

```
1 response.generic_patterns = ['*'] if request.is_local else []
```

- 允许所有通用视图

```
response.generic_patterns = ['*']
```

- 允许 only.json

```
1 response.generic_patterns = ['*.json']
```

`generic_patterns` 是 glob 全局模式，它意味着你能使用与你应用匹配的任何模式或传递模式列表。

为了能在旧版本 web2py 应用中使用它，需要从一个后面的基本构建应用（v1.60 后）中复制 “generic.*” 文件。

下面是 “generic.html” 代码

```
1 {{extend 'layout.html'}}
2
3 {{=BEAUTIFY(response._vars)}}
4
5 <button onclick="document.location='{URL("admin", "default", "design",
6 args=request.application)}'">admin</button>
7 <button onclick="jQuery('#request').slideToggle()">request</button>
8 <div class="hidden" id="request"><h2>request</h2>{{=BEAUTIFY(request)}}</div>
9 <button onclick="jQuery('#session').slideToggle()">session</button>
10 <div class="hidden" id="session"><h2>session</h2>{{=BEAUTIFY(session)}}</div>
11 <button onclick="jQuery('#response').slideToggle()">response</button>
12 <div class="hidden" id="response"><h2>response</h2>{{=BEAUTIFY(response)}}</div>
13 <script>jQuery('.hidden').hide();</script>
```

下面是 “generic.xml” 代码

```
1 {{
2 try:
3 from gluon.serializers import xml
4 response.write(xml(response._vars), escape=False)
5 response.headers['Content-Type']='text/xml'
```



```

6 except:
7 raise HTTP(405, 'no xml')
8 }}

```

下面是“generic.json”代码

```

1 {{
2 try:
3 from gluon.serializers import json
4 response.write(json(response._vars), escape=False)
5 response.headers['Content-Type'] = 'text/json'
6 except:
7 raise HTTP(405, 'no json')
8 }}

```

任何字典都能被呈现在HTML、XML 和 JSON，只要它仅包含python原始类型（int、float、string、list、tuple、dictionary）。

如果字典包含其它用户自定义或web2py指定的对象，它们必须用自定义视图呈现。

10.1.3 呈现 Rows 对象

如果需要呈现一组查询语句返回的 Rows，在 XML 或 JSON 或其它格式，首先用 `as_list()` 方法转换 Rows 对象为字典列表。

考虑下面模式例子：

```

1 db.define_table('person', Field('name'))

```

下面的动作能以 HTML 呈现，但是不能用 XML 或 JSON：

```

1 def everybody():
2 people = db().select(db.person.ALL)
3 return dict(people=people)

```

而下面的动作能以 XML 和 JSON 呈现：

```

1 def everybody():
2 people = db().select(db.person.ALL).as_list()
3 return dict(people=people)

```

10.1.4 自定义格式

如果，例如你想以 Python pickle 呈现动作：

```

1 http://127.0.0.1:8000/app/default/count.pickle

```

只需创建新视图文件“default/count.pickle”包含：

```

1 {{
2 import cPickle
3 response.headers['Content-Type'] = 'application/python.pickle'
4 response.write(cPickle.dumps(response._vars), escape=False)

```

```
5 }}
```

如果想能够以 pickled 文件呈现任何动作，只需用名字 “generic.pickle” 保存上面的文件。

不是所有的对象是 pickleable，也不是所有的 pickled 对象能反 pickled，坚持原始 Python 对象并组合它们，这是安全的，对象不包含到文件流或数据库连接的引用，其通常是 pickleable，但它们只能在所有 pickled 对象的类已定义的环境中 un-pickled。

10.1.5 RSS

web2py 包含 “generic.rss” 视图，能以 RSS feed (RSS 订阅) 形式呈现动作返回的字典。

因为 RSS feed (RSS 订阅) 有固定的结构 (标题、链接、描述、项目等)，那么这要执行，动作返回的字典必须有合适的结构：

```
1 {'title' : '',
2  'link' : '',
3  'description' : '',
4  'created_on' : '',
5  'entries' : []}
```

每个实体类中的实体必须有相同的结构：

```
1 {'title' : '',
2  'link' : '',
3  'description' : '',
4  'created_on' : ''}
```

例如，下面的动作能够呈现为 RSS feed (RSS 订阅)：

```
1 def feed():
2     return dict(title="my feed",
3                 link="http://feed.example.com",
4                 description="my first feed",
5                 entries=[
6                     dict(title="my feed",
7                         link="http://feed.example.com",
8                         description="my first feed")
9                 ])
```

只要通过访问 URL：

```
1 http://127.0.0.1:8000/app/default/feed.rss
```

此外，假设下面的模型：

```
1 db.define_table('rss_entry',
2                 Field('title'),
3                 Field('link'),
4                 Field('created_on', 'datetime'),
5                 Field('description'))
```

下面的动作也能呈现为 RSS feed (RSS 订阅)：

```
1 def feed():
2     return dict(title="my feed",
3                 link="http://feed.example.com",
```

```
4 description="my first feed",
5 entries=db().select(db.rss_entry.ALL).as_list())
```

Rows对象的`as_list()`方法把行转换为字典列表。
如果存在没有显式列出键名的附加字典项，它们被忽略：

下面是web2py 提供的“generic.rss”视图：

```
1 {{
2 try:
3 from gluon.serializers import rss
4 response.write(rss(response._vars), escape=False)
5 response.headers['Content-Type']='application/rss+xml'
6 except:
7 raise HTTP(405, 'no rss')
8 }}
```

再举RSS应用的一个例子，我们考虑RSS聚合对象，从“slashdot” feed收集数据并返回新的web2py rss feed (RSS订阅)。

```
1 def aggregator():
2 import gluon.contrib.feedparser as feedparser
3 d = feedparser.parse(
4 "http://rss.slashdot.org/Slashdot/slashdot/to")
5 return dict(title=d.channel.title,
6 link = d.channel.link,
7 description = d.channel.description,
8 created_on = request.now,
9 entries = [
10 dict(title = entry.title,
11 link = entry.link,
12 description = entry.description,
13 created_on = request.now) for entry in d.entries])
```

它能被访问：

```
1 http://127.0.0.1:8000/app/default/aggregator.rss
```

10.1.6 CSV

逗号分隔值格式是表示表格式数据的协议。

考虑下面的模型：

```
1 db.define_model('animal',
2 Field('species'),
3 Field('genus'),
4 Field('family'))
```

和下面的动作：

```
1 def animals():
2 animals = db().select(db.animal.ALL)
3 return dict(animals=animals)
```

web2py不提供“generic.csv”，必须定义自定义视图“default/animals.csv”序列化

animals到CSV。下面是可能实现：

```
1 {{
2 import cStringIO
3 stream=cStringIO.StringIO()
4 animals.export_to_csv_file(stream)
5 response.headers['Content-Type']='application/vnd.ms-excel'
6 response.write(stream.getvalue(), escape=False)
7 }}
```

注意也可能定义“generic.csv”文件，但必须指定要被序列化对象的名字（例子中的“animal”），这就是我们为什么不提供“generic.csv”文件的原因。

10.2 远程过程调用 Remote procedure calls

web2py 提供了把任何函数转换为 web 服务的机制，这里描述的机制与前面描述的不一样，因为：

- 函数可能有参数
- 函数可能在模型或模块中定义而不是控制器
- 你可能想指定细节支持 RPC 方法
- 它采用了更严格的 URL 命名规则
- 它比之前的方法智能，因为它对固定的协议簇有用，同理，它不易扩展。

要用这功能：

首先，必须导入和实例化 service 对象。

```
1 from gluon.tools import Service
2 service = Service()
```

这已经在基本构建应用的“db.py”模型文件中实现。

其次，必须公开控制器的服务句柄：

```
1 def call():
2 session.forget()
3 return service()
```

这已经在基本构建应用的“db.py”模型文件中实现，如果计划使用服务会话 cookie，那就删除 session.forget()。

第三，必须装饰那些你想公开为服务的函数。下面是当前支持的装饰器列表：

```
1 @service.run
2 @service.xml
3 @service.json
4 @service.rss
5 @service.csv
6 @service.xmlrpc
7 @service.jsonrpc
8 @service.amfrpc3('domain')
9 @service.soap('FunctionName', returns={'result':type}, args={'param1':type,})
```

作为例子，考虑下面被装饰的函数：

```
1 @service.run
2 def concat(a,b):
3 return a+b
```

这个函数在模型或控制器中被定义，在那儿 call 动作被定义，这个函数现在能用两种方法远程调用：

```
1 http://127.0.0.1:8000/app/default/call/run/concat?a=hello&b=world
2 http://127.0.0.1:8000/app/default/call/run/concat/hello/world
```

两种情况 http 请求返回：

```
1 helloworld
```

如果@service.xml装饰器被使用，函数被调用，通过：

```
1 http://127.0.0.1:8000/app/default/call/xml/concat?a=hello&b=world
2 http://127.0.0.1:8000/app/default/call/xml/concat/hello/world
```

输出以XML返回：

```
1 <document>
2 <result>helloworld</result>
3 </document>
```

它能序列化函数输出，即使这是一个DAL Rows对象，这种情况下，事实上它会自动调用as_list()。

如果@service.json装饰器被使用，函数被调用，通过：

```
1 http://127.0.0.1:8000/app/default/call/json/concat?a=hello&b=world
2 http://127.0.0.1:8000/app/default/call/json/concat/hello/world
```

输出以JSON返回。

如果@service.csv装饰器被使用，服务句柄要求迭代对象之一作为返回值，比如list列表。下面是例子：

```
1 @service.csv
2 def table1(a,b):
3 return [[a,b],[1,2]]
```

这个服务通过访问下面URL之一被调用：

```
1 http://127.0.0.1:8000/app/default/call/csv/table1?a=hello&b=world
2 http://127.0.0.1:8000/app/default/call/csv/table1/hello/world
```

它返回：

```
1 hello,world
2 1,2
```

@service.rss装饰器

期待与上一章讨论的“generic.rss”视图同样格式的返回值。

每个函数允许多个装饰器。

目前为止，本章讨论的都只是前面章节描述方法的替代。服务对象的力量，与XMLRPC、JSONRPC and AMFRPC一道，讨论如下。

10.2.1 XMLRPC

考虑下面的代码，例如在“default.py”控制器：

```
1 @service.xmlrpc
2 def add(a,b):
3     return a+b
4
5 @service.xmlrpc
6 def div(a,b):
7     return a/b
```

现在，在python shell 你能做

```
1 >>> from xmlrpclib import ServerProxy
2 >>> server = ServerProxy(
3     'http://127.0.0.1:8000/app/default/call/xmlrpc')
4 >>> print server.add(3,4)
5 7
6 >>> print server.add('hello','world')
7 'helloworld'
8 >>> print server.div(12,4)
9 3
10 >>> print server.div(1,0)
11 ZeroDivisionError: integer division or modulo by zero
```

Python xmlrpclib模块给XMLRPC协议提供客户，web2py做服务器。

客户通过ServerProxy连接服务器，并能远程调用服务器上的装饰过的函数，data(a,b)传递给函数，不是通过GET/POST变量，而是用XMLRPC协议适当地编码在请求中，并因此它携带了其自身类型信息（整型或字符串，或其它）。返回值也是一样的，而且任何服务器上引发的异常被传回客户端。

对许多编程语言（包括C、C++、Java、C#、Ruby和Perl）有XMLRPC库，它们之间能相互操作，这是最好的方法，创建能够彼此交谈的应用并独立于编程语言。

XMLRPC客户也可以在web2py动作内被实现，以便一个动作能与另外一个web2py应用（即使在同一个安装当中）通过使用XMLRPC进行对话，注意这种情况下的会话死锁，如果动作通过XMLRPC调用同一个应用的函数，调用者在调用前必须释放会话锁：

```
1 session.forget(response)
```

10.2.2 JSONRPC

这一节，我们将使用和XMLRPC一样的代码例子，但我们会公开使用JSONRPC的服务：

```
1 @service.jsonrpc
2 def add(a,b):
3     return a+b
4
5 def call():
6     return service()
```

JSONRPC与XMLRPC非常相似，但使用JSON而不是XML序列化协议。

当然，我们能从任何语言的任何程序调用服务，但在这我们将用Python完成，web2py提供Mariano Reingart创建的模块“gluon/contrib/simplejsonrpc.py”。下面是如何使用它调用上面服务的例子：

```
1 >>> from gluon.contrib.simplejsonrpc import
2 >>> URL = "http://127.0.0.1:8000/app/default/call/jsonrpc"
3 >>> service = ServerProxy(URL, verbose=True)
4 >>> print service.add(1, 2)
```

10.2.3 JSONRPC 和 Pyjamas

JSONRPC与XMLRPC非常相似，但使用基于JSON的协议而不是XML编码数据，作为应用的一个例子，我们讨论它与Pyjamas的使用，Pyjamas是Google Web Toolkit（最初Java编写）的Python端口，Pyjamas允许使用Python写客户应用，Pyjamas把这段代码翻译为JavaScript，web2py服务JavaScript，并通过源于客户和用户动作触发的AJAX请求来与它通信。

下面我们描述如何让Pyjamas与web2py一同工作，它不需要除了web2py与Pyjamas之外的任何额外库。

我们将构建一个简单的“todo”应用，有Pyjamas客户（全部JavaScript），只通过JSONRPC与服务器对话。

首先，创建一个叫“todo”的新应用：

其次，在“models/db.py”文件中，输入下列代码：

```
1 db=DAL('sqlite://storage.sqlite')
2 db.define_table('todo', Field('task'))
3 service = Service()
```

（注意：Service服务类来自gluon.tools）

第三步，在“controllers/default.py”中，输入下列代码：

```
1 def index():
2     redirect(URL('todoApp'))
3
4 @service.jsonrpc
5 def getTasks():
6     todos = db(db.todo).select()
7     return [(todo.task, todo.id) for todo in todos]
8
9 @service.jsonrpc
10 def addTask(taskFromJson):
11     db.todo.insert(task= taskFromJson)
12     return getTasks()
13
14 @service.jsonrpc
15 def deleteTask(idFromJson):
16     del db.todo[idFromJson]
17     return getTasks()
18
```



```
19 def call():
20 session.forget()
21 return service()
22
23 def todoApp():
24 return dict()
```

每个函数的目的是显而易见的。

第四步，在“views/default/todoApp.html”中，输入下列代码：

```
1 <html>
2 <head>
3 <meta name="pygwt:module"
4 content="{%=URL('static','output/ToDoApp')}%" />
5 <title>
6 simple todo application
7 </title>
8 </head>
9 <body bgcolor="white">
10 <h1>
11 simple todo application
12 </h1>
13 <i>
14 type a new task to insert in db,
15 click on existing task to delete it
16 </i>
17 <script language="javascript"
18 src="{%=URL('static','output/pygwt.js')}%">
19 </script>
20 </body>
21 </html>
```

这个视图在“static/output/todoapp”执行Pyjamas代码，我们还没有创建-code。

第五步，在“static/ToDoApp.py”（注意它是ToDoApp，不是todoApp！），输入下面的客户端代码：

```
1 from pyjamas.ui.RootPanel import RootPanel
2 from pyjamas.ui.Label import Label
3 from pyjamas.ui.VerticalPanel import VerticalPanel
4 from pyjamas.ui.TextBox import TextBox
5 import pyjamas.ui.KeyboardListener
6 from pyjamas.ui.ListBox import ListBox
7 from pyjamas.ui.HTML import HTML
8 from pyjamas.JSONService import JSONProxy
9
10 class ToDoApp:
11 def onModuleLoad(self):
12 self.remote = DataService()
13 panel = VerticalPanel()
14
15 self.todoTextBox = TextBox()
16 self.todoTextBox.addKeyboardListener(self)
17
18 self.todoList = ListBox()
19 self.todoList.setVisibleItemCount(7)
20 self.todoList.setWidth("200px")
```

```

21 self.todoList.addClickListener(self)
22 self.Status = Label("")
23
24 panel.add(Label("Add New Todo:"))
25 panel.add(self.todoTextBox)
26 panel.add(Label("Click to Remove:"))
27 panel.add(self.todoList)
28 panel.add(self.Status)
29 self.remote.getTasks(self)
30
31 RootPanel().add(panel)
32
33 def onKeyUp(self, sender, keyCode, modifiers):
34 pass
35
36 def onKeyDown(self, sender, keyCode, modifiers):
37 pass
38
39 def onKeyPress(self, sender, keyCode, modifiers):
40 """
41 This function handles the onKeyPress event, and will add the
42 item in the text box to the list when the user presses the
43 enter key. In the future, this method will also handle the
44 auto complete feature.
45 """
46 if keyCode == KeyboardListener.KEY_ENTER and \
47 sender == self.todoTextBox:
48 id = self.remote.addTask(sender.getText(), self)
49 sender.setText("")
50 if id<0:
51 RootPanel().add(HTML("Server Error or Invalid Response"))
52
53 def onClick(self, sender):
54 id = self.remote.deleteTask(
55 sender.getValue(sender.getSelectedIndex()), self)
56 if id<0:
57 RootPanel().add(
58 HTML("Server Error or Invalid Response"))
59
60 def onRemoteResponse(self, response, request_info):
61 self.todoList.clear()
62 for task in response:
63 self.todoList.addItem(task[0])
64 self.todoList.setValue(self.todoList.getItemCount()-1,
65 task[1])
66
67 def onRemoteError(self, code, message, request_info):
68 self.Status.setText("Server Error or Invalid Response: " \
69 + "ERROR " + code + " - " + message)
71 class DataService(JSONProxy):
72 def __init__(self):
73 JSONProxy.__init__(self, "../../default/call/jsonrpc",
74 ["getTasks", "addTask", "deleteTask"])
75
76 if __name__ == '__main__':
77 app = TodoApp()

```

第六步，在服务应用前运行Pyjamas:

```
1 cd /path/to/todo/static/
2 python /python/pyjamas-0.5p1/bin/pyjsbuild TodoApp.py
```

这将把Python 代码翻译为JavaScript，以便它能在浏览器中执行。

要访问这个应用，访问 URL:

```
1 http://127.0.0.1:8000/todo/default/todoApp
```

这节由Chris Prinos撰写，得到了Luke Kenneth Casson Leighton (Pyjamas发明人) 的帮助，由Alexei Vinidiktov更新。它已被用Pyjamas 0.5p1测试过，这个例子受参考文献 [77] Django页面的启发。

10.2.4 Amfrpc

AMFRPC是 Flash客户端与服务器通信使用的远程过程调用协议，web2py支持AMFRPC，但它要求你从源文件运行web2py，而且预装了PyAMF库，这能从Linux 或Windows shell键入下面命令安装:

```
1 easy_install pyamf
```

(请咨询PyAMF文档获得更多细节)。

本节我们假设你已经熟悉ActionScript编程。

我们将创建简单服务，采用两个数值参数，把它们相加，返回求和结果，我们会调用我们的web2py应用“pyamf_test”，并且调用服务addNumbers。

首先，使用Adobe Flash (从MX 2004开始后的任何版本) 通过开始一个新的Flash FLA文件创建Flash客户端应用，在文件一开始的框架中，添加下面的行:

```
1 import mx.remoting.Service;
2 import mx.rpc.RelayResponder;
3 import mx.rpc.FaultEvent;
4 import mx.rpc.ResultEvent;
5 import mx.remoting.PendingCall;
6
7 var val1 = 23;
8 var val2 = 86;
9
10 service = new Service(
11 "http://127.0.0.1:8000/pyamf_test/default/call/amfrpc3",
12 null, "mydomain", null, null);
13
14 var pc:PendingCall = service.addNumbers(val1, val2);
15 pc.responder = new RelayResponder(this, "onResult", "onFault");
16
17 function onResult(re:ResultEvent):Void {
18 trace("Result : " + re.result);
19 txt_result.text = re.result;
20 }
21
```

```

22 function onFault(fault:FaultEvent):Void {
23 trace("Fault: " + fault.fault.faultstring);
24 }
25
26 stop();

```

该代码允许Flash客户端连接到服务，其相应于在文件“/pyamf_test/default/gateway”中的“addNumbers”函数，你也必须导入ActionScript 版本2 MX 远程类来启用Flash的远程功能。添加路径到这些类在Adobe Flash IDE中的classpath设置，或者就放置“mx”文件夹在新建的文件旁边。

注意Service构造函数的参数，第一个参数是我们想创建的相应服务的URL，第三个参数是服务的域名。我们选择称这个域名“mydomain”。

其次，创建一个称为“txt_result”的动态文本字段，把它放在stage。

第三步，你需要设置web2py网关，能与上面定义的Flash客户端通信。

继续创建一个新的web2py应用，其叫做pyamf_test，主持新服务和给Flash客户端AMF网关，编辑“default.py”控制器并确认它包含

```

1 @service.amfrpc3('mydomain')
2 def addNumbers(val1, val2):
3 return val1 + val2
4
5 def call(): return service()

```

第四步，编译和导出/发布SWF flash客户为pyamf_test.swf，放置“pyamf_test.amf”、“pyamf_test.html”、“AC_RunActiveContent.js”和“crossdomain.xml”文件在新创建应用的“static”文件夹中，该应用主持“pyamf_test”网关。

现在可以测试客户端，通过访问：

```
1 http://127.0.0.1:8000/pyamf_test/static/pyamf_test.html
```

当客户端连接到addNumbers，网关在后台调用。

如果使用AMF0而不是AMF3，也可以使用装饰器：

```
1 @service.amfrpc
```

而不是：

```
1 @service.amfrpc3('mydomain')
```

在这种情况下，也需要修改服务的URL到：

```
1 http://127.0.0.1:8000/pyamf_test/default/call/amfrpc
```

10.2.5 SOAP

web2py包括由Mariano Reingart创建的SOAP客户端和服务端，它能非常类似XML-RPC一样被使用。

考虑下面代码，例如在“default.py”控制器：

```
1 @service.soap('MyAdd', returns={'result':int}, args={'a':int, 'b':int,})
```

```
2 def add(a,b):
3 return a+b
```

现在，在 python shell，你可以做：

```
1 >>> from gluon.contrib.pysimplesoap.client import SoapClient
2 >>> client = SoapClient(wsdl="http://localhost:8000/app/default/call/soap?WSDL")
3 >>> print client.MyAdd(a=1,b=2)
4 {'result': 3}
```

当返回文本值时，为了得到合适的编码，指定字符串u' 为合适的utf8 text' 。

你可以为服务获得WSDL，在

```
http://127.0.0.1:8000/app/default/call/soap?WSDL
```

并且能获得任何公开方法的文档：

```
1 http://127.0.0.1:8000/app/default/call/soap
```

10.3 低级别 API 和其它方法

10.3.1 simplejson

web2py包括由Bob Ippolito开发的gluon.contrib.simplejson，该模块提供大部分标准的Python-JSON 编码-解码。

Simplejson由两个函数组成：

gluon.contrib.simplejson.dumps(a) 把Python对象a编码成JSON。

gluon.contrib.simplejson.loads(b) 把JavaScript对象b解码成Python对象。

对象类型包括原始类型、lists（列表）和 dictionaries（字典），其能被序列化，复合对象除用户定义的类外，都能被序列化。

下面是例程动作（例如，控制器” default.py”） 用低层的 API 序列化 Python 包含 weekdays（星期）的 list（列表）：

```
1 def weekdays():
2 names=['Sunday','Monday','Tuesday','Wednesday',
3 'Thursday','Friday','Saturday']
4 import gluon.contrib.simplejson
5 return gluon.contrib.simplejson.dumps(names)
```

下面是HTML页的例子，发送Ajax请求到上面动作，接收JSON消息并存储list在相应JavaScript变量：

```
1 {{extend 'layout.html'}}
2 <script>
3 $.getJSON('/application/default/weekdays',
4 function(data){ alert(data); });
5 </script>
```

该代码使用了 jQuery 函数\$.getJSON，其执行 Ajax 调用，当响应时，存储 weekdays 名字到

本地 JavaScript 变量 `data` 并传递变量给回调函数。例子中，回调函数只提示访问者该数据已被接收。

10.3.2 PyRTF

另一个web站点常见的需求是生成词可读的（Word-readable）文本文档，实现这个的最简单的方法是使用丰富文档格式（RTF）的文档格式，这个格式由Microsoft发明，并成为了标准。web2py包含由Simon Cusack开发的，由Grant Edwards修定的`gluon.contrib.pyrtf`。该模块允许你编程生成RTF文档，其包括彩色格式的文本和图片。

下面的例子，我们实例化两个基本RTF类：Document 和 Section，追加后者到前者并插入一些虚拟文本在后者

```
1 def makertf():
2     import gluon.contrib.pyrtf as q
3     doc=q.Document()
4     section=q.Section()
5     doc.Sections.append(section)
6     section.append('Section Title')
7     section.append('web2py is great. '*100)
8     response.headers['Content-Type']='text/rtf'
9     return q.dumps(doc)
```

最后，Document被`q.dumps(doc)`序列化，注意在返回RTF文档之前，需要在头部指定内容类型，否则浏览器不知道如何处理文件。

依赖于配置，浏览器可能要求你要么存储这个文件要么使用文本编辑器打开它。

10.3.3 ReportLab 和 PDF

web2py 也能生成 PDF 文档，通过叫做“ReportLab”[76]的额外库。

如果从源文件运行web2py，它足以安装了ReportLab，如果运行Windows二进制发行版，需要解压ReportLab到“web2py/”文件夹；如果运行Mac二进制发行版，需要解压ReportLab到下面的文件夹：

```
1 web2py.app/Contents/Resources/
```

从现在开始，我们假定ReportLab已经安装并且web2py能找到它，我们将创建一个叫做“`get_me_a_pdf`”的简单动作来生成PDF文档。

```
1 from reportlab.platypus import *
2 from reportlab.lib.styles import getSampleStyleSheet
3 from reportlab.rl_config import defaultPageSize
4 from reportlab.lib.units import inch, mm
5 from reportlab.lib.enums import TA_LEFT, TA_RIGHT, TA_CENTER, TA_JUSTIFY
6 from reportlab.lib import colors
7 from uuid import uuid4
8 from cgi import escape
9 import os
10
11 def get_me_a_pdf():
```

```

12 title = "This The Doc Title"
13 heading = "First Paragraph"
14 text = 'bla ' * 10000
15
16 styles = getSampleStyleSheet()
17 tmpfilename=os.path.join(request.folder,'private',str(uuid4()))
18 doc = SimpleDocTemplate(tmpfilename)
19 story = []
20 story.append(Paragraph(escape(title), styles["Title"]))
21 story.append(Paragraph(escape(heading), styles["Heading2"]))
22 story.append(Paragraph(escape(text), styles["Normal"]))
23 story.append(Spacer(1, 2*inch))
24 doc.build(story)
25 data = open(tmpfilename, "rb").read()
26 os.unlink(tmpfilename)
27 response.headers['Content-Type'] = 'application/pdf'
28 return data

```

注意我们如何生成PDF到一个唯一临时文件tmpfilename，我们从该文件读取生成的PDF，然后删除该文件。

有关更多ReportLab API的信息，参考ReportLab文档，我们强烈推荐使用ReportLab的Platypus API，比如Paragraph、Spacer等。

10.4 Restful Web 服务

REST代表“REpresentational State Transfer”（表示状态转移），它是web服务结构的一种类型而不是协议，如SOAP。事实上，REST没有标准。

粗略地说，REST指出服务能被考虑为资源的集合，每个资源用URI标识，每个资源有四种动作，它们是POST（create 创建）、GET（read 读取）、PUT(update更新)和 DELETE（删除），首字母缩写为CRUD（create-read-update-delete 创建-读取-更新-删除）来代表。客户与通过产生到标识资源的URL的HTTP请求的资源通信，使用HTTP方法给资源传递POST/PUT/GET/DELETE指令，URL可能有扩展名，例如json，指定协议如何编码数据。

因此，例如POST要求

```
1 http://127.0.0.1/myapp/default/api/person
```

意味着你想创建新person，这种情况person对象可以响应表person中的记录，但也可能是其它资源类型（例如文件）。

相似地，GET请求到

```
1 http://127.0.0.1/myapp/default/api/persons.json
```

表明以json格式请求person记录的列表（从数据person表来的记录）。

GET请求到

```
1 http://127.0.0.1/myapp/default/api/person/1.json
```

表明以json格式请求关联到person/1（id==1的记录）的信息。

在web2py的这种情况，每个请求可以被分成三部分：

- 第一部分表示资源的位置，即公开服务的动作：

```
1 http://127.0.0.1/myapp/default/api/person/1.json
```

- 资源的名字（person、persons、person/1等）
- 由扩展指定的通信协议

注意我们总能使用路由器去除在URL中任何不想要的前缀，例如简化如下：

```
1 http://127.0.0.1/myapp/default/api/person/1.json
```

成为：

```
1 http://127.0.0.1/api/person/1.json
```

但这是个测试问题，我们已经在第四章详尽讨论了。

我们的例子中，我们使用叫做api的动作，但这不是必须的，事实上我们可以给那些公开RESTful服务的动作，以任何我们想取的名字命名，而且实际上我们可以不止创建一个。同理对于参数也一样，我们将继续假定我们的RESTful动作叫做api。

假定我们已经定义了下面两张表：

```
1 db.define_table('person',Field('name'),Field('info'))
2 db.define_table('pet',Field('owner',db.person),Field('name'),Field('info'))
```

并且它们是我们想公开的资源。

第一件我们要做的事是创建 RESTful 动作：

```
1 def api():
2 return locals()
```

现在，我们修改它以使扩展名过滤掉request.args（以便request.args能用来标识资源）而且它能分别处理不同方法：

```
1 @request.restful()
2 def api():
3 def GET(*args,**vars):
4 return dict()
5 def POST(*args,**vars):
6 return dict()
7 def PUT(*args,**vars):
8 return dict()
9 def DELETE(*args,**vars):
10 return dict()
11 return locals()
```

现在当我们发出 GET http 请求到

```
1 http://127.0.0.1:8000/myapp/default/api/person/1.json
```

它调用并返回GET(' person' , ' 1')，GET是定义在动作里的函数。注意：

- 我们不需要定义所有四个方法，只定义那些我们想公开的。
- 方法函数能用命名的参数。
- 扩展名存储在request.extension，内容类型自动设置。

@request.restful() 装饰器确认在路径信息里的扩展名存储在request.extension，映射请求方法为相应的函数在动作（POST、GET、PUT、DELETE）内，并传递request.args和request.vars给选择到的函数。

现在我们构建到 POST 和 GET 个人记录的服务

```
1 @request.restful()
2 def api():
3 response.view = 'generic.json'
4 def GET(tablename,id):
5 if not tablename=='person': raise HTTP(400)
```

```

6 return dict(person = db.person(id))
7 def POST(tablename,**fields):
8 if not tablename=='person': raise HTTP(400)
9 return db.person.validate_and_insert(**fields)
10 return locals()

```

注意：

- GET 和 POST 用不同的方法处理
- 函数期望正确的参数（未命名参数被request.args解析，命名的参数来自request.vars）
- 它们检查输入是否正确，否则引发异常
- GET执行查询并返回记录db.person(id)，因为通用视图调用，输出自动转换为JSON。
- POST执行validate_and_insert(..)并返回新记录id或，另外，验证错误。POST变量**fields是post变量。

10.4.1 parse_as_rest 方法(实验性的)

parse_as_rest（实验的）

迄今解释的逻辑足够创建任何RESTful类型web服务但web2py能帮助我们更多。

事实上，web2py提供语法去描述哪一个我们想公开的数据库表，及如何映射资源到URL，反之亦然。

这用URL模式实现。模式是映射URL请求参数为数据库查询的字符串，有四种类型原子模式：

- 字符串常量比如“friend”。
- 表相应的字符串常量，例如“friend[person]”匹配URL“friend”到“person”表。
- 过滤器用的变量，例如“person.id”会用到db.person.name=={person.id}过滤器。
- 字段的名称，“:field”表示的

原子模式用“/”可以组合成复杂的URL，比如

```
1 "/friend[person]/{person.id}/:field"
```

给出表单的url

```
1 http://.../friend/1/name
```

查询返回人名的person.id，“friend[person]”匹配“friend”并过滤表“person”，“{person.id}”匹配“1”并过滤“person.id==1”，“:field”匹配“name”并返回：

```
1 db(db.person.id==1).select().first().name
```

多URL模式能组合为列表以便单个RESTful动作能服务不同类型的请求。

DAL有方法parse_as_rest(pattern、args、vars)，其给出模式列表，request.args 和 request.vars匹配模式并返回响应（仅GET）。

下面是更复杂的例子：

```

1
2 @request.restful()
3 def api():
4 response.view = 'generic.'+request.extension
5 def GET(*args,**vars):
6 patterns = [
7 "/friends[person]",
8 "/friend/{person.name.startswith}",

```

```

9  "/friend/{person.name}/:field",
10 "/friend/{person.name}/pets[pet.owner]",
11 "/friend/{person.name}/pet[pet.owner]/{pet.name}",
12 "/friend/{person.name}/pet[pet.owner]/{pet.name}/:field"
13 ]
14 parser = db.parse_as_rest(patterns, args, vars)
15 if parser.status == 200:
16 return dict(content=parser.response)
17 else:
18 raise HTTP(parser.status, parser.error)
19 def POST(table_name, **vars):
20 if table_name == 'person':
21 return db.person.validate_and_insert(**vars)
22 elif table_name == 'pet':
23 return db.pet.validate_and_insert(**vars)
24 else:
25 raise HTTP(400)
26 return locals()

```

理解下列相应列表模式的URL:

- GET所有人

```
1 http://.../api/friends
```

- GET名字以“t”开始的人

```
1 http://.../api/friend/t
```

- GET第一个名字等于“Tim”的人“info”字段值

```
1 http://.../api/friend/Tim/info
```

- GET上面人(friend)的宠物列表

```
1 http://.../api/friend/Tim/pets
```

- GET名叫“Tim”的人的宠物“Snoopy”

```
1 http://.../api/friend/Tim/pet/Snoopy
```

- GET宠物“info”字段值

```
1 http://.../api/friend/Tim/pet/Snoopy/info
```

该动作公开两个POST url:

- POST 新朋友
- POST 新宠物

如果你已安装了“curl”设备，你可以试试:

```

1 $ curl -d "name=Tim" http://127.0.0.1:8000/myapp/default/api/friend.json
2 {"errors": {}, "id": 1}
3 $ curl http://127.0.0.1:8000/myapp/default/api/friends.json
4 {"content": [{"info": null, "name": "Tim", "id": 1}]}
5 $ curl -d "name=Snoopy&owner=1" http://127.0.0.1:8000/myapp/default/api/pet.json
6 {"errors": {}, "id": 1}
7 $ curl http://127.0.0.1:8000/myapp/default/api/friend/Tim/pet/Snoopy.json
8 {"content": [{"info": null, "owner": 1, "name": "Snoopy", "id": 1}]}

```

声明更复杂的查询，URL中的值被用来构建不涉及公平性的查询。例如

patterns = [' friends/{person.name.contains}' 映射

```
1 http://....../friends/
```

为

```
1 db.person.name.contains('i')
```

相似的:

```
patterns = [' friends/{person.name.ge}/{person.name.gt.not}'] 映射
1 http://.... /friends/aa/uu
```

为

```
1 (db.person.name>='aa')&('~(db.person.name>'uu'))
```

模式中字段的合法属性是: contains、startswith、le、ge、lt、gt、eq (equal、default)、ne (not equal), 其它属性特别是日期和日期时间字段的是 day、month、year、hour、minute、second。

注意该模式语法不是设计通用的, 不是每个可能的查询都能通过模式描述, 但是其中大量是可以的。语法可能在将来扩展。

常常你想公开一些RESTful URL, 但又想限制可能的查询, 这能通过传递额外参数queries给 parse_as_rest的方法实现, queries是(tablename, query)字典, query是DAL查询限制到表 tablename的访问。

也能用GET排序变量排序结果。

```
1 http://.... /api/friends?order=name|~info
```

字母排序 (name), 也可以相反的信息 (~info) 排序。

我们也可以通过指定limit 和 offset GET变量来限制记录数

```
1 http://.... /api/friends?offset=10&limit=1000
```

它返回上限1000朋友(人), 并跳过开始10条, Limit默认到1000, offset默认0。

现在考虑极端情况, 我们想给所有表(除auth_表)构建全部可能的模式, 想用用任意文本字段、任意整型字段、任意浮点字段(范围)和任意日期(也是范围)进行搜索, 也能POST任意表:

通常情况, 我们需要许多模式, web2py让它变得简单:

```
1 @request.restful()
2 def api():
3     response.view = 'generic.' + request.extension
4     def GET(*args, **vars):
5         patterns = 'auto'
6         parser = db.parse_as_rest(patterns, args, vars)
7         if parser.status == 200:
8             return dict(content=parser.response)
9         else:
10            raise HTTP(parser.status, parser.error)
11     def POST(table_name, **vars):
12         return db[table_name].validate_and_insert(**vars)
13     return locals()
```

设置patterns=' auto', 让web2py给所有非auth表生成全部可能模式, 甚至有查询有关模式的模式:

```
1 http://.... /api/patterns.json
```

其为输出person 和 pet 表结果是:

```
1 {"content": [
2     "/person[person]",
3     "/person/id/{person.id}",
4     "/person/id/{person.id}:/field",
5     "/person/id/{person.id}/pet[pet.owner]",
6     "/person/id/{person.id}/pet[pet.owner]/id/{pet.id}",
7     "/person/id/{person.id}/pet[pet.owner]/id/{pet.id}:/field",
8     "/person/id/{person.id}/pet[pet.owner]/owner/{pet.owner}",
9     "/person/id/{person.id}/pet[pet.owner]/owner/{pet.owner}:/field",
```

```

10 "/person/name/pet[pet.owner]",
11 "/person/name/pet[pet.owner]/id/{pet.id}",
12 "/person/name/pet[pet.owner]/id/{pet.id}/:field",
13 "/person/name/pet[pet.owner]/owner/{pet.owner}",
14 "/person/name/pet[pet.owner]/owner/{pet.owner}/:field",
15 "/person/info/pet[pet.owner]",
16 "/person/info/pet[pet.owner]/id/{pet.id}",
17 "/person/info/pet[pet.owner]/id/{pet.id}/:field",
18 "/person/info/pet[pet.owner]/owner/{pet.owner}",
19 "/person/info/pet[pet.owner]/owner/{pet.owner}/:field",
20 "/pet[pet]",
21 "/pet/id/{pet.id}",
22 "/pet/id/{pet.id}/:field",
23 "/pet/owner/{pet.owner}",
24 "/pet/owner/{pet.owner}/:field"
25 ]}

```

可以为一些表指定自动模式，只需：

```
1 patterns = [':auto[person]', ':auto[pet]']
```

10.4.2 smart_query 方法(实验性的)

智能查询（实验的）

有时需要更多灵活性，你想能够传递任意如下查询到 RESTful 服务

```
1 http://.../api.json?search=person.name starts with 'T' and person.name contains 'm'
```

可以用下面实现

```

1 @request.restful()
2 def api():
3     response.view = 'generic.' + request.extension
4     def GET(search):
5         try:
6             rows = db.smart_query([db.person, db.pet], search).select()
7             return dict(result=rows)
8         except RuntimeError:
9             raise HTTP(400, "Invalid search string")
10    def POST(table_name, **vars):
11        return db[table_name].validate_and_insert(**vars)
12    return locals()

```

方法 db.smart_query 采取两个参数：

- 字段或表的列表，在查询中允许的
- 字符串，包含自然语言形式的查询

它返回找到的包含记录的 db.set 对象。

注意搜索字符串被解析，不是被计算或者执行，因此它没有安全风险。

10.4.3 访问控制

访问 API 被限制，其通常用装饰器实现。因此例如：

```
1 auth.settings.allow_basic_login = True
2
3 @auth.requires_login()
4 @request.restful()
5 def api():
6 def GET(s):
7 return 'access granted, you said %s' % s
8 return locals()
```

现在可以访问，用

```
1 $ curl --user name:password http://127.0.0.1:8000/myapp/default/api/hello
2 access granted, you said hello
```

10.5 服务与认证 Services and Authentication

前面的章节，我们讨论了下面装饰器的使用：

```
1 @auth.requires_login()
2 @auth.requires_membership(...)
3 @auth.requires_permission(...)
```

对通常的动作（不装饰为服务），这些装饰器可被使用，即使输出呈现为 HTML 以外的格式。

定义为服务并用@service... 和@auth... 装饰器装饰的函数，装饰器不该使用，两种类型装饰器不能混用。如果认证不执行，正是 call 动作需要被装饰。

```
1 @auth.requires_login()
2 def call(): return service()
```

注意也可能实例化多服务对象，用它们注册一样的不同的函数，用认证公开其中一些而一些不公开。

```
1 public_services=Service()
2 private_services=Service()
3
4 @public_service.jsonrpc
5 @private_service.jsonrpc
6 def f(): return 'public'
7
8 @private_service.jsonrpc
9 def g(): return 'private'
10
11 def public_call(): return public_service()
12
13 @auth.requires_login()
14 def private_call(): return private_service()
```

这假定调用函数在 HTTP 报头传递凭据（有效绘画 cookie 或使用基本认证，如前一章讨论），客户需要支持它，但不是所有的客户都支持。

第11章 jQuery 和 Ajax

虽然web2py主要为服务器端开发，但**welcome**基本构建应用附带基本jQuery库[32]、jQuery日历（日期拾取、日期时间拾取和时钟）、“superfish.js”菜单和一些附加的基于jQuery的JavaScript函数。

web2py根本不阻碍你使用其它Ajax [78]库，比如Prototype、ExtJS或YUI，我们决定打包jQuery因为我们发现它更易于使用，而且比其它相当的库更有效，也发现它抓住了web2py有效而简洁的精髓。

11.1 web2py_ajax.html

web2py 基本构建应用 “welcome” 包含文件，叫做

```
1 views/web2py_ajax.html
```

它看起来像下面这样：

```
1  {{
2  response.files.insert(0, URL('static', 'js/jquery.js'))
3  response.files.insert(1, URL('static', 'css/calendar.css'))
4  response.files.insert(2, URL('static', 'js/calendar.js'))
5  response.include_meta()
6  response.include_files()
7  }}
8  <script type="text/javascript"><!--
9  // These variables are used by the web2py_ajax_init
10 // function in web2py.js (which is loaded below).
11 var w2p_ajax_confirm_message =
12  "{%=T('Are you sure you want to delete this object?')%}";
13 var w2p_ajax_date_format = "{%=T('%Y-%m-%d')%}";
14 var w2p_ajax_datetime_format = "{%=T('%Y-%m-%d %H:%M:%S')%}";
15 //--></script>
16 <script src="{%=URL('static', 'js/web2py.js')%}"
17 type="text/javascript"></script>
```

- 这个文件包含在默认“layout.html”的HEAD，它提供下列服务：
- 包含“static/jquery.js”。
- 包含“static/calendar.js”和“static/calendar.css”，用做弹出日历。
- 包含response.meta头。
- 包含所有response.files（要求CSS和JS，在代码中描述的）
- 设置表单变量并包含“static/js/web2py.js”

“web2py.js”实现下面：

- 定义ajax函数（基于jQuery \$.ajax）。
- 使任何“error”类DIV或任何“flash”类标签对象向下滑动。
- 防止在“integer”类INPUT区域输入无效整型。
- 防止“double”类INPUT区域输入无效浮点型。

- 用弹出日期拾取连接“date”类 INPUT 区域。
- 用弹出日期时间拾取连接“datetime”类 INPUT 区域。
- 用弹出时间拾取连接“time”类 INPUT 区域。
- 定义 web2py_ajax_component，一个非常重要的工具将在第 12 章描述。
- 定义 web2py_comet，一个能用在 HTML5 websockets 的函数（不在本书描述但可以阅读“gluon/contrib/comet_messaging.py”里的代码例子）。

为了后向兼容性，它也包含 popup、collapse 和 fade 函数。
下面例子展示了其它效果如何一起表现更好。

考虑下面模型的一个 test 应用：

```
1 db = DAL("sqlite://db.db")
2 db.define_table('child',
3     Field('name'),
4     Field('weight', 'double'),
5     Field('birth_date', 'date'),
6     Field('time_of_birth', 'time'))
7
8 db.child.name.requires=IS_NOT_EMPTY()
9 db.child.weight.requires=IS_FLOAT_IN_RANGE(0,100)
10 db.child.birth_date.requires=IS_DATE()
11 db.child.time_of_birth.requires=IS_TIME()
```

有“default.py”这个控制器：

```
1 def index():
2     form = SQLFORM(db.child)
3     if form.process().accepted:
4         response.flash = 'record inserted'
5     return dict(form=form)
```

以及下面“default/index.html”这个视图：

```
1 {{extend 'layout.html'}}
2 {{=form}}
```

“index”动作生成下面的表单：

Name:	<input type="text"/>
Weigth:	<input type="text"/>
Birth Date:	<input type="text"/>
Time Of Birth:	<input type="text"/>
<input type="submit" value="Submit"/>	

如果无效表单被提交，服务器返回修改包含错误消息的页面，错误消息是“error”类的 DIV，因为上述 web2y.js 代码，错误呈现向下滑动的效果：

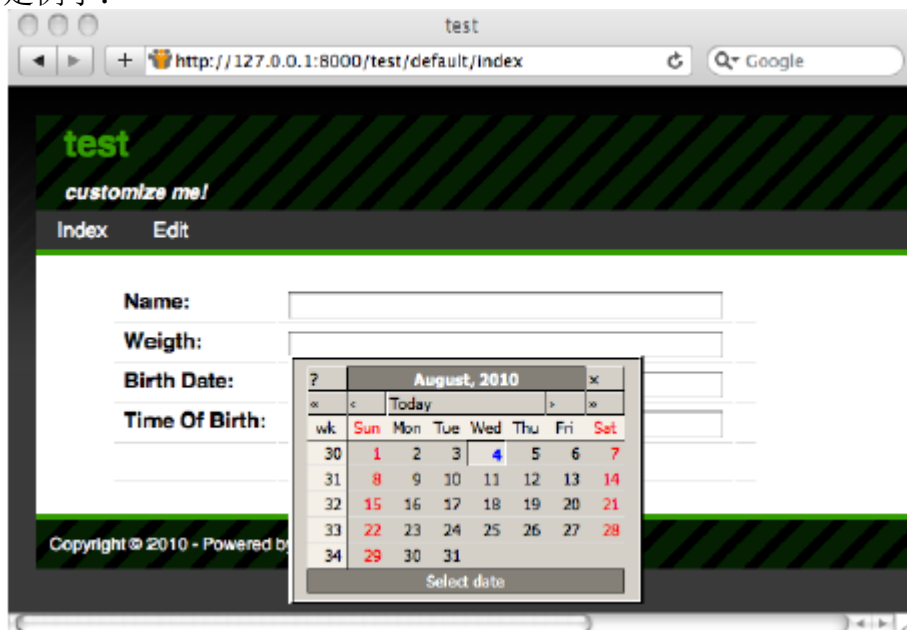
Name:	<input type="text"/>
	enter a value
Weigth:	<input type="text"/>
	enter a number between 0.0 and 100.0
Birth Date:	<input type="text"/>
	enter date and time as 1963-08-28 14:30:59
Time Of Birth:	<input type="text"/>
	enter time as hh:mm:ss (seconds, am, pm optional)
	<input type="button" value="Submit"/>

错误消息的颜色在“layout.html” CSS代码给定。

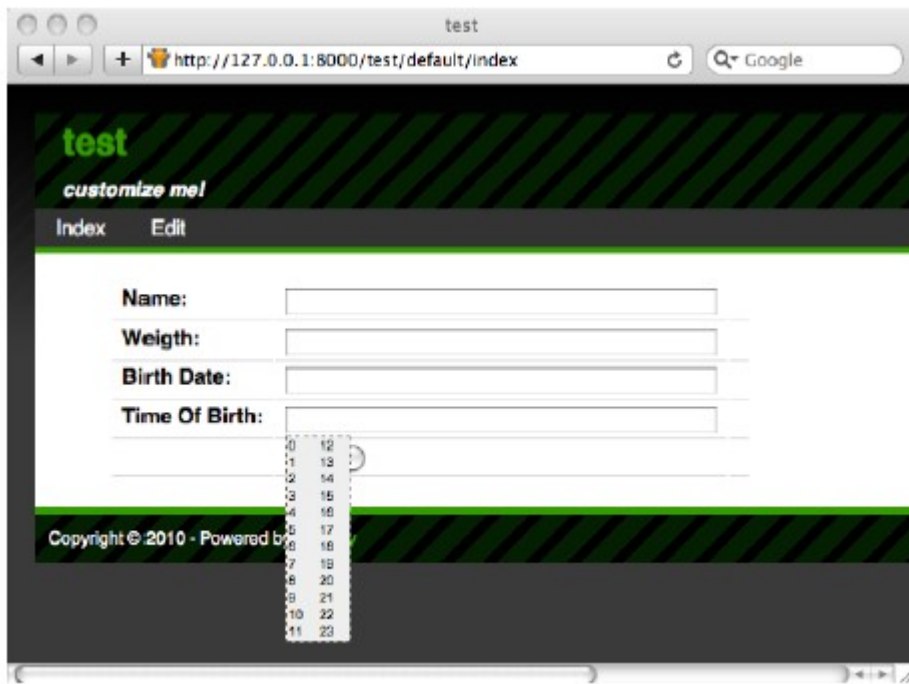
web2y.js代码防止你在输入区域输入非法值，这在之前完成而且不是一个替代，服务器端验证。

当在“date”类INPUT字段输入时，web2y.js代码显示日期拾取，当在“datetime”类INPUT字段输入时，它显示日期时间拾取。

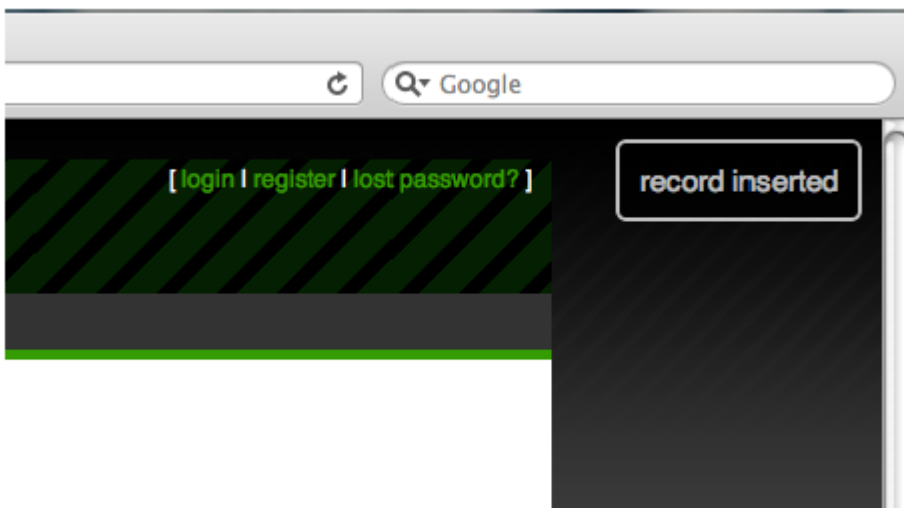
下面是例子：



当编辑“time”类 INPUT 字段输入，web2y.js 代码也显示下面的时间拾取。



一旦提交，控制器动作设置相应flash到消息“record inserted”，默认布局在一个DIV中用id=“flash”呈现该消息，web2y.js代码负责使该DIV出现并且当你点击它时让它消失。



通过控制器帮助对象，这些以及其它效果在视图中编程访问。

11.2 jQuery 效果

这里描述的基本效果无需额外的文件，所有需要的已经包含在web2py_ajax.html文件。HTML/XHTML对象能用它们的类型（例如DIV对象）、它们的类或它们的id标识。例如：

```
1 <div class="one" id="a">Hello</div>
2 <div class="two" id="b">World</div>
```

它们分别属于类“one”和“two”，它们有分别等于“a”和“b”的id。

jQuery里，你可以用下列CSS-like等效的注释应用前者

```
1 jQuery('.one') // address object by class "one"
2 jQuery('#a') // address object by id "a"
3 jQuery('DIV.one') // address by object of type "DIV" with class "one"
4 jQuery('DIV #a') // address by object of type "DIV" with id "a"
```

对于后者用

```
1 jQuery('.two')
2 jQuery('#b')
3 jQuery('DIV.two')
4 jQuery('DIV #b')
```

或引用它们两个

```
1 jQuery('DIV')
```

标签对象关联到事件，比如“onclick”，jQuery允许链接这些事件到效果，例如“slideToggle”：

```
1 <div class="one" id="a" onclick="jQuery('.two').slideToggle()">Hello</div>
2 <div class="two" id="b">World</div>
```

现在如果点击“Hello”，“World”消失，如果再次点击，“World”重新出现，通过给它隐藏类，可以使标签默认隐藏：

```
1 <div class="one" id="a" onclick="jQuery('.two').slideToggle()">Hello</div>
2 <div class="two hidden" id="b">World</div>
```

也可以链接动作到标签本身外事件，前面的代码可重写如下：

```
1 <div class="one" id="a">Hello</div>
2 <div class="two" id="b">World</div>
3 <script>
4 jQuery('.one').click(function() {jQuery('.two').slideToggle()});
5 </script>
```

效果返回调用对象，因此它们可以被链接。

此时click设置回调函数为点击时调用，同样类似于change、keyup、keydown、mouseover等。在整个文档载入之后，通常的情况需要执行一些jQuery代码，经常通过BODY onload属性实现，但是jQuery提供另外的方法无需编辑布局：

```
1 <div class="one" id="a">Hello</div>
2 <div class="two" id="b">World</div>
3 <script>
4 jQuery(document).ready(function() {
5 jQuery('.one').click(function() {jQuery('.two').slideToggle()});
6 });
7 </script>
```

未命名函数部分只有当文档准备好且已经被载入后才执行。

下面是一些有用事件的名字列表：

Form events 表单事件

- onchange：当元素改变时脚本运行
- onsubmit：当表单提交时脚本运行
- onreset：当表单重置时脚本运行
- onselect：当元素被选中时脚本运行
- onblur：当元素失去焦点时脚本运行
- onfocus：当元素得到焦点时脚本运行

Keyboard events 键盘事件

- onkeydown：当键按下时脚本运行
- onkeypress：当键按下并释放时脚本运行

- onkeyup: 当键释放时脚本运行

Mouse events 鼠标事件

- onclick: 当鼠标单击时脚本运行
- ondblclick: 当鼠标双击时脚本运行
- onmousedown: 当按下鼠标按键时脚本运行
- onmousemove: 当鼠标指针移动时脚本运行
- onmouseout: 当鼠标指针移出元素时脚本运行
- onmouseover: 当鼠标指针移到元素时脚本运行
- onmouseup: 当鼠标按键释放时脚本运行

下面是jQuery定义的有用效果的列表:

Effects 效果

- jQuery(...).attr(name): 返回属性值的名字
- jQuery(...).attr(name, value): 设置属性名为值
- jQuery(...).show(): 使对象可见
- jQuery(...).hide(): 使对象隐藏
- jQuery(...).slideToggle(speed, callback): 使对象上滑或下滑
- jQuery(...).slideUp(speed, callback): 使对象上滑
- jQuery(...).slideDown(speed, callback): 使对象下滑
- jQuery(...).fadeIn(speed, callback): 使对象渐入
- jQuery(...).fadeOut(speed, callback): 使对象淡出

speed参数通常是“slow”、“fast”或忽略(默认), callback回调函数是可选函数, 当效果完成时调用, jQuery效果可以容易地嵌入到帮助对象, 例如在视图:

```
1 {=DIV(' click me!', _onclick="jQuery(this).fadeOut()")}
```

jQuery是非常紧凑和简介的Ajax库, 因此web2py无需jQuery之上额外的抽象层(除了下面讨论的ajax函数), 当需要时, jQuery API能够以本来的表单形式访问和准备好访问。

咨询文档以获得更多有关这些效果和其它jQuery API的信息。

jQuery库也能用插件和用户接口Widget (User Interface Widget) 扩展, 这个主题不在本书中讨论, 参看[80]了解更多。

11.2.1 表单的条件域

jQuery 效果典型应用是表单, 其改变基于它的区域值的形式。

在web2py中容易实现因为SQLFORM帮助对象生成“CSS friendly”表单, 表单包含行的表, 每行含标签、输入区域和可选第三列, 项目有从表名和字段名严格继承的id。

规定是每个INPUT字段有id tablename_fieldname, 用id tablename_fieldname__row包含在row(行)中。

作为例子, 创建输入表单要求纳税人的名字和纳税人配偶的名字, 只有当他/她结婚。

创建有下列模型的 test 应用:

```
1 db = DAL('sqlite://db.db')
```

```
2 db.define_table('taxpayer',
3 Field('name'),
4 Field('married', 'boolean'),
5 Field('spouse_name'))
```

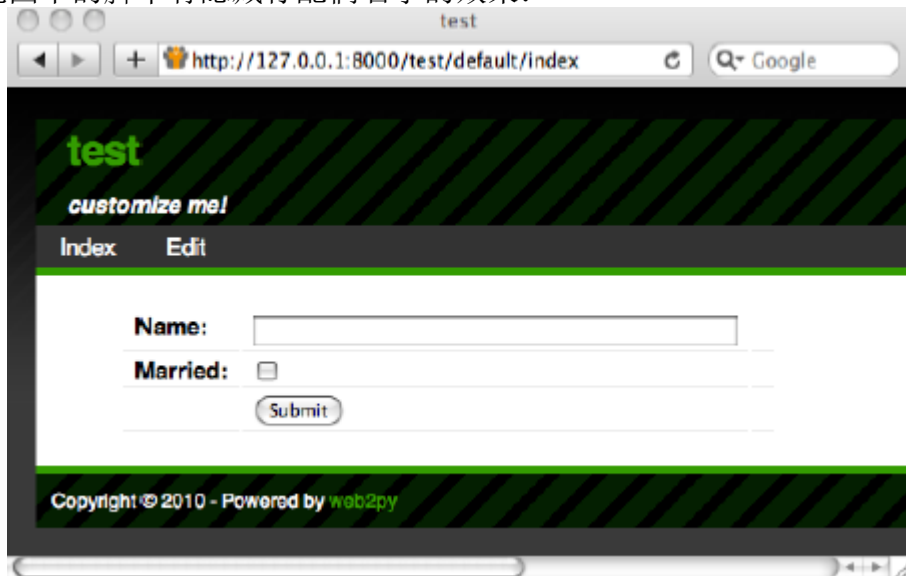
下述“default.py”控制器：

```
1 def index():
2 form = SQLFORM(db.taxpayer)
3 if form.process().accepted:
4 response.flash = 'record inserted'
5 return dict(form=form)
```

下述“default/index.html”视图：

```
1 {{extend 'layout.html'}}
2 {{=form}}
3 <script>
4 jQuery(document).ready(function() {
5 jQuery('#taxpayer_spouse_name__row').hide();
6 jQuery('#taxpayer_married').change(function() {
7 if(jQuery('#taxpayer_married').attr('checked'))
8 jQuery('#taxpayer_spouse_name__row').show();
9 else jQuery('#taxpayer_spouse_name__row').hide();});
10 });
11 </script>
```

视图中的脚本有隐藏行配偶名字的效果：



当纳税人检查“married”复选框，配偶的名字字段重新显示：



此处“taxpayer_married”是复选框，关联到表“taxpayer”的字段“married”其类型“boolean”，“taxpayer_spouse_name__row”是给表“taxpayer”的“spouse_name”输入字段的行。

11.2.2 删除确认

另外一个有用的应用是，当勾选“delete”复选框比如出现在编辑表单的删除复选框时，要求确认。

考虑上面的例子并添加下面的控制器动作：

```
1 def edit():
2     row = db.taxpayer[request.args(0)]
3     form = SQLFORM(db.taxpayer, row, deletable=True)
4     if form.process().accepted:
5         response.flash = 'record updated'
6     return dict(form=form)
```

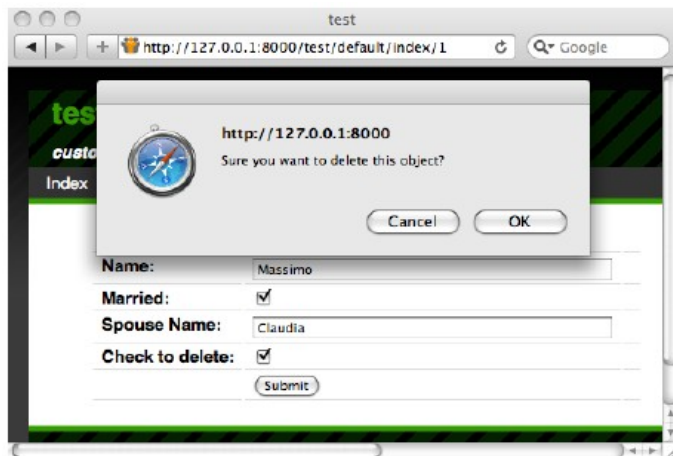
相应的视图“default/edit.html”

```
1 {{extend 'layout.html'}}
2 {{=form}}
```

在SQLFORM构造函数中的参数deletable=True指示web2py在编辑表单显示“delete”复选框，默认它是False，web2py的“web2py.js”包含下列代码：

```
1 jQuery(document).ready(function() {
2     jQuery('input.delete').attr('onclick',
3     'if(this.checked) if(!confirm(
4     "{{T('Sure you want to delete this object?')}}")
5     this.checked=false;');
6 });
```

按照规定，该复选框有等于“delete”的类，上面的jQuery代码带确认对话框（JavaScript标准的）链接该复选框的onclick事件，如果纳税人不勾选就不检查复选框。



11.3 ajax 函数

在web2py.js里，web2py定义一个叫ajax的函数，其基于jQuery函数\$.ajax，但其不要与jQuery函数\$.ajax混淆，后者比前者更强大，使用方法，我们推荐你看参考文献[32]和[79]。但是，前一个函数对许多复杂任务足够并且更容易使用。

ajax函数是JavaScript函数，有下面的语法：

```
1 ajax(url, [name1, name2, ...], target)
```

它异步调用url（第一个参数），传递名字与列表中名字（第二个参数）相等的输入字段值，然后存储id等于目标（第三个参数）的标签的内HTML的响应。

下面是default控制器的例子：

```
1 def one():
2     return dict()
3
4 def echo():
5     return request.vars.name
```

关联的“default/one.html”视图：

```
1 {{extend 'layout.html'}}
2 <form>
3 <input name="name" onkeyup="ajax('echo', ['name'], 'target')" />
4 </form>
5 <div id="target"></div>
```

当在INPUT字段输入内容时，只要释放按键（onkeyup），ajax函数被调用，name="name"字段的值传递给动作“echo”，它送回文本给视图，ajax函数收到响应并在“target”DIV显示回声响应。

11.3.1 Eval target

ajax函数的第三个参数是字符串“:eval”，这意味着被服务器返回的字符串不会嵌在文档中，而是被评估。

下面是一个default控制器例子：

```
1 def one():
```

```

2 return dict()
3
4 def echo():
5 return "jQuery('#target').html(%s);" % repr(request.vars.name)

```

关联的“default/one.html”视图：

```

1 {{extend 'layout.html'}}
2 <form>
3 <input name="name" onkeyup="ajax('echo', ['name'], ':eval')" />
4 </form>
5 <div id="target"></div>

```

这允许更多复杂的响应来更新多目标。

11.3.2 自动完成 Auto-completion

web2py 包含内建自动完成小工具（widget），在 Forms 章描述过，这里我们将从零开始创建一个简单点的。

上面的 ajax 函数的另外一个应用是自动完成，这里我们希望创建输入字段期望月名（month name），当访问者敲入不完整的名字，通过 Ajax 请求执行自动完成，响应时，自动完成的下拉框出现在输入字段下方。

这能够通过下面的 default 控制器实现：

```

1 def month_input():
2 return dict()
3
4 def month_selector():
5 if not request.vars.month: return ''
6 months = ['January', 'February', 'March', 'April', 'May',
7 'June', 'July', 'August', 'September', 'October',
8 'November', 'December']
9 month_start = request.vars.month.capitalize()
10 selected = [m for m in months if m.startswith(month_start)]
11 return DIV(*[DIV(k,
12 _onclick="jQuery('#month').val('%s')" % k,
13 _onmouseover="this.style.backgroundColor='yellow'",
14 _onmouseout="this.style.backgroundColor='white'"
15 ) for k in selected])

```

相应的“default/month_input.html”视图：

```

1 {{extend 'layout.html'}}
2 <style>
3 #suggestions { position: relative; }
4 .suggestions { background: white; border: solid 1px #55A6C8; }
5 .suggestions DIV { padding: 2px 4px 2px 4px; }
6 </style>
7
8 <form>
9 <input type="text" id="month" name="month" style="width: 250px" /><br />
10 <div style="position: absolute;" id="suggestions"
11 class="suggestions"></div>
12 </form>
13 <script>
14 jQuery("#month").keyup(function() {
15 ajax('month_selector', ['month'], 'suggestions');
16 </script>

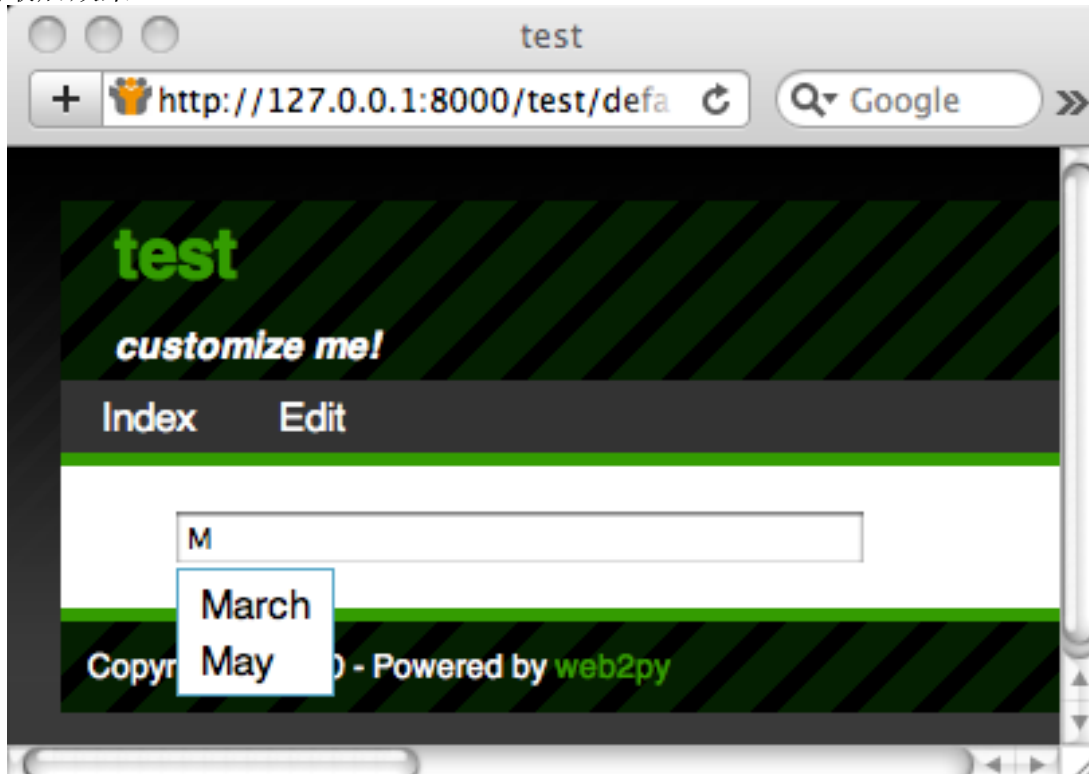
```

每次访问者在“month”输入框敲入内容，视图中的 jQuery 脚本发出 Ajax 请求，输入框的值用 Ajax 请求提交给“month_selector”动作，该动作找到以提交文本开头（选到的）月

份名的列表，创建DIV列表（每个包含推荐的月份名），并返回带序列化DIV的字符串。视图在“suggestions” DIV显示响应HTML，“month_selector” 动作生成建议和嵌在DIV的JavaScript代码，当访问者点击每个建议时必须执行。例如，当访问者敲入” M” 回调动作返回：

```
1 <div>
2 <div onclick="jQuery('#month').val(' March')"
3 onmouseout="this.style.backgroundColor=' white' "
4 onmouseover="this.style.backgroundColor=' yellow' ">March</div>
5 <div onclick="jQuery('#month').val(' May')"
6 onmouseout="this.style.backgroundColor=' white' "
7 onmouseover="this.style.backgroundColor=' yellow' ">May</div>
8 </div>
```

下图是最后效果：



如果月份存储在数据库表中，比如：

```
1 db.define_table('month', Field('name'))
```

那么只要用下面的替换month_selector动作：

```
1 def month_input():
2     return dict()
3
4 def month_selector():
5     if not request.vars.month:
6         return ''
7     pattern = request.vars.month.capitalize() + '%'
8     selected = [row.name for row in db(db.month.name.like(pattern)).select()]
9     return ''.join([DIV(k,
10 _onclick="jQuery('#month').val('%s')"% k,
11 _onmouseover="this.style.backgroundColor=' yellow'",
12 _onmouseout="this.style.backgroundColor=' white'"
13 ).xml() for k in selected])
```

jQuery提供可选自动完成插件具有更多的功能，但这里不讨论。

11.3.3 Ajax 表单提交

这里我们考虑页面允许访问者用 Ajax 提交消息又不重载整个页面，使用 LOAD 帮助对象，web2py 提供比这里描述的更好的机制来实现，会在第 12 章描述。此处，我们想展示给你如何简单地用 jQuery 实现。

它包含表单 “myform” 和 “target” DIV，当表单提交时，服务器可能接受它（并执行数据库插入）或拒绝它（因为它没有通过验证），相应的通知用 Ajax 响应返回并显示在 “target” DIV。

创建有下面的模型的 test 应用：

```
1 db = DAL('sqlite://db.db')
2 db.define_table('post', Field('your_message', 'text'))
3 db.post.your_message.requires = IS_NOT_EMPTY()
```

注意每个 post 有单个字段 “your_message” 要求为非空。

编辑 default.py 控制器并写两个动作：

```
1 def index():
2     return dict()
3
4 def new_post():
5     form = SQLFORM(db.post)
6     if form.accepts(request, formname=None):
7         return DIV("Message posted")
8     elif form.errors:
9         return TABLE(*[TR(k, v) for k, v in form.errors.items()])
```

第一个动作仅返回一个视图。

第二个动作是 Ajax 回调，它期望 request.vars 里的表单变量处理它们，成功则返回 DIV (“Message posted”) 或失败返回错误消息 TABLE。

现在编辑 “default/index.html” 视图：

```
1 {{extend 'layout.html'}}
2
3 <div id="target"></div>
4
5 <form id="myform">
6     <input name="your_message" id="your_message" />
7     <input type="submit" />
8 </form>
9
10 <script>
11     jQuery('#myform').submit(function() {
12         ajax('{{URL('new_post')}}',
13             ['your_message'], 'target');
14         return false;
15     });
16 </script>
```

注意在这个例子中，表单是如何用 HTML 手动创建，但它被 SQLFORM 以不同于显式表单的动作处理，SQLFORM 对象从不序列化为 HTML，这个情况的 SQLFORM.accepts 不用回话，并设置 formname=None，因为我们选择不设置表单名字而且表单键是手工 HTML 形式。

视图底部的脚本连接 “myform” 提交按钮到内联函数，该函数用 web2py ajax 函数提交 id=“your_message” 的 INPUT，在 id= “target” 的 DIV 内显示回答。

11.3.4 投票和打分 Voting and rating

另一个Ajax应用是页面内的投票和打分项，这里我们考虑一个应用，其允许访问者对粘贴的图像投票，应用由单个页面组成，根据他们的票排列显示图像，我们将允许访问者多次投票，但是如果用户认证很容易修改这个举动，通过跟踪个人投票数据库，并且用投票者的request.env.remote_addr关联它们。

下面是例子模型：

```
1 db = DAL('sqlite://images.db')
2 db.define_table('item',
3   Field('image', 'upload'),
4   Field('votes', 'integer', default=0))
```

下面是 default 控制器：

```
1 def list_items():
2   items = db().select(db.item.ALL, orderby=db.item.votes)
3   return dict(items=items)
4
5 def download():
6   return response.download(request, db)
7
8 def vote():
9   item = db.item[request.vars.id]
10  new_votes = item.votes + 1
11  item.update_record(votes=new_votes)
12  return str(new_votes)
```

下载动作需要允许list_items视图下载存储在“uploads”文件夹的图像，投票动作被用于Ajax回调。

下面是“default/list_items.html”视图：

```
1 {{extend 'layout.html'}}
2
3 <form><input type="hidden" id="id" name="id" value="" /></form>
4 {{for item in items:}}
5 <p>
6   
8   <br />
9   Votes=<span id="item{{=item.id}}">{{=item.votes}}</span>
10   [<span onclick="jQuery('#id').val(' {{=item.id}} ');
11   ajax('vote', [ 'id' ], 'item{{=item.id}} ');>vote up</span>]
12 </p>
13 {{pass}}
```

当访问者点击“[vote up]” JavaScript代码存储item.id在隐藏“id” INPUT字段，并且通过Ajax请求提交该值给服务器，服务器给相应记录增加投票计数并以字符串返回新的票数，这个值于是被插入到目标item{{=item.id}} SPAN中。

Ajax回调能用在后台执行计算，但我们推荐使用CRON或后台进程（第4章讨论过），因为web服务器执行线程超时措施，如果计算耗时太长，web服务器终止它，参考你的web服务器参数来设置超时值。

第 12 章 组件和插件 Components and plugins

组件和插件是 web2py 相对较新的功能，在开发者内部，关于它们是什么和它们应该是什么存在一些不同意见，大部分的困惑源于在其它软件项目对这些术语不同的使用，以及开发者继续完善技术要求的事实。

但是，插件支持是重要特性而且我们需要提供一些定义，这些定义不意味着是最终的，而是与编程模式保持一致，我们将在本章讨论。

- 我们如何能构建模块应用最小化服务器负荷，并最大化代码重用？
- 我们如何能发布代码为或多或少即插即用的风格？

Components 解决第一个问题，plugins 解决第二个问题。

12.1 组件 Components

component 是 web 页面功能自治的部分。

组件可以由模块、控制器和视图组成，没有严格的要求，除了嵌在 web 页面，它必须局部化在 html 标签（例如 DIV、SPAN 或 IFRAME）范围内，并且必须独立于页面其它部分执行任务，我们尤其关注加载在页面内并且通过 Ajax 和组件控制器通信的页面。

组件的一个例子是“comments component”，包含在 DIV 中，并显示用户的评论以及粘贴新评论表单，当表单提交，它通过 Ajax 发送到服务器，列表更新并且评论存储在服务器的数据库，DIV 内容无需重载页面的其余部分就得到刷新。

web2py LOAD 函数使它易于实现，且不需要太多 JavaScript/Ajax 知识和编程。

我们的目标是能通过装配组件到页面布局开发 web 应用。

考虑简单 web2py 应用“test”，用“models/db_comments.py”中自定义的模型扩展默认基本构建应用。

```
1 db.define_table('comment',
2     Field('body', 'text', label='Your comment'),
3     Field('posted_on', 'datetime', default=request.now),
4     Field('posted_by', db.auth_user, default=auth.user_id))
5 db.comment.posted_on.writable=db.comment.posted_on.readable=False
6 db.comment.posted_by.writable=db.comment.posted_by.readable=False
```

在“controllers/comments.py”中的动作

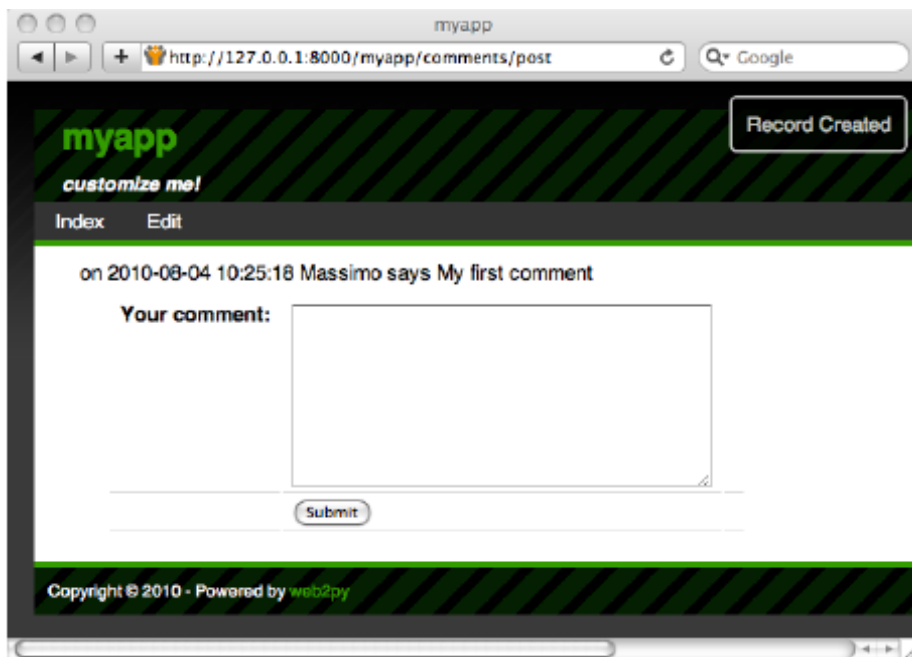
```
1 @auth.requires_login()
2 def post():
3     return dict(form=crud.create(db.comment),
4                 comments=db(db.comment).select())
```

相应的视图“views/comments/post.html”

```
1 {{extend 'layout.html'}}
2 {{for comment in comments:}}
3 <div class="comment">
4     on {{=comment.posted_on}} {{=comment.posted_by.first_name}}
5     says <span class="comment_body">{{=comment.body}}</span>
6 </div>
7 {{pass}}
8 {{=form}}
```

可以像以往一样访问，在：

```
1 http://127.0.0.1:8000/test/comments/post
```



到目前，该动作没有什么特殊的，但通过定义带扩展名“.load”的新视图，我们可以把它转换为一个不扩展布局的组件。

因此，我们创建“views/comments/post.load”：

```
1 {{#extend 'layout.html' <- notice this is commented out!}}
2 {{for comment in comments:}}
3 <div class="comment">
4   on {{=comment.posted_on}} {{=comment.posted_by.first_name}}
5   says <span class="comment_body">{{=comment.body}}</span>
6 </div>
7 {{pass}}
8 {{=form}}
```

我们可以访问它，在URL：

```
1 http://127.0.0.1:8000/test/comments/post.load
```

它看起来会像下面这样：



我们可以嵌入该组件到另外任何其他页面，只需通过：

```
1 {{=LOAD('comments', 'post.load', ajax=True)}}
```

例如在“controllers/default.py”，可以编辑

```
1 def index():
2     return dict()
```

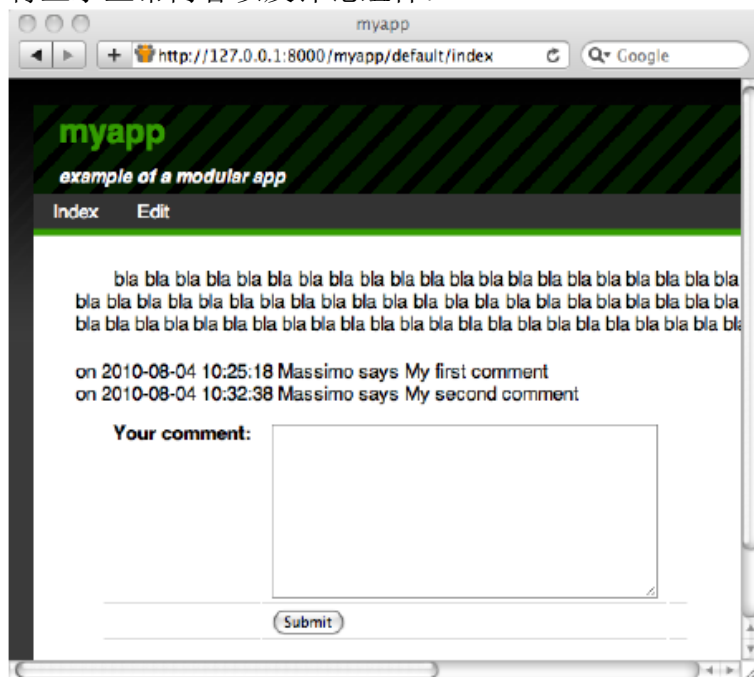
并且在相应的视图添加组件：

```
1 {{extend 'layout.html'}}
2 <p>{{='bla '*100}}</p>
3 {{=LOAD('comments', 'post.load', ajax=True)}}
```

访问该页面

```
1 http://127.0.0.1:8000/test/default/index
```

将显示正常内容以及评论组件：



{{=LOAD(...)}} 组件呈现如下：

```
1 <script type="text/javascript"><!--
2 web2py_component("/test/comment/post.load", "c282718984176")
3 //--></script><div id="c282718984176">loading...</div>
```

（实际生产的代码取决于传递给LOAD函数的选项）。

web2py_component(url, id) 函数定义在 “web2py_ajax.html”，它执行所有的魔法：它通过Ajax调用url并依据相应的id把响应嵌到DIV，它陷阱每个表单提交进入到DIV，并通过Ajax提交那些表单，Ajax目标总是DIV自己。

完整的LOAD帮助对象的代码如下：

```
1 LOAD(c=None, f='index', args=[], vars={},
2      extension=None, target=None,
3      ajax=False, ajax_trap=False,
4      url=None, user_signature=False,
5      content='loading...', **attr):
```

此处：

- 头两个参数c和f分别是控制器和我们想调用的函数。
- args 和 vars是我们想传递给函数的参数和变量。前者是列表，后者是字典。
- extension 是可选的扩展。注意扩展名也能像在函数f='index.load' 部分那样传递。
- target 是目标DIV的 id。如果没有指定，生成随机的目标id。
- 如果DIV没有通过Ajax填充，ajax应被设置为True，如果DIV在当前页面返回以前已被填充（如此避免Ajax调用），设置为False。
- ajax_trap=True意味着DIV任何表单提交必须通过Ajax捕获和提交，响应必须呈现在DIV里，ajax_trap=False表明表单必须正常提交，如此重载整个页面，如果ajax=True，ajax_trap被忽略。

略并假定为True。

- url, 如果指定, 则覆盖c、f、args、vars和extension的值, 并在url载入组建, 它被用于像其它应用服务的组建页面那样载入 (可能或也可不是用web2py创建的)。
- user_signature默认为False, 但如果你登录了, 它该被设置为True, 这将确保ajax回调是数字签名的, 它可能是帮助对象, 如在content=IMG(..)。
- 可选**att (属性) 能被传递给contained DIV。

如果没有指定.load视图, generic.load呈现那些通过没有布局动作返回的字典, 如果字典包含单个项, 它工作得更好。

如果你LOAD一个有.load扩展名的组件, 并且这个相应的控制器函数重定向到另外一个动作 (例如, 登录表单), .load 扩展名传播且新的url (也是重定向到的) 也用.load扩展名载入。

*请注意: *因为Ajax post不支持多部分表单, 即文件上传、上传字段用LOAD组件不工作, 你可能觉得被愚弄, 认为它能用, 因为如果POST从单个组件.load视图完成, 上传字段会正常工作。然而, 上传要用ajax兼容第三方widget并且web2py手动上传存储命令。

12.1.1 客户-服务器组件通信

当通过 Ajax 调用组件动作, web2py 用请求传递两个 HTTP 报头:

```
1 web2py-component-location
2 web2py-component-element
```

通过变量被动作访问:

```
1 request.env.http_web2py_component_location
2 request.env.http_web2py_component_element
```

后者也能通过下面访问:

```
1 request.cid
```

前者包含调用组件动作页面 URL, 后者包含 DIV 的 id 且该 DIV 包含响应。

组件动作也存储数据到两个特殊的 HTTP 响应报头, 一旦响应, 就会被整个页面解释。它们是:

```
1 web2py-component-flash
2 web2py-component-command
```

它们可被设置

```
1 response.headers['web2py-component-flash']='...'
2 response.headers['web2py-component-command']='...'
```

或 (如果动作被组件调用) 自动通过:

```
1 response.flash='...'
2 response.js='...'
```

前者包含关于响应你想flash显示的文本。后者包含关于响应你想要执行的JavaScript代码, 它不包含新行。

作为例子, 让我们定义一个在 “controllers/contact/ask.py” 中的联系表单组件, 该组件允许用户提问题, 组件用电子邮件发送问题给系统管理员, flash “thank you” 消息, 从页面移出组件:

```
1 def ask():
2     form=SQLFORM.factory(
3         Field('your_email',requires=IS_EMAIL()),
4         Field('question',requires=IS_NOT_EMPTY()))
5     if form.process().accepted:
6         if mail.send(to='admin@example.com',
7                     subject='from %s' % form.vars.your_email,
```

```

8         message = form.vars.question):
9         response.flash = 'Thank you'
10        response.js = "jQuery('#%s').hide()" % request.cid
11    else:
12        form.errors.your_email = "Unable to send the email"
13    return dict(form=form)

```

头四行定义表单并接受它，用来发送的 mail 对象定义在默认的基本构建应用中；最后四行通过从 HTTP 请求报头得到数据以及设置 HTTP 响应报头实现全部组件指定的逻辑。

现在可以嵌入这个联系表单到任意页面，通过

```

1    {{=LOAD('contact','ask.load',ajax=True)}}

```

注意我们不给我们的 ask 组件定义 .load 视图，也不需要因为它返回单个对象（表单），因此“generic.load”就非常好了，注意通用视图是开发工具。在生产中，你要复制“views/generic.load”到“views/contact/ask.load”。

我们能阻止访问通过 Ajax 调用的函数，用 user_signature 参数数字签名 URL：

```

1    {{=LOAD('contact','ask.load',ajax=True,user_signature=True)}}

```

它添加了数字签名到 URL，那么数字签名在回调函数中使用装饰器验证。

```

1    @auth.requires_signature()
2    def ask(): ...

```

12.1.2 捕获 Ajax 链接

通常链接不被捕获，点击组件内的链接，整个链接页面被载入。有时，你想链接的页面载入到组件内，这能用 A 帮助对象实现：

```

1    {{=A('linked page',_href='http://example.com',cid=request.cid)}}

```

如果 cid 被指定，链接的页面通过 Ajax 载入，cid 是 html 元素的 id，放置被载入页面的内容。这个例子，我们设置它为 request.cid，即组件的 id 生成链接，链接的页面可能是并通常是使用 URL 命令生成的内部 URL。

12.2 插件 Plugins

plugin 是应用文件的任意子集。

而且我们认为是任意：

- 插件不是模块，不是模型，它不是控制器不是视图，但它可能包含模块、模型、控制器和/或视图。
- 插件不需要为功能性自治，它可能依赖其它插件或特定用户代码。
- plugin 不是 plugin system，因此没有注册概念也没有隔离，但是我们制定一些规则来尝试实现一些隔离。
- 我们讨论你的应用插件，而不是 web2py 的插件。

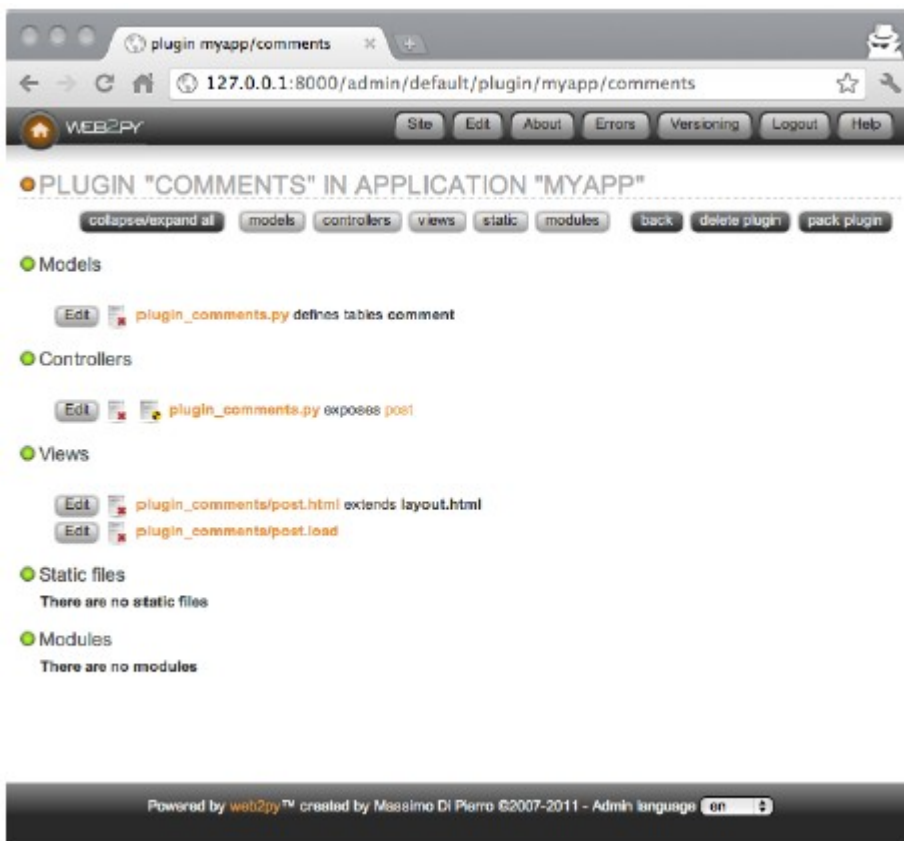
因此，为什么把它叫做 plugin？因为它提供一种机制打包应用子集并给另外应用拆包（即 plugin），在这个定义下，你应用中的任何文件可以视为插件。

当应用程序发布时，它的插件被打包并和其一同发布。

实践中，admin 提供界面从你的应用分开地打包和拆包插件，你应用的文件和文件夹，有前缀 plugin_name 名字的能被一起打包到一个文件，叫做：
web2py.plugin.name.w2p 并一起发布。



web2py 处理组成插件的文件与其它文件没有什么区别，除了 admin 从它们的名字理解它们意味着要一同发布外，admin 在分开的页面中显示它们：



作为实际问题，通过上面的定义，这些插件比哪些被认识的，比如被 admin 更常见。

在实践中我们仅关心两类插件：

- Component Plugins，这些是包含前面章所定义的组件。组件插件可以包含一个或更多组件，我们可以思考例子 plugin_comments，包含上面建议的 comments 组件。另外一个例子是 plugin_tagging，包含 tagging 组件和 tag-cloud 组件，共享一些被插件定义的数据库表。
- Layout Plugins，这些是包含布局视图及该视图所需静态文件的插件，当插件应用时，它给应用新面貌和感受。

根据上面定义，之前章节生成的组件，例如 “controllers/contact.py” 也是插件，我们能移动它们从一个应用到另一个，以及使用它们定义的组件。但是，它们不被 admin 视为如此，因为没有把它们标

识为插件的东西，因此我们有两个问题需要解决：

- 用规则命名插件文件，以便 admin 能识别它们属于同样的插件。
- 如果插件有模型文件，建立规则以便它定义的对象不污染命名空间而且不相互冲突。

让我们假定插件叫做 name，下面是应该遵守的规则：

Rule 1: 插件模型和控制器应该分别命名为

- models/plugin_name.py
- controllers/plugin_name.py

插件视图、模块、静态和私有文件应该分别在文件夹内被调用：

- views/plugin_name/
- modules/plugin_name/
- static/plugin_name/
- private/plugin_name/

Rule 2: 插件模型仅能用下列开头的名字定义对象

- plugin_name
- PluginName
- -

Rule 3: 插件模型仅能用下列开头的名字定义会话变量

- session.plugin_name
- session.PluginName

Rule 4: 插件应该包含许可和文档，这些被放在：

- static/plugin_name/license.html
- static/plugin_name/about.html

Rule 5: 插件仅依赖于定义在基本构建应用“db.py”中的全局对象存在，即

- 数据库连接db
- Auth 实例 auth
- Crud 实例 crud
- Service实例service

有些插件可能更高级，有配置以防参数超过一个 db 的实例存在。

Rule 6: 如果插件需要配置参数，这些该通过 PluginManager 设置，下面描述。

遵守上面的规则，我们能确认：

- admin 识别所有 plugin_name 文件和文件夹为当个实体的一部分。
- 插件不会相互干扰。

上面的规则不解决插件版本问题和依赖性，这超出了我们的范围。

12.2.1 组件插件 Component plugins

组件插件是定义组件的插件，组件经常访问数据库并用它们自己的模型定义。

这里我们通过使用之前相同的代码，把前面的 comments 组件变为 comments_plugin，遵守所有前面规则。

首先，我们创建的模型叫做“models/plugin_comments.py”：

```
1 db.define_table('plugin_comments_comment',
```



```

2      Field('body', 'text', label='Your comment'),
3      Field('posted_on', 'datetime', default=request.now),
4      Field('posted_by', db.auth_user, default=auth.user_id))
5  db.plugin_comments_comment.posted_on.writable=False
6  db.plugin_comments_comment.posted_on.readable=False
7  db.plugin_comments_comment.posted_by.writable=False
8  db.plugin_comments_comment.posted_by.readable=False
9
10 def plugin_comments():
11     return LOAD('plugin_comments', 'post', ajax=True)

```

(注意最后两行定义的函数，会简化插件嵌入)

其次，我们定义 “controllers/plugin_comments.py”

```

1  @auth.requires_login()
2  def post():
3      comment = db.plugin_comments_comment
4      return dict(form=crud.create(comment),
5                  comments=db(comment).select())

```

第三步，我们创建视图叫 “views/plugin_comments/post.load”：

```

1  {{for comment in comments:}}
2  <div class="comment">
3      on {{=comment.posted_on}} {{=comment.posted_by.first_name}}
4      says <span class="comment_body">{{=comment.body}}</span>
5  </div>
6  {{pass}}
7  {{=form}}

```

现在，为了发布我们能用admin打包插件，Admin会保存该插件为：

```

1  web2py.plugin.comments.w2p

```

我们可以在任意视图中使用插件，只需通过在admin里edit页面安装插件并添加这个到我们自己的视图

```

1  {{=plugin_comments()}}

```

当然通过用带参数和配置选项的组件我们可以使插件更高级，组件越复杂，避免名字冲突就变得越难，下面描述Plugin Manager设计来避免这个问题。

12.2.2 插件管理对象 Plugin manager

PluginManager 是在gluon.tools定义的类，在我们解释它如何在内工作之前，我们解释如何使用它。

这里我们考虑之前的comments_plugin并使它更好，我们想能够定制：

```

1  db.plugin_comments_comment.body.label

```

无需编辑插件代码自身。

下面是我们如何实现它：

首先，用下面方式重写插件 “models/plugin_comments.py”：

```

1  db.define_table('plugin_comments_comment',
2      Field('body', 'text', label=plugin_comments.comments.body_label),
3      Field('posted_on', 'datetime', default=request.now),
4      Field('posted_by', db.auth_user, default=auth.user_id))
5

```

```

6     def plugin_comments()
7         from gluon.tools import PluginManager
8         plugins = PluginManager('comments', body_label='Your comment')
9
10        comment = db.plugin_comments_comment
11        comment.label=plugins.comments.body_label
12        comment.posted_on.writable=False
13        comment.posted_on.readable=False
14        comment.posted_by.writable=False
15        comment.posted_by.readable=False
16        return LOAD('plugin_comments','post.load',ajax=True)

```

注意所有的代码除了表定义的是如何封装在单个函数内的，也要注意函数如何创建 PluginManager 实例。

现在在你应用中的任何模型，例如 “models/db.py”，你可以配置该插件如下：

```

1     from gluon.tools import PluginManager
2     plugins = PluginManager()
3     plugins.comments.body_label = T('Post a comment')

```

plugins 对象已经在默认基本构建应用 “models/db.py” 中实例化。

PluginManager 对象是 Storage 对象中的一个线程级的独个存储对象，意味着在同个应用中你可以实例化你想要数量的，但（不论它们有相同的名字或没有）它们行为如同仅有单个 PluginManager 实例。

特别是，每个插件文件生成它们自己的 PluginManager 对象并注册自己以及它默认的参数，用：

```

1     plugins = PluginManager('name', param1='value', param2='value')

```

可以在其它地方覆盖这些参数（例如 “models/db.py”），用下面的代码：

```

1     plugins = PluginManager()
2     plugins.name.param1 = 'other value'

```

也可以在一个地方配置多个插件：

```

1     plugins = PluginManager()
2     plugins.name.param1 = '...'
3     plugins.name.param2 = '...'
4     plugins.name1.param3 = '...'
5     plugins.name2.param4 = '...'
6     plugins.name3.param5 = '...'

```

当插件被定义，PluginManager 必须用参数：plugin 名字和可选命名的参数均是默认参数。但是，当插件被配置时，PluginManager 构造函数必须不带参数，配置必须在插件定义之前（即，它必须是以第一个字母顺序的模型文件）。

12.2.3 布局插件 Layout plugins

布局插件比组件插件简单，因为通常它不包含代码，仅含视图和静态文件，但你仍要遵从好的实践：

首先，创建叫做 “static/plugin_layout_name/” 的文件夹（name 是你布局的名字），放置所有的静态文件于此。

其次，创建一个叫做 “views/plugin_layout_name/layout.html” 的布局文件，包含布局链接 “static/plugin_layout_name/” 图像、CSS 和 JavaScript files 文件。

再次，修改 “views/layout.html” 以便它仅能读取：

```

1     {{extend 'plugin_layout_name/layout.html'}}

```

该设计的好处是，这个插件的用户能安装多个布局并通过编辑“views/layout.html”选择应用哪个，而且“views/layout.html”不被admin与插件一道打包，因此不会有在插件覆盖之前安装的布局中的用户代码的风险。

12.3 plugin_wiki

免责声明：plugin_wiki 是仍正在开发的，因此我们不保证与 web2py 核心功能同一个级别的后向兼容性。

plugin_wiki 是插件 on steroids 类固醇，我们的意思是它定义多个应用的组件，它能改变你开发你应用的方式：

可以从以下链接下载：

1 <http://web2py.com/examples/static/web2py.plugin.wiki.w2p>

plugin_wiki 背后的理念是大部分应用包括页面是半静态的，这些是不包含复杂自定义逻辑的页面，它们包含结构化的文本（想想帮助页面）、图像、音频、视频、crud 表单或标准组件的集合（评论、标签、表和地图）等。这些页面可能是公开的，需要登录或有其它认证限制，这些页面可能用菜单链接或只有通过向导表单到达，plugin_wiki 提供简便的方法增加页面适应这个类别到你的规则的 web2py 应用。

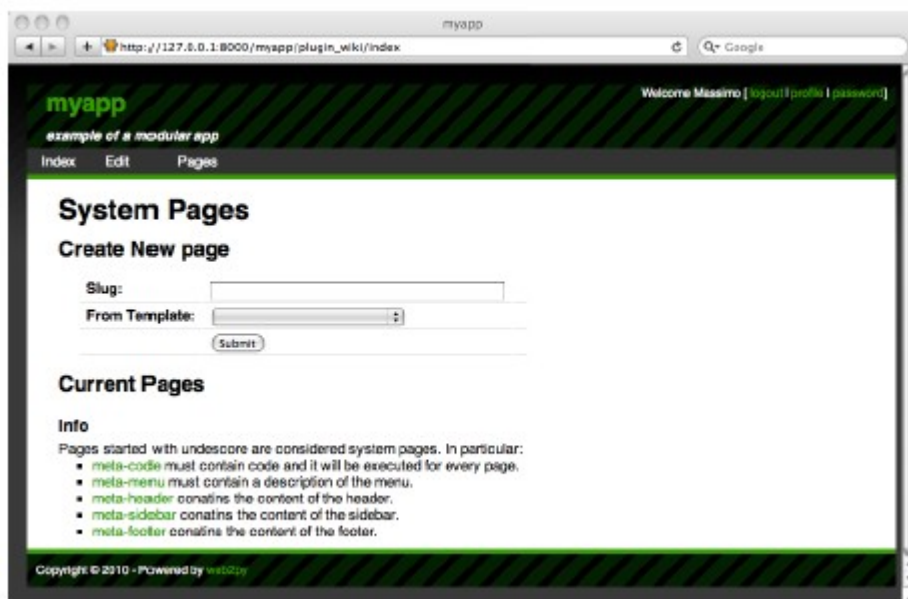
特别是，plugin_wiki 提供：

- 类似wiki的界面允许添加页面到你的应用并通过slug引用它们，这些页面（我们称它们为wiki页面）有版本并存储在数据库。
- 公开和私有页面（需要登录），如果页面要求登录，它可能需要用户有特定组成员资格。
- 三个级别：1、2 和 3，级别 1 表示页面只包括文本、图像、音频和视频。级别 2 表示页面也包括 widget（小工具）（这些是之前章节定义能嵌在 wiki 页的组件）。级别 3 表示页面也能包含 web2py 模板代码。
- 用markin语法编辑页面选择或在HTML用WYSIWYG编辑器（编辑对象）
- 一组 widget（小工具）：以组件实现，它们是自己记录并作为规则组件嵌入通常 web2py 视图或使用简化的语法，到 wiki 页面。
- 一组特殊页面（meta-code, meta-menu等）能用来定制插件（例如，定义插件运行的代码，定制菜单等）

welcome基本构建应用加上plugin_wiki可以视为开发环境，这个环境下适于构建简单web应用比如博客。

安装插件之后你注意到的第一件事是它增加了一个叫pages的新菜单项。
点击pages菜单项，你会被重定向到插件动作：

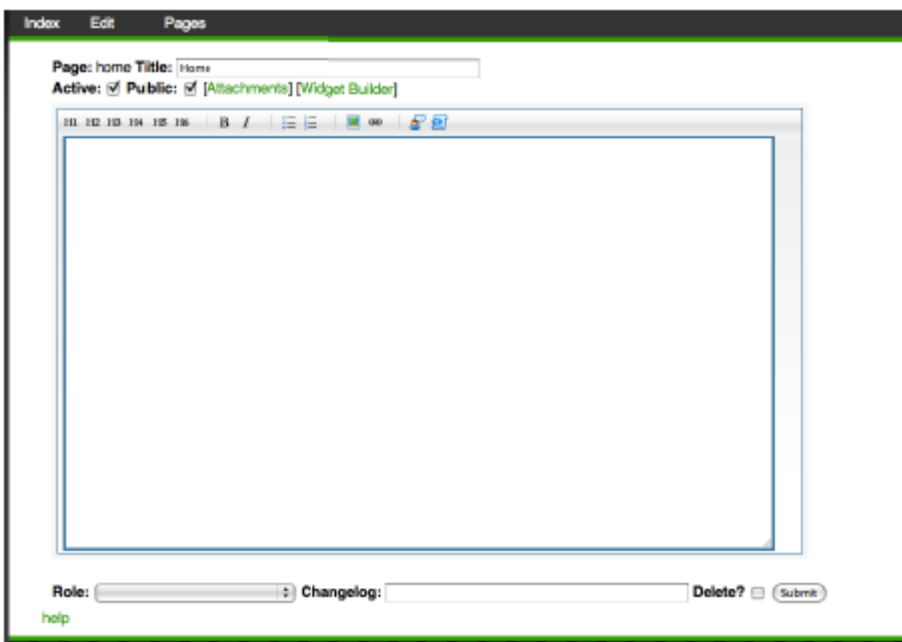
1 http://127.0.0.1:8000/myapp/plugin_wiki/index



插件索引页列出用插件本身创建的页面并允许你通过选择slug创建新的，尝试创建home页面，你将重定向到

1 http://127.0.0.1:8000/myapp/plugin_wiki/page/home

点击create page编辑它的内容。



默认情况，该plugin为级别3，意味着你能插入widget（小工具）和代码到页面。默认使用markmin语法来描述页面内容。

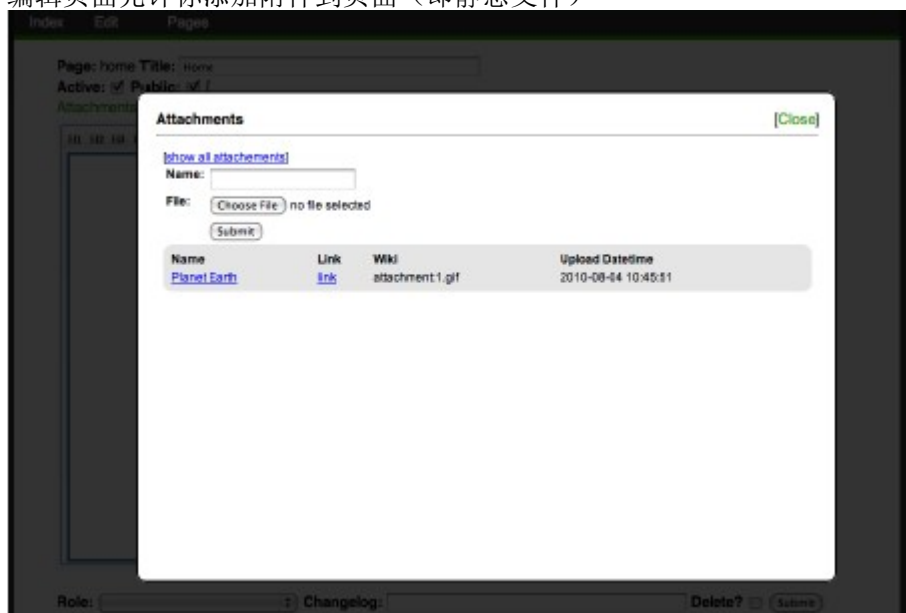
12.3.1 MARKMIN 语法

下面是主要的markmin语法：

markmin	html
# title	<h1>title</h1>
## subtitle	<h2>subtitle</h2>
### subsubtitle	<h3>subsubtitle</h3>
bold	bold
"italic"	<i>italic</i>
http://...	http:...
http://...png	
http://...mp3	<audio src="http://...mp3"></audio>
http://...mp4	<video src="http://...mp4"></video>
qr:http://...	
embed:http://...	<iframe src="http://..."></iframe>

注意链接、图像、音频和视频文件是自动嵌入的。
更多 MARKIN 语法信息，请参考第 5 章。

如果页面不存在，你将被要求创建一个。
编辑页面允许你添加附件到页面（即静态文件）



你也能链接它们：

```
1 [[mylink name attachment:3.png]]
```

或嵌入它们，用：

```
1 [[myimage attachment:3.png center 200px]]
```

大小（200px）是可选的，center 不是可选但它可以用 left 或 right 替换。

也可以用下面方式嵌入块引用文本

```
1 -----
2 this is blockquoted
3 -----
```

表也一样

```
1 -----
2 0 | 0 | X
```

```

3      0 | X | 0
4      X | 0 | 0
5      -----

```

逐字文本

```

1  ``
2      verbatim text
3  ``

```

也可以在前面加选项:class 到最后---或最后，对于块引用文本和表，它被翻译为标签类，例如：

```

1      -----
2      test
3      -----:abc

```

呈现为：

```

1      <blockquote class="abc">test</blockquote>

```

对于逐字文本，类能用来嵌入到不同类型的内容，你能，例如用:code_language 指定语言高亮语法嵌入代码

```

1  ``
2      def index(): return 'hello world'
3  ``:code_python

```

也可以嵌入 widget：

```

1  ``
2      name: widget_name
3      attribute1: value1
4      attribute2: value2
5  ``:widget

```

从编辑页面，点击“widget builde”从列表交互地插入 widget：



（widget 列表参看下一节）。

也可以嵌入 web2py 模版语言代码：

```

1  ``
2      {{for i in range(10):}}<h1>{{=i}}</h1>{{pass}}
3  ``:template

```

12.3.2 页面权限 Page permissions

当编辑一个页面将找到下列字段：

- `active`（默认 `True`），如果页面不是活动的，它不能被访问者访问（即便是 `public`）
- `public`（默认 `True`），如果页面是公开的，它能被访问者访问且无需登录。
- `role`（默认 `None`），如果页面有角色，页面只能被登录用户有相应角色的组的成员访问。

12.3.3 特殊页面 *Special pages*

meta-menu 包含菜单，如果该页面不存在，web2py使用定义在“`models/menu.py`”的 `response.menu`，**meta-menu** 页面内容覆盖菜单。语法如下：

```
1      Item 1 Name http://link1.com
2          Submenu Item 11 Name http://link11.com
3          Submenu Item 12 Name http://link12.com
4          Submenu Item 13 Name http://link13.com
5      Item 2 Name http://link1.com
6          Submenu Item 21 Name http://link21.com
7          Submenu Item 211 Name http://link211.com
8          Submenu Item 212 Name http://link212.com
9          Submenu Item 22 Name http://link22.com
10         Submenu Item 23 Name http://link23.com
```

缩进决定了子菜单结构，每项由菜单项文本紧随链接组成，链接可以是 `page:slug`，链接 `None` 不链接到任何页面，额外的空格被忽略。

下面是另一个例子：

```
1      Home page:home
2      Search Engines None
3          Yahoo http://yahoo.com
4          Google http://google.com
5          Bing http://bing.com
6      Help page:help
```

呈现如下效果：



meta-code 是另一个特殊页面而且它必须包含 web2py 代码，这是你模型的扩展，事实上你能在这里放置模型代码，当“`models/plugin_wiki.py`”代码执行时，它被执行。

可以在 **meta-code** 里定义表。

例如，你可以通过放置这写代码在 **meta-code** 里，创建简单表“`friends`”：

```
1      db.define_table('friend',Field('name',requires=IS_NOT_EMPTY()))
```

并且你可以通过嵌入下面代码到你的选择页面来创建 `friend` 管理界面：

```
1      # List of friends
2      ``
3      name: jqgrid
4      table: friend
5      ``:widget
```

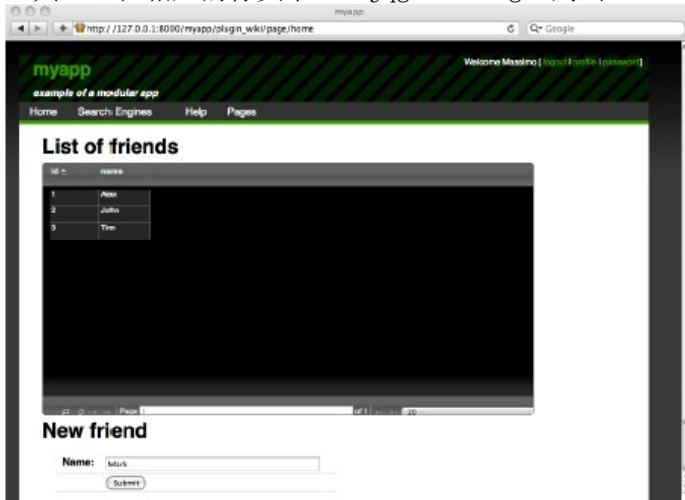


```

6
7     # New friend
8     ``
9     name: create
10    table: friend
11    ``:widget

```

该页面有两个标头（以#开头）：“List of friends”和“New friend”，该页面包含两个widget（小工具）（在相应的标头下）：jqgrid widget列出friends，crud widget添加新的friend。



在“welcome/views/layout.html”中，meta-header、meta-footer和meta-sidebar不被默认布局视图使用。如果你想使用它们，用admin（或shell）编辑“layout.html”并放置下列标签在合适的位置：

```

1     {{=plugin_wiki.embed_page('meta-header') or ''}}
2     {{=plugin_wiki.embed_page('meta-sidebar') or ''}}
3     {{=plugin_wiki.embed_page('meta-footer') or ''}}

```

用这种方式，那些页面的内容会显示在布局的页头、侧边栏和页脚。

12.3.4 配置 *plugin_wiki*

至于在“models/db.py”中的其他任何插件，你可以做：

```

1     from gluon.tools import PluginManager
2     plugins = PluginManager()
3     plugins.wiki.editor = auth.user.email == mail.settings.sender
4     plugins.wiki.level = 3
5     plugins.wiki.mode = 'markmin' or 'html'
6     plugins.wiki.theme = 'ui-darkness'

```

此处

- 如果当前已登录用户授权到编辑**plugin_wiki**页面，**editor** 是True。
- **level** 是权限：1 编辑规则页面，2 嵌入widget到页面，3 嵌入代码。
- **mode** 决定使用“markin”编辑器还是WYSIWYG“html”编辑器。
- **theme** 是要求jQuery UI Theme的名字，默认仅中性色彩“ui-darkness”被安装。

可以在这添加主题：

```

1     static/plugin_wiki/ui/%(theme)s/jquery-ui-1.8.1.custom.css

```

12.3.5 当前小工具 Current widgets

每个 widget 都能嵌入到 `plugin_wiki` 页面和通常的 web2py 模板。
例如，嵌入 YouTube 视频到 `plugin_wiki` 页面，你可以做

```
1 ``
2     name: youtube
3     code: 17AwnfFRc7g
4 ``:widget
```

或嵌入同样的 widget 到 web2py 视图，你可以做

```
1 {{=plugin_wiki.widget('youtube',code='17AwnfFRc7g')}}}
```

无论哪种情况，输出如下：



widget 参数被要求没有默认值。
下面是所有当前的 widget 列表：

read

```
1 read(table,record_id=None)
```

读取并显示记录

- table 表的名字
- record_id 记录号

create

```
1 create(table,message='',next='',readonly_fields='',
2 hidden_fields='',default_fields='')
```

显示记录创建表单

- table 表的名字
- message 记录创建后显示的消息
- next 要重定向到的地方，例如 “page/index/[id]”
- readonly_fields 逗号分隔的字段列表
- hidden_fields 逗号分隔的字段列表
- default_fields 逗号分隔的fieldname=value列表

update

```
1 update(table, record_id='', message='', next='',
2        readonly_fields='', hidden_fields='', default_fields='')
```

显示记录更新表单

- table 表的名字
- record_id 是要被更新的记录或{{=request.args(-1)}}
- message 记录创建后显示的消息
- next 要重定向到的地方，例如 “page/index/[id]”
- readonly_fields 逗号分隔的字段列表
- hidden_fields 逗号分隔的字段列表
- default_fields 逗号分隔的fieldname=value列表

select

```
1 select(table, query_field='', query_value='', fields='')
```

列出表中所有记录

- table 表的名字
- query_field 和 query_value 如果存在，用查询query_field == query_value过滤记录
- fields是要显示的、逗号分隔的字段列表

search

```
1 search(table, fields='')
```

查询记录的widget

- table 表的名字
- fields是要显示的、逗号分隔的字段列表

jqgrid

```
1 jqgrid(table, fieldname=None, fieldvalue=None, col_widths='',
2        colnames=None, _id=None, fields='', col_width=80, width=700, height=300)
```

嵌入jqgrid 插件

- table 表的名字
- fieldname、fieldvalue是可选过滤器： fieldname==fieldvalue
- col_widths是每列的宽度
- colnames 是要显示的列名字的列表
- _id是包含jqGrid的TABLE “id”
- fields 是要显示的列的列表
- col_width是每列的默认宽度
- height是jqGrid的高度
- width是jqGrid的宽度

一旦安装了plugin_wiki，也能在你其它视图容易地使用jqGrid，使用示例（显示用fk_id==47过滤你的

表格)：

```
1      {{=plugin_wiki.widget('jqgrid','yourtable','fk_id',47,'70,150',  
      'Id,Comments',None,'id,notes',80,300,200)}}
```

latex

```
1      latex(expression)
```

使用Google图表API嵌入LaTeX

pie_chart

```
1      pie_chart(data='1,2,3',names='a,b,c',width=300,height=150,align='center')
```

嵌入饼图

- data 是逗号分隔值的列表
- names 是逗号分隔标签列表（数据一个）
- width 是图像宽度
- height 是图像高度
- align 决定图像对齐

bar_chart

```
1      bar_chart(data='1,2,3',names='a,b,c',width=300,height=150,align='center')
```

使用 Google 图表 API 嵌入条状图

- data 是逗号分隔值的列表
- names 是逗号分隔标签列表（数据一个）
- width 是图像宽度
- height 是图像高度
- align 决定图像对齐

slideshow

```
1      slideshow(table, field='image', transition='fade', width=200, height=200)
```

嵌入幻灯片，它从表得到图片。

- table 是表名
- field 是表中上传字段包含的图片
- transition 决定过渡类型，比如渐变等
- width 是图像宽度
- height 是图像高度

youtube

```
1      youtube(code, width=400, height=250)
```

嵌入 YouTube 视频（用代码）

- code 是视频代码
- width 是图像宽度
- height 是图像高度

vimeo

```
1      vimeo(code, width=400, height=250)
```

嵌入 Vimeo 视频（用代码）

- code 是视频代码
- width 是图像宽度
- height 是图像高度

mediaplayer

```
1 mediaplayer(src, width=400, height=250)
```

嵌入媒体文件（比如Flash 视频或mp3 文件）

- src 是视频的 scr
- width 是图像宽度
- height 是图像高度

comments

```
1 comments(table='None', record_id=None)
```

嵌入评论到页面

评论可能被链接到表和/或记录

- table 是表名
- record_id 是记录的id

tags

```
1 tags(table='None', record_id=None)
```

嵌入标签到页面，标签可能被链接到表和/或记录

- table 是表名
- record_id 是记录的id

tag_cloud

```
1 tag_cloud()
```

嵌入标签云

map

```
1 map(key='...', table='auth_user', width=400, height=200)
```

嵌入 Google 地图

它从表得到地图上的点

- key 是 google 地图 api 的键（默认为 127.0.0.1 工作）
- table 是表名
- width 是地图宽度
- height 是地图高度

地图必须有列：latitude, longitude和map_popup，当点击一个点，map_popup消息会出现。

Iframe

```
1 iframe(src, width=400, height=300)
```

在<iframe></iframe>中嵌入页面

load_url

```
1 load_url(src)
```

用LOAD 函数载入url 内容

load_action

```
1 load_action(action, controller='', ajax=True)
```

用LOAD函数载入url内容（(request.application, 控制器, 动作）

12.3.6 扩展 widget

`plugin_wiki`的widget能通过创建新模型文件“`models/plugin_wiki_`” name被添加，此处name是任意的也是包含下面类似内容的文件：

```
1 class PluginWikiWidgets(PluginWikiWidgets):
2     @staticmethod
3     def my_new_widget(arg1, arg2='value', arg3='value'):
4         """
5         document the widget
6         """
7     return "body of the widget"
```

第一行说明正扩展widget列表，类型内，你能定义你需要的函数，每个静态函数都是新的widget，除了以下划线开始的函数不是，函数带有任意数量的参数，也许有或没有默认值，函数的docstring必须使用markin语法归档函数自己。

当widget嵌到`plugin_wiki`页面，参数以字符串传递给widget，这意味着widget函数能够给每个参数接受字符串并最终把它们转换为所需要的表示，你能决定哪种字符串表示，就确认它在docstring中被记录。

Widget可以返回web2py帮助对象的字符串，后一种情况，通过使用 `.xml()`，它们会被序列化。注意新widget如何访问任何声明在全局名称空间内的变量。

例如，我们打算创建一个新的widget，其显示本章开头创建的“`contact/ask`”表单，这能通过创建文件“`models/plugin_wiki_contact`”实现，该文件包含：

```
1 class PluginWikiWidgets(PluginWikiWidgets):
2     @staticmethod
3     def ask(email_label='Your email', question_label='question'):
4         """
5         This plugin will display a contact us form that allows
6         the visitor to ask a question.
7         The question will be emailed to you and the widget will
8         disappear from the page.
9         The arguments are
10
11         - email_label: the label of the visitor email field
12         - question_label: the label of the question field
13
14         """
15     form=SQLFORM.factory(
16         Field('your_email', requires=IS_EMAIL(), label=email_label),
17         Field('question', requires=IS_NOT_EMPTY()), label=question_label)
18     if form.process().accepted:
19         if mail.send(to='admin@example.com',
20                     subject='from %s' % form.vars.your_email,
21                     message = form.vars.question):
22             command="jQuery('#%s').hide()" % div_id
23             response.flash = 'Thank you'
24             response.js = "jQuery('#%s').hide()" % request.cid
25         else:
26             form.errors.your_email="Unable to send the email"
27     return form.xml()
```

`plugin_wiki` widget不被视图呈现，除非`response.render(...)`函数被该widget显式调用。

第 13 章 部署方法

在生产环境中，有多种方法部署 web2py，其详细情况依赖于配置和主机提供的服务。在本章，我们将考虑以下几个问题：

- 生产部署 (Apache、Lighttpd、Cherokee)
- 安全性
- 可扩展性
- Google App Engine 平台上部署 (GAE^[13])

web2py 附带了一个 SSL^[21]，其能启用 web 服务器 Rocket wsgiserver^[22]，尽管这是一个快速的 Web 服务器，但它的配置能力有限，因此最好部署 web2py 在 Apache^[82]、Lighttpd^[89] 或者 Cherokee^[90] 上，这些都是免费的且开源的 Web 服务器，其是可定制的，并且在高流量的生产环境中，已被证明是可靠的。它们可以被配置来直接地服务静态文件，处理 HTTPS，为动态内容传递控制给 web2py。

直到几年前，Web 服务器和 Web 应用程序之间的通信标准接口是通用网关接口 (CGI)^[81]，CGI 的主要问题是：它为每个 HTTP 请求创建一个新的进程，如果 Web 应用程序用一种解释性的语言编写，那么 CGI 脚本服务的每一个 HTTP 请求都将启动这个解释器的一个新实例。在生产环境中，这很慢且应当避免，而且 CGI 只能处理简单的响应，它不能处理，例如文件流。web2py 提供了一个文件 `cgihandler.py` 接口到 CGI。

解决这个问题的一种方案是，对于 Apache 使用 `mod_python` 模块，我们在此讨论它是因为尽管 `mod_python` 项目已经被 Apache 软件基金会正式地放弃了，然而它的使用仍然很广泛，当 Apache 启动时，`mod_python` 启动 Python 解释器的一个实例，并在其自己的线程中服务每个 HTTP 请求，而无需每次重启 Python，这是一个比 CGI 更好的解决方案，然而它却不是是一个最优解决方案，因为为了让 Web 服务器和 Web 应用程序能相互通信，`mod_python` 使用其专属接口。`mod_python` 中，所有托管的应用程序在相同的 `user-id/group-id` 中运行，这就引起了一个安全问题，`issues.web2py` 提供了一个文件 `modpythonhandler.py` 接口到 `mod_python`。

在过去的几年里，用 Python 编写的一个能在 Web 服务器和 Web 应用程序之间通信的新标准接口出现后，Python 社区社员们走到一起，它被称为 Web 服务器网关接口 (WSGI)^[17, 18]，web2py 被构建在 WSGI 上，当 WSGI 不可用时，它提供了使用其他接口的处理程序。

通过格雷厄姆邓普尔顿 (Graham Dumpleton) 开发的 `mod_wsgi`^[88] 模块，Apache 支持 WSGI，web2py 提供了一个文件 `wsgihandler.py` 接口到 WSGI。

一些 web 主机服务并不支持 `mod_wsgi` 模块，在这种情况下，我们必须把 Apache 作为代理使用，将所有传入的请求传到 web2py 内置的 Web 服务器（例如运行在 `localhost:8000`）。

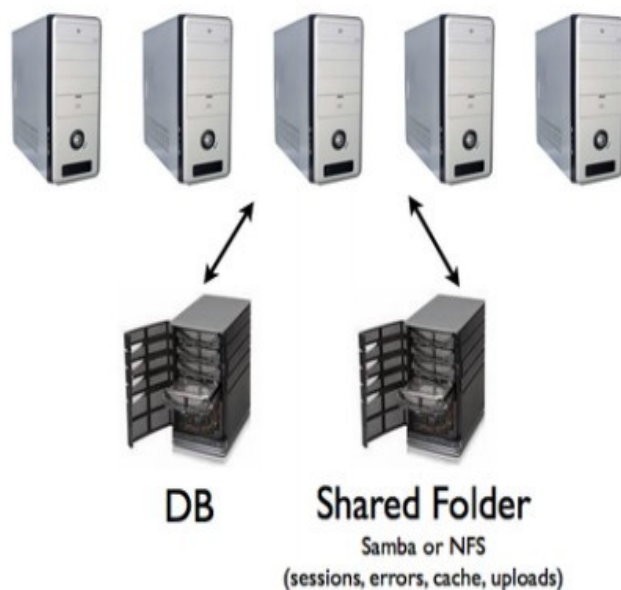
在这两种情况下，使用 `mod_wsgi` 和/或 `mod_proxy` 模块，Apache 都能被配置服务静态文件，直接处理 SSL 加密，大大减少 web2py 的负担。

当前，`lighttpd` Web 服务器不支持 WSGI 接口，但它支持被改进过后的 CGI 接口——`FastCGI`^[91] 接口，`FastCGI` 接口的主要目的是减少开销，其有关 Web 服务器和 CGI 程序接口间的开销，并允许一个服务器立刻处理更多的 HTTP 请求。

依据 lighttpd, “lighttpd 模式的几种流行的 Web 2.0 网站, 如 YouTube 和 Wikipedia, 它的这种高速 I/O 基础设施, 允许他们在使用相同的硬件时比用可替换的 Web 服务器, 能更好地扩展几倍。实际上, 使用 FastCGI 的 Lighttpd 比使用 mod_wsgi 的 Apache 更快, web2py 中提供了一个文件 fcgihandler.py 接口到 FastCGI, web2py 也包含了 gaehandler.py 去连接 Google App Engine (GAE), 在 GAE 上, Web 应用程序运行 “in the cloud” (在云中), 这意味着, 该框架完全抽象了任何硬件细节, Web 应用程序会根据需要自动复制多次, 并响应所有并发请求。在这种情况下, 复制意味着更多, 比在单个服务器上多线程更多, 同时也意味着在不同的服务器上的多个进程, GAE 通过阻断写访问进文件系统, 实现这一水平的可扩展性, 并且所有的持久性信息必须存储在谷歌的 BigTable 数据库中或在 memcache 中。

在非 GAE 平台, 可扩展性是一个需要加以解决的问题, 在 web2py 应用程序中, 它可能需要一些调整。实现可扩展性的最常用的方法是使用多台 Web 服务器后台支撑的负载平衡器 (一个简单的循环, 或一些更复杂的循环, 接收来自服务器的心跳反馈)。

即使有多个 Web 服务器, 然而必须有一个, 且只有一个数据库服务器。默认情况下, web2py 会使用文件系统存储会话、错误票据、已上传文件以及缓存。这意味着, 在默认配置情况下, 相应的文件夹必须是共享文件夹:



在本章的余下部分, 我们将要考虑多种方法, 提供对该稚嫩方法的改进, 包括:

- 存储会话于数据库中、高速缓存中或者根本不存储会话。
- 存储票据于本地文件系统上, 并分批移动到数据库中。
- 使用 memcache 而不用 cache.ram 和 cache.disk。
- 将上传文件存放在数据库中而不存放于共享文件系统中。

尽管我们推荐前三种方法，但第四种方法可能在小文件处理情况下有很多优势，对于大文件可能就适得其反。

13.0.7 文件 anyserver.py

tornado: inxx twisted: inxx wsgrief

web2py 附带了一个称为 anyserver.py 的文件，实施 WSGI 接口到以下各种流行服务器：bjoern、cgi、cherrypy、diesel、eventlet、fapws、up、gevent、gunicorn、mongrel2、paste、rocket、tornado、twisted 和 wsgiref。

你可以使用这些服务器中的任意一个，例如 Tornado，只需如下做：

```
1 python anyserver.py -s tornado -i 127.0.0.1 -p 8000 -l -P
    (-l 表示 logging (记录日志)，-p 表示 profiler (分析器)，所有命令行上的信息选项用“-h”：
```

```
1 python anyserver.py -h
```

13.1 Linux 和 Unix

13.1.1 生产部署一步到位

下面是从零开始安装 apache+python+mod_wsgi+web2py+postgresql 的几个步骤。
在 Ubuntu 系统：

```
1 wget http://web2py.googlecode.com/hg/scripts/setup-web2py-ubuntu.sh
2 chmod +x setup-web2py-ubuntu.sh
3 sudo ./setup-web2py-ubuntu.sh
```

在 Fedora 系统：

```
1 wget http://web2py.googlecode.com/hg/scripts/setup-web2py-fedora.sh
2 chmod +x setup-web2py-fedora.sh
3 sudo ./setup-web2py-fedora.sh
```

这两组脚本直接可以运行，但是每一个 Linux 的安装都存在点区别，因此在运行它们之前，一定要确保已经检查了源代码。在 Ubuntu 的情况下，它们所做的解释如下，即它们不实施下面讨论的可扩展性优化。

13.1.2 Apache 设置

在这一部分，我们使用 Ubuntu 8.04 服务编辑器作为参考平台，该配置命令与在其他基于 Debian 的 Linux 发行版极其相似，但它们也可能会有所不同于基于 Fedora 的系统（使用 yum 而不是 apt-get 的）。

首先，确保所有必要的 Python 和 Apache 的安装包都被安装，通过键入以下 shell 命令：

```
1 sudo apt-get update
```

```

2    sudo apt-get -y upgrade
3    sudo apt-get -y install openssh-server
4    sudo apt-get -y install python
5    sudo apt-get -y install python-dev
6    sudo apt-get -y install apache2
7    sudo apt-get -y install libapache2-mod-wsgi
8    sudo apt-get -y install libapache2-mod-proxy-html

```

然后，启用 Apache 中的 SSL 模块、代理模块和 WSGI 模块：

```

1    sudo ln -s /etc/apache2/mods-available/proxy_http.load \
2        /etc/apache2/mods-enabled/proxy_http.load
3    sudo a2enmod ssl
4    sudo a2enmod proxy
5    sudo a2enmod proxy_http
6    sudo a2enmod wsgi

```

创建 SSL 文件夹，并把 SSL 证书放进 Apache 中：

```

1    sudo mkdir /etc/apache2/ssl

```

应该从受信任的证书颁发机构得到你的 SSL 证书，如 verisign.com，但出于测试的目的，也可以按照参考文献^[87]中的说明生成你自己的自签名的证书。

然后重启 web 服务器：

```

1    sudo /etc/init.d/apache2 restart

```

Apache 配置文档：

```

1    /etc/apache2/sites-available/default

```

Apache 日志在：

```

1    /var/log/apache2/

```

13.1.3 mod_wsgi 模块

在安装了上述 Web 服务器的机器上，下载并解压缩 web2py 源代码。

在 /users/www-data/ 下安装 web2py，例如给用户 www-data 和组 www-data 所有权，这些步骤可按照下面的 shell 命令完成：

```

1    cd /users/www-data/
2    sudo wget http://web2py.com/examples/static/web2py_src.zip
3    sudo unzip web2py_src.zip
4    sudo chown -R www-data:www-data /user/www-data/web2py

```

为了建立具有 mod_wsgi 模块的 web2py，创建一个新的 Apache 配置文档：

```

1    /etc/apache2/sites-available/web2py

```

并包括以下代码：

```

1    <VirtualHost *:80>
2        ServerName web2py.example.com
3        WSGIDaemonProcess web2py user=www-data group=www-data \
4            display-name=%{GROUP}
5        WSGIProcessGroup web2py
6        WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py
7
8        <Directory /users/www-data/web2py>
9            AllowOverride None

```

```

10     Order Allow,Deny
11     Deny from all
12     <Files wsgihandler.py>
13         Allow from all
14     </Files>
15 </Directory>
16
17 AliasMatch ^/([^/]+)/static/(.*) \
18     /users/www-data/web2py/applications/$1/static/$2
19 <Directory /users/www-data/web2py/applications/*/static/>
20     Order Allow,Deny
21     Allow from all
22 </Directory>
23
24 <Location /admin>
25     Deny from all
26 </Location>
27
28 <LocationMatch ^/([^/]+)/appadmin>
29     Deny from all
30 </LocationMatch>
31
32 CustomLog /private/var/log/apache2/access.log common
33 ErrorLog /private/var/log/apache2/error.log
34 </VirtualHost>

```

当重启 Apache，它应该传递所有的请求给 web2py，而不用通过 Rocket wsgiserver。
下面是一些解释：

```

1 WSGIDaemonProcess web2py user=www-data group=www-data
2     display-name=%{GROUP}

```

在“web2py.example.com”网站中，定义一个守护进程组，通过在虚拟主机内定义它，只有该虚拟主机能够访问正在使用中的 WSGIProcessGroup，包括那些具有相同服务器名的但在不同端口的任何虚拟主机。“user”和“group”选项应该设置给用户，这些用户具有能够写访问安装了 web2py 的目录，如果让 web2py 安装目录能够被像 Apache 一样默认用户那样可写，那么就不必设置用户和组。“display-name”选项，使进程名称出现在 ps 的输出“（WSGI 的 web2py）”，而不是作为 Apache Web 服务器的可执行文件名称。由于“processes”或“threads”选项没有被指定，守护进程组将有一个单一进程，在这一进程中运行着 15 个线程。通常情况下，对于大多数站点是绰绰有余，并应保留原样，如果重载它，请不要使用“processes = 1”，因为这样做将禁用任何在浏览器中的 WSGI 调试工具，那些选中“wsgi.multiprocess”标志的 WSGI 调试工具，这是因为，任何使用了“processes”选项的都将导致该标志被设置为 true，即使是单一进程，这样的工具也期望它被设置为 false。注意：如果您的应用程序代码或第三方扩展模块都不是线程安全的，请使用选项“processes=5 threads=1”代替。这将在守护进程组中创建 5 个进程，在那儿每个进程都是单线程的，如果您的应用程序泄漏了 Python 对象，可以考虑使用“maximum-requests=1000”，因为它无法正确地收集垃圾。

```

1 WSGIProcessGroup web2py

```

委托所有 WSGI 应用程序的运行给守护进程组，通过使用 WSGIDaemonProcess 指令，它被配置。

```

1 WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py

```

安装 web2py 应用程序，在这种情况下，它将被安装在网站的根上。

```
1 <Directory /users/www-data/web2py>
2 ...
3 </Directory>
```

授权 Apache 访问 WSGI 脚本文件：

```
1 <Directory /users/www-data/web2py/applications/*/static/>
2     Order Allow,Deny
3     Allow from all
4 </Directory>
```

当搜索静态文件时，指示 Apache 绕过 web2py：

```
1 <Location /admin>
2     Deny from all
3 </Location>
```

和：

```
1 <LocationMatch ^/(^/+)/appadmin>
2     Deny from all
3 </LocationMatch>
```

阻止公共访问 admin 和 appadmin。

通常情况下，我们会只允许 WSGI 脚本文件所在的整个目录，但 web2py 的 WSGI 脚本文件却放置在一个包含其他源代码、管理界面密码的目录下，开放整个目录会引起安全问题的出现，因为从技术层面讲，Apache 将获准服务所有文件的权利，任何用户都可通过映射 URL 访问那个目录，为了避免安全问题，明确拒绝访问该目录中的内容，除了 WSGI 脚本文件，并禁止用户从 a.htaccess 文件做任何覆盖，以便获取额外的安全性。

可以找到一个完整的、有注释的、Apache wsgi 配置文件：

```
1 scripts/web2py-wsgi.conf
```

这部分能被创建有赖于 mod_wsgi 模块开发者 Graham Dumbleton 的帮助。

13.1.4 mod_wsgi 模块和 SSL（安全套接层）

为了强制一些应用程序（例如 admin 和 appadmin）检验 HTTPS 协议、存储 SSL 证书和密钥文件：

```
1 /etc/apache2/ssl/server.crt
2 /etc/apache2/ssl/server.key
```

以及编辑 Apache 配置文件 web2py.conf 并附加：

```
1 <VirtualHost *:443>
2     ServerName web2py.example.com
3     SSLEngine on
4     SSLCertificateFile /etc/apache2/ssl/server.crt
5     SSLCertificateKeyFile /etc/apache2/ssl/server.key
6
7     WSGIProcessGroup web2py
8
9     WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py
10
11 <Directory /users/www-data/web2py>
```

```

12     AllowOverride None
13     Order Allow,Deny
14     Deny from all
15     <Files wsgihandler.py>
16         Allow from all
17     </Files>
18 </Directory>
19
20 AliasMatch ^/([^/]+)/static/(.*) \
21     /users/www-data/web2py/applications/$1/static/$2
22
23 <Directory /users/www-data/web2py/applications/*/static/>
24     Order Allow,Deny
25     Allow from all
26 </Directory>
27
28 CustomLog /private/var/log/apache2/access.log common
29 ErrorLog /private/var/log/apache2/error.log
30
31 </VirtualHost>

```

重启 Apache，就能够进入：

```

1 https://www.example.com/admin
2 https://www.example.com/examples/appadmin
3 http://www.example.com/examples

```

而不是：

```

1 http://www.example.com/admin
2 http://www.example.com/examples/appadmin

```

13.1.5 mod_proxy 模块（代理模块）

一些 Unix/ Linux 发行版可以运行 Apache，但不支持 mod_wsgi 模块。在这种情况下，最简单的解决方案是作为代理运行 Apache，让 Apache 只处理静态文件。

下面是一个最低限度的 Apache 配置：

```

1 NameVirtualHost *:80
2 ##### deal with requests on port 80
3 <VirtualHost *:80>
4     Alias / /users/www-data/web2py/applications
5     ### serve static files directly
6     <LocationMatch "^/welcome/static/.*">
7         Order Allow, Deny
8         Allow from all
9     </LocationMatch>
10    ### proxy all the other requests
11    <Location "/welcome">
12        Order deny,allow
13        Allow from all
14        ProxyRequests off
15        ProxyPass http://localhost:8000/welcome

```

```

16     ProxyPassReverse http://localhost:8000/
17     ProxyHTMLURLMap http://127.0.0.1:8000/welcome/ /welcome
18 </Location>
19     LogFormat "%h %l %u %t \"%r\" %>s %b" common
20     CustomLog /var/log/apache2/access.log common
21 </VirtualHost>

```

上面的脚本仅公开了“welcome”这个应用程序，为了公开其他应用程序，需要使用像为“welcome”应用程序所做的那样，使用同样的语法添加相应的<Location>...</Location>。

这个脚本假定存在一个 web2py 服务器且在端口 8000 上运行，重启 Apache 之前，务必是在这种情况：

```

1     nohup python web2py.py -a '<recycle>' -i 127.0.0.1 -p 8000 &

```

可以使用 -a 选项指定一个密码，或使用“<recycle>”参数替代一个密码；后一种情况，先前存储的密码会被再次使用且密码也没有存储在 shell 历史记录中。

还可以使用参数“<ask>”，会提示输入密码。

当关闭此 shell 脚本时，nohup 命令能确保服务器不死掉，nohup 的所有记录都输出到的 nohup.out 中。

为了迫使基于 HTTPS 协议的 admin 和 appadmin 使用下面的 Apache 配置文件，替换：

```

1     NameVirtualHost *:80
2     NameVirtualHost *:443
3     ##### deal with requests on port 80
4     <VirtualHost *:80>
5         Alias / /usres/www-data/web2py/applications
6         ### admin requires SSL
7         <LocationMatch "^/admin">
8             SSLRequireSSL
9         </LocationMatch>
10        ### appadmin requires SSL
11        <LocationMatch "^/welcome/appadmin/.*">
12            SSLRequireSSL
13        </LocationMatch>
14        ### serve static files directly
15        <LocationMatch "^/welcome/static/.*">
16            Order Allow,Deny
17            Allow from all
18        </LocationMatch>
19        ### proxy all the other requests
20        <Location "/welcome">
21            Order deny,allow
22            Allow from all
23            ProxyPass http://localhost:8000/welcome
24            ProxyPassReverse http://localhost:8000/
25        </Location>
26        LogFormat "%h %l %u %t \"%r\" %>s %b" common
27        CustomLog /var/log/apache2/access.log common
28    </VirtualHost>
29    <VirtualHost *:443>
30        SSLEngine On
31        SSLCertificateFile /etc/apache2/ssl/server.crt
32        SSLCertificateKeyFile /etc/apache2/ssl/server.key
33        <Location "/">
34            Order deny,allow

```



```

35     Allow from all
36     ProxyPass http://localhost:8000/
37     ProxyPassReverse http://localhost:8000/
38 </Location>
39     LogFormat "%h %l %u %t \"%r\" %>s %b" common
40     CustomLog /var/log/apache2/access.log common
41 </VirtualHost>

```

当 web2py 通过 mod_proxy 模块运行在共享主机上时，管理界面必须被禁用，否则它将被暴露给其他用户。

13.1.6 开启 Linux 守护进程

只要正在使用 mod_wsgi 模块，就应该设置 web2py 的服务器，以便它可以启动/停止/重启任何其他 Linux 守护进程，因此它可以在电脑启动阶段自动启动。

这个进程的这种设置是特定于各种 Linux/ Unix 的发行版。

在 web2py 中的文件夹中，有两种可以用于此目的脚本：

```

1     scripts/web2py.ubuntu.sh
2     scripts/web2py.fedora.sh

```

在 Ubuntu 上，或其他基于 Debian 的 Linux 发行版上，编辑 “web2py.ubuntu.sh”，用你的 web2py 的安装路径取代 “/usr/lib/web2py” 路径，然后键入下面的 shell 命令，以便将文件移动到合适的文件夹，并将其注册为启动服务，再启动 web2py：

```

1     sudo cp scripts/web2py.ubuntu.sh /etc/init.d/web2py
2     sudo update-rc.d web2py defaults
3     sudo /etc/init.d/web2py start

```

在 Fedora 系统或基于 Fedora 的其他发行版，编辑 “web2py.fedora.sh”，用你的 web2py 安装路径取代 “/usr/lib/web2py” 路径，然后键入以下 shell 命令以便将文件移动到合适的文件夹，并注册它为启动服务，再启动 web2py：

```

1     sudo cp scripts/web2py.fedora.sh /etc/rc.d/init.d/web2pyd
2     sudo chkconfig --add web2pyd
3     sudo service web2py start

```

13.1.7 Lighttpd 开源服务器

用下面的 shell 命令，可以在 Ubuntu 或者其他基于 Debian 的 Linux 发行版上安装 Lighttpd 服务器：

```

1     apt-get -y install lighttpd

```

一旦安装完成，编辑 /etc/rc.local 并创建一个快速通用网关接口 (FCGI) web2py 后台进程。

```

1     cd /var/www/web2py && sudo -u www-data nohup python fcgihandler.py &

```

然后需要编辑 Lighttpd 服务器配置文件

```

1     /etc/lighttpd/lighttpd.conf

```

以便它能找到上面进程创建的套接字，在配置文件中，写入像这样代码：

```

1  server.modules          = (
2      "mod_access",
3      "mod_alias",
4      "mod_compress",
5      "mod_rewrite",
6      "mod_fastcgi",
7      "mod_redirect",
8      "mod_accesslog",
9      "mod_status",
10 )
11
12 server.port = 80
13 server.bind = "0.0.0.0"
14 server.event-handler = "frebsd-kqueue"
15 server.error-handler-404 = "/test.fcgi"
16 server.document-root = "/users/www-data/web2py/"
17 server.errorlog      = "/tmp/error.log"
18
19 fastcgi.server = (
20     "/handler_web2py.fcgi" => (
21         "handler_web2py" => ( #name for logs
22             "check-local" => "disable",
23             "socket" => "/tmp/fcgi.sock"
24         )
25     ),
26 )
27
28 $HTTP["host"] = "(^|.)*example.com$" {
29     server.document-root="/var/www/web2py"
30     url.rewrite-once = (
31         "^(/.+?/static/.+)$" => "/applications$1",
32         "(^|/.*)$" => "/handler_web2py.fcgi$1",
33     )
34 }

```

现在检查语法错误:

```
1  lighttpd -t -f /etc/lighttpd/lighttpd.conf
```

并启动（重启）web 服务器:

```
1  /etc/init.d/lighttpd restart
```

注意: FastCGI 绑定在 web2py 服务器的 Unix 套接字, 而不是 IP 套接字上。

```
1  /tmp/fcgi.sock
```

lighttpd 转发 HTTP 请求和接收响应, Unix 套接字较互联网套接字更轻, 这是 lighttpd+ FastCGI+ web2py 速度快的原因之一, 至于在 Apache 情况下, 设置 Lighttpd 直接去处理静态文件以及迫使一些应用使用 HTTPS 协议是可能的, 有关详细信息, 参考 Lighttpd 文档。

本节中的例子取自于 John Heenan 编写 web2py 程序片段。

(当 web2py 通过 mod_proxy 模块在共享主机上运行时, 管理界面必须被禁用, 否则, 它将被暴露给其他用户。)

13. 1. 8 具有 mod_python 模块的共享式主机

有时，尤其在共享式主机，当某一个没有直接配置 Apache 配置文件的权限，在这次写的过程当中，即使在 Apache 不再维持赞同 mod_wsgi 模块，大部分这种主机仍然可以运行 mod_python 模块。

仍然可以运行 web2py，下面的例子将告诉你如何设置它。

将 web2py 中的内容移到“htdocs”文件夹中。

在 web2py 文件夹中，创建一个“web2py_modpython.py”文件，其包含下面内容：

```
1 from mod_python import apache
2 import modpythonhandler
3
4 def handler(req):
5     req.subprocess_env['PATH_INFO'] = req.subprocess_env['SCRIPT_URL']
6     return modpythonhandler.handler(req)
```

用下面的内容 Create/update（创建或更新）文件“.htaccess”：

```
1 SetHandler python-program
2 PythonHandler web2py_modpython
3 #PythonDebug On
```

这个例子由 Niktar 提供。

13. 1. 9 具有 FastCGI 的 cherokee 服务器

Cherokee 是一款非常快速的 web 服务器，并且像 web2py 一样，它为自身配置提供了能启用基于 web 界面的 AJAX，它的 Web 界面是用 Python 编写的。此外，也不必重启所需的大多数变化。

这里是在 Cherokee 上设置 web2py 需要的步骤：

下载 Cherokee ^[90]

解压、编译和安装：

```
1 tar -xzf cherokee-0.9.4.tar.gz
2 cd cherokee-0.9.4
3 ./configure --enable-fcgi && make
4 make install
```

至少正常启动 web2py 一次，以确保它创建了“application”文件夹。

使用下面的代码编写 shell 脚本并命名为“startweb2py.sh”：

```
1 #!/bin/bash
2 cd /var/web2py
3 python /var/web2py/fcgihandler.py &
```

并给这些脚本执行特权和运行它，这将启动在 FastCGI 处理器下的 web2py。

启动 Cherokee 和 Cherokee 管理员：

```
1 sudo nohup cherokee &
2 sudo nohup cherokee-admin &
```

默认情况下，Cherokee 管理员只侦听端口 9090 上的本地接口，在那台机器上如果你有充足的物理访问接口，这不是一个问题。如果不是这种情况，你可以使用下面选项，强制它绑定到一个 IP 地址和端口。

```
1 -b, --bind[=IP]
2 -p, --port=NUM
```

或做一个 SSH 端口转发（更安全，推荐）：

```
1 ssh -L 9090:localhost:9090 remotehost
```

在你的浏览器上打开“http://localhost:9090”，如果一切顺利，将到达 cherokee-admin 界面。

在 cherokee-admin 的 web 界面中，单击“info sources”按钮，选择“Local Interpreter”选项，写入下面的代码，然后单击“Add New”按钮。

```
1 Nick: web2py
2 Connection: /tmp/fcgi.sock
3 Interpreter: /var/web2py/startweb2py.sh
```

最后，请执行以下剩余步骤：

- 单击“Virtual Servers”，然后单击“Default”。
- 单击“Behavior”，然后，在此，请单击“Default”。
- 选中“FastCGI”，而不是来自列表框中的“List and Send”。
- 在底部，选择“web2py 中”作为“Application Server”
- 在所有的复选框（可以留下 Allow-X-sendfile）勾选一个复选框，如果有警告显示、禁用和启用一个复选框。（它会自动重新提交应用程序服务器参数。有时没有，这是一个漏洞）。
- 将你的浏览器置为“http://yoursite”，那么“Welcome to web2py”将出现。

13.1.10 PostgreSQL 数据库

PostgreSQL 是一个免费的开源数据库，它在要求严格的生产环境中使用，例如，存储 the.org 域名数据库，并已被证明能很好地扩展到数百 TB 的数据，它有非常快速和坚实的事务支持，并提供了自动清理功能，使管理员从大部分的数据库维护任务中解放出来。

在 Ubuntu 或其他基于 Debian 的 Linux 发行版中，它很容易安装 PostgreSQL 和它的 Python 应用编程接口（API）：

```
1 sudo apt-get -y install postgresql
2 sudo apt-get -y install python-psycopg2
```

明智的做法是在不同的机器上运行 Web 服务器和数据库服务器，在这种情况下，运行 web 服务器的机器应该连接一个安全的内部（物理）网络，或者应该建立 SSL 隧道去安全地连接数据库服务器。

编辑 PostgreSQL 配置文件

```
1 sudo nano /etc/postgresql/8.4/main/postgresql.conf
```

并确保它包含这两行：

```
1 ...
```

```
2 track_counts = on
3 ...
4 autovacuum = on # Enable autovacuum subprocess? 'on'
5 ...
```

启动数据库服务器:

```
1 sudo /etc/init.d/postgresql restart
```

当重启 PostgreSQL 服务器时, 它应该知道哪一个端口在运行, 除非你有多个数据库服务器, 那它就应该 是 5432。

PostgreSQL 的日志在:

```
1 /var/log/postgresql/
```

一旦数据库服务器已启动并运行着, 创建一个用户和一个数据库, 以便 web2py 中应用程序可以使用它:

```
1 sudo -u postgres createuser -PE -s myuser
2 postgresql> createdb -O myself -E UTF8 mydb
3 postgresql> echo 'The following databases have been created:'
4 postgresql> psql -l
5 postgresql> psql mydb
```

第一个命令将授予超级用户访问权限给这个叫 myuser 的新用户, 它会提示输入密码。使用这个命令, 任何 web2py 应用程序都能够连接到这个数据库:

```
1 db = DAL("postgres://myuser:mypassword@localhost:5432/mydb")
```

当提示在我的密码处输入密码时, 数据库服务器正在 5432 端口处运行。

通常情况下, 你的每一个应用程序使用一个数据库, 同一应用程序的多个实例连接到同一个数据库。另外, 不同的应用程序共享同一数据库也是可能的。

对于数据库备份的详细信息, 请参阅 PostgreSQL 文档, 具体的命令即 pg_dump 和 pg_restore。

13.2 Windows 系统

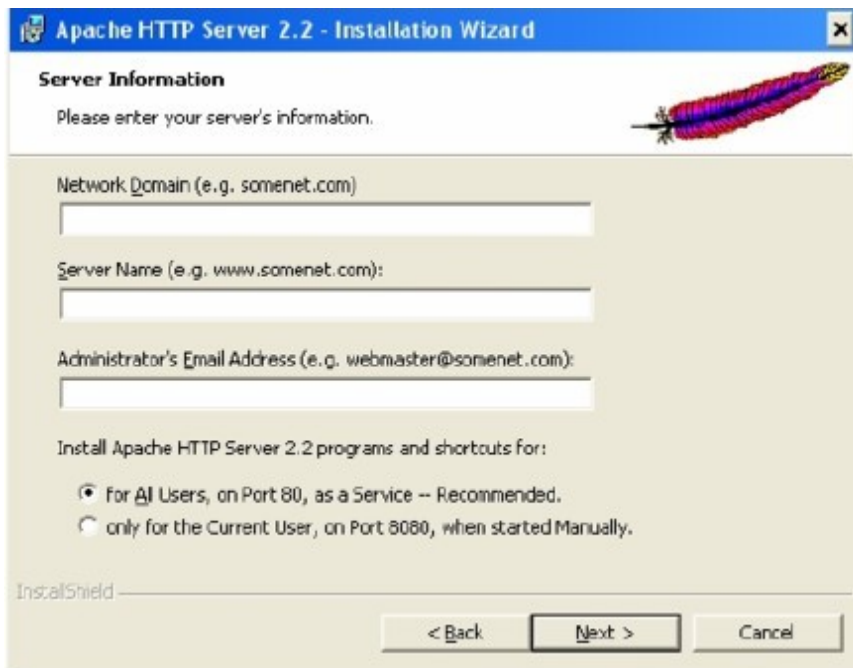
13.2.1 Apache 和 mod_wsgi

在 Windows 下安装 Apache 和 mod_wsgi 需要不同的程序, 这里假设 Python 2.5 被安装, 您正在运行来自源和位于 C :/ web2py 的 web2py。

首先下载所需要的包:

- 从参考文献^[83]下载 apache_2.2.11-win32-x86-openssl-0.9.8i.msi
- 从参考文献^[84]下载 mod_wsgi

第二, 运行 Apache...msi, 然后按照向导屏幕上的提示进行操作, 在服务器信息屏幕上:



输入所有请求的值：

- 网络域名：进入 DNS 域名，你的服务器或许是你的服务器注册在它里面。例如，如果你的服务器的 DNS 全名是 `server.mydomain.net`，可以键入 `mydomain.net`。
- 服务器名称：你的服务器是完整的 DNS 域名。从上面的例子中，你会在这里键入 `server.mydomain.net`，从安装的 `web2py` 而不是快捷方式，进入一个完全合格的域名或 IP 地址，更多详细信息，请参阅[86]。
- 管理员的电子邮件地址。进入服务器管理员或网站管理员的电子邮件地址，默认情况下，该地址以及错误消息一同显示给客户。

继续典型安装直到结束，除非有其他要求。

默认情况下，安装了 Apache 的向导在这个文件夹中：

```
1 C:/Program Files/Apache Software Foundation/Apache2.2/
```

从现在起，我们把这个文件夹简单称为 `Apache2.2`。

第三，拷贝下载的、并由 Chris Travers 编写的，于 2007 年 12 月在微软开源软件实验室发表的 `mod_wsgi.so` 到 `Apache2.2/modules` 文件夹中。

第四，创建 `server.crt` 和 `server.key` 证书（如前节讨论的那样），并把它们放到 `Apache2.2/conf` 文件夹中，注意把 `cnf` 文件放在 `Apache2.2/conf/openssl.cnf` 文件夹中。

第五，编辑 `Apache2.2/conf/httpd.conf`，从行中删除注释标记（# 字符）

```
1 LoadModule ssl_module modules/mod_ssl.so
```

添加下面的这行代码于所有其他 `LoadModule` 行之后

```
1 LoadModule wsgi_module modules/mod_wsgi.so
```

寻找“`listen80`”，加入下面代码于这一行后面

```
1 Listen 443
```

最后根据数值数据添加以下几行改变驱动器号、端口号、服务器名称

```
1 NameVirtualHost *:443
2 <VirtualHost *:443>
```

```

3      DocumentRoot "C:/web2py/applications"
4      ServerName server1
5
6      <Directory "C:/web2py">
7          Order allow,deny
8          Deny from all
9      </Directory>
10
11     <Location "/">
12         Order deny,allow
13         Allow from all
14     </Location>
15
16     <LocationMatch "^(/[\\w_]*static/.*)">
17         Order Allow,Deny
18         Allow from all
19     </LocationMatch>
20
21     WSGIScriptAlias / "C:/web2py/wsgihandler.py"
22
23     SSLEngine On
24     SSLCertificateFile conf/server.crt
25     SSLCertificateKeyFile conf/server.key
26
27     LogFormat "%h %l %u %t \"%r\" %>s %b" common
28     CustomLog logs/access.log common
29 </VirtualHost>

```

使用下面步骤保存和检查这些配置：“开始”>“程序”>“Apache HTTP 服务器 2.2”>配置 Apache 服务器>测试配置]

如果没有问题，你会看到一个命令窗口的打开和关闭，现在可以启动 Apache：

[“开始”>“程序”> Apache HTTP 服务器 2.2>控制 Apache 服务器>启动]或更好的便是启动任务栏上的显示器

[“开始”>“程序”> Apache HTTP 服务器 2.2>控制 Apache 服务器]

现在可以用鼠标右键单击红色羽毛般的任务栏图标“打开 Apache 监视器”，然后按照要求启动，停止和重启 Apache。

本节是由 Jonathan Lundell 编写。

13.2.2 开始为 Windows 服务

Linux 称之为守护进程，Windows 叫做服务，作为 Windows 服务，web2py 服务器能容易地被 installed/started/stopped（安装/启动/停止）。

为了把 web2py 作为一个 Windows 服务使用，必须创建文件“options.py”，其包含启动参数：

```

1      import socket, os
2      ip = socket.gethostname()
3      port = 80
4      password = '<recycle>'
5      pid_filename = 'httpserver.pid'
6      log_filename = 'httpserver.log'
7      ssl_certificate = "

```



```

8     ssl_private_key = "
9     numthreads = 10
10    server_name = socket.gethostname()
11    request_queue_size = 5
12    timeout = 10
13    shutdown_timeout = 5
14    folder = os.getcwd()

```

不需要从头开始创建“options.py”，因为在 web2py 的文件夹中已经有“options_std.py”文件，可以把它作为一个模型来使用。

在 web2py 安装文件夹中创建“options.py”文件后，可以把 web2py 作为一个服务安装：

```

1    python web2py.py -W install

```

并启动/停止这个服务：

```

1    python web2py.py -W start
2    python web2py.py -W stop

```

13.3 安全会话和 admin(管理)

除非 admin 应用程序和 appadmin 控制器通过 HTTPS 协议运行，否则公开暴露它们会是非常危险的，而且密码和证书绝不能不加密就传输，这对于 web2py 和其他 Web 应用程序都是对的。

在你的应用程序中，如果它们需要身份验证，应该使会话 cookie 安全，用：

```

1    session.secure()

```

在一个服务器上设置安全环境的一种容易方式就是先停止运行 web2py，然后删除来自 web2py 安装文件夹中的所有 parameters_*.py 文件，最后以一种无密码方式启动 web2py，这样就会完全禁止 admin 和 appadmin 运行。

```

1    nohup python web2py --nogui -p 8001 -i 127.0.0.1 -a '' &

```

接下来，启动只能从 localhost 访问的又一个 web2py 实例：

```

1    nohup python web2py --nogui -p 8002 -i 127.0.0.1 -a '<ask>' &

```

并从本地主机（你想访问的管理界面）到服务器（web2py 正在其上运行，如 example.com）建立一个 SSH 隧道，使用：

```

1    ssh -L 8002:127.0.0.1:8002 username@example.com

```

现在，通过在 localhost: 8002 上的 Web 浏览器，可以局部访问管理界面。

因为当隧道被关闭（用户退出登录）时，admin 不可访问，故此配置是安全的。

当且仅当其他用户不具有读取访问包含 web2py 文件夹的权限时，在共享主机上该解决方案才是安全的，否则用户可能直接从服务器上窃取会话 cookie。

13.4 高效性和扩展性

web2py 被设计得易于部署和设置，这并不意味着它折中了效率或可扩展性，但它意味

着可能需要调整它，以便其具有可扩展性。

在本节，我们假设多个 web2py 安装在提供本地负载均衡的 NAT 服务器后。

在这种情况下，如果某些条件得到满足，web2py 可以立即运行工作，尤其是每个基于 web2py 的应用程序的所有实例必须能访问同样数据库的服务器，且必须看到相同的文件，这后一种情况可以根据下面的文件夹共享实现：

```
1 applications/myapp/sessions
2 applications/myapp/errors
3 applications/myapp/uploads
4 applications/myapp/cache
```

共享文件夹必须支持文件锁定，可能的解决方案是 ZFS（ZFS 是 Sun Microsystems 开发的，也是最佳的选择。）、NFS（使用 NFS 时，可能需要运行 thenlockmgr 守护进程去允许文件锁定。）或 Samba（SMB）。

共享整个 web2py 文件夹或者整个应用文件夹是可能的，但却不是一个好主意，因为这样会引起网络带宽使用的无限增长。

我们认为上面讨论的配置是具有良好的扩展性的，因为它通过移动共享文件系统减少数据库的负载，这些资源需要能被共享却不需要安全的事务处理（一次只有一个客户端访问会话文件、缓存，且缓存总是需要一个全局锁，上传和错误一次性写入/读出多个文件）

理想情况是，数据库和共享存储器应该具有 RAID 功能，不要把存储在同一存储器上的数据库错误当着共享文件夹，否则会创建一个新的瓶颈。

在逐案基础下，可能需要执行额外的优化，我们将在下面讨论它们。特别是，我们将讨论如何摆脱这些一个接一个的共享文件夹，以及如何在数据库中存储相关数据，尽管这是可能的，然而它却不是一个好的解决方法，但是无论如何，仍然有理由让我们这样做，一个理由就是有时我们并不能随意的设置这些共享文件夹。

13.4.1 高效策略

web2py 应用程序代码是在每一个请求上执行，因此要最大限度地减少这种代码量，这里是你能做的：

- 运行一次 migrate= True，然后设置所有的表 migrate= FALSE。
- 字节码使用 admin 编译你的应用程序
- 可以尽量多的使用 cache.ram，但确保使用一组有限的键设置，否则缓存量的使用将任意的增加。
- 最小化模型中的代码：不要在那里定义函数，在控制器中定义，控制器需要它们——甚至更好的情况——定义模块中的函数，引进它们并按需使用这些函数。
- 不要把许多函数放在同一控制器中，而是使用许多含有少许函数的控制器。
- 调用在所有控制器中的 session.forget（响应）和/或调用没有改变会话的函数。
- 尽量避免使用 web2py 的 cron，而是使用一个后台进程，web2py 的 cron 会启动太多的 Python 实例，并引起过多的内存使用。

13.4.2 数据库中的会话

指示 web2py 存储会话于数据库中，而不存储于会话文件夹中，这是可能实现的，尽管它们都可能使用同一个数据库来存储会话，但是为每个单独的 web2py 应用程序这样做是必须的。

考虑一个数据库连接

```
1 db = DAL(...)
```

可以通过简单的下面表述来存储这些会话于这个数据库（db）中，在同一个模型文件中建立这个连接：

```
1 session.connect(request, response, db)
```

如果它已经不存在，在后台 web2py 在数据库中创建一个名为 web2py_session_appname 的表格，包含有以下字段：

```
1 Field('locked', 'boolean', default=False),
2 Field('client_ip'),
3 Field('created_datetime', 'datetime', default=now),
4 Field('modified_datetime', 'datetime'),
5 Field('unique_key'),
6 Field('session_data', 'text')
```

“unique_key”是一个 uuid 键，用于标识 cookie 中的会话，“session_data”是 cPickled 会话数据。

为了减少数据库访问，当会话不需要时，尽量避免存储它们，用：

```
1 session.forget()
```

有了这样的调整，“sessions”文件夹并不需要设置为共享文件夹，因为它将不再被访问。

注意： 如果会话被禁用，不准传递会话到 *form.accepts*，也不能使用 *session.flash* 和 *CRUD*。

13.4.3 HAProxy 一种高可用性、负载均衡器

如果需要多个 web2py 进程在多台机器上运行，而不是存储会话于数据库或者缓存中，可以选择使用粘性会话的负载均衡器。

Pound [92]和 haproxy[93]是两个使用 HTTP 的负载均衡和反向代理，其提供粘性会话，这里，我们讨论后者，因为在商业 VPS 主机中，它似乎使用更普遍。

我们的意思是，通过粘性会话，一旦一个会话 cookie 已经发出，负载均衡器总是路由那些来自与该会话关联的客户端请求到同一台服务器，这可以让存储会话于本地文件系统，而不必存于共享文件系统中。

要使用 HAProxy：

First, install it, on out Ubuntu test machine:

首先，安装它，就出现了 Ubuntu 的试验机：

```
1 sudo apt-get -y install haproxy
```

其次，向下面表述的一样编辑配置文件“/etc/haproxy.cfg”：

```
1  ## this config needs haproxy-1.1.28 or haproxy-1.2.1
2
3  global
4      log 127.0.0.1    local0
5      maxconn 1024
6      daemon
7
8  defaults
9      log        global
10     mode       http
11     option     httplog
12     option     httpchk
13     option     httpclose
14     retries    3
15     option     redispatch
16     timeout    5000
17     clitimeout 50000
18     srvtimeout 50000
19
20     listen 0.0.0.0:80
21         balance url_param WEB2PYSTICKY
22         balance roundrobin
23         server  L1_1 10.211.55.1:7003 check
24         server  L1_2 10.211.55.2:7004 check
25         server  L1_3 10.211.55.3:7004 check
26         appsession WEB2PYSTICKY len 52 timeout 1h
```

侦听指令告诉 HAProxy，哪个端口在等待连接。

服务器指令告诉 HAProxy 到哪去寻找代理服务器，为了实现这个目标，应用会话目录使用粘性会话和使用名为 WEB2PYSTICKY 的 cookie。

第三，启用此配置文件和启动的 HAProxy：

```
1  /etc/init.d/haproxy restart
```

可以找到类似的指示去安装 Pound，在 URL 上：

```
1  http://web2pyslices.com/main/slices/take_slice/33
```

13.4.4 清除会话

应该注意到，在生产环境中，会话快速堆积，web2py 提供了一个脚本调用：

```
1  scripts/sessions2trash.py
```

在后台运行时，定期删除在一定时间内没有被访问的所有会话， web2py 提供了一个脚本来清除这些会话（它适用于基于文件的会话和数据库会话）。

这里是一些典型的应用情况：

- 每五分钟删除过期会话

```
1  nohup python web2py.py -S app -M -R scripts/sessions2trash.py &
```

- 无论到期，删除超过 60 分钟的会话，详细输出，然后退出：

```
1 python web2py.py -S app -M -R scripts/sessions2trash.py -A -o -x 3600 -f -v
```

- 删除所有的会话，而不管到期与否并退出：

```
1 python web2py.py -S app -M -R scripts/sessions2trash.py -A -o -x 0
```

这里 app（应用程序）是你的应用程序的名称。

13.4.5 上传文件于数据库

默认情况下，所有上传被 SQLFORM 处理的文件，都将被安全地重命名并存储在“uploads”文件夹下的文件系统中，指示 web2py 存储上传的文件于数据库中是可能的。

现在，考虑下面的表：

```
1 db.define_table('dog',
2     Field('name')
3     Field('image', 'upload'))
```

在那儿 dog.image 的类型是上传，要使上传作为狗的名字的图片在同一个记录中，必须增加一个 blob 字段来修改表定义，并将其链接到这个上传字段：

```
1 db.define_table('dog',
2     Field('name')
3     Field('image', 'upload', uploadfield='image_data'),
4     Field('image_data', 'blob'))
```

这里的“image_data”只是一个新的 blob 字段中的任意名称。

第 3 行指示 web2py 像往常一样，去安全地重命名上传的图片，在图像领域中存储为新名称，并在名为“image_data”的 uploadfield 中存储这些数据，而不是存储在文件系统中，通过 SQLFORM 所有这一切都被自动完成，并且没有其他代码需要改变。

通过这个调整，“uploads”文件夹不再需要。

在 Google App Engine 中，文件被默认存储在数据库中，而不需要定义 uploadfield，因为它默认被创建。

13.4.6 收集票据

默认情况下，web2py 在本地文件系统中存储票据（错误），直接在数据库中存储票据是毫无意义的，因为在生产环境，最常见的错误来源是数据库失败。

存储票据绝不是一个瓶颈，因为这通常是一个罕见的事件，因此在生产环境中有多台并发的服务器，将它们存储在一个共享文件夹，它是绰绰有余的，不过，因为只有管理员才需取得票据，在当地一个非共享的“错误”文件夹中保存票据也是可行的，并定期收集和/或清除它们。

一种可能性就是周期的移动所有本地票据到数据库。

为了这个目的，web2py 提供了下面脚本：

```
1 scripts/tickets2db.py
```

默认情况下，脚本从保存在私有文件夹的文件(ticket_storage.txt)中得到 db uri，这个文件应该包含一个字符串，它被直接传递到一个 DAL 实例，像：

```

1 mysql://username:password@localhost/test
2 postgres://username:password@localhost/test
3 ...

```

这允许脚本保持不变：如果有多个应用程序，它会为每一个应用程序动态地选择正确的连接，如果想在它里面硬编码 uri，编辑第二个参考给 db_string，恰好在除线之后，可以使用这个命令运行该脚本：

```

1 nohup python web2py.py -S myapp -M -R scripts/tickets2db.py &

```

在那儿，myapp(我的应用)就是你的应用名。

这个脚本在后台运行，并且每 5 分钟将所有票据移动到一个表和删除本地的票据，稍后，可以使用管理应用程序查看错误，点击顶部的按钮“switch to: db”（“切换到：db”），有相同确切功能，好像它们被存储在文件系统中。

通过这个调整，“errors”文件夹不必也不再是共享文件夹，因为错误将被存在数据库中。

13.4.7 Memcache 缓存

我们已经表明，web2py 提供两种类型的缓存：cache.ram 和 cache.disk，它们都工作在具有多个并发服务器的分布式环境中，但他们并不如预期般地工作，特别是，cache.ram 将只缓存在服务器级别，因此它变成无用的了，cache.disk 也将缓存在服务器级别，除非“cache”文件夹是共享并支持锁的文件夹，那样，取代加速，它变成了一个主要的瓶颈。

这种解决方案不是使用它们，而是使用缓存代替，Web2py 附带了一个缓存 API。

为了使用缓存，创建一个新的模型文件，例如 0_memcache.py，并在这个文件中写入（或附加）下面的代码：

```

1 from gluon.contrib.memcache import MemcacheClient
2 memcache_servers = ['127.0.0.1:11211']
3 cache.memcache = MemcacheClient(request, memcache_servers)
4 cache.ram = cache.disk = cache.memcache

```

第一行导入 memcache，第二行是一个 memcache 套接字的列表（服务器：端口），第三行定义 cache.memcache，就 memcache 而言，第四行重新定义 cache.ram 和 cache.disk。

可以选择重新定义其中一个，用它去定义了一个全新的高速缓存对象指向的 Memcache 对象。

有了这个调整的“cache”文件夹也就不再需要共享文件夹了，因为它将不再被访问。

这个代码需要有 memcache 服务器在本地网络运行，关于如何设置这些服务器的信息，应该咨询 memcache 文档。

13.4.8 memcache 会话

如果确实需要会话并且又不想用粘性会话的负载平衡器，可以选择在 memcache 中存储会话：

```

1 from gluon.contrib.memdb import MEMDB
2 session.connect(request, response, db=MEMDB(cache.memcache))

```


13.4.9 Redis 的高速缓存

[103] 存储到 Memcache 的另一种方法是使用 Redis。

假设我们已经安装 Redis 和运行在本地主机端口 6379，我们可以使用下面的代码（模型）连接到它：

```
1 from gluon.contrib.redis import RedisCache
2 cache.redis = RedisCache('localhost:6379', db=None, debug=True)
```

在那儿，'localhost: 6379' 是连接字符串并且 db 不是一个 DAL 对象而是一个 Redis 的数据库名。

现在我们可以使用 cache.redis 取代（或者一起共存）cache.ram 和 cache.disk。

我们也可以通过调用获取 Redis 的统计信息：

```
1 cache.redis.stats()
```

13.4.10 删除应用

在产品设置中，不安装默认的应用程序，它可能会更好：admin、examples 和 welcome，虽然这些应用程序是相当小的，但它们不是必要的。

删除这些应用和删除在应用文件夹中的相应文件夹一样容易。

13.4.11 使用复制数据库

在高性能环境中，你可能有一个主从数据库架构，附带许多复制从站与可能到一对复制服务器，DAL 可以处理这种情况，并根据请求参数有条件地连接到不同的服务器，在第 6 章中描述的 API 能做到这一点。下面是一个例子：

```
1 from random import sample
2 db = DAL(sample(['mysql://...1', 'mysql://...2', 'mysql://...3'], 3))
```

这种情况下，不同 HTTP 请求将被不同的数据库随意服务，并且每一 DB 将以相同的概率或多或少受到冲击。

我们可以执行一个简单的 Round-Robin

```
1 def fail_safe_round_robin(*uris):
2     i = cache.ram('round-robin', lambda: 0, None)
3     uris = uris[i:] + uris[:i] # rotate the list of uris
4     cache.ram('round-robin', lambda: (i+1)%len(uris), 0)
5     return uris
6 db = DAL(fail_safe_round_robin('mysql://...1', 'mysql://...2', 'mysql://...3'))
```

在这个意义上讲，这是故障安全，即如果数据库服务器指派请求连接失败，DAL 将按顺序尝试下一个。

依赖于请求行为或者控制器来连接不同的数据库是可能的，在主从数据库配置中，一些行为表现为仅只读而一些人可读写，前者可以安全地连接到从 db 服务器，而后者应该连接主服务器。因此你可以做：


```

1     if request.function in read_only_actions:
2         db = DAL(sample(['mysql://...1','mysql://...2','mysql://...3'], 3))
3     if request.action in read_only_actions:
4         db = DAL(shuffle(['mysql://...1','mysql://...2','mysql://...3']))
5     else:
6         db = DAL(sample(['mysql://...3','mysql://...4','mysql://...5'], 3))

```

在此处，1、2、3 是从数据库服务器，3、4、5 是主数据库服务器。

13.5 部署在 Google App Engine

在 Google App Engine (GAE) ^[13] 上运行 web2py 代码是可能的，包括 DAL 代码。

GAE 支持两种版本的 Python: 2.5 (默认版) 和 2.7 (测试版)，web2py 支持两种，但默认情况下使用了 2.5 (这可能会在未来发生改变)，在下面描述的“app.yaml”文件中查看配置的细节。

GAE 还支持谷歌 SQL 数据库 (兼容 MySQL) 和谷歌 NoSQL (参照“Datastore”)，web2py 两者都支持，如果想使用谷歌 SQL 数据库请参照第 6 章的说明，本节指导您使用谷歌数据存储。

GAE 平台提供了正常的托管解决方案的几个优势：

- 易于部署，谷歌完全抽象化基础架构。
- 可扩展性，谷歌将多次复制你的应用程序，因为它需要服务所有并发请求。
- 可以在 SQL 和 NoSQL 数据库间选一个 (或两个都选)。

但也有一些缺点：

- 不能读或写访问文件系统
- 没有 HTTPS，除非你有 Google 证书就可使用 appspot.com 域

和一些数据存储特定缺点：

- 无典型事务处理。
- 无复杂的数据查询。特别是有没有 JOIN、LIKE 和 DATE / DATETIME 运算符。
- 无多个或子查询，除非它们涉及一个和相同字段。

因为这个只读文件系统，在文件系统中 web2py 不能存储会话、错误票据、缓存文件以及上传文件，它们必须存储在数据存储中，而不是在文件系统中。

这里，我们提供了一个 GAE 的快速概述并且我们关注在 web2py 中的具体问题，我们建议参考官方 GAE 的在线文档以求详细信息。

注意：在写的时候，GAE 只支持 Python 2.5，任何其他版本会出现的问题，您还必须运行 web2py 源码发布，而不是一个二进制发布。

13.5.1 配置

有三个配置文件需要注意：

```

1     web2py/app.yaml
2     web2py/queue.yaml
3     web2py/index.yaml

```

app.example.yaml 和 queue.example.yaml 作为出发点，通过使用模板文件，app.yaml 和 queue.yaml 是最容易创建的，index.yaml 被谷歌部署软件自动创建。

app.yaml 具有以下结构（它已经被使用... 缩短）：

```
1 application: web2py
2 version: 1
3 api_version: 1
4 runtime: python
5 handlers:
6 - url: /_ah/stats.*
7   ...
8 - url: /(?P<a>.+?)/static/(?P<b>.+?)
9   ...
10 - url: /_ah/admin/.
11   ...
12 - url: /_ah/queue/default
13   ...
14 - url: .*
15   ...
16 skip_files:
17 ...
```

app.example.yaml（当被复制到 app.yaml 时）被配置以便部署 web2py 的欢迎应用程序，但不是管理员或示例应用程序，当注册在 Google App Engine 时，必须更换 web2py 中你使用的应用程序 id。

url: 为了速度，/(.+?)/static/(.+?) 指示 GAE 直接服务于你的应用程序的静态文件，而无需调用 web2py 的逻辑。

url:.* 为了每个其他要求，指示 web2py 使用 gaehandler.py。

skip_files: 会话是正则表达式的文件列表，其并不需要部署到 GAE 上，特别的行：

```
1 (applications/(admin|examples)/.*)|
2 ((admin|examples|welcome).(w2p|tar))|
```

告知 GAE 不要部署默认应用程序，除了这解压后的 welcome 基本构建应用，可以添加更多的在这里被忽略的应用程序。

除了应用程序的 id 和版本，可能不必编辑 app.yaml，虽然你也许想排除这个欢迎应用程序。

文件 queue.yaml 用于配置 GAE 任务队列。

当运行你的本地使用的 GAE appserver（Web 服务器自带的 Google SDK）的应用程序时，文件 index.yaml 自动地生成，它包含了这样的内容：

```
1 indexes:
2 - kind: person
3   properties:
4     - name: name
5     direction: desc
```

在这个例子中，它告诉 GAE 创建一个索引，用于“person”表，其将被用于以“name”逆字母顺序排序的，没有相应的索引在应用程序中，将无法搜索和排序记录。

重要的是始终运行您的在本地附带 appserver 的应用程序，并在部署之前，尝试检验您的应用程序每个功能，对于测试目的来讲，这将是很重要，而且还自动生成“index.yaml”文件，有时候，你可能想编辑这个文件，并进行清理，如删除重复项。

13.5.2 运行和部署

Linux

在这里，我们假设已经安装了 GAE 的 SDK，在写本书时，GAE 运行在 Python2.5.2，可以使用 `appserver`（应用程序服务器）命令，运行来自 “web2py” 文件夹里面的你的应用程序：

```
1 python2.5 dev_appserver.py ../web2py
```

这将启动 `appserver` 以及可以在 URL 上运行您的应用程序：

```
1 http://127.0.0.1:8080/
```

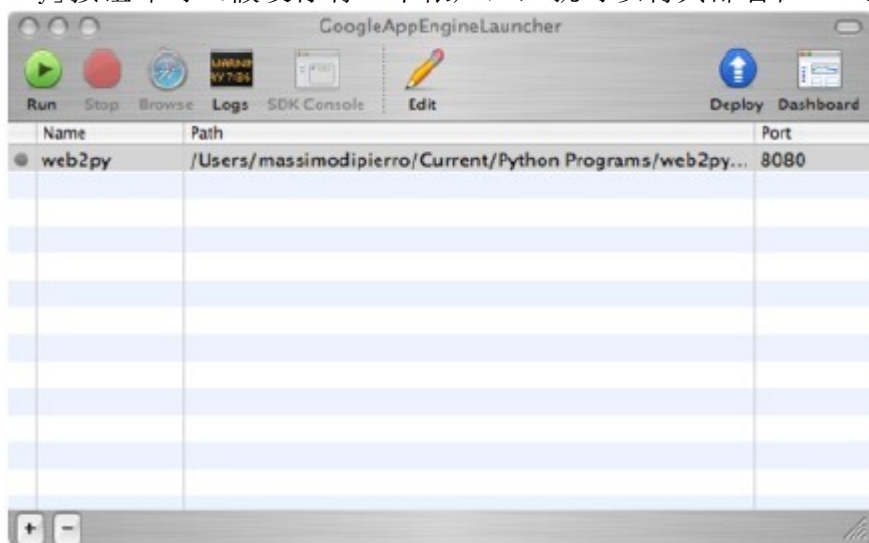
在 GAE 上，为了上传您的应用程序，确保已按照如之前解释那样编辑了 “`app.yaml`” 文件，并设置适当的应用程序 id，然后运行：

```
1 python2.5 appcfg.py update ../web2py
```

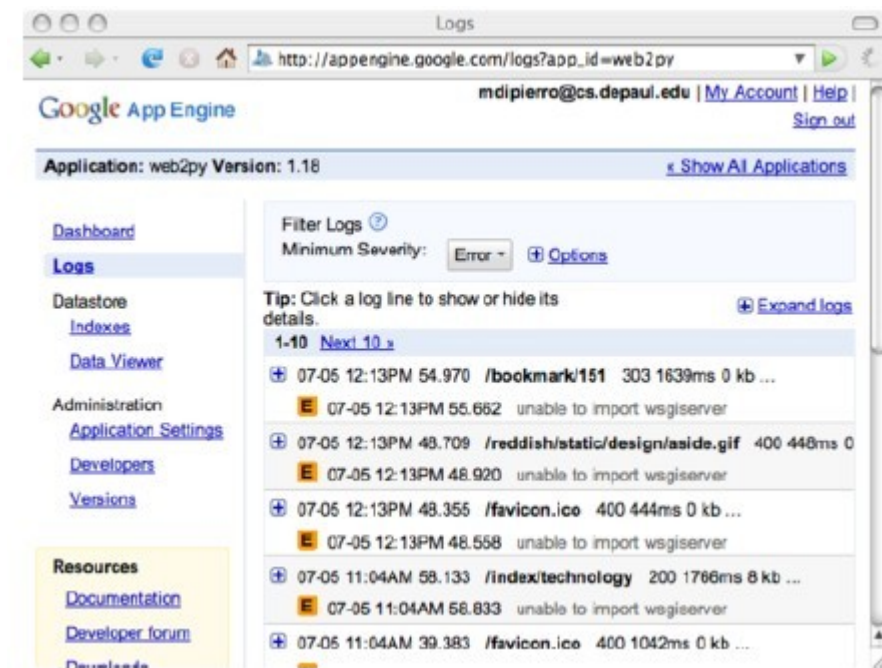
Mac, Windows

在 Mac 和 Windows 上，也可以使用 Google App Engine Launcher，可以从参考文献[13]中下载这个软件。

选择[File] [Add Existing Application]，设置路径到顶层的 web2py 文件夹路径，然后按在工具栏上的[Run]按钮，已测试后，它在本地上运行工作，只需点击在工具栏上的[Deploy]按钮即可（假设你有一个帐户），就可以将其部署在 GAE 上。



在 GAE 上，web2py 票据/错误也会被记录到 GAE 管理控制台上，在那儿，日志能被在线访问和搜索。



13.5.3 配置处理器

文件 `gaehandler.py` 负责在 GAE 上服务文件并且它有很少的选项，这儿是它们的默认值：

```
1 LOG_STATS = False
2 APPSTATS = True
3 DEBUG = False
```

`LOG_STATS` 将在 GAE 日志中记录网页的服务时间。

`APPSTATS` 将启用 GAE `appstats`，其提供分析统计，它们将在 URL 上可用：

```
1 http://localhost:8080/_ah/stats
```

`DEBUG` 设置调试模式，它实际上没有什么区别，除非通过 `gluon.settings.web2py_runtime` 在代码中明确检查。

13.5.4 避免文件系统

在 GAE 上，你没有权限访问文件系统，不能为了写而打开任何文件。

为了这个目的，在 GAE 上，`web2py` 自动地将所有上传文件存储于数据存储中，无论是否是“upload”字段都会有一个上传字段属性。

也应该将会话和票据存储在数据库中，并且必须明确：

```
1 if request.env.web2py_runtime_gae
2     db = DAL('gae')
3     session.connect(request, response, db)
4 else:
5     db = DAL('sqlite://storage.sqlite')
```

上面的代码是检测您是否在 GAE 上运行，是否连接到 BigTable，并且指示 `web2py` 将会

话和票据存储在这儿，另外它连接了一个嵌入式数据库，这个代码已经在文件“db.py”中的脚手架应用程序中。

13.5.5 Memcache 缓存

如果愿意，您可以将会话存储在 memcache 中：

```
1 from gluon.contrib.gae_memcache import MemcacheClient
2 from gluon.contrib.memdb import MEMDB
3 cache.memcache = MemcacheClient(request)
4 cache.ram = cache.disk = cache.memcache
5
6 db = DAL('gae')
7 session.connect(request, response, MEMDB(cache.memcache))
```

注意：在 GAE 上，cache.ram 和 cache.disk 不能被使用，所以我们让它们指向 cache.memcache。

13.5.6 数据存储问题

多实体事务处理和关系数据库典型功能的缺乏就使得 GAE 与其他主机环境相区别，这是支付高扩展性的价格，如果这些限制可忍受，GAE 是一个卓越的平台；如果不能，那应该考虑一个关系型数据库的普通主机平台。

如果一个 web2py 应用程序不在 GAE 上运行，那是因为上面讨论的某个限制，通过删除来自 web2py 队列的 JOIN 和去正常化数据库，许多问题可以解决。

Google App Engine 支持一些特定类型的字段，例如 ListProperty 和 StringListProperty，使用如下旧语法，可以使用这些类型的 web2py：

```
1 from gluon.dal import gae
2 db.define_table('product',
3     Field('name'),
4     Field('tags', type=gae.StringListProperty()))
```

或者等效的新的语法：

```
1 db.define_table('product',
2     Field('name'),
3     Field('tags', 'list:string'))
```

两种情况下，“tags”字段是一个 StringListProperty，因此它的值一定是字符串列表，兼容 GAE 文档，第二个表示法首选，因为 web2py 会在上下文的形式中以一种更智能的方法对待这个字段，而且因为它也将和关系型数据库一起工作。

相似的是，web2py 支持 list: interger 以及 list: reference，它们映射为一个 ListProperty(整型)。

list(列表)类型在第 6 章中有更详细讨论。

13.5.7 GAE 和 https

如果您的应用程序有 id “myapp”，你的 GAE 域名是：

```
1 http://myapp.appspot.com/
```

并且它也能通过 HTTPS 访问

```
1 https://myapp.appspot.com/
```

这种情况下，它将使用一个 Google 提供的 “appspot.com” 证书。

为了您的应用程序，可以注册一个 DNS 条目，并使用您拥有的任何其他域名，但在它上面将无法使用 HTTPS。在写本书时，这是一个 GAE 限制。

13.6 Jython

Web2py 通常在 Cpython 上（用 C 语言编写 Python 解释器）运行，但它也可以在 Jython 中（用 Java 编写的 Python 解释器）运行，这允许 web2py 在 Java 基础设施中去运行。

即便 web2py 在 Jython，开箱即用的运行，但仍有一些技巧涉及设置 Jython 和 zxJDBC（Jython 的数据库适配器）中。下面是这个指导：

- Jython.org 下载文件 “jython_installer-2.5.0.jar”（或 2.5.x）
- 安装它：

```
1 java -jar jython_installer-2.5.0.jar
```

- 从[101]下载安装 “zxJDBC.jar”
- 从[102]中下载安装文件 “sqlitejdbc-v056.jar”
- 添加 zxJDBC 和 sqlitejdbc 到 Java CLASSPATH
- 用 Jython 启动 web2py

```
1 /path/to/jython web2py.py
```

在写本书时，我们只支持在 Jython 上的 SQLite 和 Postgres。

第 14 章 其他方法

14.1 升级

在“site”页面的管理界面中有一个“upgrade now”按钮，以防这不可行，或不能正常工作（例如，因为一个文件锁定问题），手动升级 web2py 是很容易的。

只要解压缩最新版本的 web2py 覆盖旧版本的安装。

这将更新所有的库和应用程序：**admin**、**examples** 以及 **welcome**，它还将创建一个新的空文件“NEWINSTALL”，当重新启动时，web2py 将删除空文件和打包 welcome 应用程序到“welcome.w2p”，其将作为新的基本构建应用程序，web2py 不会升级在其他现有应用程序中的任何文件。

14.2 如何以二进制文件发布您的应用程序

用 web2py 二进制版本捆绑您的应用程序并一起发布是可能的，该许可证允许这，只要在您的应用程序的许可证中，您明确正在用 web2py 绑定并且添加一个到 web2py.com 的链接。

这里，我们解释如何为 Windows 实现：

- 像平常一样创建您的应用程序
- 使用 admin(管理)，字节码编译您的应用程序（点击一下）
- 使用 admin(管理)，打包您的编译过应用程序（再点击一下）
- 创建文件夹“myapp”
- 下载 web2py 的 windows 二进制发行版
- 在文件夹“myapp”中解压并启动它（双击）
- 使用 admin(管理)上传先前包装的和编译的名为“init”的应用程序（单击）
- 创建文件“myapp/start.bat”，其包含“cd web2py; web2py.exe”
- 创建文件“myapp/license”，其包含一个您的应用程序的许可证，并确保它表述为：它正在被“用来自 web2py.com 未经修改的 web2py 副本”发布。
- 压缩 myapp 文件夹为文件“myapp.zip”
- 发布和/或销售“myapp.zip”

当用户解压“myapp.zip”和单击“run”时，他们将看到您的应用程序而不是“welcome”应用程序，在用户侧没有要求，即便是 Python 的预安装。

对于 Mac 二进制发布过程是相同的，但不需要“bat”文件。

14.3 建立一个最低限度的 web2py

有时我们需要在很小内存占用的服务器上部署 web2py，这种情况下，我们想要拆分 web2py 直达到它的最低限度。

一种容易的方式就是照着下面去做：

- 在一个生产机器上，从源安装所有的 web2py
- 从内主要的 web2py 文件夹运行

```
1 python scripts/make_min_web2py.py /path/to/minweb2py
```

- 现在在“/path/to/minweb2py/applications”下拷贝您想部署的应用程序；
- 部署“/ path/to/minweb2py”到小足迹服务器；

脚本“make_min_web2py.py”构建一个最小 web2py 版本，其不包括：

- admin（管理）
- examples（实例）
- welcome（欢迎）
- scripts（脚本）
- 很少使用的 contrib 模块（rarely used contrib modules）

它确实不包括“welcome”应用程序，其包括一个简单的文件允许测试部署，寻找该脚本，在它顶部，它包含了一个详细的列表，其包含有那些应该有的和那些可以忽略不管的，你可以容易的修改它以及按照您的需要裁剪。

14.4 取出一个外部 URL

Python 包括用来抓取 url 的 urllib 库：

```
1 import urllib
2 page = urllib.urlopen('http://www.web2py.com').read()
```

这常常是很好的，但 urllib 模块在 Google App Engine 上不工作，Google 提供了一个不同的 API，其用于下载 URL，它只能在 GAE 上工作。

为了让代码便于移植，web2py 包括一个在 GAE 上工作的抓取函数，和其他的 Python 安装一样：

```
1 from google.tools import fetch
2 page = fetch('http://www.web2py.com')
```

14.5 漂亮的日期

用来表示日期不为“2009-7-25 14:34:56”，而是“one year ago”，它常常是有用的，web2py 对此提供了一个应用函数：

```
1 import datetime
2 d = datetime.datetime(2009, 7, 25, 14, 34, 56)
3 from gluon.tools import prettydate
4 pretty_d = prettydate(d, T)
```

第二个参数（T）必须被通过，以便允许国际化的输出。

14.6 地理编码 Geocoding

如果需要转换一个地址（例如：“243 S Wabash Ave, Chicago, IL, USA”）为地理坐标（经度和纬度），web2py 提供了一个函数来做到。

```
1 from gluon.tools import geocode
2 address = '243 S Wabash Ave, Chicago, IL, USA'
3 (latitude, longitude) = geocode(address)
```

这个函数 geocode（地理编码）要求一个网络连接和它连接用于这个地理编码的 Google 地理编码服务，在失败的情况下，这个函数返回（0,0）。注意：Google 地理编码服务覆盖了这些请求，所以应该检查他们的服务协议，geocode（地理编码）函数建立在抓取函数顶部并且这样它可以在 GAE 上运作。

14.7 分页

在分页的情况下，这个方法是一个尽量减少数据库访问的有用技巧，例如，当需要显示数据库中的一个行的列表，但想要在多个页面分发行时。

首先，创建一个素数的应用程序，它存储了在一个数据库中的前 1000 个素数。

这儿是 db.py 模型：

```
1 db = DAL('sqlite://primes.db')
2 db.define_table('prime',Field('value','integer'))
3 def isprime(p):
4     for i in range(2,p):
5         if p%i==0: return False
6     return True
7 if len(db().select(db.prime.id))==0:
8     p=2
9     for i in range(1000):
10         while not isprime(p): p+=1
11         db.prime.insert(value=p)
12         p+=1
```

现在，在“default.py”控制器中创建一个动作 list_items，它读起来像这样：

```
1 def list_items():
2     if len(request.args): page=int(request.args[0])
3     else: page=0
4     items_per_page=20
5     limitby=(page*items_per_page, (page+1)*items_per_page+1)
6     rows=db().select(db.prime.ALL, limitby=limitby)
7     return dict(rows=rows, page=page, items_per_page=items_per_page)
```

注意：这个代码选择了一个比需要更多的条目，20+1，额外的元素告诉视图是否存在下一个页面。

这儿是“default/list_items.html”视图：

```
1 {{extend 'layout.html'}}
2 {{for i,row in enumerate(rows):}}
3 {{if i==items_per_page: break}}
4 {{=row.value}}<br />
5 {{pass}}
6 {{if page:}}
```

```

7 <a href="{URL(args=[page-1])}">previous</a>
8 {{pass}}
9 {{if len(rows)>items_per_page:}}
10<a href="{URL(args=[page+1])}">next</a>
11{{pass}}

```

在这种方式中，我们已经用每个动作单个选取获得了分页，并且那个选取只选取比我们需要的多一行。

14.8 httpserver.log 和日志文件格式

Web2py 网页服务器日志所有的请求到一个文件，名为：

```
1 httpserver.log
```

在 web2py 根目录下，可以通过 web2py 命令行选项指定另一个可替换文的件名和位置。每次发出请求时，新的条目追加到该文件的尾部，每一行看起来像这样：

```
1 127.0.0.1, 2008-01-12 10:41:20, GET, /admin/default/site, HTTP/1.1, 200, 0.270000
```

其格式为：

```
1 ip, timestamp, method, path, protocol, status, time_taken
```

此处：

- ip 是发出请求用户的 IP 地址
- 时间戳是要求的日期和时间，用 ISO8601 格式表示，YYYYMM-DDT HH: MM: SS
- 方法是 GET 或 POST
- path 是用户请求的路径
- protocol（协议）是 HTTP 协议，其用于发送给用户，通常是 HTTP/1.1
- status 是 HTTP 状态编码中[95]的一个
- time_taken 是服务器获取请求进程花费的时间，以秒为单位，不包括上传/下载时间。

在应用库[34]，会发现一个用于日志分析的应用，当使用 mod_wsgi 时，默认情况下该日志不可用，因为它和 Apache 日志相同。

14.9 用虚拟数据填充数据库

出于测试的目的，它能很方便的用虚拟数据填充数据库表，web2py 包括一个经训练过的贝叶斯分类器，它生成虚拟的但可读的用于此目的文本。

这里是最简单地使用它的方式：

```

1 from gluon.contrib.populate import populate
2 populate(db.mytable, 100)

```

它将插入 100 个伪记录入 db.mytable，将尝试智能地做去完成，通过使用字符串字段生成简短的文本、用文本字段生成更长文本、整数、双精度、日期、日期时间、时间、布尔等相应字段，它会尝试遵守验证器提出的要求，因为字段包含单词 “name”，它会尝试生成假名称，对于参考字段，它将会产生有效的参考。

如果有两个表（A 和 B）且 B 参考 A，确保按照首先填入 A 和 B 次之的原则。

因为在一个事务处理中填入已经完成，不要试图马上填入太多的记录，特别是如果涉及到引用，相反，一次填入 100、提交、循环。

```

1 for i in range(10):
2     populate(db.mytable, 100)

```

可以使用贝叶斯分类器去学习一些文本和生成假文本，它听起来相似但应该是没有意义的：

```
1 from gluon.contrib.populate import Learner, IUP
2 ell=Learner()
3 ell.learn('some very long input text ...')
4 print ell.generate(1000,prefix=None)
```

14.10 接受信用卡付款

有多种方式接受信用卡在线付款，Web2py 提供专门的 API，用于最流行的一些和部分：

- Google 钱包 [96]
- 贝宝 paypalcite
- Stripe.com [98]
- Authorize.net [99]
- DowCommerece [100]

上述前两个机制委托验证受款人过程到外部服务，虽然这是最好的安全解决方案（您的应用程序不处理任何信用卡信息），但它使过程繁琐（用户必须登录两次，例如，一次与您的应用程序，另一次与谷歌）并不允许您的应用程序用一个自动化的方式来处理经常性支付。

有很多次，当您需要更多的控制以及要生成自己的信用卡信息的报名表时，而不是处理器编程要求将资金从信用卡转到您的帐户。

出于这个原因，web2py 提供了集成的开箱即用的 Stripe、Authorize.net（模块是由 John Conde 开发并略有修改）和 DowCommerce，Stripe 是最简单的操作，也是用于低交易量的方式的最便宜方式（他们不会收取任何固定费用，但每笔交易的成本大约 3%），Authorize.net 更好使用于高容量交易（每年有一个固定的成本，加上较低的交易费用）。

记住，在 Stripe 和 Authorize.net 的情况下，您的程序将接受信用卡信息，您不必存储这些信息并且我们建议不要存储，因为涉及到法律要求（检查 Visa 或 Mastercard），当有多少次您可能想要存储经常性收支或复制亚马逊的点击支付按钮的信息。

14.10.1 Google 钱包

使用 Google 钱包（一级）的最简单的方式包括嵌入一个按钮到您的网页，当点击时，您的访问者被重定向到一个由谷歌提供的支付页面。

首先，需要在 URL 上注册一个谷歌商业账户：

```
1 https://checkout.google.com/sell
```

需要提供您的银行信息给 Google，Google 将分配给您一个 merchant_id 和 merchant_key（不要混淆了它们，让它们处于保密状态）。

那么，仅需要在您的页面上创建下面的编码：

```
1 {{from gluon.contrib.google_wallet import button}}
2 {{=button(merchant_id="123456789012345",
3           products=[dict(name="shoes",
4                           quantity=1,
5                           price=23.5,
6                           currency='USD',
```

当访问者点击这个按钮时，访问者会被重定向到谷歌网页，在那儿他/她可以为该项目支付，这里，产品是一个产品列表，每个产品是参数的一本字典，一本您想通过它描述您的项目（名称、数量、价格、货币、说明和其他可选项，您可以在谷歌钱包文档找到这些）。

如果选择使用这种机制，您也许希望通过这个编程的基于您的库存和游客购物图的按钮生成这些值。

谷歌方面将处理所有的税金和运费信息，账户信息亦相同，默认情况下，您的应用程序不会被告知，该交易已经完成，因此必须访问您的 Google Merchant 网站，以便知道，哪款产品已购买和支付，以及哪款产品需要运送到您的买家，谷歌也将发送给您一封包含这些信息的电子邮件。

如果想要一个更紧密的集成，必须使用 2 级通知 API，在那种情况下，可以传递更多的信息到谷歌并且谷歌将调用您的 API 告知有关购买情况，这可以让您保持在您的应用程序的账户信息，但它要求公开 Web 服务，让其可以与谷歌钱包对话。

这是一个相当困难的问题，但这样的 API 已经实现，并且它可以作为插件形式：

```
1 http://web2py.com/plugins/static/web2py.plugin.google\_checkout.w2p
```

可以找到插件文档和该插件本身。

14.10.2 Paypal

这儿不描述 Paypal 集成，但是可以找到更多关于它的信息，在如下资源：

```
1 http://www.web2pyslices.com/main/slices/take\_slice/9
```

14.10.3 Stripe.com

对于接受信用卡付款，这可能是最简单的和灵活的方式中的一种。

需要在 Stripe.com 上注册而且那是非常容易的过程，实际上 Stripe 会分配给您一个 API 密钥甚至在您创建的任何凭证之前去尝试。

一旦有了 API 密钥，你、可以用下面的代码接受信用卡：

```
1 from gluon.contrib.stripe import Stripe
2 stripe = Stripe(api_key)
3 d = stripe.charge(amount=100,
4                   currency='usd',
5                   card_number='4242424242424242',
6                   card_exp_month='5',
7                   card_exp_year='2012',
8                   card_cvc_check='123',
9                   description='the usual black shoes')
10 if d.get('paid', False):
11     # payment accepted
12 elif:
13     # error is in d.get('error', 'unknown')
```

这个响应“d”是一本字典，一本可以暴露您自己的字典，在本例中使用的卡号是一个沙箱，并且它总是成功的，每一笔交易都与存储在 d['id'] 中的交易 id 有关联。

Stripe 也允许在以后的时间来验证一笔交易：

```
1 d = Stripe(key).check(d['id'])
```

和退款交易：

```
1 r = Stripe(key).refund(d['id'])
2 if r.get('refunded', False):
3     # refund was successful
4 elif:
5     # error is in d.get('error', 'unkown')
```

Stripe 很容易在您的应用程序中保持这账单。

您的应用程序与 Stripe 的所有通信都是基于 RESTful web 服务上的，实际上 Stripe 公开了更多的服务并提供了一个更大的 Python API 集，可以在他们的 web 网站上读到更多信息。

14.10.4 Authorize.Net

另一接受信用卡的简单方式就是使用 Authorize.net，像平常一样，需要注册并且您将获取一个登录名和交易的密钥（transkey），一旦有了它们，它工作基理非常像 Stripe：

```
1 from gluon.contrib.AuthorizeNet import process
2 if process(creditcard='4427802641004797',
3           expiration='122012',
4           total=100.0, cvv='123', tax=None, invoice=None,
5           login='cnpdev4289', transkey='SR2P8g4jdEn7vFLQ', testmode=True):
6     # payment was processed
7 else:
8     # payment was rejected
```

如果有一个有效的 Authorize.Net 帐户，应该用有关您帐户的内容更换沙箱中的登录名和 transkey，设置 testmode= False，以便它在真正的平台，而不是沙箱中运行，并使用访问者提供的信用卡信息。

如果 process（进程）返回为真，访问者信用卡账户中的钱就转到您的 Authorize.Net 账户，invoice（发票）只是一个字符串，一串您可以设置和将被 Authorize.Net 存储的字符串，有了这次交易以便您可以调和在您的应用程序中的数据信息。

这儿是一个更复杂的工作流实例，有更多的变量被公开：

```
1 from gluon.contrib.AuthorizeNet import AIM
2 payment = AIM(login='cnpdev4289',
3               transkey='SR2P8g4jdEn7vFLQ',
4               testmod=True)
5 payment.setTransaction(creditcard, expiration, total, cvv, tax, invoice)
6 payment.setParameter('x_duplicate_window', 180) # three minutes duplicate windows
7 payment.setParameter('x_cust_id', '1324')      # customer ID
8 payment.setParameter('x_first_name', 'Agent')
9 payment.setParameter('x_last_name', 'Smith')
10 payment.setParameter('x_company', 'Test Company')
11 payment.setParameter('x_address', '1234 Main Street')
12 payment.setParameter('x_city', 'Townsville')
13 payment.setParameter('x_state', 'NJ')
14 payment.setParameter('x_zip', '12345')
15 payment.setParameter('x_country', 'US')
16 payment.setParameter('x_phone', '800-555-1234')
17 payment.setParameter('x_description', 'Test Transaction')
18 payment.setParameter('x_customer_ip', socket.gethostbyname(socket.gethostname()))
19 payment.setParameter('x_email', 'you@example.com')
```



```

20 payment.setParameter('x_email_customer', False)
21
22 payment.process()
23 if payment.isApproved():
24     print 'Response Code: ', payment.response.ResponseCode
25     print 'Response Text: ', payment.response.ResponseText
26     print 'Response: ', payment.getResultResponseFull()
27     print 'Transaction ID: ', payment.response.TransactionID
28     print 'CVV Result: ', payment.response.CVVResponse
29     print 'Approval Code: ', payment.response.AuthCode
30     print 'AVS Result: ', payment.response.AVSResponse
31 elif payment.isDeclined():
32     print 'Your credit card was declined by your bank'
33 elif payment.isError():
34     print 'It did not work'
35 print 'approved', payment.isApproved()
36 print 'declined', payment.isDeclined()
37 print 'error', payment.isError()

```

注意：上面使用的编码是一个伪的测试账户，您需要在 Authorize.NetS（它不是一个免费的服务）上注册并且提供您的登录、transkey、testmode=True 或 False 给这个 AMI 构造函数。

14.11 Dropbox API

Dropbox 是一个非常流行的存储服务，它不仅存储您的文件而且保持与您的机器同步的云存储，允许您创建群并且把读/写变量文件夹的权利给予个人用户或群体；它也保持所有您的所有文件版本历史，包括文件夹“Public”和放在这儿的每一个文件都将有它自己的公共 URL。Dropbox 是一个进行协作的很好方式：

通过注册，可以轻松访问 dropbox：

```
1 https://www.dropbox.com/developers
```

您将获得 APP_KEY 和 APP_SECRET，一旦有了它们，就可以使用 Dropbox 来验证您的用户身份。

创建一个叫做“yourapp/private/dropbox.key”的文件，并在它里面写：

```
1 <APP_KEY>:<APP_SECRET>:app_folder
```

此处<APP_KEY> 和 APP_SECRET 是您的钥匙和密码。

然后在“models/db.py”中做：

```

1 from gluon.contrib.login_methods.dropbox_account import use_dropbox
2 use_janrain(auth, filename='private/dropbox.key')
3 mydropbox = auth.settings.login_form

```

这将允许用户使用他们的 dropbox 证书登录进您的应用程序，并且您的程序将会上传文件到他们的 dropbox 账户：

```

1 stream = open('localfile.txt', 'rb')
2 mydropbox.put('destfile.txt', stream)

```

下载文件：

```
1 stream = mydropbox.get('destfile.txt')
```



```
2 open('localfile.txt','wb').write(read)
```

并获取目录列表:

```
1 contents = mydropbox.dir(path = '/') ['contents']
```

14.12 Twitter API

这里是一些关于如何 post/get (发布/获取) tweet 的快速例子, 不需要第三方库, 因为 Twitter 使用简单的 RESful API。

这里是一个如何发布 tweet 的例子:

```
1 def post_tweet(username,password,message):
2     import urllib, urllib2, base64
3     import gluon.contrib.simplejson as sj
4     args= urllib.urlencode([('status',message)])
5     headers={}
6     headers['Authorization'] = 'Basic '+base64.b64encode(
7         username+':'+password)
8     req = urllib2.Request(
9         'http://twitter.com/statuses/update.json',
10        args, headers)
11     return sj.loads(urllib2.urlopen(req).read())
```

这里是一个如何接收 tweet 的例子:

```
1 def get_tweets():
2     user='web2py'
3     import urllib
4     import gluon.contrib.simplejson as sj
5     page = urllib.urlopen('http://twitter.com/%s?format=json' % user).read()
6     tweets=XML(sj.loads(page) ['#timeline'])
7     return dict(tweets=tweets)
```

更多复杂操作, 参考 Twitter API 文档。

14.13 流虚拟文件

恶意攻击者扫描网站的漏洞是很常见的, 它们使用像 Nessus 的安全扫描器去探索目标网站上的已知漏洞脚本, 从扫描机或直接在 Nessus 的数据库中的 Web 服务器日志分析, 揭示了大部分在 PHP 脚本和 ASP 脚本上的已知漏洞, 因为我们运行的是 web2py, 所以我们没有这些漏洞, 但是我们仍然会被扫描。这是令人烦恼的, 因此我们乐意回应那些漏洞扫描并让攻击者明白他们正在浪费时间。

一种可能性是重定向所有的 .php、.asp 请求以及任何可疑的到一个虚设行为, 通过响应这个攻击, 让攻击者花费大量的时间忙于此, 最后攻击者会放弃并不再扫描我们。

这个方案需要两部分。

一个被称为 jammer, 其附带 “default.py” 控制器干扰的专门应用程序, 定义如下:

```
1 class Jammer():
2     def read(self,n): return 'x'*n
3     def jam(): return response.stream(Jammer(),40000)
```

当这个动作被调用，它会一次用一个无限的充满“x”-es 的 40000 个字符数据流响应。

第二成分是一个“route.py”文件，该文件重定向任何以.php，.asp 等结束的请求（包括大写和小写）给这个控制器。

```
1         route_in=(  
2             ('.*.(php|PHP|asp|ASP|jsp|JSP)', 'jammer/default/jam'),  
3         )
```

第一次你被攻击，可能会产生一个小的开销，但我们的经验是，同一个攻击者不会再尝试第二次。

参考文献：

- [1] <http://www.web2py.com>
- [2] <http://www.python.org>
- [3] <http://en.wikipedia.org/wiki/SQL>
- [4] <http://www.sqlite.org/>
- [5] <http://www.postgresql.org/>
- [6] <http://www.mysql.com/>
- [7] <http://www.microsoft.com/sqlserver>
- [8] <http://www.firebirdsql.org/>
- [9] <http://www.oracle.com/database/index.html>
- [10] <http://www-01.ibm.com/software/data/db2/>
- [11] <http://www-01.ibm.com/software/data/informix/>
- [12] <http://www.ingres.com/>
- [13] <http://code.google.com/appengine/>
- [14] <http://en.wikipedia.org/wiki/HTML>
- [15] <http://www.w3.org/TR/REC-html40/>
- [16] <http://www.php.net/>
- [17] http://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
- [18] <http://www.python.org/dev/peps/pep-0333/>
- [19] <http://www.owasp.org>
- [20] <http://www.pythonsecurity.org>
- [21] http://en.wikipedia.org/wiki/Secure_Sockets_Layer
- [22] <https://launchpad.net/rocket>
- [23] <http://www.cdolivet.net/editarea/>
- [24] <http://nicedit.com/>
- [25] <http://pypi.python.org/pypi/simplejson>
- [26] <http://pyrtf.sourceforge.net/>
- [27] <http://www.dalkescientific.com/Python/PyRSS2Gen.html>
- [28] <http://www.feedparser.org/>
- [29] <http://code.google.com/p/python-markdown2/>
- [30] <http://www.tummy.com/Community/software/python-memcached/>
- [31] <http://www.gnu.org/licenses/lgpl.html>
- [32] <http://jquery.com/>
- [33] <https://www.web2py.com/cas>
- [34] <http://www.web2py.com/appliances>
- [35] <http://www.web2py.com/AlterEgo>
- [36] <http://www.python.org/dev/peps/pep-0008/>
- [37] <http://www.network-theory.co.uk/docs/pytut/>
- [38] <http://oreilly.com/catalog/9780596158071>
- [39] <http://www.python.org/doc/>
- [40] http://en.wikipedia.org/wiki/Cascading_Style_Sheets
- [41] <http://www.w3.org/Style/CSS/>
- [42] <http://www.w3schools.com/css/>
- [43] <http://en.wikipedia.org/wiki/JavaScript>
- [44] <http://www.amazon.com/dp/0596000480>
- [45] http://en.wikipedia.org/wiki/Cron#crontab_syntax
- [46] <http://www.xmlrpc.com/>

[47] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[48] <http://www.w3.org/Protocols/rfc2616/rfc2616.html>

[49] <http://www.modernizr.com/>

[50] <http://getskeleton.com/>

[51] <http://en.wikipedia.org/wiki/XML>

[52] <http://www.w3.org/XML/>

[53] <http://www.ez-css.org>

[54] <http://en.wikipedia.org/wiki/XHTML>

[55] <http://www.w3.org/TR/xhtml1/>

[56] <http://www.w3schools.com/xhtml/>

[57] <http://www.web2py.com/layouts>

[58] <http://sourceforge.net/projects/zxjdbc/>

[59] <http://pypi.python.org/pypi/psycopg2>

[60] <https://github.com/petehunt/PyMySQL>

[61] <http://sourceforge.net/projects/mysql-python>

[62] http://python.net/crew/atuining/cx_Oracle/

[63] <http://pyodbc.sourceforge.net/>

[64] <http://kinterbasdb.sourceforge.net/>

[65] <http://informixdb.sourceforge.net/>

[66] <http://pypi.python.org/simple/ingresdbi/>

[67] http://docs.python.org/library/csv.html#csv.QUOTE_ALL

[68] <http://www.faqs.org/rfcs/rfc2616.html>

[69] <http://www.faqs.org/rfcs/rfc2396.html>

[70] <http://tools.ietf.org/html/rfc3490>

[71] <http://tools.ietf.org/html/rfc3492>

[72] <http://mail.python.org/pipermail/python-list/2007-June/617126.html>

[73] <http://mail.python.org/pipermail/python-list/2007-June/617126.html>

[74] <http://www.recaptcha.net>

[75] http://en.wikipedia.org/wiki/Pluggable_Authentication_Modules

[76] <http://www.reportlab.org>

[77] <http://gdwarner.blogspot.com/2008/10/brief-pyjamal-django-tutorial.html>

[78] <http://en.wikipedia.org/wiki/AJAX>

[79] <http://www.learningjquery.com/>

[80] <http://ui.jquery.com/>

[81] http://en.wikipedia.org/wiki/Common_Gateway_Interface

[82] <http://www.apache.org/>

[83] <http://httpd.apache.org/download.cgi>

[84] http://adal.chiriliuc.com/mod_wsgi/revision_1018_2.3/mod_wsgi_py25_apache22/mod_wsgi.so

[85] http://httpd.apache.org/docs/2.0/mod/mod_proxy.html

[86] <http://httpd.apache.org/docs/2.2/mod/core.html>

[87] <http://sial.org/howto/openssl/self-signed>

[88] <http://code.google.com/p/modwsgi/>

[89] <http://www.lighthouse.net/>

[90] <http://www.cherokee-project.com/download/>

[91] <http://www.fastcgi.com/>

[92] <http://www.apsis.ch/pound/>

[93] <http://haproxy.lwt.eu/>

[94] <http://pyamf.org/>

[95] http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

[96] <https://wallet.google.com/manage>

[97] <https://www.paypal.com/>

[98] <https://stripe.com/>

- [99] <http://www.authorize.net/>
- [100] <http://www.dowcommerce.com/>
- [101] <http://sourceforge.net/projects/zxjdbc/>
- [102] <http://www.zentus.com/sqlitejdbc/>
- [103] <http://redis.io/>