# Odoo 10 Development Essentials

# Odoo10.0 中文开发手册 (2017版)

**作者:Daniel Reis**

**中文版翻译:**

**刘春涛(远鼎Maker),唐振宽,黄伟杰,彭 明,时威,庄春梅,胡 恒,陈厚源,孙保锁,**

**曾发军,季蜗牛,乔明洋,香港胡sir,高守庆,赵祥涛,罗英川布,台北蘇sir,苏州-n37r06u3**

**,石家庄-leo,周传红,福州- 南湘子,北京-生东击西,李旭**

修订记录:

杭州Chevrolet李旭 2018-03-02

目录

[Odoo 10 Development Essentials 1](#_Toc502671977)

[Odoo10.0 中文开发手册 (2017版) 1](#_Toc502671978)

[1 Odoo 大V社概况 9](#_Toc502671979)

[Odoo 大V社概况 9](#_Toc502671980)

[Odoo中文版技术书籍项目 9](#_Toc502671981)

[项目成员: 9](#_Toc502671982)

[统筹: 9](#_Toc502671983)

[审核: 9](#_Toc502671984)

[成员: 10](#_Toc502671985)

[2 前言 10](#_Toc502671986)

[3 第一章 Odoo 入门 11](#_Toc502671987)

[3.1 安装Odoo主机服务器 11](#_Toc502671988)

[3.1.1 准备Debian 主机 12](#_Toc502671989)

[3.1.2 为Odoo创建用户帐户 13](#_Toc502671990)

[3.2 从源码安装Odoo 14](#_Toc502671991)

[3.3 创建新的Odoo应用数据库 16](#_Toc502671992)

[3.3.1 管理Odoo数据库 18](#_Toc502671993)

[3.4 Odoo 版本 19](#_Toc502671994)

[3.5 查阅更多服务器选项 20](#_Toc502671995)

[3.5.1 Odoo 服务器配置文件 20](#_Toc502671996)

[3.5.2 更改侦听端口 21](#_Toc502671997)

[3.5.3 数据库的过滤选项 21](#_Toc502671998)

[3.5.4 管理服务器日志消息 22](#_Toc502671999)

[3.6 在工作站上进行开发 23](#_Toc502672000)

[3.6.1 使用Linux 文本编辑器 23](#_Toc502672001)

[3.6.2 安装和配置 Samba 23](#_Toc502672002)

[3.6.3 激活开发者工具 26](#_Toc502672003)

[3.7 安装第三方模块 27](#_Toc502672004)

[3.7.1 查找社区模块 27](#_Toc502672005)

[3.7.2 配置 addons 路径 28](#_Toc502672006)

[3.7.3 更新应用程序列表 28](#_Toc502672007)

[3.8 第一章 小结 28](#_Toc502672008)

[4 第二章 构建第一个应用程序 29](#_Toc502672009)

[4.1 基本概念 30](#_Toc502672010)

[4.1.1 了解应用程序和模块 30](#_Toc502672011)

[4.1.2 修改和扩展模块 31](#_Toc502672012)

[4.1.3 创建模块的基本框架 31](#_Toc502672013)

[4.1.4 关于许可证 34](#_Toc502672014)

[4.1.5 添加 addons 路径 34](#_Toc502672015)

[4.1.6 安装新模块 35](#_Toc502672016)

[4.1.7 升级模块 35](#_Toc502672017)

[4.1.8 服务器开发模式 36](#_Toc502672018)

[4.2 模型层 37](#_Toc502672019)

[4.2.1 创建数据模型 37](#_Toc502672020)

[4.2.2 加入自动化测试 39](#_Toc502672021)

[4.3 视图层 41](#_Toc502672022)

[4.3.1 添加菜单选项 41](#_Toc502672023)

[4.3.2 创建窗体视图 43](#_Toc502672024)

[4.3.3 业务凭证窗体视图 44](#_Toc502672025)

[4.3.4 添加动作按钮 45](#_Toc502672026)

[4.3.5 用groups来组织forms 45](#_Toc502672027)

[4.3.6 完整的Form视图 46](#_Toc502672028)

[4.3.7 添加列表和搜索视图 47](#_Toc502672029)

[4.4 业务逻辑层 48](#_Toc502672030)

[4.4.1 添加业务逻辑 48](#_Toc502672031)

[4.4.2 添加测试 49](#_Toc502672032)

[4.5 设置安全访问权限 50](#_Toc502672033)

[4.5.1 测试安全访问权限 50](#_Toc502672034)

[4.5.2 添加访问控制安全 51](#_Toc502672035)

[4.5.3 Row-level访问规则 52](#_Toc502672036)

[4.6 更好的模块描述 54](#_Toc502672037)

[4.7 第二章 小结 54](#_Toc502672038)

[5 第三章 继承–扩展现有的应用程序 55](#_Toc502672039)

[5.1 向待办事宜应用程序添加共享功能 55](#_Toc502672040)

[5.2 扩展模型 56](#_Toc502672041)

[5.2.1 向模型中添加字段 57](#_Toc502672042)

[5.2.2 修改现有字段 58](#_Toc502672043)

[5.2.3 修改模型的方法 59](#_Toc502672044)

[5.3 扩展视图 61](#_Toc502672045)

[5.3.1 扩展表单视图 63](#_Toc502672046)

[5.3.2 扩展树视图和搜索视图 64](#_Toc502672047)

[5.4 更多的模型继承机制 65](#_Toc502672048)

[5.4.1 使用原型继承复制特征 65](#_Toc502672049)

[5.4.2 使用委托继承嵌入模型 66](#_Toc502672050)

[5.4.3 添加社交网络功能 67](#_Toc502672051)

[5.5 修改数据 68](#_Toc502672052)

[5.5.1 修改菜单和操作记录 69](#_Toc502672053)

[5.5.2 修改安全记录规则 69](#_Toc502672054)

[5.6 第三章 小结 71](#_Toc502672055)

[6 第四章 模块数据 71](#_Toc502672056)

[6.1 了解外部标识符 72](#_Toc502672057)

[6.1.1 寻找外部标识符 73](#_Toc502672058)

[6.2 导出和导入数据 74](#_Toc502672059)

[6.2.1 导出数据 74](#_Toc502672060)

[6.2.2 导入数据 75](#_Toc502672061)

[6.2.3 CSV数据文件相关记录 76](#_Toc502672062)

[6.3 模块数据 78](#_Toc502672063)

[6.3.1 演示数据 78](#_Toc502672064)

[6.4 XML数据文件 79](#_Toc502672065)

[6.4.1 数据的noupdate属性 80](#_Toc502672066)

[6.4.2 在XML里定义记录 81](#_Toc502672067)

[6.4.3 常用模型的快捷键 83](#_Toc502672068)

[6.4.4 XML数据文件的其它操作 84](#_Toc502672069)

[6.5 第四章小结 85](#_Toc502672070)

[7 第五章 模型 - 结构化应用程序数据 86](#_Toc502672071)

[7.1 将应用程序特性组织到模块中 86](#_Toc502672072)

[7.1.1 todo\_ui模块介绍 87](#_Toc502672073)

[7.2 创建模型 88](#_Toc502672074)

[7.2.1 模型属性 89](#_Toc502672075)

[7.2.2 模型和Python类 89](#_Toc502672076)

[7.2.3 瞬态和抽象模型 90](#_Toc502672077)

[7.2.4 检查现有模型 91](#_Toc502672078)

[7.3 创建字段 92](#_Toc502672079)

[7.3.1 基本字段类型 92](#_Toc502672080)

[7.3.2 公共字段属性 94](#_Toc502672081)

[7.3.3 特殊字段名称 96](#_Toc502672082)

[7.4 模型之间的关系 97](#_Toc502672083)

[7.4.1 多对一关系 98](#_Toc502672084)

[7.4.2 多对多关系 99](#_Toc502672085)

[7.4.3 一对多反向关系 101](#_Toc502672086)

[7.4.4 分层关系 101](#_Toc502672087)

[7.4.5 使用动态关系的参考字段 102](#_Toc502672088)

[7.5 计算字段 103](#_Toc502672089)

[7.5.1 在计算字段上搜索和写入 104](#_Toc502672090)

[7.5.2 存储计算字段 105](#_Toc502672091)

[7.5.3 关联字段 105](#_Toc502672092)

[7.6 模型约束 106](#_Toc502672093)

[7.7 第五章 小结 107](#_Toc502672094)

[8 第六章 视图 - 设计用户界面 107](#_Toc502672095)

[8.1 使用XML文件定义用户界面 107](#_Toc502672096)

[8.1.1 菜单项 108](#_Toc502672097)

[8.1.2 窗口动作 110](#_Toc502672098)

[8.2 上下文和域 111](#_Toc502672099)

[8.2.1 上下文数据 112](#_Toc502672100)

[8.2.2 域表达式 112](#_Toc502672101)

[8.3 窗体视图 115](#_Toc502672102)

[8.3.1 处理同一类型的几个视图 115](#_Toc502672103)

[8.3.2 业务文档视图 116](#_Toc502672104)

[8.4 查看语义组件 121](#_Toc502672105)

[8.4.1 字段 121](#_Toc502672106)·

[8.4.2 按钮 124](#_Toc502672107)

[8.4.3 智能按钮 125](#_Toc502672108)

[8.5 动态视图 127](#_Toc502672109)

[8.5.1 动态属性 127](#_Toc502672110)

[8.6 列表视图 128](#_Toc502672111)

[8.7 搜索视图 129](#_Toc502672112)

[8.8 日历视图 131](#_Toc502672113)

[8.9 图形和枢轴视图 132](#_Toc502672114)

[8.10 其他视图类型 133](#_Toc502672115)

[8.11 第六章 小结 134](#_Toc502672116)

[9 第七章 ORM应用逻辑 - 支持业务流程 134](#_Toc502672117)

[9.1 创建向导 134](#_Toc502672118)

[9.1.1 向导模型 135](#_Toc502672119)

[9.1.2 向导窗体 136](#_Toc502672120)

[9.1.3 向导业务逻辑 137](#_Toc502672121)

[9.1.4 记录 139](#_Toc502672122)

[9.1.5 抛出异常 140](#_Toc502672123)

[9.1.6 向导中的助手操作 140](#_Toc502672124)

[9.2 使用ORM API 142](#_Toc502672125)

[9.2.1 装饰器方法 142](#_Toc502672126)

[9.2.2 重写 ORM 的默认方法 143](#_Toc502672127)

[9.2.3 RPC和Web客户端调用的方法 144](#_Toc502672128)

[9.2.4 shell命令 146](#_Toc502672129)

[9.2.5 服务器环境 147](#_Toc502672130)

[9.2.6 修改执行环境 148](#_Toc502672131)

[9.2.7 事务和底层SQL 148](#_Toc502672132)

[9.3 使用记录集 150](#_Toc502672133)

[9.3.1 查询模型 150](#_Toc502672134)

[9.3.2 单例模式 151](#_Toc502672135)

[9.3.3 写在记录上 151](#_Toc502672136)

[9.3.4 使用时间和日期 153](#_Toc502672137)

[9.3.5 对记录集的操作 154](#_Toc502672138)

[9.3.6 操作记录集 155](#_Toc502672139)

[9.3.7 使用关系字段 157](#_Toc502672140)

[9.3.8 处理关系字段 157](#_Toc502672141)

[9.4 第七章 小结 158](#_Toc502672142)

[10 第八章 编写测试和调试代码 158](#_Toc502672143)

[10.1 单元测试 158](#_Toc502672144)

[10.2 添加单元测试 159](#_Toc502672145)

[10.3 写测试用例 161](#_Toc502672146)

[**注意** 162](#_Toc502672147)

[10.4 设置测试 162](#_Toc502672148)

[10.5 测试异常 164](#_Toc502672149)

[10.6 运行测试 165](#_Toc502672150)

[10.7 关于YAML测试 165](#_Toc502672151)

[10.8 开发工具 167](#_Toc502672152)

[10.8.1 服务器开发选项 167](#_Toc502672153)

[**注意** 168](#_Toc502672154)

[**$ pip install watchdog** 168](#_Toc502672155)

[**注意** 168](#_Toc502672156)

[10.8.2 调试 168](#_Toc502672157)

[**小贴士** 169](#_Toc502672158)

[10.8.3 检查运行过程 174](#_Toc502672159)

[10.9 第八章 小结 175](#_Toc502672160)

[11 第九章 QWeb和Kanban视图 175](#_Toc502672161)

[11.1 关于Kanban 175](#_Toc502672162)

[11.1.1 Kanban视图 176](#_Toc502672163)

[11.2 设计Kanban视图 178](#_Toc502672164)

[11.2.1 优先级，Kanban状态和颜色 179](#_Toc502672165)

[11.2.2 Kanban卡元素 180](#_Toc502672166)

[11.2.3 Kanban卡布局 182](#_Toc502672167)

[11.2.4 添加Kanban卡选项菜单 184](#_Toc502672168)

[11.2.5 Kanban视图中的动作 185](#_Toc502672169)

[11.3 QWeb模板语言 186](#_Toc502672170)

[11.3.1 QWeb JavaScript评估上下文 187](#_Toc502672171)

[11.3.2 使用t-attf进行属性字符串替换 188](#_Toc502672172)

[11.3.3 使用t-att作为动态属性 189](#_Toc502672173)

[11.3.4 使用t-foreach循环 189](#_Toc502672174)

[11.3.5 使用t-if进行条件渲染 191](#_Toc502672175)

[11.3.6 使用t-esc和t-raw来渲染值 191](#_Toc502672176)

[11.3.7 使用t-set设置变量的值 192](#_Toc502672177)

[11.3.8 使用t-call插入其他模板 193](#_Toc502672178)

[11.3.9 更多的方法来使用t-attf 194](#_Toc502672179)

[11.4 Kanban视图的继承 195](#_Toc502672180)

[11.5 自定义CSS和JavaScript 196](#_Toc502672181)

[11.6 第九章 小结 197](#_Toc502672182)

[12 第十章 创建QWeb报告 197](#_Toc502672183)

[12.1 安装wkhtmltopdf 198](#_Toc502672184)

[12.2 创建业务报告 199](#_Toc502672185)

[**小贴士** 200](#_Toc502672186)

[12.3 QWeb报告模板 201](#_Toc502672187)

[12.4 在报告中提供数据 202](#_Toc502672188)

[**注意** 203](#_Toc502672189)

[12.5 渲染图像 205](#_Toc502672190)

[12.6 总计摘要和运行总计 205](#_Toc502672191)

[12.7 定义纸张格式 207](#_Toc502672192)

[**小贴士** 208](#_Toc502672193)

[12.8 在报告中启用语言翻译 208](#_Toc502672194)

[12.9 基于自定义SQL的报表 209](#_Toc502672195)

[12.10 第十章 小结 211](#_Toc502672196)

[13 第十一章 创建网站前端功能 211](#_Toc502672197)

[13.1 路线图 212](#_Toc502672198)

[13.2 我们的第一个网页 212](#_Toc502672199)

[13.2.1 Hello World！ 213](#_Toc502672200)

[13.2.2 Qweb模板的Hello World！ 214](#_Toc502672201)

[**注意** 215](#_Toc502672202)

[**小贴士** 215](#_Toc502672203)

[13.2.3 扩展Web功能 216](#_Toc502672204)

[13.2.4 HelloCMS！ 219](#_Toc502672205)

[**注意** 220](#_Toc502672206)

[13.3 建设网站 220](#_Toc502672207)

[13.3.1 添加CSS和JavaScript 221](#_Toc502672208)

[13.3.2 待办事项列表控制器 222](#_Toc502672209)

[13.3.3 待办事项列表模板 223](#_Toc502672210)

[13.3.4 待办事宜任务详细信息页面 225](#_Toc502672211)

[13.4 网站表单 226](#_Toc502672212)

[13.4.1 表单页面 227](#_Toc502672213)

[13.4.2 访问安全和菜单项 231](#_Toc502672214)

[13.4.3 添加自定义逻辑 232](#_Toc502672215)

[13.5 第十一章 小结 233](#_Toc502672216)

[14 第十二章 外部API - 与其他系统集成 233](#_Toc502672217)

[14.1 设置一个Python客户端 234](#_Toc502672218)

[**注意** 235](#_Toc502672219)

[14.2 使用XML-RPC调用Odoo API 236](#_Toc502672220)

[14.2.1 打开XML-RPC连接 236](#_Toc502672221)

[14.2.2 从服务器读取数据 237](#_Toc502672222)

[14.2.3 调用其他方法 239](#_Toc502672223)

[14.3 编写Notes桌面应用程序 240](#_Toc502672224)

[14.3.1 Odoo的通信层 240](#_Toc502672225)

[14.3.2 创建GUI 242](#_Toc502672226)

[14.4 介绍ERPpeek客户端 245](#_Toc502672227)

[14.4.1 ERPpeek API 245](#_Toc502672228)

[14.4.2 ERPpeek CLI 246](#_Toc502672229)

[14.5 第十二章 小结 248](#_Toc502672230)

[15 第十三章 实战部署 248](#_Toc502672231)

[15.1 可用的预构建软件包 249](#_Toc502672232)

[15.2 安装依赖关系 249](#_Toc502672233)

[15.2.1 准备专用系统用户 251](#_Toc502672234)

[15.2.2 从源代码安装 252](#_Toc502672235)

[**小贴士** 252](#_Toc502672236)

[15.2.3 设置配置文件 253](#_Toc502672237)

[**小贴士** 253](#_Toc502672238)

[15.3 多线程操作 255](#_Toc502672239)

[15.3.1 设置系统服务 256](#_Toc502672240)

[15.3.2 创建systemd服务 257](#_Toc502672241)

[15.3.3 创建Upstart/sysvinit服务 258](#_Toc502672242)

[15.3.4 用命令行检查Odoo服务 260](#_Toc502672243)

[15.4 使用反向代理 261](#_Toc502672244)

[15.4.1 设置Nginx反向代理 262](#_Toc502672245)

[15.4.2 执行HTTPS 264](#_Toc502672246)

[**注意** 264](#_Toc502672247)

[**小贴士** 266](#_Toc502672248)

[15.4.3 Nginx优化 266](#_Toc502672249)

[15.4.4 Long polling长轮循 267](#_Toc502672250)

[15.5 服务器和模块更新 268](#_Toc502672251)

[**小贴士** 269](#_Toc502672252)

[15.6 第十三章 小结 270](#_Toc502672253)

# 1 Odoo 大V社概况

## Odoo 大V社概况

Odoo 大V社成员来自五湖四海，他们热爱开源，无私奉献，甘于把自己的业余时间投入到公益的 Odoo 相关资料的整理中去，

为 Odoo 在中国区的推广无私奉献。

odoo 中文书籍交流群:343593855(验证信息:参与odoo中文书籍编写), 不参加书籍编写勿加

odoo 实施定制交流群:446389096(验证信息:odoo 实施), 开发实施定制交流的加这个群

## Odoo中文版技术书籍项目

项目一期目标: Odoo 中文开发手册

项目性质：公益，保留所有参与人员的署名权；项目成果归全人类

项目使命：Odoo 中文书籍编写整理，推倒 odoo 语言壁垒，助力 odoo 中国推广

## 项目成员:

### 统筹:

01.广州-唐振宽(稿件初审) 02.广州-黄伟杰(任务分派) 03.广州-彭 明(进度跟踪)

### 审核:

01.苏州-刘春涛(远鼎Maker)

### 成员:

01.苏州时 威 02.苏州庄春梅 03.杭州胡 恒 04.广州陈厚源

05. 06.苏州孙保锁 07.成都曾发军 08.深圳季蜗牛

09.北京乔明洋 10.香港胡sir 11.上海高守庆 12.上海赵祥涛

13.上海罗英川布 14.台北蘇sir 15.苏州-n37r06u3 16.石家庄-leo

17.武汉-周传红 18.福州- 南湘子 19.北京-生东击西 20.杭州-李旭

# 2 前言

Odoo 是一款功能强大，面向企业应用的ERP开源软件开发平台。基于 odoo 紧密集成，包含了从 CRM、到销售、到库存、到会计等一系列企业应用。

Odoo 还建立了一个围绕它动态扩展的社区，不断增加新的功能、新的连接器和附加的企业增值应用。

本书以逐步推进的方式，提供了开发指导，可以让读者高效的使用 Odoo 应用开发平台，缩短学习时间。

前两章目的使读者熟悉 odoo系统,学习 odoo 建立开发环境的基本技术，熟悉模块开发方法和工作流程的配置。

后续的章节详细介绍 odoo 模块开发需要了解的各个主题，比如继承机制、扩展、数据文件，模型、视图、业务逻辑，等等。

最末一章介绍在生产环境中部署 odoo 实例需要考虑的问题。

# 3 第一章 Odoo 入门

我们在进行Odoo开发之前,需要先设置我们的开发环境和学习odoo的基本管理任务。

在这一章,我们将学习如何设置工作环境,并在这之后建立我们的Odoo应用程序。我们将学习如何建立一个Debian或Ubuntu系统主机开发服务器实例，以及如何从GitHub以源代码的方式安装Odoo。

然后,我们将学习如何设置Samba文件共享, samba 会允许我们使用运行windows的工作站或者其他操作系统来管理“ Odoo” 的程序文件。

Odoo是使用Python编程语言构建的,它使用PostgreSQL数据库数据存储;这也是Odoo服务器的两个主要配置要求。要想从源代码运行Odoo,我们将首先需要安装Pythony依赖库。

Odoo的源代码可以从GitHub下载。虽然我们可以下载ZIP文件或tar包,但是我们会发现使用git版本控制程序会更好，它会帮助我们把源码下载到我们的Odoo主机上。

## 3.1 安装Odoo主机服务器

我们在进行Odoo开发之前,需要先设置我们的开发环境和学习odoo的基本管理任务。

在这一章,我们将学习如何设置工作环境,并在这之后建立我们的Odoo应用程序。我们将学习如何建立一个Debian或Ubuntu系统主机开发服务器实例，以及如何从GitHub以源代码的方式安装Odoo。

然后,我们将学习如何设置Samba文件共享, samba 会允许我们使用运行windows的工作站或者其他操作系统来管理“ Odoo” 的程序文件。

Odoo是使用Python编程语言构建的,它使用PostgreSQL数据库数据存储;这也是Odoo服务器的两个主要配置要求。要想从源代码运行Odoo,我们将首先需要安装Pythony依赖库。

Odoo的源代码可以从GitHub下载。虽然我们可以下载ZIP文件或tar包,但是我们会发现使用git版本控制程序会更好，它会帮助我们把源码下载到我们的Odoo主机上。

### 3.1.1 准备Debian 主机

正如前面所介绍的，我们需要为Odoo服务器准备一台基于Debian操作系统的主机。 如果这是您初次接触Linux，您可能需要注意Ubuntu是一个基于Debian的Linux发行版，因此它们非常相似。

Odoo与目前稳定版的Debian和Ubuntu运行良好。本书写作时，它们的版本分别是：Debian 8 "Jessie", Ubuntu 16.04.1 LTS (Xenial Xerus)。它们都装有运行Odoo所必需的Python 2.7。值得注意的是，Odoo10.0及以前版本,还不支持Python 3。

如果您已经运行了Ubuntu或其他基于Debian的发行版，那么也是可以的，这也可以用作Odoo的主机。 对于Windows和Mac操作系统，请安装Python 2.7, PostgreSQL以及它们的所有依赖包。然后从本地源运行Odoo。然而配置过程可能有点困难。我们建议使用虚拟机来运行Debian或Ubuntu服务器。您也可以在虚拟机中选择自己喜欢的虚拟化软件来安装配置Debian系统。 如果您需要一些指导，这里有一些关于虚拟化软件的建议。

可供选择的有：

1. Microsoft Hyper-V（目前Windows server 2008、windows server 2012，windows server 2016,win8,win10系统中某些版本可用）

2. Oracle VirtualBox，下载地址是：https://www.virtualbox.org/

3. VMWare Workstation Player (VMWare Fusion for Mac)

VMWare Workstation Player可能使用起来比较简单，免费使用的下载地址 https://my.vmware.com/web/vmware/downloads

关于使用的Linux镜像，安装Ubuntu服务器比安装Debian更人性化。如果您对Linux不太熟悉，建议您尝试可立即使用的镜像。TurnKey Linux提供了易于使用的预装镜像，并有几种格式（包括ISO）。ISO格式都可以运行，甚至裸机也可以。无论您选择什么虚拟化软件，LAPP镜像是一个比较不错的选择，它包含了Python和PostgreSQL，下载地址 http://www.turnkeylinux.org/lapp 只要安装并启动，您就可以登录到命令行界面。

### 3.1.2 为Odoo创建用户帐户

如果正在使用有超级管理员权限的root账号登录，考虑到用root 用户运行不是一个好主意，首先应该为工作环境创建一个普通用户账号。实际上，当使用root用户启动odoo 服务器时，它也会拒绝运行。 如果正在使用Ubuntu，可能不需要这个步骤，安装过程中已经引导建立了一个用户。 首先，确认系统安装了sudo，odoo系统的工作需要这个程序。 如果是使用root登录的，执行下面的命令:

**apt-get update && apt-get upgrade # 安装系统更新**

**apt-get install sudo # 确保 sudo 程序已安装**

下面的命令将建立odoo用户：

**useradd -m -g sudo -s /bin/bash odoo # 创建'odoo' 用户并授予使用sudo的权限**

**passwd odoo # 设置新用户密码**

可以将odoo更改为可能想要的任何用户名。

-m选项 确保其主目录已创建。

-g sudo选项将其添加到sudoers 列表，因此它可以用root权限运行命令。

-s / bin / bash选项设置 默认shell到bash，这比使用默认sh更好。

完成以上操作，现在就可以用新用户登录并设置Odoo。

## 3.2 从源码安装Odoo

可以在 http://nightly.odoo.com/ 上找到准备安装的Odoo软件包，可以使用Windows（.exe），Debian（.deb），CentOS（.rpm）和源代码tarball（.tar.gz）。作为开发人员，我们更喜欢直接从 GitHub 代码库安装它们。这将最终使我们更好的控制 odoo 版本和更新。 为了保持整洁，让我们在我们的主目录下的 /odoo-dev 目录下工作。

注意

在本书中，我们将假定/odoo-dev 是安装 Odoo 服务器的目录。

首先，确保您以我们现在创建的用户或在安装过程中创建的用户登录，而不是以 root 身份登录。假设您的用户是 odoo，请使用以下命令进行确认：

**$ whoami**

**odoo**

**$ echo $HOME**

**/ home / odoo**

现在我们可以使用这个脚本。它向我们展示了如何从源代码将 Odoo 安装到 Debian/Ubuntu系统中。 首先，安装基本依赖，让我们开始：

**$ sudo apt-get update && sudo apt-get upgrade＃安装系统更新**

**$ sudo apt-get install git＃安装Git**

**$ sudo apt-get install npm＃安装NodeJs及其软件包管理器**

**$ sudo ln -s /usr/bin/nodejs /usr/bin/node ＃创建链接**

**$ sudo npm install -g less less-plugin-clean-css #Install less编译器**

从版本9.0开始，Odoo Web客户端需要在系统中安装较少的 CSS 预处理器，以便正确呈现网页。要安装这个，我们需要Node.js和npm。

接下来，我们需要获取Odoo源代码并安装所有其依赖项。 Odoo 源代码包括一个实用程序脚本，在 odoo/setup/ 目录中，以帮助我们在 Debian/Ubuntu系统中安装所需的依赖项：

**$ mkdir 〜/odoo-dev＃创建一个工作目录**

**$ cd 〜/odoo-dev＃进入我们的工作目录**

**$ git clone https://github.com/odoo/odoo.git -b 10.0 --depth = 1＃获取Odoo源代码**

**$ ./odoo/setup/setup\_dev.py setup\_deps＃安装Odoo系统依赖项**

**$ ./odoo/setup/setup\_dev.py setup\_pg＃为unix用户安装PostgreSQL和db超级用户**

最后，Odoo应该可以使用了。 〜符号是我们主目录的快捷方式（例如/home/odoo）。 git -b 10.0选项告诉 Git 只下载Odoo的10.0分支。在写本书时，这是多余的，因为 10.0 是默认分支;然而，这可能会改变，所以它可以使脚本面向未来。 --depth = 1选项指示 Git 只下载最后一个版本，而不是完整的更改历史，使下载更小更快。 要启动Odoo服务器实例，只需运行：

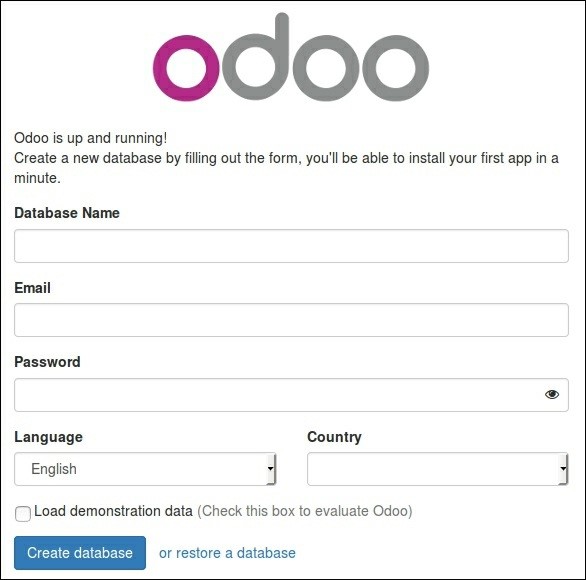
**$〜/odoo-dev/odoo/odoo-bin**

提示

在Odoo 10中，在以前版本中用于启动服务器的odoo.py脚本被替换为odoo-bin。

默认情况下，Odoo实例侦听端口8069，所以如果我们将浏览器指向http//<server-address>：8069，我们将访问这些实例。当我们第一次访问它时，

它向我们显示一个助手来创建一个新的数据库，如下面的截图所示：



作为开发人员，我们需要使用多个数据库，因此从命令行创建它们更方便，因此我们将学习如何做到这一点。 现在，在终端中按Ctrl+C 可停止 Odoo 服务器，并返回到命令提示符。

## 3.3 创建新的Odoo应用数据库

如果要创建新的数据库, 您的系统用户必须是 PostgreSQL 超级用户。 下面通过命令为当前Unix 户创建PostgreSQL 超级用户:

**$ sudo createuser --superuser $(whoami)**

创建新的数据库, 请使用 createdb 命令。下面我们来创建演示数据库:

**$ createdb demo**

如果要初始化Odoo数据模式的数据库,我们使用-d 参数运行空数据库的Odoo:

**$~/odoo-dev/odoo/odoo-bin -d demo**

这将需要几分钟的时间来初始化演示数据库,最终会显示日志信息、已加载的模块。

备注:

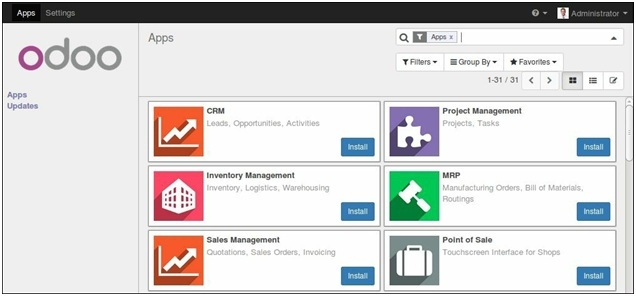
请注意, 它可能不是最后记录的信息, 它可能是最后第三或者第四行信息。 在这种情况下, 服务器将准备侦听客户端请求。

默认情况下, 这将初始化一个带有演示数据的数据库,这往往有利于数据库开发.

如果仅仅初始化数据库而无需演示数据,只需要在命令行添加参数:

**--without-demo-data=all**

现在使用浏览器打开 http://<server-name>:8069 , 将会显示登录页面。 如果您不知道服务器的名称, 请在终端键入 hostname 命令, 以便找到服务器名称或者 输入ifconfig 命令来查找 IP 地址。 如果您 Odoo 布在虚拟机上, 您可能需要进行一些网络配置,使得从您的主机系统能够访问它。最简单的解决方法是更改虚拟机网络类型从NAT 变为桥接。 这种情况下,无需共享主机的 IP 地址,客户虚拟机将拥有自己的 IP 地址.它也可以使用NAT,但是这需要您配置端口转发功能,使您的系统知道某些应该由虚拟机使用的端口, 如 8069. 为了防止您遇到麻烦,希望这些细节可以帮助您从虚拟软件文档中找到相关资料. 默认的管理员帐户是 admin 的密码 admin。 登录后,您的 应用程序 菜单, 将向您显示可用的应用程序:



当你想要停止Odoo服务器实例并返回到命令行, 按下Ctrl+C 在 bash 提示符处。按向上箭头键将显示您以前的 shell 命令, 因此它是一种快速启动带有相同参数Odoo 的方法。在开发过程中。 按 Ctrl+C键,然后按向上箭头键和回车键,是一种常用的组合键用来重新启动Odoo服务器 。

### 3.3.1 管理Odoo数据库

我们如何使用命令去创建和初始化新的Odoo数据库？这里有许多已知命令去管理数据库。

我们不仅可以使用createdb命令去创建空数据库，而且还可以使用—template选项去复制一个已有的数据库。

确认Odoo服务没有启用，并且有一个其它的数据库管理工具连接上了我们创建好的demo数据库。然后使用以下命令：

**$ created –template=demo demo-test**

实际上，每次我们创建数据库都使用了一个模板。如果没有特殊状况，那么使用的就是template的模板。

想查看你的系统中已存在了哪些数据库，可以使用PostgreSql的 psql命令，也可以附带-/选项：

**$psql –l**

使用这个命令，将会展示出我们已经创建的两个数据库，demo 和demo-test。它也会展示出数据库所使用的编码。默认编码是UTF-8，也是Odoo数据库使用的编码。

如果想删除一个长时间不用的数据库（或想重新创建），可以使用dropdb命令：

**$ dropdb demo-test**

现在您已经知道了基本的数据库使用。想知道更多，请参考PG官方文档：

**http://www.postgresql.org/docs/**

注意 警告 删除数据库将彻底删除保存数据信息。所以要特别小心的使用它，在使用删除命令时，也要确保重要的数据已经备份。

## 3.4 Odoo 版本

目前为止，Odoo的最新稳定版本是Odoo10，10.0版本标记在GitHub上，Odoo10这个版本即本书介绍所用的版本。

注意： 最重要的是要注意odoo不同的版本，数据库是不兼容的。如果数据库是以前版本创建的，Odoo 10将无法使用。 需要经过复杂的的升级作为前提，数据才能被新版本的使用。 相同的addon模块如下，作为一般规则，一个插件模块，无法用不同版本的odoo开发，你从网上下载的版本，确认好是你所需要的。 另一方面，odoo版本（9，10）将收到频繁的更新。这些新更新主要是修复一些bug，以保证程序的稳定，这代表着保持模型数据结构和视图元素标识符的稳定。这很重要，因为这些意味着将不会有任何风险,以保证自定义模块兼容主要核心模块。

分支的版本会发展到主要的稳定版,但在那之前,它不是最稳定的版本,你不应该使用它构建定制的模块。如果使用分支的版本，你不能确定分支版本什么时候改变，而影响到必须中断定制模块的开发。

## 3.5 查阅更多服务器选项

Odoo 服务器支持很多其他选项,我们可以用 --help来检查所有可用的选项：

**$ ./odoo-bin --help**

我们将在下面的章节中回顾一些最重要的选项，让我们先看看当前活动的选项如何保存在配置文件中。

### 3.5.1 Odoo 服务器配置文件

大多数选项可以保存在配置文件中。 默认情况下，Odoo将使用home目录中.odoorc文件。 在Linux系统中它的默认位置在home目录（$ HOME）中，在Windows系统中，位置是在用于启动Odoo的可执行文件相同的目录中。

注释

在以前的Odoo / OpenERP版本中，默认配置文件的名称是.openerp-serverrc。 为了向后兼容，如果.odoorc文件没有找到，Odoo 10仍将使用它。 在首次安装时，不会自动创建.odoorc配置文件。我们应该使用--save选项创建默认配置文件，如果它不存在，并将当前实例配置存储在其中：

**$ ~/odoo-dev/odoo/odoo-bin --save --stop-after-init #保存配置文件**

在这里，我们还可以使用--stop-after-init选项来停止服务器后执行初始化。 此选项通常在运行测试或运行模块升级时使用，以检查其是否正确安装。

现在我们可以检查保存在默认配置文件中的内容：

**$ more ~/.odoorc # 显示配置文件**

这将显示所有可用的配置选项及其默认值，编辑它们将在下次启动Odoo实例时生效。 按键盘‘q’退出并返回提示。

我们还可以选择使用--conf =<filepath>选项指定特定的配置文件。 配置文件不需要具有这些所有选项，只有需要更改默认值时才需要。

### 3.5.2 更改侦听端口

--xmlrpc-port=<端口> 命令选项充许我们将服务器实例的侦听端口从默认8069改为指定端口。这样可以在同一服务器上同时运行多个实例。

让我们来试试这个。打开两个终端窗口。在第一个终端，运行下面命令︰

**$ ~/odoo-dev/odoo1/odoo-bin --xmlrpc-port=8070**

在第二个终端上运行下面的命令:

**$ ~/odoo-dev/odoo2/odoo-bin --xmlrpc-port=8071**

这时你看到的: 两个 Odoo 实例运行在同一服务器不同侦听端口上！

这两个实例可以使用相同或不同的数据库，具体取决于使用的配置参数。两个可运行相同或不同版本的 Odoo 实例.

### 3.5.3 数据库的过滤选项

进行Odoo开发时，需要频繁在多个数据库上切换来工作，有时甚至工作在不同的Odoo版本。在相同的端口停止和启动不同的服务器实例，以及切换不同的数据库，可能导致 web 客户端会话有不正确的行为。

在浏览器窗口使用私密模式访问我们运行的实例可以帮助避免这些问题。

另一个好的做法是在服务器实例上使用数据库的过滤选项，确保只允许请求我们指定的数据库，而忽略其他的。这只需要指定--db-filter 选项。它接受一个正则表达式用作过滤有效的数据库名称。若要匹配一个确切的名称，表达式应该用 ^ 开始与用 $ 结束。

例如，只充许 domo 这个数据库，我们用的下面的命令：

**$ ~/odoo-dev/odoo/odoo-bin --db-filter=^demo$**

译者注： 更详细可参考： Odoo(OpenERP)应用实践: 使用db-filter参数实现通过域名指定访问哪个数据库

参考:http://blog.csdn.net/wangnan537/article/details/44229369

### 3.5.4 管理服务器日志消息

--log-level选项允许我们设置日志级别。 这可以有助于了解服务器中发生了什么。

例如，要启用调试日志级别，请使用--log-level = debug选项。

以下日志级别可能特别有趣：

**debug\_sql** 检查服务器生成的SQL查询

**debug\_rpc** 详细说明服务器收到的请求

**debug\_rpc\_answer** 详细说明服务器发送的响应

参数详细参考:

http://bbs.echarts.com.cn/thread-45-1-1.html

默认情况下，日志输出被定向到标准输出（您的控制台屏幕），但它可以使用--logfile = <filepath>选项定向到日志文件。

最后，当出现异常时，--dev = all选项将调出Python调试器（pdb）。 对服务器错误进行事后分析很有用。 请注意，它对记录器的详细程度没有任何影响。

更多详情Python调试器命令可以参考：

**https://docs.python.org/2/library/pdb.html#debugger-commands**

## 3.6 在工作站上进行开发

你可能用本地虚拟机或者服务器上的Debian/Ubuntu系统运行odoo。但是你可能更喜欢在你的个人工作站上用你喜欢的文本编辑集成开发环境来完成开发工作。

这种情况在使用windows 工作站的开发者的工作过程中可能经常发生。但是linux使用者在服务器上或者本地使用odoo的时候也会发生这种情况。

一种解决办法就是在odoo主机中设置允许文件共享，以便在工作站中能更简单方便的编辑修改文件。对于Odoo 服务器的操作，比如重启服务器，我们可以根据个人喜好使用SSH shell（例如在windows里使用PuTTY）。

### 3.6.1 使用Linux 文本编辑器

我们迟早都会需要从命令行上编辑文件。在大多数的Debian系统中vi都是缺省的文本编辑器。如果不是特别习惯vi，你也可以选择使用你熟悉的编辑器。

在Ubuntu系统中，缺省的文本编辑器是nano。它更易于使用，可能会成为你的首选编辑器。如果服务器上没有安装nano，可以使用下列命令安装：

**$ sudo apt-get install nano**

在下面的部分，我们假定nano是首选编辑器。如果你习惯其它的编辑器，请用相应的操作指令将之进行替换。

### 3.6.2 安装和配置 Samba

Samba服务让我们能够在Linux下实现与Windows兼容的文件共享服务。在Debian/Ubuntu系统中，可以使用下列命令安装Samba服务：

**$ sudo apt-get install samba samba-common-bin**

samba包用于安装文件共享服务，smbpasswd工具需要samba-common-bin包。缺省情况下，用户需要通过smbpasswd注册后才能够访问文件的文件。我们需要注册一个用户odoo，然后为之设置一个密码用于文件共享访问：

**$ sudo smbpasswd -a odoo**

输入此命令后，将提示为为此用户输入密码，此密码用于访问共享的目录。完成后odoo用户将能够访问他的用户主目录，尽管只是只读访问。如果需要有写入权限，我们需要按下面的内容修改Samba的配置文件。

**$ sudo nano /etc/samba/smb.conf**

在配置文件中，查找到[homes]部分。编辑其相应配置行使其匹配如下配置：

**[homes]**

**comment = Home Directories**

**browseable = yes**

**read only = no**

**create mask = 0640**

**directory mask = 0750**

为了使配置生效，需要重启服务：

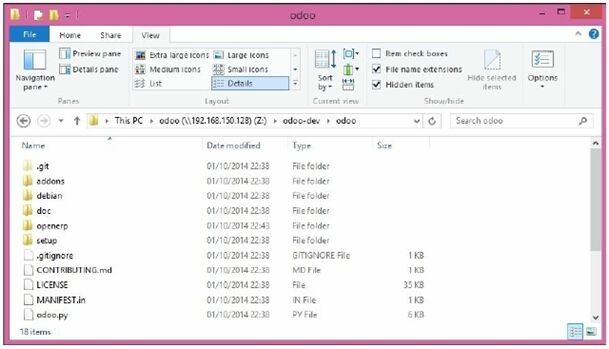
**$sudo /etc/init.d/smbd restart**

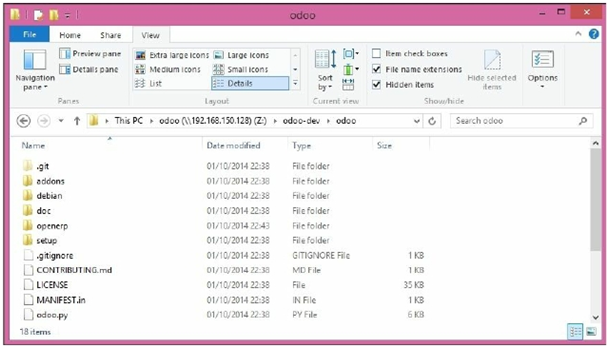
建议

下载示例代码

在Packt 账户中，您可以下载所有购买过的书籍的示例代码，Packt链接为：http://www.packtpub.com. 如果是从其他地方购买的书籍，您可以访问http://www.packtpub.com/support , 注册后便能通过邮件获取示例代码文件。

为了从Windows访问这些文件，我们可以使用特殊的用户名和定义为smbpasswd的密码来为路径 \\<my-server-name>\odoo 映射一个网络驱动。当使用odoo用户登录时，您可能会遇到Windows将计算机的域添加到用户名的问题（例如：MYPC\odoo）。为了避免这种情况，我们通过增加 \ 前缀的空域来登录（例如，\odoo）：

如果我们用Windows Explorer来打开网络驱动，我们将能够访问和编辑odoo用户的主目录的内容：

Odoo包含一些对开发者很有帮助的工具，我们将在整本书中用到它们。包括技术特点和开发者模式，默认状态下它们是关闭的，所以这是一个好机会去了解和启用它们。

### 3.6.3 激活开发者工具

开发工具提供先进的服务器配置和功能。包括一个在顶部菜单栏中调试菜单，和Setting菜单的附加菜单一起，尤其是Technical菜单。

这个菜单默认是隐藏的，如果要启用它，需要用admin账号登陆。在顶部菜单栏，选着Setting菜单。在右下方有个Odoo version, 你有两个选项去打开开发者模式；Debug 和 Technical菜单都会被启用。第二个选项，Activate the developer mode（with assets），不压缩客户端已使用的JavaScript 和Css，使其更容易的调试客户端。



之后，页面会重新加载，你可以在顶部菜单看见一个调试图标，就在用户头像旁边，用来展示调试模式的名称。并且，在顶部菜单的Settings选项中，我们也可以看见一个新的Technical菜单，它允许访问许多Odoo的接口。

小贴士

开发者模式允许我们查看和编辑Odoo在数据库中保存的配置信息、用户权限和其它的系统配置。你浏览后，就会学习到更多。

## 3.7 安装第三方模块

在Odoo实例中提供新模块，以便可以安装它们；是令Odoo新手常常感到困惑的地方。但它并不是那么复杂，所以让我们揭开神秘面纱吧。

### 3.7.1 查找社区模块

Internet上有许多Odoo模块可用。 apps.odoo.com上的Odoo应用商店是一个可以下载并安装在系统上的模块目录。 Odoo社区协会（OCA）协调社区贡献，并在GitHub上**https://github.com/OCA/**维护了很多模块存储库。

要将模块添加到Odoo安装，我们可以将其复制到官方模块的addons目录中。在我们的示例中，addons目录位于 ~/odoo-dev/odoo/addons/。由于我们的Odoo安装是基于版本控制的代码存储库，在我们仍然要保持它与GitHub存储库同步的情况下，这可能不是我们最好的选择。

幸运的是，我们可以为模块使用额外的位置，所以我们可以将我们的自定义模块保存在不同的目录，而不必与官方混合。

例如，我们将从GitHub下载这本书中的代码，并在Odoo安装中提供这些附加模块。

要从GitHub获取源代码，请运行以下命令：

**$ cd ~/odoo-dev**

**$ git clone https://github.com/dreispt/todo\_app.git -b 10.0**

使用-b选项来确保我们正在下载10.0版本的模块。

之后，在/odoo同级目录有一个新的/todo\_app目录，其中包含模块。现在我们需要让Odoo知道这个新的模块目录。

### 3.7.2 配置 addons 路径

Odoo服务器有一个addons\_path的配置选项，用于设置服务器运行时从哪里加载模块。默认情况下，Odoo服务器运行时，它使用根目录下/addons。

我们不仅可以提供一个，而且可以提供模块的目录列表。 这允许我们将我们的自定义模块保存在不同的目录中，而不必与官方默认的模块目录混合。 让我们使用包含我们的新模块目录的addons\_path启动服务器：

**$ cd ~/odoo-dev/odoo**

**$ ./odoo-bin -d demo --addons-path="../todo\_app,./addons"**

如果你仔细看看服务器日志，你会注意到一行记录模块加载使用的路径：INFO？ odoo：addons paths：[...] 确认它包含我们的todo\_app目录。

### 3.7.3 更新应用程序列表

在这些新模块可供安装之前，我们需要让Odoo更新其模块列表。

为此，我们需要启用开发人员模式，因为它提供了Update Apps List菜单选项。 它可以在Apps顶部菜单中找到。

更新模块列表后，我们可以搜索到新模块并安装。 使用Apps菜单选项查看本地模块的列表。搜索oodo，您应该看到新的模块。

请注意，第二个App Store菜单选项显示Odoo应用商店的模块列表，而不是本地模块：

## 3.8 第一章 小结

在本章中，我们学习了如何设置一个Debian系统来托管Odoo并从GitHub源代码安装它。 我们还学习了如何创建Odoo数据库和运行Odoo实例。 为了允许开发人员在个人工作站中使用他们喜欢的工具，我们解释了如何在Odoo主机中配置文件共享。 我们现在应该有一个运行的Odoo环境，管理数据库和实例。 有了这个，我们就可以直接采取行动了。 在下一章中，我们将从头创建我们的第一个Odoo模块，并理解它涉及的主要元素。 让我们开始吧！

# 4 第二章 构建第一个应用程序

我们在用odoo做开发的大部分时间就是在写符合我们自己需求的模块。在这一章节，我们将教你写出属于自己的第一个应用，并学习使Odoo可用并安装它所需的步骤。

参考著名的http://todomvc.com/ 项目，我们将创建一个简单的 ToDo 应用，它将允许我们添加新的任务，当任务完成的时候标记它们，并最终清除所有的已完成任务的列表。

我们将从学习开发工作流程的基础知识开始：为您的工作设置一个新实例，创建并安装一个新的模块，并应用您随开发迭代所做的更改。

Odoo遵循 MCV-like 架构设计，当实现 To-Do 应用的时候，我们会接触不同的层。

模型层，定义app 的数据结构

视图层，描述用户界面

控制器层，支持应用的业务逻辑

接下来，我们讲学习如何安全有效的设置访问控制器，最后我们将添加一些模块的描述和品牌信息。

注意

这里所描述的控制器这个概念，跟 odoo 网页开发的控制器是不同的。这些是网页可以调用以执行操作的程序端点。

使用这种方法，您将能够逐渐了解构成应用程序的基本构造，并体验从头开始构建 Odoo 模块的迭代过程。

## 4.1 基本概念

您很可能只是刚入门 Odoo，现在显然是一个非常好的时机，去了解Odoo模块，和怎样在Odoo开发中使用它们 。

### 4.1.1 了解应用程序和模块

在Odoo中，经常听见模块(modules)和应用程序(application)。那么，它们有什么区别呢？

模块元件是Odoo应用程序的组成块。一个模块可以将新功能添加到 Odoo，或修改现有的功能。

它是一个包含名为\_\_manifest\_\_.py的清单或描述符文件的目录，再加上实现其功能的其余文件。

应用程序是将主要功能添加到Odoo的方式。他们是核心功能元素，如会计或人力资源，其它的模块在此基础上修改或扩展功能。正因为如此，他们在Odoo应用程序菜单中高亮显示。

如果您的模块很复杂，并将新的或重大的功能添加到 Odoo，您可以考虑将它创建为应用程序。如果您的模块只是对现有的 Odoo 中的功能进行更改，则可能不需要将它创建为应用程序。

在清单中定义模块是否是应用程序。技术上它在模块使用上，没有任何的影响。它只是在应用程序清单中高亮显示。

### 4.1.2 修改和扩展模块

在示例中，我们要记住，在新模块创建时，要尽可能少的使用依赖关系。

然而，这不是常见情况。大多数情况下，我们将修改或扩展现有的模块。

一般情况下，通过直接更改它们的源代码,以修改现有模块的功能。这是一个很糟糕的方法。对于 Odoo 提供的官方模块，尤其如此。

如果这样做，你不能明确的区分原始模块代码和修改，并且这使它难以升级，因为升级会覆盖所做的修改。

相反，我们应该创建一个额外的模块，去实现我们需要的改变或者增加的功能。事实上，Odoo 的主要优势之一是继承机制，允许自定义模块扩展现有模块，无论是官方的还是社区的。

继承可以是任意一个层次︰ 数据模型、业务逻辑和用户界面层。

在这一章，我们将创建一个完全新的模块，但不扩展任何现有的模块，重点放在模块组成部分和所涉及的步骤。我们将会在后面的章节中详细介绍，这里只是简要介绍一下每个部分。

一旦我们可以轻松的创建一个新的模块，我们可以深入的学习继承机制。

在第3章将会介绍 ：继承扩展现有的应用程序。

想高效开发 Odoo，我们应该能自如地开发工作流︰管理开发环境、 应用代码更改，并检查结果。这一节将指导您这些基础知识。

### 4.1.3 创建模块的基本框架

按照第一章的指导，我们用odoo开发的时候，需要有odoo sever安装在

~/odoo-dev/odoo/目录。为了保持整洁，我们在其旁边建新的目录来存放自定义模块

~/odoo-dev/custom-addons。

Odoo包括一个scaffold命令来自动创建一个新的模块目录，其基本结构已经就位。 你可以用下面的命令来了解更多。

**$ ~/odoo-dev/odoo/odoo-bin scaffold --help**

在处理下一个模块时，您可能想要记住这一点，但是我们现在不会使用它，因为我们宁愿手动创建模块的所有结构。

Odoo addon模块是一个包含 \_\_manifest\_\_.py 描述文件的目录

备注

在以前的版本中，此描述符文件命名为 \_\_openerp\_\_.py。 此名称仍然受支持，但已被弃用。

它还需要是 Python 可导入的，因此它还必须有一个 \_\_init\_\_.py 文件。

模块的目录名称是其技术名称。 我们将使用oodo\_app。 文件名称必须是有效的Python标识符：它应以字母开头，并且只能包含字母，数字和下划线字符。

以下命令将创建模块目录并在其中创建一个空的\_\_init\_\_.py文件，〜/ odoo-dev / customaddons / todo\_app / \_\_ init\_\_.py 如果想通过命令行，下面的命令行你将会用到：

**$ mkdir ~/odoo-dev/custom-addons/oodo\_app**

**$ touch ~/odoo-dev/custom-addons/todo\_app/\_\_init\_\_.py**

接下来，我们需要创建清单文件。 它只包含一个Python字典，里面大约有十几个可能的属性; 其中，只有 name 属性是必填的。描述属性（对于较长的描述）和作者属性是建议填写的。 我们现在应该在\_\_init\_\_.py文件旁边添加一个\_\_manifest\_\_.py文件，其中包含以下内容

**{**

**'name': 'To-Do Application',**

**'description': 'Manage your personal To-Do tasks.',**

**'author': 'Daniel Reis',**

**'depends': ['base'],**

**'application': True,**

**}**

**depends** 属性可以包含所依赖的其他模块的列表。 当安装此模块时，Odoo 将自动安装它们。 这不是一个强制性的属性，但建议总是包含它。 如果不需要特定的依赖，我们应该依赖于核心 base 模块。

你应该注意确保所有依赖关系在这里明确设置; 如果其所依赖的模块在其之后加载，模块可能无法安装在干净的数据库（由于缺少依赖关系）或有加载错误。

对于我们的应用程序，我们不需要任何特定的依赖关系，因此我们仅依赖于 base 模块 为了简明起见，我们选择使用非常少的描述键，但是在真实的情况下，我们建议您还使用附加键，因为它们与 Odoo 应用商店相关：

**summary** 显示为模块的副标题

**version** 默认为 1.0。 它应该遵循版本语义规则（详见http://semver.org/）

**license** 许可证标识符，默认为LGPL-3

**website** 网站是一个用于查找有关模块的更多信息的URL。 这可以帮助人们找到更多的文档或问题跟踪，以提出 Bug 和建议。

**category**是模块的功能类别，默认为Uncategorized。 现有类别的列表可以在【应用程序】下拉列表中的安全组表单（设置|用户|组）中找到其它可用的描述符：

**installable** 默认为 True，但可以设置为 False 以禁用模块

**auto\_install** 如果设置为True，此模块将被自动安装，前提是其所有依赖项都已安装。 它用于必装模块

从Odoo 8.0 开始，我们可以在模块的顶层目录中使用 README.rst 或 README.md 文件替代描述键。

### 4.1.4 关于许可证

为您的工作选择许可证非常重要，您应该仔细考虑您最佳的选择及其影响。 Odoo模块使用最多的许可证是GNU通用公共许可证，（LGLP）和Affero通用公共许可证（AGPL）。 LGPL更为宽容，允许商业衍生工作，而不需要共享相应的源代码。 **AGPL**是一个更强大的开源许可证，需要派生工作和服务托管来共享其源代码。了解有关GNU许可证的更多信息，请访问https://www.gnu.org/licenses/。

### 4.1.5 添加 addons 路径

现在我们有一个简单的新模块，我们希望将它提供给 Odoo 实例。 为此，我们需要确保包含该模块的目录位于addons路径中，然后更新Odoo模块列表。 在上一章中已经详细解释了这两个动作，但是在这里，我们将继续简要介绍一下需要的内容。 我们将在我们的工作目录中定位并使用适当的addons路径配置启动服务器：

**$ cd ~/odoo-dev**

**$ ./odoo/odoo-bin -d todo --addons-path ="custom-addons, odoo/addons" --save**

**--save**选项保存您在配置文件中使用的选项。这使我们在重新启动服务器时重复使用它们：只需运行./odoo-bin，并使用上次保存的选项。 仔细查看服务器日志。应该有一个**INFO？ odoo：addons paths：[...] line**。它应该包括我们的 custom-addons 目录。 记住还要包括您可能正在使用的任何其他插件目录。例如，如果您还有一个 ~/odoo-dev/extra 目录，其中包含要使用的其他模块，则可能还要使用--addons-path选项来包含它们：

**--addons-path =“custom-addons, extra, odoo/addons”**

现在我们可以从Odoo实例来识别刚添加的新模块。

### 4.1.6 安装新模块

在“应用程序”顶部菜单中，选择“更新应用程序列表”选项。这将更新模块列表，添加自上次更新以来可能添加到列表中的任何模块。请记住，我们需要启用开发人员模式才能看到。

注意

确保您的Web客户端会话正在使用正确的数据库。您可以在右上角查看：数据库名称显示在括号中，紧随用户名。强制使用正确数据库的方法是使用附加选项

**--db-filter = ^MYDB$**

启动服务器实例。

应用程序选项向我们显示可用模块的列表。默认情况下，它只显示应用程序模块。由于我们创建了一个应用程序模块，因此我们不需要删除该过滤器来查看它。在搜索中键入todo，您应该看到我们的新模块。

现在点击模块的安装按钮。

### 4.1.7 升级模块

开发一个模块是一个迭代的过程，它会把你在源代码中更改的功能，在 Odoo 中体现出来。

在大多数情况下，只需要升级你的模块：在应用程序列表中的模块，一旦它已经安装，你就可以使用升级按钮。

然而，如果只修改了 Python 代码，升级就可能没有效果。除了模块的升级，还需要重新启动应用程序服务器。只要 Odoo 成功加载了 Python 代码，以后任何代码的更改都需要重新启动服务器才能应用。

在某些情况下，如果模块更改了数据文件和 Python 代码，您可能需要同时做这两种操作。对于Odoo新开发人员来说，这是很容易混淆。

但幸运的是，还有更好的方式。更改我们的模块最安全和最快的方式，是停止并重新启动服务器，并按要求把我们的模块升级到我们工作的数据库。

在终端服务器实例运行时，使用ctrl 键 + C 来停止它。然后，启动服务器和升级 todo\_app 模块，使用下面的命令︰

**$./ odoo-bin -d todo -u todo\_app**

-u 选项(或 —— update) 需要 -d 选项,并需要一个逗号分隔更新的模块列表。例如，我们可以使用 -u todo\_app,mail。一个模块更新时，也将更新所有其他依赖它的已安装的模块。这是必须保持的继承机制，用于扩展功能的完整性。

总体来说，当你需要更新应用模块时，最安全的方法是利用前面讲述的命令重新启动 Odoo 实例。按 up 键就可以使用前面的命令。所以，大多数情况下，你会发现自己经常使用 Ctrl + C ，up 和 Enter 键组合。

值得特别注意的是，更新模块列表和卸载模块这两种操作，都不可以通过命令行操作。需要在 web 界面的应用程序菜单中进行。

### 4.1.8 服务器开发模式

Odoo 10 新的选项特性，更方便开发者进行开发。要使用它，在启动服务器实例时，附加选项**--dev=all**。

这有几个小技巧，可以缩短我们的开发周期。最重要的是︰

一旦保存Python文件，自动重新加载 Python 代码，避免手动重启服务器；

从 XML 文件中，直接获取视图定义，避免手动模块升级；

--dev选项接受一个逗号分隔的列表，然而大部分时间都使用的是合适的all选项。我们还可以指定使用的调试器。默认情况下，Python 调试器，使用pdb 。有些人可能喜欢安装和使用其他调试程序。它也支持ipdb和 pudb调试.

## 4.2 模型层

让我们开始在Odoo中添加一个简单的新模型。

模型描述业务对象，例如商机、销售订单或合作伙伴（客户，供应商等）。 模型具有属性列表，并且还可以定义特定的业务逻辑。

模型是从 Odoo 模板类派生的 Python 类实现的。 它们直接转换为数据库对象，Odoo在安装或升级模块时会自动处理。负责此机制的是对象关系模型（ORM）。

我们的模块将是一个非常简单，记录保留待办事项的应用程序。

应用程序的主要功能有：用于描述的简单文本字段、用于将其标记为完成的复选框、最后需要添加一个按钮来清除已完成任务的旧的待办事项列表。

### 4.2.1 创建数据模型

Odoo 开发指导指出，模型的 Python 文件应放在一个 models 子目录中。为了简单起见，我们先不遵循这一准则，我们在 todo\_app 模块的主目录中创建一个todo\_model.py文件。

在文件中添加以下内容：

**# -\*- coding: utf-8 -\*-**

**from odoo import models, fields**

**class TodoTask(models.Model):**

**\_name = 'todo.task'**

**\_description = 'To-do Task'**

**name = fields.Char('Description', required=True)**

**is\_done = fields.Boolean('Done?')**

**active = fields.Boolean('Active?', default=True)**

第一行是一个特殊的标记告诉Python解释器，这个文件有UTF-8，以便它可以期望和处理非ASCII字符。我们不会使用任何非 ANSI 字符，但无论如何这是一个很好的做法。

第二行是Python代码import语句，从Odoo核心导入模型和字段对象。

第三行声明了我们的新模型。它是从models.Model派生的类。

第四行设置\_name属性，定义此模型的标识符，它将在整个Odoo中被引用。注意，实际的Python类名：TodoTask，对其他Odoo模块是无意义的。 \_name值将用作标识符。

请注意，此行和以下行是缩进的。如果你不熟悉Python，你必须知道这是很重要的：缩进定义一个嵌套的代码块，所以这四行应该是同样缩进。

然后我们有\_description模型属性。它不是强制性的，但它为模型记录提供了一个用户友好的名称，可用于用户友好的消息。

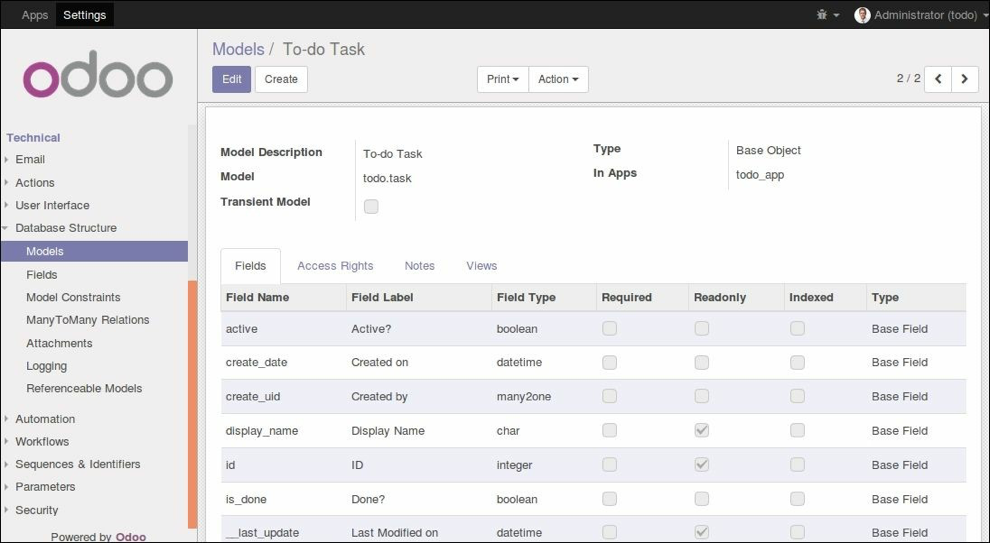
最后三行定义模型的字段。值得注意的是name和active是特殊的字段名。默认情况下，当从其他模型引用它时，Odoo将使用name字段作为记录的标题。active字段用于停用记录，默认情况下，仅显示活动记录。我们将使用它来清除已完成的任务，而不会从数据库中删除它们。

现在，该文件尚未被模块使用。我们必须告诉Python使用\_\_init\_\_.py文件中的模块加载它。让我们编辑它以添加以下行：

**from . import todo\_model**

就是这样而已！为了使我们的Python代码更改生效，服务器实例需要重新启动（除非它使用--dev模式）。

我们不会看到任何菜单选项来访问这个新模型，因为我们还没有添加它们。我们仍然可以使用技术菜单查看新创建的模型。在顶部菜单设置中，**转到技术|数据库结构|模型**，在列表中搜索**todo.task**模型，并单击它以查看其定义：



如果一切正常，则确认已创建模型和字段。如果您在此处看不到它们，请尝试重新启动服务器并进行模块升级，如前所述。

我们还可以看到一些我们没有声明的额外字段。这些是Odoo自动添加到每个新模型的保留字段。它们如下所示：

**id**是模型中每个记录的唯一数字标识符。

**create\_date**和**create\_uid**指定创建记录的时间和由谁创建它。

**write\_date**和**write\_uid**确认记录的上次修改时间以及谁对其进行了修改。

**\_\_last\_update**是一个实际上不存储在数据库中的帮助信息。它用于并发检查。

### 4.2.2 加入自动化测试

最佳编程实践包括自动测试你的代码。 这对动态语言更为重要，如 Python语言。 因为没有编译的步骤, 代码实际是解释运行的，所以无法确定有无语法错误的。 一个优秀的编辑器可以提前帮助我们发现这些问题, 但却不能帮助我们确保代码按照预期的执行，比如自动化测试。

Odoo支持使用两种方式来描述测试: 使用 YAML 数据文件或使用 Python 代码、基于 Unittest2 库。 YAML 测试是一种遗留下来的旧老版本, 不建议使用。 我们会更乐意使用 Python 测试并将添加基本的测试实例到我门的模块。

测试代码文件**必须用test\_开头** , 并且从tests/\_\_init\_\_.py文件导入。 但 测试的 目录(或 Python 的子模块)不应从模块的顶部的\_\_init\_\_.py导入， 因为仅在测试的执行的时候会自动被发现并装载。

测试必须放置在 tests/的 子目录中。 添加一个tests/\_\_init\_\_.py 文件，包含如下内容：

**from . import test\_todo**

现在添加实际的测试代码放在tests/test\_todo.py文件里：

**# -\*- coding: utf-8 -\*-**

**from odoo.tests.common import TransactionCase**

**class TestTodo(TransactionCase):**

**def test\_create(self):**

**"Create a simple Todo"**

**Todo = self.env['todo.task']**

**task = Todo.create({'name': 'Test Task'})**

**self.assertEqual(task.is\_done, Fa**lse)

这里增加了一种简单的测试例子来创建新的to-do任务来验证测试例子是否被执行? 测试例子的字段具有合适的默认值。

现在我们即将运行我们的测试。 当我们在安装模，可以添加 --test-enable选项

**$ ./odoo-bin -d todo -i todo\_app --test-enable**

该Odoo服务器将寻找一个tests/子目录的升级模块 并运行。如果有任何测试失败了，服务器日志都会在这里显示。

## 4.3 视图层

视图层作用是定义用户界面。视图是使用XML定义的，它被Web客户端框架用来转换成数据感知的HTML视图。

我们有菜单项，这些菜单可以触发能够呈现视图的动作。例如，“用户”菜单项触发一个也称为“用户”的动作，这个动作就会去渲染一系列视图。这里有几种视图类型可用，例如列表视图和表单视图，而且特定类型视图还可以定义过滤选项，如搜索视图。

Odoo开发准则指出：定义用户界面的XML文件应放在views /子目录中。

让我们开始为我们的To-Do应用程序创建用户界面吧。

### 4.3.1 添加菜单选项

现在我们有一个存储数据的模型，我们应该在用户界面上可以操作它。

为此，我们应该添加一个菜单选项来打开To-do Task模型，以便可以使用它。

创建views/todo\_menu.xml文件以定义菜单项及其执行的动作：

**<?xml version="1.0"?>**

**<odoo>**

**<act\_window id="action\_todo\_task"**

**name="To-do Task"**

**res\_model="todo.task"**

**view\_mode="tree,form" />**

**<menuitem id="menu\_todo\_task"**

**name="Todos"**

**action="action\_todo\_task" />**

**</odoo>**

用户界面接口，包括菜单和动作，存储在数据库表中。当模块安装或升级时，xml数据文件中的这些用户接口定义就会被加载到数据库中。上面的代码就是一个Odoo数据文件，定义了两条要添加到Odoo的记录：

**<act\_window>**元素定义了一个客户端窗口动作，该动作将以列表和表单视图的顺序打开todo.task模型。

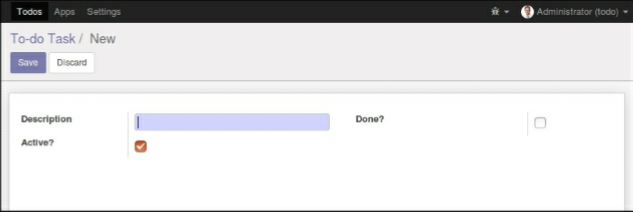
**<menuitem>**定义了一个顶级菜单项，调用前面定义的action\_todo\_task动作。

两个元素都包含了**id属性**。 此id属性也称为XML ID，它用于唯一标识模块内的每个数据元素，并且可以由其他元素使用来引用它。 在本例中，<menuitem>元素需要指定要处理的动作，就使用了<act\_window> 的ID来引用它。 XML ID的相关内容，在第4章“模块数据”中有更详细的讨论。

目前，我们的模块还不知道这个新定义的XML数据文件。 为了让模块加载这个文件，我们需要将它添加到\_\_manifest\_\_.py文件的**data属性中**。 data属性中的文件列表都将被模块加载。 manifest清单data属性定义如下：

**'data': ['views/todo\_menu.xml'],**

现在，我们需要重新升级模块以使这些更改生效。 升级后转到Todos顶部菜单，您应该看到我们的新菜单选项效果如下：



即使我们没有定义用户界面视图，点击Todos菜单，仍将打开一个根据模型自动生成的表单视图，允许我们添加和编辑记录。 正是因为Odoo可以很好地自动创建视图，这样我们就可以立即开始使用我们的模型。 目前为止一切都很顺利！现在让我们来开始改善用户界面。试着按照下一节的演示逐步改进、升级模块，不要害怕进行实验。您可能还会想尝试--dev = all服务器选项。使用它，视图定义可以直接从XML文件中读取，这样您的改动可以立即在Odoo上生效，而不需要进行模块升级。

小贴士

如果由于XML错误导致升级失败，请不要惊慌！ 注释掉最后编辑的XML部分或者从\_\_manifest\_\_.py文件中删除该XML文件，并重新升级。 服务器就可以正常启动。 然后可以仔细阅读服务器日志中的错误消息：它应该会告诉你问题所在。 Odoo支持几种类型的视图，但最重要的三个是：

**tree**（通常称为列表视图），

**form** 窗体视图和

**search**搜索视图。

每种我们将向模块添加一个示例。

### 4.3.2 创建窗体视图

创建窗体视图 所有的视图都存储在数据库中，在 ir.ui.view模型下。要向模块添加视图，我们在XML文件中声明一个描述视图的<record>元素，该模块在安装模块时将被加载到数据库中。

添加这个新的views / todo\_view.xml文件来定义我们的表单视图：

**<?xml version="1.0"?>**

**<odoo>**

**<record id="view\_form\_todo\_task" model="ir.ui.view">**

**<field name="name">To-do Task Form</field>**

**<field name="model">todo.task</field>**

**<field name="arch" type="xml">**

**<form string="To-do Task">**

**<group>**

**<field name="name"/>**

**<field name="is\_done"/>**

**<field name="active" readonly="1"/>**

**</group>**

**</form>**

**</field>**

**</record>**

**</odoo>**

记住要将这个新文件添加到清单文件中的数据键，否则我们的模块不会识别它，它将不会被加载

这将以标识符view\_form\_todo\_task向**ir.ui.view模型**添加一条记录。 该视图用于todo.task模型，并命名为To-do Task Form。 名称仅供参考; 它不必是唯一的，但它应该允许容易地识别它指的是哪个记录。 实际上，名称可以完全省略，在这种情况下，它将根据模型名称和视图类型自动生成.

**最重要的属性是arch**，它包含视图定义，在上面的XML代码中突出显示。 <form>标签定义视图类型，在本例中，包含三个字段。 我们还向活动字段添加了一个属性，使其为只读。

### 4.3.3 业务文档窗体视图

前面的部分提供了一个基本的表单视图，但我们可以做一些改进。 对于文档模型，Odoo有一个模仿纸页文档的表现风格。 此表单包含两个元素：<header>包含操作按钮，<sheet>包含数据字段

窗体顶端 我们现在可以使用以下代码替换上一节中定义的基本<form>：

**<form>  
<header>  
<!-- Buttons go here-->  
</header>  
<sheet>  
<!-- Content goes here: -->  
<group>  
<field name="name"/>  
<field name="is\_done"/>  
<field name="active" readonly="1"/>  
</group>  
</sheet>  
</form>**

### 4.3.4 添加动作按钮

Form视图可以添加按钮以完成特定动作。这些按钮可以打开一个包含Form表单的窗口或运行定义在模块中的Python函数。

它们可以定义在Form视图内的任意位置，但是对于文档形式的窗体，建议把它们放在<header>标签中。 对于我们的应用程序，我们将添加两个按钮去运行定义在todo.task模型中的方法。

**<header>**

**<button name="do\_toggle\_done" type="object"**

**string="Toggle Done" class="oe\_highlight" />**

**<button name="do\_clear\_done" type="object"**

**string="Clear All Done" />**

**</header>**

按钮的基本属性有以下这些：

**string:**展示在按钮上的文本

**type:**动作的类型（执行的操作）

**name：**定义的动作（操作的标识符）

**class：**一个可选的属性，用于展示Css样式，同HTML规则一样

### 4.3.5 用groups来组织forms

<group>标签允许组织Form表单里的内容。在<group>中放置<group>，可以创建一个两列的列表。Group标签使用时，建议定义它的name属性，这样可以更方便的让其它模块扩展它。

使用<group>标签，可以更好的规划需要展示内容。将<sheet>改写成以下内容：

**<sheet>**

**<group name="group\_top">**

**<group name="group\_left">**

**<field name="name"/>**

**</group>**

**<group name="group\_right">**

**<field name="is\_done"/>**

**<field name="active" readonly="1"/>**

**</group>**

**</group>**

**</sheet>**

### 4.3.6 完整的Form视图

此时，todo.task模块的Form视图为：

**<form>**

**<header>**

**<button name="do\_toggle\_done" type="object" string="Toggle Done" class="oe\_highlight" />**

**<button name="do\_clear\_done" type="object" string="Clear All Done" />**

**</header>**

**<sheet>**

**<group name="group\_top">**

**<group name="group\_left">**

**<field name="name"/>**

**</group>**

**<group name="group\_right">**

**<field name="is\_done"/>**

**<field name="active" readonly="1" />**

**</group>**

**</group>**

**</sheet>**

**</form>**

小贴士

记住，将模块升级后，才能把这些改变加载到Odoo数据库中。在客户端查看变化，要刷新界面时：可以重新点击菜单按钮或刷新浏览器（大部分浏览器的快捷键是F5）。

在我们添加业务逻辑的具体实现代码之前，按钮是不能正常使用的。

### 4.3.7 添加列表和搜索视图

在列表模式下查看模型时，将使用<tree>视图。 树视图能够显示按层级结构组织的行，但大多数时候，它们用于显示简单列表。

我们可以将以下树视图定义添加到 todo\_view.xml：

**<record id="view\_tree\_todo\_task" model="ir.ui.view">**

**<field name="name">To-do Task Tree</field>**

**<field name="model">todo.task</field>**

**<field name="arch" type="xml">**

**<tree colors="decoration-muted:is\_done==True">**

**<field name="name"/>**

**<field name="is\_done"/>**

**</tree>**

**</field>**

**</record>**

这将定义一个只有两列的列表：name和is\_done。 我们还添加了一个很好的交互界面：已完成任务的行（is\_done == True）会显示成灰色。 这是通过应用Bootstrap类完成的。

有关Bootstrap及其上下文颜色的更多信息，请参阅

http://getbootstrap.com/css/#helper-classes-colors

在列表的右上角，Odoo显示一个搜索框。 它搜索的字段和可用的过滤器是由<search>视图定义的。

和前面一样，我们将它添加到todo\_view.xml：

**<record id="view\_filter\_todo\_task" model="ir.ui.view">**

**<field name="name">To-do Task Filter</field>**

**<field name="model">todo.task</field>**

**<field name="arch" type="xml">**

**<search>**

**<field name="name"/>**

**<filter string="Not Done"**

**domain="[('is\_done','=',False)]"/>**

**<filter string="Done"**

**domain="[('is\_done','!=',False)]"/>**

**</search>**

**</field>**

**</record>**

<field>元素定义了在搜索框中输入时能搜索的字段。 <filter>元素添加了预定义的过滤条件(过滤条件可以通过用户点击来切换），是使用特定语法来定义的。

## 4.4 业务逻辑层

现在我们将为按钮添加一些逻辑。这是用Python代码完成的，使用了模型的Python类里面的方法。

### 4.4.1 添加业务逻辑

我们应该编辑todo\_model.py Python文件，将按钮所调用的方法添加到类中。 首先，我们需要导入新的API，因此将其添加到Python文件顶部的import语句：

**from odoo import models, fields, api**

Toggle Done按钮的操作将非常简单：只需切换Is Done? 标记。对于记录上的逻辑，使用@api.multi装饰器。 这里，self将表示一个记录集，然后我们应该遍历每个记录。

在TodoTask类中，添加：

**@api.multi**

**def do\_toggle\_done(self):**

**for task in self:**

**task.is\_done = not task.is\_done**

**return True**

代码循环遍历所有待完成的任务记录，并对每个任务记录修改is\_done字段，将值取反。 该方法不需要返回任何东西，但我们应该让它至少返回一个True值。 原因是客户端可以使用XML-RPC来调用这些方法，并且此协议不支持只返回None值的服务器函数。

对于Clear All Done按钮，我们需要更进一步的改进。它应该查找已完成的所有活动记录，并使它们处于非活动状态。 通常表单按钮只能对选定的记录起作用，但在这种情况下，我们希望它也对除当前记录之外的记录起作用：

**@api.model**

**def do\_clear\_done(self):**

**dones = self.search([('is\_done', '=', True)])**

**dones.write({'active': False})**

**return True**

在用 @api.model装饰的方法上，自变量表示没有记录的模型。 我们将构建一个包含所有标记为done的任务的dones记录集。 然后，我们将active标志设置为False。

搜索方法是一种返回满足某些条件的记录的API方法。 这些条件写在一个域中，这是一个三元组列表。

我们将在第6章“视图 - 设计用户界面”中更详细地探讨域。

写入方法同时对记录集的所有元素设置值。 要写入的值使用字典进行描述。 在这里使用write比遍历记录集更有效率，以便逐一为每个记录集赋值。

### 4.4.2 添加测试

现在我们应该为业务逻辑添加测试。 理想情况下，我们希望每行代码至少被一个测试用例覆盖到。在tests/test\_todo.py文件中，向test\_create（）方法再添加几行代码：

**# def test\_create(self):**

**# ...**

**# Test Toggle Done**

**task.do\_toggle\_done()**

**self.assertTrue(task.is\_done)**

**# Test Clear Done**

**Todo.do\_clear\_done()**

**self.assertFalse(task.active)**

如果我们现在运行测试并且上面模型方法正确写入，那么我们在服务器日志中应该看不到错误消息：

**$ ./odoo-bin -d todo -i todo\_app --test-enable**

## 4.5 设置安全访问权限

您可能已经注意到，在加载过程中，我们的模块在服务器日志中有一条警告消息：

The model todo.task has no access rules, consider adding one.

这条消息意思很明确：我们的新模型没有访问规则。这样会使除admin超级用户之外的任何人都不能使用它。 admin作为超级用户，会忽略数据访问规则，这就是之前我们可以使用表单而没有出现错误的原因。 但我们必须解决这个问题，其他用户才能使用我们的模型。 另外我们还需要解决的问题是：我们希望待办事项对每个用户都是私有的。 Odoo支持“行级”（row-level）访问规则，我们将用它来实现这一点。。

### 4.5.1 测试安全访问权限

事实上，对于安全访问测试，由于缺少访问规则，目前我们的测试应该是会失败的。没有失败，是因为我们使用的是admin用户。因此，我们应该更改测试，以使它们使用Demo用户。 为此，我们应该修改tests/test\_todo.py文件，添加一个setUp方法：

**＃class TestTodo（TransactionCase):**

**def setUp（self，\* args，\*\* kwargs）：**

**result = super（TestTodo，self）.setUp（\* args，\**

**\*\* kwargs）**

**user\_demo = self.env.ref（'base.user\_demo'）**

**self.env = self.env（user = user\_demo）**

**return result**

方法中第一个语句调用父类的setUp代码。接下来语句改变了运行测试的环境变量self.env，使用Demo用户给它赋予了新值。根据上面的写法，不需要再做其他改变。

为确保用户只能看到自己的任务，我们还应该添加一个测试用例。添加之前，首先在顶部添加一个额外的导入语句：

**from odoo.exceptions import AccessError**

然后，添加一个额外的方法到测试类：

**def test\_record\_rule（self）：**

**“测试每个用户记录规则”**

**Todo = self.env [ todo.task']**

**task = Todo.sudo（）.create（{'name'：'Admin Task'}）**

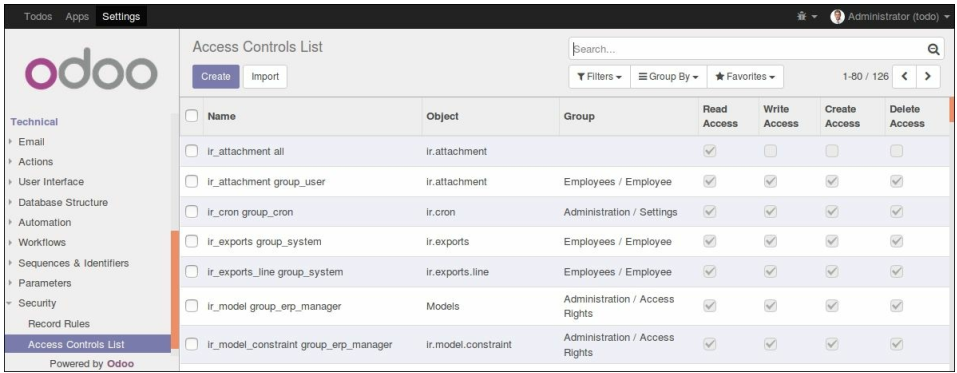
**with self.assertRaises（AccessError）：**

**Todo.browse（[task.id]）**

因为env使用的是Demo用户，我们使用sudo（）方法将上下文更改为admin用户。然后，我们使用admin来创建一条任务，这个任务Demo 用户应该无法访问。 当尝试访问这条任务数据时，我们期望抛出一个AccessError异常。 如果我们现在运行测试，它们应该失败，然后抛出异常让我们来处理。

### 4.5.2 添加访问控制安全

使用 web 客户端，然后转到设置|技术|安全|访问控制列表，就可以给模型添加需要的访问规则，如图：



小贴士

在这里我们可以看到一些模块的 ACL（访问控制列表）。它表明：对于每个安全组，对记录可允许的操作。

这些信息是模块通过ir.model.access的数据文件加载的。我们将给员工组添加模块的所有访问权限。员工组几乎是每个人都属于的基本访问组。

创建CSV 文件security/ir.model.access.csv。添加以下内容︰

**id,name,model\_id:id,group\_id:id,perm\_read,perm\_write,perm\_create,perm\_unlink**

**acess\_todo\_task\_group\_user,todo.task.user,model\_todo\_task,base.group\_user,1,1,1,1**

文件名与要加载的数据的模块名相对应，并且该文件的第一行包含列名称。以下是我们的 CSV 文件中提供的列︰

**id**是记录的外部标识符 (也称为 XML ID)。在我们的模块中它应该是唯一。

**name**是一个描述标题。它只是内容展示，在模块中最好也保持其唯一性。官方模块通常使用模型名称和组的圆点分隔的字符串。遵循此约定，我们使用 todo.task.user

**model\_id** 是该模型的外部标识符。ORM给模型自动生成的 O的 XML Id︰ 如 todo.task，该标识符是model\_todo\_task

**group\_id**标识授予权限的访问组。最重要的一点是提供定义它的模块名前缀。比如员工组，它的标识符为base.group\_user。

**perm**字段标记授予的读, 写, 创建，或删除权限。

我们不能忘记将引用添加到文件\_\_manifest\_\_.py的data属性中。就像这样︰

**'data': [**

**' security/ir.model.access.csv'，**

**' views/todo\_view.xml'，**

**'views/todo\_menu.xml'，**

**]，**

和以前一样，升级模块,添加的内容才能生效。现在，警告消息应该没有了，我们可以用demo(密码也是demo)用户登录，验证权限的正确性。现在如果我们重新测试，应该只会在测试 test\_record\_rule 用例时失败.

### 4.5.3 Row-level访问规则

我们可以在技术菜单中找到和访问控制列表在一起的记录规则菜单。

记录规则的定义模型是ir.rule。像往常一样，我们需要提供一个独特的名称。我们还需要操作符和规则过滤器组成的模型以达到访问限制。在Odoo中，规则过滤器通常是多个元组组成的列表。

通常情况下，规则适用于某些特定的访问组。在我们的例子中，我们将它应用于员工组。如果它没有访问组，这种特殊情况，被认为是全局 ( global字段将自动设置为 True)。全局规则是不同的，因为他们实施了限制，不会被非全局规则重写。

添加记录规则，我们应该创建security/todo\_access\_rules.xml文件并添加下面的内容︰

**<?xml version="1.0" encoding="utf-8"?>**

**<odoo>**

<data noupdate="1">

**<record id="todo\_task\_user\_rule" model="ir.rule">**

**<field name="name">ToDo Tasks only for owner</field>**

**<field name="model\_id" ref="model\_todo\_task"/>**

**<field name="domain\_force">[('create\_uid','=',user.id)]</field>**

**<field name="groups" eval="[(4,ref('base.group\_user'))]"/>**

**</record>**

</data>

**</odoo>**

注意

小心**noupdate ="1"**属性。这意味着此数据在模块的升级时将不会更新。这就使它能够进行定制，因为后面模块的升级不会破坏用户进行的更改。但请注意，在开发的时候也会这样，所以在开发的时候你可以设置noupdate ="0" ，直到你对你的数据文件满意为止。

在groups字段，你还会发现一个特殊的表达式。它是一个一对多的关系字段，他们有特殊的操作语法。在这种情况下， (4，x) 元组指要追加 x 记录，这里 的x 是关联的员工组，用base.group\_user标识的。一对多特殊的语法将在章 4模块数据 中更详细的介绍。

和前面一样，加载模块之前，我们必须将该文件添加到\_\_manifest\_\_.py文件中︰

**'data': [**

**'security/ir.model.access.csv'，**

**'security/todo\_access\_rules.xml'，**

**'todo\_view.xml'，**

**'todo\_menu.xml'，**

**],**

如果我们做对了所有步骤，现在我们运行测试模块，他们应该是通过的。

## 4.6 更好的模块描述

我们的模块的看起来很好。为什么不给它添加图标，让它看起来更好呢？为此，我们只是需要向模块中添加要使用的图标文件 static/description/icon.png 。

我们将重用现有的Notes应用程序的图标，所以复制

odoo/addons/static/description/icon.png文件到

customaddons/todo\_app/static/description目录。 我们可以使用下面的命令︰

**$ mkdir-p ~/odoo-dev/custom-addons/todo\_app/static/description**

**$ cp ~/odoo-dev/odoo/addons/note/static/description/icon.png ~/odoo-dev/custom-addons/todo\_app/static/description**

现在，如果我们更新模块列表，我们的模块就可以显示新图标了。 我们还可以添加更好的描述它来解释它做什么和怎样的效果。在\_\_manifest\_\_.py文件中可以添加description选项来做到 。不过，首选的方法是添加 README.rst 文件到模块的根目录。

## 4.7 第二章 小结

在这一章中，我们创建了一个拥有最常用元素的新模块︰ 模块、 三种基本视图 （窗体、 列表和搜索），模型的业务逻辑方法和访问权限。

在这个过程中，我们不仅熟悉了模块的开发过程，还涉及到，要使更改在Odoo中有效，必须进行模块的升级和应用程序服务器重新启动。

永远记住，当添加模型字段，需要升级。当改变 Python 代码，包括清单文件，需要重新启动。当更改 XML 或 CSV 文件，需要升级；此外，当不清楚时，两者都做︰ 重新启动服务器和升级模块。

在下一章中，您将学习如何对现有的模块添加功能

# 5 第三章 继承–扩展现有的应用程序

odoo最强大功能之一就是在可以在不改变底层对象的时候添加新的功能。

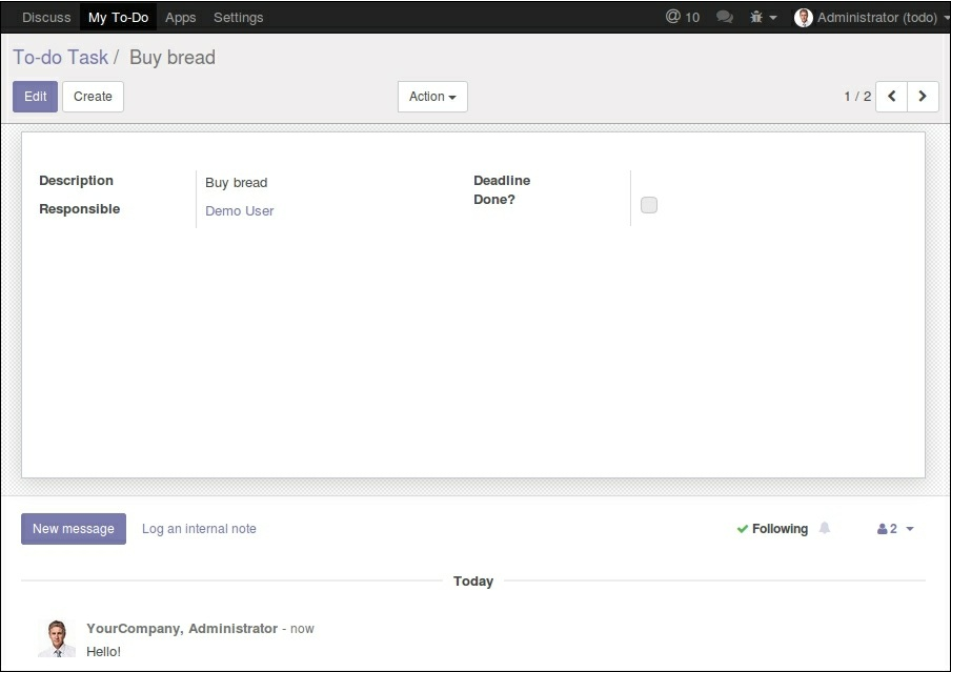
这是通过继承机制来实现的，作为在现有对象之上的修改层。这些修改可以发生在所有级别：模型，视图和业务逻辑。不是直接修改现有模块，而是创建一个新模块以添加预期的修改。

在本章中，您将学习如何编写自己的扩展模块，使您能够利用现有的核心或社区应用程序。作为相关示例，您将了解如何将Odoo的社交和消息传递功能添加到您自己的模块中。

## 5.1 向待办事宜应用程序添加共享功能

我们的To-Do应用程序现在允许用户私下管理自己的待办事项。 通过添加协作和社交网络功能，将应用程序带到另一个级别不会是伟大的吗？ 我们将能够与其他人分享任务和讨论。

我们将使用一个新模块来扩展先前创建的To-Do应用程序，并使用继承机制添加这些新功能。 这里是我们期望在本章结束时实现的：



这将是我们要实现的功能扩展的工作计划：

* 扩展任务模型，例如负责该任务的用户
* 修改业务逻辑，仅能操作当前用户的任务，而不是用户能够看到的所有任务
* 扩展视图以向视图添加必要的字段
* 添加社交网络功能：留言墙和追随者

我们将开始为todo\_user模块和todo\_app模块一起创建一个基本框架。 按照第1章“Odoo开发入门”中的安装示例，我们在〜/ odoo-dev / custom-addons /上放置我们的模块。 我们应该为模块添加一个新的todo\_user目录，其中包含一个空的\_\_init\_\_.py文件。

现在创建todo\_user / \_\_ manifest\_\_.py，包含这段代码：

**{ 'name': 'Multiuser To-Do',**

**'description': 'Extend the To-Do app to multiuser.',**

**'author': 'Daniel Reis',**

**'depends': ['todo\_app'], }**

当发布模块到Odoo在线应用商店时，提供包括摘要和类别等是重要的，这里我们没这样做。

注意，我们添加了对todo\_app模块的显式依赖。这对于继承机制正常工作是必要和重要的。从现在起，当todo\_app模块更新时，所有依赖于它的模块，例如todo\_user模块，也将被更新

接下来安装它。 使用应用程序下的更新应用程序列表菜单，更新模块列表; 在应用程序列表中找到新模块，然后单击其安装按钮。 请注意，此时您将需要删除默认的Apps过滤器，以便在列表中查看新模块，因为它未标记为应用程序。 有关发现和安装模块的更详细的说明，请参阅第1章，Odoo开发入门。

现在，让我们开始添加新的功能。

## 5.2 扩展模型

新的模型是通过 Python 的类定义，扩展模型也是通过 Python 的类来定义, 但需要在 Odoo 特定的继承机制的帮助下来实现。

想要扩展现有的模型, 我们使用一个具有继承属性的Python类， 这个特征决定模型可以用来进行扩展。 新类从父 Odoo 模型继承的所有功能, 我们只需要声明一下我们想提出修改部分。

实际上, Odoo 模型存在 Python 的模块之外, 在中间注册表那里。对于这个注册表，可以通过模型的方法使用self.env[<model name>]访问。 例如, 通过res.partner 模型获取对象的引用， 代码书写如下 self.env['res.partner']。

要修改某个 Odoo 模型, 我们可以从他的注册类获取参考，然后执行本地修改。这意味着，这些修改后的模型，也能在任何其他使用这种新模型的地方使用

在 Odoo 服务器启动的时候，载入模块顺序是有关系的， 通过add-on模块修改的模块，只能在add-on模块装载后才可见

因此正确设置模块的依赖是非常重要的， 所以它是很重要的模块依赖关系才能进行正确的设置, 确保模块提供我们使用的模型都包括在我们的依赖关系树中

### 5.2.1 向模型中添加字段

我们将添加几个字段到todo.task模型：负责的任务用户、任务截止日期。 代码风格指引推荐有一个 models/ subdirectory ，这是每个odoo模型的一个文件。所以我们应该从创建model子目录开始，以便用Python导入。

编辑todo\_user / \_\_ init\_\_.py文件，包含以下内容

from .import models

使用以下代码创建todo\_user / models / \_\_ init\_\_.py：

from . import todo\_task

上一行内容指引 Python 在同一目录去寻找一个叫odoo\_task.py的文件，并完成导入。对于每个目录中的 Python ，你通常都有这一段从属行代码。 现在创建todo\_user / models / todo\_task.py文件，以扩展原来的模型：

**# -\*- coding: utf-8 -\*-**

**from odoo import models, fields, api**

**class TodoTask(models.Model):**

**\_inherit = 'todo.task'**

**user\_id = fields.Many2one('res.users', 'Responsible')**

**date\_deadline = fields.Date('Deadline')**

这个名为todo.task的类，对于 Python文件来说，是属于本地的，一般来讲，和其他模块没有任何关系， 类的属性\_inherit是关键，这表明Odoo的这个类是继承的，来自于对todo.task模型的修改。

备注：

注意到这里的\_name属性是不存在的，因为是继承父模型的，所以不是必要的一个属性。

上述代码下的两行是正则字段的声明。 user\_id字段表示一个用户是从用户模型res.users引用而来。 这是一个多对一的Many2one字段，这个字段是相当于数据库术语中的外键。 这里的截止日期是一个简单的日期型的字段。 在第5章“结构化应用数据模型”，我们将更详细地讲解Odoo中可用的字段类型。

### 5.2.2 修改现有字段

如你所见，向现有模型添加新字段非常简单。Odoo 8及以后的版本，开发人员可以修改现有继承字段上的属性。为了改变属性，你只需要为属性添加一个具有相同名称和值的字段。

例如，为了添加帮助提示信息到name字段，我们会将下面这行代码添加到todo\_ task.py中：

**name = fields.Char（help =“What needs to be done?”）**

这将修改具有指定属性的字段，保留未修改的所有其他未在此处明确使用的属性。如果我们升级模块，请转到待办事项任务表单，并将鼠标指针悬停在“说明”字段上;将显示工具提示的文本。

### 5.2.3 修改模型的方法

继承也在业务逻辑级别起作用。添加新方法很简单：只需在继承类中声明新的函数。

要扩展或更改现有逻辑，可以通过声明具有完全相同名称的方法来覆盖相应的方法。新方法将替换前一个方法，它也可以只是扩展继承类的代码，使用Python的super（）方法来调用父方法。可以在调用super（）方法之前和之后，在原有逻辑周围添加新逻辑。

小窍门

最好避免更改方法的函数的功能识别标志（即保持相同的参数），以确保与其关联的现有的调用能继续正常工作。如果需要添加其他参数，请将它们作为可选的关键字参数（使用默认值）。

原来的“**Clear All Done**”方法不再适用于我们的任务共享模块，因为它清除所有任务，而不管其他用户。我们需要修改它，以便它只清除当前用户任务。

为此，我们将使用新版本重写（或替换）原始方法，首先查找当前用户的已完成任务的列表，然后停止它们的活动：

**@api.multi**

**def do\_clear\_done(self):**

**domain = [('is\_done', '=', True),**

**'|', ('user\_id', '=', self.env.uid),**

**('user\_id', '=', False)]**

**dones = self.search(domain)**

**dones.write({'active': False})**

**return True**

为了清楚起见，我们首先构建要用于查找要清除记录的过滤器表达式。

这个过滤器表达式遵循称为domain的Odoo特定语法：它由条件组成的列表表示，其中每个条件是元组。

这些条件用AND（＆）运算符隐式连接。OR操作，使用管道符号| ，用来连接后面两个条件元组。我们将在第6章“视图 - 设计用户界面”中介绍有关域的更多详细信息。

此处域使用过滤将所有已完成的任务设置为（'is\_done'，'='，True），将仍有当前用户的情况设置为（'user\_id'，'='，self.env.uid），或将没有当前用户的情况设置为（'user\_id'，'='，False）。

然后，我们使用search方法获取一个具有完成记录的记录集，并对其进行批量写入，将active字段设置为False。这里的Python False值表示数据库中的NULL值。

在这种情况下，我们完全重写了父方法，将其替换为一个新的实现方法，但这不是我们通常想要做的。相反，我们应该扩展现有的逻辑与一些额外的操作。否则，我们可能会破坏已经存在的功能。

为了使重写的方法保持已经存在的逻辑，我们使用Python的super（）语句来调用父方法的版本。让我们看一个例子。

我们可以改进do\_toggle\_done（）方法，以便它只对分配给当前用户的任务执行其操作。这是实现的代码：

**from odoo.exceptions import ValidationError**

**# ...**

**# class TodoTask(models.Model):**

**# ...**

**@api.multi**

**def do\_toggle\_done(self):**

**for task in self:**

**if task.user\_id != self.env.user:**

**raise ValidationError(**

**'Only the responsible can do this!')**

**return super(TodoTask, self).do\_toggle\_done()**

继承类中的方法以for循环开始，以检查任何要切换的任务都不属于另一个用户。如果这些检查通过，它然后继续调用父类方法，使用super（）。如果没有提出错误，我们应该使用这个Odoo内置异常。与此处情况最相关的是ValidationError和UserError。

这些是重写和扩展模型类中定义的业务逻辑的基本技术。接下来，我们将了解如何扩展用户界面视图。

## 5.3 扩展视图

表单、列表和搜索视图是使用arch XML结构定义的。 要扩展视图，我们需要一种方法来修改这个XML。 这意味着需要定位XML中的元素位置，然后在这些位置引入修改。

继承视图正可以实现这个功能。一个继承视图声明如下：

**<record id="view\_form\_todo\_task\_inherited" model="ir.ui.view">**

**<field name="name">Todo Task form - User extension</field>**

**<field name="model">todo.task</field>**

**<field name="inherit\_id" ref="todo\_app.view\_form\_todo\_task"/>**

**<field name="arch" type="xml">**

**</field>**

**</record>**

inherit\_id字段使用特定属性ref引用一个外部标识符来指定要被扩展的视图。 外部标识符将在第4章“模块数据”中更详细地讨论。

对于XML，在其中定位元素的最好方法是使用XPath表达式。 比如，以上一章中定义的表单视图为例，其中一个XPath表达式是// field [@name] ='is\_done',这个表达式就是用于定位<field name =“is\_done”>元素的。 这个表达式查找name属性等于is\_done的任意field元素。

有关XPath的详细信息，请访问

https://docs.python.org/2/library/xml.etree.elementtree.html#xpath-support。

如果XPath表达式匹配到多个元素，只有第一个元素会被修改。 因此，表达式应该使用独特的属性以使其指定尽可能具体。 使用name属性是确保我们找到扩展点的确切xml元素的最简单方法。 因此，在我们的视图XML元素上定义name属性是很重要的。

扩展点定位后，您可以修改它或在其附近添加XML元素。 下面是一个写在arch中的实现在is\_done字段之前添加date\_deadline字段的具体例子：

**<xpath expr="//field[@name]='is\_done'" position="before">**

**<field name="date\_deadline" />**

**</xpath>**

幸运的是，Odoo为此提供了快捷符号，因此大多数时候我们可以完全避免XPath语法。 代替前面的XPath元素，我们仅使用要定位的元素的特定属性及定位类型相关信息就可以了。前面的XPath，我们可以写成：

**<field name="is\_done" position="before">**

**<field name="date\_deadline" />**

**</field>**

请注意，如果字段在同一视图中多次出现，则应始终使用XPath语法。 这是因为在字段第一次出现的时候Odoo将停止查找，而这可能会将您的更改应用在错误的字段上。

通常，我们是在现有字段旁边添加新字段，因此<field>标记经常用作定位器。 但是也可以使用任何其他标签：如<sheet>，<group>，

等。 name属性通常是匹配元素的最佳选择，但有时，我们可能需要使用其他东西：例如CSS类元素。 Odoo将找到具有指定的所有属性的第一个元素。

注意 在版本9.0之前，string属性（用于显示标签文本）也可以用作扩展定位器。 从9.0开始，这种做法不再允许了。 这个限制与对这些字符串进行操作的语言翻译机制有关。

与定位器元素一起使用的position属性是可选的，并且可以具有以下值：

**after:**将内容添加到父元素之中，匹配的节点之后。

**befire:**添加内容在匹配节点之前。

**inside**（默认值）:匹配节点内的追加内容。

**replace:**替换匹配的节点。如果使用空内容，它将删除该匹配的元素。从Odoo 10开始，它还允许用其他标记包装一个元素，通过在内容中使用$0来表示被替换的元素。

**attributes：**修改匹配元素的XML属性。在元素内容使用<attribute name =“attr-name”>实现给属性name设置新属性值attr-name。

例如，在Task表单中，我们有active字段，但它可见不是那么有用。 我们可以将它对用户隐藏。 这可以通过设置invisible属性来完成：

**<field name="active" position="attributes">**

**<attribute name="invisible">1</attribute>**

**</field>**

设置invisible属性来隐藏元素是使用定位器元素replace属性删除节点的一个很好的替代方法。 应该避免删除节点，因为它可能破环那些利用删除的节点作为选择器添加其他元素的依赖模块。

### 5.3.1 扩展表单视图

将所有先前的表单元素放在一起，我们可以添加新字段并隐藏active字段。 扩展to-do tasks表单的继承视图的完整代码如下：

**<record id="view\_form\_todo\_task\_inherited" model="ir.ui.view">**

**<field name="name">Todo Task form - User extension</field>**

**<field name="model">todo.task</field>**

**<field name="inherit\_id" ref="todo\_app.view\_form\_todo\_task"/>**

**<field name="arch" type="xml">**

**<field name="name" position="after">**

**<field name="user\_id">**

**</field>**

**<field name="is\_done" position="before">**

**<field name="date\_deadline" />**

**</field>**

**<field name="active" position="attributes">**

**<attribute name="invisible">1</attribute>**

**</field>**

**</field>**

**</record>**

这段定义应该添加到我们模块中的views/todo\_task.xml文件中的<odoo>元素内，如上一章所示。

小贴士

继承的视图还可以被继承，但由于这会创建更复杂的依赖关系，因此应该避免这样做。 您应该尽可能继承原始视图。 此外，我们不要忘记将data属性添加到\_\_manifest\_\_.py清单描述文件中，如下：

**'data'：['views / todo\_task.xml']**

### 5.3.2 扩展树视图和搜索视图

列表视图和搜索视图的扩展也是通过arch XML结构定义的，且与表单视图扩展方式相同。 我们将继续给出扩展列表和搜索视图的示例。

对于列表视图，如果要添加责任人字段，我们定义如下：

**<record id="view\_tree\_todo\_task\_inherited" model="ir.ui.view">**

**<field name="name">Todo Task tree - User extension</field>**

**<field name="model">todo.task</field>**

**<field name="inherit\_id" ref="todo\_app.view\_tree\_todo\_task"/>**

**<field name="arch" type="xml">**

**<field name="name" position="after">**

**<field name="user\_id" />**

**</field>**

**</field>**

**</record>**

对于搜索视图，我们添加一个按任务责任人的搜索和用于筛选当前用户的任务以及未分配给任何人的任务的预定义筛选器，定义如下：

**<record id="view\_filter\_todo\_task\_inherited" model="ir.ui.view">**

**<field name="name">Todo Task tree - User extension</field>**

**<field name="model">todo.task</field>**

**<field name="inherit\_id" ref="todo\_app.view\_filter\_todo\_task"/>**

**<field name="arch" type="xml">**

**<field name="name" position="after">**

**<field name="user\_id" />**

**<filter name="filter\_my\_tasks"**

**string="My Tasks" domain="[('user\_id','in',[uid,False])]" />**

**<filter name="filter\_not\_assigned"**

**string="Not Assigned" domain="[('user\_id','=',False)]" />**

**</field>**

**</field>**

**</record>**

不用太在意这些视图的具体语法。 我们将在第6章“视图 - 设计用户界面” 中更详细地介绍它们。

## 5.4 更多的模型继承机制

我们已经看到了模型的基本扩展，在官方文档中称为类继承， 这是最常用的继承，很容易想到它的应用场景，比如对现有模型的扩展， 添加新功能时，它们将添加到现有模型中，不会创建新模型。

我们还可以从多个父模型继承，在\_inherit属性将值列表设置。 这样，我们可以使用混合类。 混合类是实现我们可以添加到其他模型的通用特性的模型。 它们不会被直接使用，而是像一个容器，可以添加其他模型。

如果我们想使用具有不同于父模型的值的\_name属性，我们将获得一个新模型重用来自继承的特性，但是具有自己的数据库表和数据。 官方文档称为原型继承，在这里你的模型，将创建一个全新的，一个旧的。 添加新功能时，它们将添加到新模型中。 现有模型中没有改变。

还有代理继承方法，使用\_inherits属性，它允许一个模型以透明的方式包含其他模型，而在后台，每个模型处理自己的数据。 你拿一个模型并扩展它，添加新功能时，它们将添加到新模型中，现有模块没有变化。 新模型中的记录有到原始模型中的记录的链接，并且原始模型的字段被公开并且可以直接在新模型中使用。

让我们更详细地探讨这些可能性。

### 5.4.1 使用原型继承复制特征

我们之前使用的扩展模型的方法仅使用\_inherit属性。 我们定义了一个继承了todo.task模型的类，并向它添加了一些功能。 类属性\_name未设置; 隐藏的，它是todo.task。

但是，使用\_name属性允许我们创建一个新的模型并从继承模型复制功能。 这里是一个例子：

**from odoo import models**

**class TodoTask(models.Model):**

**\_name = 'todo.task'**

**\_inherit = 'mail.thread'**

这通过将mail.thread模型中的特性复制到todo.task模型中来扩展它。 mail.thread模型实现了Odoo消息和跟随器功能，并且是可重复使用的，因此很容易将这些功能添加到任何模型中。

复制意味着继承的方法和字段也将在继承模型中可用。 对于字段，这意味着它们也将被创建并存储在目标模型的数据库表中。 原始（继承）和新（继承）模型的数据记录保持不相关。 只有定义是共享的。

稍后，我们将详细讨论如何使用这个来添加mail.thread及其社交网络功能到我们的模块。 实际上，当使用mixins时，我们很少从常规模型继承，因为这会导致相同数据结构的重复。

Odoo还提供了避免数据结构重复的委托继承机制，因此通常在从常规模型继承时是首选。 让我们更详细地看看它。

### 5.4.2 使用委托继承嵌入模型

委托继承不常用，但它可以提供非常方便的解决方案。 它通过\_inherits属性（注意s）通过字典映射继承模型与字段的关系，并关联它们。 一个很好的例子是标准用户的模型，res.users; 它嵌入了一个合作伙伴模型：

from odoo import models, fields

class User(models.Model):

\_name = 'res.users'

\_inherits = {'res.partner': 'partner\_id'}

partner\_id = fields.Many2one('res.partner')

使用委托继承，res.users模型嵌入继承的模型res.partner，以便在创建新的User类时，还将创建一个合作伙伴，并且对其的引用保留在User类的partner\_id字段中, 它与面向对象编程中的多态性概念有一些相似之处。

通过委派机制，来自继承模型和合作伙伴的所有字段都可用，就像它们是User类字段一样。 例如，合作伙伴名称和地址字段显示为User类字段，但实际上，它们存储在链接的合作伙伴模型中，不会发生数据重复。

与原型继承相比，这样做的优点是不需要在几个表之间重复数据结构，例如地址。 任何需要包含地址的新模型都可以将其委派给嵌入式合作伙伴模型。 如果在合作伙伴地址字段中引入了修改，则这些修改会立即提供给嵌入它的所有模型！

注意：

请注意，使用委托继承，字段是继承的，但方法不是。

### 5.4.3 添加社交网络功能

社交网络模块 (技术名称mail) 提供留言板功能和关注的特征，以及消息和通知功能。这些功能是我们经常会想要添加在模块中的，所以让我们学习如何去做。

社交网络邮件功能由mail.thread模型的mail模块提供。要将其添加到自定义的模型，我们需要执行以下操作︰

依赖mail模块

继承mail.thread类

将关注者和线程小部件添加到form视图

为关注者设置记录规则。

我们按照以上步骤。

第一点，在我们模块的\_\_manifest\_\_.py清单文件中添加对mail的依赖︰

**‘depends’: ['todo\_app'，'mail'],**

第二点，使用我们以前使用的\_inherit属性继承mail.thread。但我们的类已经在使用 \_inherit 属性。幸运的是，它可以接受列表，所以我们可以写成列表并添加mail.thread:

**\_name = 'todo.task'**

**\_inherit = ['todo.task'，'mail.thread']**

mail.thread是一种抽象的模型。抽象的模型和普通模型类似，只是他们没有数据库表示;也没有实际的表。抽象的模型不是直接使用。相反，他们都是被混合使用。我们可以将它们看作模板与准备使用的功能。若要创建一个抽象类，我们只需要在创建模块时，使用models.AbstractModel 替代models.Model。 第三点，我们在form视图的底部添加社交网络部件。通过扩展原视图的方式添加。我们使用view\_form\_todo\_task\_inherited继承已经创建的视图，并其arch数据块中添加以下内容︰

**<sheet position="after">**

**<field name="message\_follower\_ids"**

**widget="mail\_followers" />**

**<field name="message\_ids" widget="mail\_thread" />**

**</sheet>**

在这里添加的两个字段未被我们进行显式声明，他们由mail.thread模型声明的。 最后的一步，即第四步，是关注者设置记录规则︰ rowlevel 访问控制。如果我们的模型需要限制其他用户无法访问记录，就需要设置记录规则。在这里，我们希望每个待办任务记录可以被任何它的关注者看见。 我们已经为待办任务模型定义了记录规则，所以我们需要进行调整以添加这项新规定。这是在下一节我们会做的一件事情。

## 5.5 修改数据

与视图不同，数据记录不包含 XML的arch数据块，也不能使用 XPath 表达式扩展。但他们仍可修改替换其字段中的值。

**<record id="x" model="y">**

数据记录加载时，实际上对模型y执行了create或update操作︰ 如果记录x不存在，则创建它; 否则，更新原来对应的值。 其它模块可以使用<model>.<identifier>全局标识符来访问，所以，可能会覆盖其它模块访问时的数据。

注意

请注意，点（.）已被保留，用以分隔模块名称和对象标识符，所有不能在标识符名称中使用它。可以用下划线替代。

### 5.5.1 修改菜单和操作记录

我们看看下面一个例子，改变todo\_app模块定义的菜单选项。我们添加todo\_user/views/todo\_task.xml 文件︰

**< ！ — — 修改菜单项-->**

**<record id="todo\_app.menu\_todo\_task" model="ir.ui.menu">**

**<field name="name">My To-Do</field>**

**</record>**

我们还可以修改该菜单项使用的action操作。action操作有一个可选的上下文属性。它可以为视图字段和筛选器提供默认值。我们启用在本章前面定义的My Tasks筛选器，︰

**<record model="ir.actions.act\_window"**

**id="todo\_app.action\_todo\_task">**

**<field name="context">**

**{'search\_default\_filter\_my\_tasks': True}**

**</field>**

**</record>**

### 5.5.2 修改安全记录规则

To-Do 应用程序包括一个记录规则，以确保每个任务只对创建它的用户可见。但现在，随着社交功能的添加，我们需要任务跟随者也访问它们。社交网络模块本身不处理这一点。 此外，现在任务可以有用户分配给他们，因此更有意义的是，访问规则在负责的用户而不是创建任务的用户上工作。 计划将与我们处理菜单项的做法相同：覆盖todo\_app.todo\_task\_user\_rule以将domain\_force字段修改为新值。 约定是将安全相关文件保存在安全子目录中，因此我们将创建具有以下内容的security / todo\_access\_rules.xml文件：

**<？xml version =“1.0”encoding =“utf-8”？>**

**<odoo>**

**<data noupdate =“1”>**

**<record id =“todo\_app.todo\_task\_per\_user\_rule”model =“ir.rule”>**

**<field name =“name”>所有者和关注者的ToDo任务</ field>**

**<field name =“model\_id”ref =“model\_todo\_task”/>**

**<field name =“groups”eval =“[（4，ref（'base.group\_user'））]”/>**

**<field name =“domain\_force”>['|'，（'user\_id'，'in'，[user.id，False]），（'message\_follower\_ids'，'in'，[user.partner\_id.id]）] </field>**

**</ record>**

**</ data>**

**</ odoo>**

这将从 todo\_app 模块覆盖 todo\_task\_per\_user\_rule 记录规则。新的域过滤器现在使任务对负责用户 user\_id可见，或者如果负责用户未设置（等于False），则对任何人都可见。它对所有的任务关注者也是可见的。 记录规则在用户变量可用且表示当前会话用户的记录的上下文中运行。由于关注者是合作伙伴，而不是用户，而不是user.id，我们需要使用user.partner\_id。 组字段是一对多关系。在这些字段中编辑数据使用特殊符号。这里使用的代码4是附加到相关记录的列表。也经常使用代码6，而是用新的列表完全替换相关的记录。我们将在第4章，模块数据中更详细地讨论这个标记。 记录元素的noupdate =“1”属性意味着此记录数据将仅在安装操作中写入，并且在模块升级时将被忽略。这允许它进行自定义，而不会承担覆盖自定义的风险，并且在将来某个时间进行模块升级时丢失它们。 小技巧 在开发时使用处理数据文件可能很棘手，因为在模块升级时将忽略对XML定义的后续编辑。为了避免这种情况，您可以重新安装模块。这通过使用-i的命令行更容易完成 像往常一样，我们不能忘记将新文件添加到\_\_manifest\_\_.py中的data属性：

**'data'：['views / todo\_task.xml'，**

**'security/todo\_access\_rules.xml']，**

## 5.6 第三章 小结

您现在应该能够通过扩展现有的模块来创建自己的模块。 为了演示如何做到这一点，我们扩展了我们在上一章中创建的 To-Do 模块，为组成应用程序的多个层添加了新功能。 我们扩展了 Odoo 模型以添加新字段并扩展其业务逻辑。 接下来，我们修改了视图以使新字段可用。 最后，我们学习了如何从其他模型继承，并使用它们将社交网络添加到To-Do应用程序。 前三章向我们概述了 Odoo 开发中涉及的常见活动，从 Odoo 安装、设置到模块创建和扩展。 接下来的章节将分别关注Odoo开发的一个特定领域，其中大部分是我们在这些第一章中简要介绍的。 在下一章中，我们将更详细地讨论数据序列化、XML和CSV数据文件的使用。

# 6 第四章 模块数据

大部分 Odoo 的模块定义，比如使用者介面及安全规则，实际上是存储在特定的数据库列表的数据记录中，模块中建立的XML和CSV文件并不使用，而是在Odoo程序在运行的時候，它们被载入数据库的表中。

因此，Odoo的一個重要的部分是关于作为資料所使用的文件因此可以較晚載入数据库(序列化)。

模块也可以擁有預設及演示数据，資料表現允許添加至我们的模块，此外，理解Odoo数据表現格式對於在專案執行中匯出和導入業務数据很重要。

## 6.1 了解外部标识符

外部ID（也稱为XML ID）是人類可讀的字串標識符，其唯一標示Odoo中特定的記錄。這在將数据加載到Odoo時很重要。

一個原因是當升級模块時，其資料文件將被再次加載至数据库，並且我们需要檢測已經存在的記錄，

是为了更新记录而不是創建新的重複的記錄。

另一個原因是为了支援互相關聯的資料：数据記錄必須能夠與其他数据記錄關聯。实际的数据库ID是在模块安裝期間

由数据库自動分配的具有順序的編號。而外部ID提供了一種引用相關記錄的方法但不需要事先知道数据库將被分配給其 记录的ID，允許我们定义Odoo数据文件中的資料關係。

Odoo則負責將外部ID名稱轉換为实际数据库分配給它们的ID。這背後的機制很簡單：Odoo管理一個表單其內容儲存所

匹配的外部IDs與對應的數字数据庫ID：ir.model.data模型。

要檢查現有映射匹配，請轉到設置上方的技術部分的目錄並選擇“序列與身份標識”下的“外部標識符”目錄項。

例如，如果我们看外部IDs列表，並透過todo\_app模块過濾它们，我们會看到模块創建前所生成的外部IDs：

我们可以看到外部標識符有一個完整的ID標籤。注意它是運用點( . )去連接模块名稱及其標識符名稱而組成例如:

todo\_app.action\_todo\_task。

外部標識符只在Odoo模塊內部是唯一的，若兩個模块意外地選擇了相同標識符名稱是沒有衝突的風險。要構建一個通用

的唯一ID，Odoo結合模塊名稱與实际外部標識符名稱。你可以在Complete(通用) ID欄位中看到。

在資料文件中使用外部ID時，可以選擇要使用完整ID或只是外部ID名稱。通常使用外部ID名稱較为簡單，但是完整ID使

我们能夠參考來自其他模塊的数据記錄。當要這樣做時，請確保那些模块包含在模块依賴項中，以確保這些記錄在先前即 加載。

在列表的上方，我们有todo\_app.action\_todo\_task的通用ID。這是我们为模块創建的目錄操作，其關聯到相應的目錄

項中。通過點擊它，我们去表單查看詳細資料在todo\_app模塊中action\_todo\_task的外部ID映射到一個 ir.actions.act\_window模型中特定的記錄ID--72。

此外提供一種方便地引用其他記錄的方法，外部ID還允許您避免重複導入数据時造成資料重複。如果外部ID已經存在，

現有記錄將被更新;你不需要創建新記錄。這就是为什麼在後續模塊升級，先前加載的記錄被更新而不是被複製。

### 6.1.1 寻找外部标识符

当准备和定义模块的数据文件时，我们经常需要查找引用所需的现有外部标识符。

我们可以使用前面显示的外部标识符菜单，但是开发人员菜单可以提供更方便的方法。 您可以在第1章“Odoo开发入门”中查看它，开发人员菜单在设置仪表板右下角的某个选项中激活。

要找到数据记录的外部标识符，在适当的表单视图上，从开发者菜单中选择查看元数据选项。 这将显示一个对话框，其中包含数据库ID和记录的外部标识符（也称为XML ID）。

例如，要查找演示用户ID，我们可以导航到表单视图，在设置|下 用户并选择“查看元数据”选项，

图片

要查找视图元素（如窗体，树，搜索或操作）的外部标识符，“开发人员”菜单也是一个很好的帮助来源。 为此，我们可以使用其管理视图选项或使用编辑<视图类型>选项打开所需视图的信息。 然后，选择其“查看元数据”选项。

## 6.2 导出和导入数据

我们将开始探索数据导出和导入如何从Odoo的用户界面，并从那里，我们将移动到更多的技术细节如何使用我们的addon模块中的数据文件。

### 6.2.1 导出数据

数据导出是任何列表视图中的标准功能。要使用它，我们必须首先选择要导出的行，方法是选择最左边的相应复选框，然后从更多按钮中选择导出选项。

下面是一个使用最近创建的待办事项任务的示例：

图

我们还可以勾选列标题中的复选框。它将一次检查所有记录，并将导出与当前搜索条件匹配的所有记录。

注意

在以前的Odoo版本中，只有屏幕上看到的记录（当前页）实际上将被导出。自从Odoo 9，这是改变和复选框的标题将导出所有匹配当前过滤器的记录，而不只是当前显示的。这对导出不适合屏幕的大量记录非常有用。

“导出”选项使我们转到对话框窗体，在其中我们可以选择要导出的内容。 “导入兼容导出”选项确保导出的文件可以导回到Odoo。我们将需要使用这个。

导出格式可以是CSV或Excel。我们更喜欢使用CSV文件来更好地了解导出格式。接下来，我们选择要导出的列，然后单击导出到文件按钮。这将开始下载包含导出数据的文件：

图

如果我们按照这些说明并选择上面屏幕截图中显示的字段，我们最终会得到一个类似于以下内容的CSV文本文件：

**"id","name","user\_id/id","date\_deadline","is\_done" "todo\_user.todo\_task\_a","Install Odoo","base.user\_root","2015- 01-30","False" "\_\_export\_\_.todo\_task\_9","Create my first module","base.user\_root","","False"**

注意

注意Odoo自动导出一个额外的id列。这是分配给每个记录的外部标识符。如果尚未分配，则使用\_\_export\_\_自动生成一个新的代替实际模块名称。新标识符仅分配给尚未有标识符的记录，并且从那开始，它们保持绑定到同一记录。这意味着后续导出将保留相同的外部标识符。

### 6.2.2 导入数据

首先，我们必须确保导入功能已启用。从Odoo 9开始导入功能在默认情况下被启用。如果没有，则可从顶部的设置菜单的常规设置选项中进行设置。导入|导出部分下面有允许用户从CSV/XLS/XLSX/ODS文件导入数据复选框，应该启用。

注意[编辑]

此功能由“初始设置工具”插件（base\_setup是技术名称）提供。允许用户导入...复选框的实际效果是安装或卸载base\_setup。

启用此选项后，列表视图将在列表顶部的创建按钮旁边显示导入选项。

让我们先对我们的待办事项数据进行批量编辑。在电子表格或文本编辑器中打开我们刚刚下载的CSV文件，并更改几个值。此外，添加一些新行，将id列留空。

如前所述，第一列id为每一行提供唯一的标识符。这允许更新现有记录，而不是在将数据导入Odoo时重复它们。对于添加到CSV文件中的新行，我们可以选择可选择的外部标识符，或者为id列留空，将为它们创建一个新记录。

将更改保存到CSV文件后，点击导入选项（在创建按钮旁边），我们将向导入助手显示。

在那里，我们应该选择磁盘上的CSV文件位置，然后单击验证以检查其格式的正确性。由于要导入的文件基于Odoo导出，因此它将有效：

图

现在我们可以点击导入，然后可以看见;我们的修改和新的记录应该已经加载到Odoo。

### 6.2.3 CSV数据文件相关记录

在前面的示例中，负责每个任务的用户是用户模型中的相关记录，具有多对一（或外键）关系。用于它的列名称是user\_id / id，字段值是相关记录的外部标识符，例如管理员用户的base.user\_root。

技巧

仅当您从/向同一数据库导出和导入时，才建议使用数据库ID。通常，您更喜欢使用外部标识符。

如果使用外部标识符，则关系列应该在其名称后附加/ id，如果使用数据库（数字）ID，则应该/.id。或者，可以使用冒号（:)替换斜杠以获得相同的效果。

类似地，也支持多对多关系。多对多关系的示例是用户和组之间的关系：每个用户可以在许多组中，并且每个组可以具有许多用户。此类型字段的列名称应附加/ id。字段值接受以逗号分隔的外部标识符列表，由双引号括起。

例如，待办任务跟随者在待完成任务和合作伙伴之间具有多对多关系。其列名称应为follower\_ids / id，并且具有两个关注者的字段值可能如下所示：

"\_\_export\_\_.res\_partner\_1,\_\_export\_\_.res\_partner\_2"

最后，一对多关系也可以通过CSV导入。这种类型关系的典型示例是具有几行的文档头。注意，一对多关系总是多对一关系的逆。每个文档头可以有多行。并且每行都有一个头。

我们可以在公司模型中看到这种关系的一个例子（表单视图在设置菜单中可用）：每个公司可以有多个银行帐户，每个帐户都有自己的详细信息;相反，每个银行帐户记录属于并且与仅一个公司具有多对一关系。

我们可以将公司及其银行帐户导入到一个文件中。这里有一个例子，我们加载一家公司有三家银行：

**id,name,bank\_ids/id,bank\_ids/acc\_number,bank\_ids/state base.main\_company,YourCompany,\_\_export\_\_.res\_partner\_bank\_4,123 456789,bank ,,\_\_export\_\_.res\_partner\_bank\_5,135792468,bank ,,\_\_export\_\_.res\_partner\_bank\_6,1122334455,bank**

我们可以看到前两列（id和name）在第一行中有值，在下两行中为空。他们有头记录的数据，这是公司。

其他三列都以bank\_ids /为前缀，并且在所有三行上都有值。他们有三个相关行的数据为公司的银行帐户。第一行有公司和第一家银行的数据，接下来的两行只有额外的公司和银行的数据。

当从GUI导出和导入时，这些是必需的。在新的Odoo实例中设置数据或准备要包含在Odoo模块中的数据文件非常有用。接下来，我们将了解有关在模块中使用数据文件的更多信息。

## 6.3 模块数据

模块使用数据文件将其配置加载到数据库，默认数据和演示数据中。可以通过CSV和XML文件完成。为了完整性，也可以使用YAML文件格式，但是它很少用于加载数据; 因此，我们不会讨论它。

模块使用的CSV文件与我们已经看到并用于导入功能的CSV文件完全相同。当在模块中使用它们时，一个附加的限制是文件名必须与要加载数据的模型的名称匹配，这样系统才可以推断应该将数据导入相应的模型。

数据CSV文件的常见用法是访问加载到ir.model.access模型中的安全定义。 它们通常使用名为ir.model.access.csv的CSV文件。

上一节：CSV数据文件相关记录

### 6.3.1 演示数据

Odoo附加模块可以安装演示数据，并且这被认为是良好的做法。这对于提供要在测试中使用的模块和数据集的使用示例很有用。使用\_\_manifest\_\_.py清单文件的demo属性声明模块的演示数据。 与data属性一样，它是一个文件名列表，其中包含模块中相应的相对路径。

现在是向todo\_user模块添加一些演示数据的时候了。 我们可以从待办事项任务导出一些数据开始，如上一节所述。 约定是将数据文件放在data /子目录中。 因此，我们应该将这些数据文件保存在todo\_user addon模块中作为data / todo.task.csv。 由于此数据将由我们的模块拥有，因此我们应编辑id值以删除标识符中的\_\_export\_\_前缀。

例如，我们的todo.task.csv数据文件可能如下所示：

**id,name,user\_id/id,date\_deadline todo\_task\_a,"Install Odoo","base.user\_root","2015-01-30" todo\_task\_b","Create dev database","base.user\_root",**""

我们不能忘记将此数据文件添加到\_\_manifest\_\_.py清单演示属性：

**'demo': ['data/todo.task.csv'],**

下次我们更新模块时，只要它已安装演示数据启用，将导入文件的内容。 请注意，每当执行模块升级时，将重新导入此数据。

XML文件也用于加载模块数据。 让我们进一步了解XML数据文件可以做的但CSV文件却不能。

## 6.4 XML数据文件

虽然CSV文件提供了一种简单而紧凑的格式来表示数据，但XML文件更加强大，可以更好地控制加载过程。 它们的文件名不需要与要加载的模型匹配。 这是因为XML格式更加丰富，并且信息由文件中的XML元素提供。

我们已经在前面的章节中使用了XML数据文件。 用户界面组件（例如视图和菜单项）实际上是存储在系统模型中的记录。 模块中的XML文件是用于将这些记录加载到服务器中的方法。

为了展示这一点，我们将向todo\_user模块data / todo\_data.xml添加第二个数据文件，其中包含以下内容：

**<?xml version="1.0"?>**

**<odoo>**

**<record model="todo.task" id="todo\_task\_c">**

**<field name="name">Reinstall Odoo</field>**

**<field name="user\_id" ref="base.user\_root" />**

**<field name="date\_deadline">2015-01-30</field>**

**<field name="is\_done" eval="False" />**

**</record>**

**</odoo>**

此XML等同于我们刚才在上一节中看到的CSV数据文件。

XML数据文件有一个<odoo> top元素，其中我们可以有多个对应于CSV数据行的<record>元素。

备注

在版本9.0中引入了数据文件中的<odoo> top元素，并替换了以前的<openerp>标记。 top元素中的部分仍然受支持，但现在它是可选的。 事实上，现在<odoo>和<data>是等效的，因此我们可以使用一个作为顶部元素用于我们的XML数据文件。

一个<record>元素有两个强制属性，即model和id（记录的外部标识符），并且每个字段包含一个<field>标签。

请注意，字段名称中的斜杠符号在此处不可用; 我们不能使用<field name =“user\_id / id”>。 相反，ref特殊属性用于引用外部标识符。 我们将在稍后讨论关系到许多字段的值。

### 6.4.1 数据的noupdate属性

重复数据加载时，将重写上次运行中加载的记录。 重要的是要记住，这意味着升级一个模块将覆盖在数据库内可能已经进行的任何手动更改。 值得注意的是，如果使用自定义修改视图，那么这些更改将在下一次模块升级时丢失。 正确的过程是为我们需要的更改创建继承的视图，如第3章“继承 - 扩展现有应用程序”中所述。

此重新导入行为是默认值，但可以更改，以便在升级模块时，某些数据文件记录保持不变。 这是通过<odoo>或元素的noupdate =“1”属性完成的。 这些记录将在安装addon模块时创建，但在后续模块升级中不会对其进行任何操作。

在同一个XML文件中可以有多个<data>节。 我们可以利用这个来分隔数据，只导入一个，noupdate =“1”，每次升级时重新导入数据，noupdate =“0”。

noupdate标志存储在每个记录的外部标识符信息中。 可以使用技术菜单中的外部标识符表单直接手动编辑，使用不可更新复选框。

提示

noupdate属性在开发模块时可能很棘手，因为稍后对数据所做的更改将被忽略。 解决方案是，不使用-u选项升级模块，而是使用-i选项重新安装它。 使用-i选项从命令行重新安装将忽略数据记录上的noupdate标志

### 6.4.2 在XML里定义记录

每个<record>元素有两个基本属性id和model，并且包含为每列分配值的<field>元素。如前所述，id属性对应于记录的外部标识符，并且模型属性对应于要写入记录的目标模型。<field>元素有几种不同的方式来分配值。 让我们详细看看他们。

#### 6.4.2.1 设置字段值

<record>元素定义数据记录并包含<field>元素以设置每个字段的值。

field元素的name属性标识要写入的字段。

要写入的值是元素内容：字段的开始和结束标记之间的文本。 对于日期和数据时间，将正确转换“YYYY-mm-dd”和“YYYY-mm-dd HH：MM：SS”的字符串。 但对于布尔字段，任何非空值都将转换为True，并且“0”和“False”值将转换为False。

注意

在Odoo 10中改进了从数据文件读取Boolean False值的方式。在以前的版本中，任何非空值，包括“0”和“False”都转换为True。对于使用下面讨论的eval属性的布尔值是被推荐的。

#### 6.4.2.2 使用表达式设置值

定义字段值的更精细的替代方法是eval属性。 它评估一个Python表达式并将结果值分配给字段。

该表达式在除了Python内置函数之外还有一些在其他标识符的上下文中求值。 让我们看看他们。

要处理日期，可以使用以下模块：time，datetime，timedelta和relativedelta。 它们允许您计算日期值，这是在演示和测试数据中经常使用的日期值，以便使用的日期接近模块安装日期。 例如，要将值设置为昨天，我们将使用：

<field name="date\_deadline"

eval="(datetime.now() + timedelta(-1)).strftime('%Y-%m-%d')" />

要处理日期，可以使用以下模块：time，datetime，timedelta和relativedelta。 它们允许您计算日期值，这是在演示和测试数据中经常使用的日期值，以便使用的日期接近模块安装日期。 例如，要将值设置为昨天，我们将使用： <field name="user\_id" eval="ref('base.group\_user')" />

#### 6.4.2.3 设置字段值的关系

我们刚刚了解了如何使用eval属性和ref（）函数在多对一关系字段（如user\_id）上设置值。 但是有一个更简单的方法。

<field>元素还有一个ref属性，用于使用外部标识符设置many-to-one字段的值。 有了这个，我们可以设置user\_id的值：

**<field name="user\_id" ref="base.user\_demo" />**

对于一对多和多对多字段，需要相关ID的列表，因此需要不同的语法; Odoo提供了一种特殊的语法来写这种类型的字段。

以下示例取自Fleet官方应用程序，替换了tag\_ids字段的相关记录列表：

**<field name="tag\_ids"**

**eval="[(6,0,**

**[ref('vehicle\_tag\_leasing'),**

**ref('fleet.vehicle\_tag\_compact'),**

**ref('fleet.vehicle\_tag\_senior')]**

**)]" />**

要在一对多字段上写，我们使用三元组列表。 每个三元组是一个写命令，根据所使用的代码执行不同的事情：

(0,\_ ,{'field': value}) 创建一个新记录并将其链接到此记录

(1,id,{'field': value}) 更新已链接记录上的值

(2,id,\_) 取消链接并删除相关记录

(3,id,\_) 取消链接但不删除相关记录

(4,id,\_) 链接已存在的记录

(5,\_,\_) 取消链接，但不会删除所有链接的记录

(6,\_,[ids]) 使用提供的列表替换链接记录的列表

前面列表中使用的下划线符号表示不相关的值，通常填充为0或False

### 6.4.3 常用模型的快捷键

在第2章"建立你的第一个Odoo应用程序"中，我们发现在xml文件中除了<record>的元素，还有如act\_window >和<menuitem>元素。

这些常用类的快捷键可以通过<record> 元素加载。他们将数据加载到支持用户界面的基础模型中，在第6章“视图-设计用户界面”中，这部分内容会进行更详细的探讨。 作为参考，下列快捷元素可与他们加载数据的相应模型一起使用：

**<act\_window>**是窗口操作模型,ir.actions.act\_window

**<menuitem>**是菜单项模型,ir.ui.menu

**<report>**是报告行动模型,ir.actions.report.xml

**<template>**是存储在ir.ui.view 模型的QWeb模板

**<URL>** is the URL行为模型，ir.actions.act\_URL

### 6.4.4 XML数据文件的其它操作

直到现在,我们已经看到了如何使用XML文件添加或更新数据。但XML文件还允许您执行其他类型的操作来设置数据。特别是,他们可以删除数据,执行任意模型方法,触发工作流事件。

#### 6.4.4.1 删除记录

我们使用<delete>元素，通过ID或搜索域来查找目标记录，进行数据记录删除操作。

例如，使用搜索域找到要删除的记录：

**<delete**

**model="ir.rule"**

**search="**

**[('id','=',ref('todo\_app.todo\_task\_user\_rule'))]"**

**/>**

在这种情况下如果我们知道要删除的具体ID，我们可以直接使用它来达到同样的效果：

**<delete model="ir.rule" id="todo\_app.todo\_task\_user\_rule" />**

#### 6.4.4.2 触发功能和工作流

xml文件还可以通过<function>元素在其加载过程中执行方法。这可以用来建立演示和测试数据.

例如，CRM应用程序使用它来建立演示数据：

**<function**

**model="crm.lead"**

**name="action\_set\_lost"**

**eval="[ref('crm\_case\_7'), ref('crm\_case\_9')**

**, ref('crm\_case\_11'), ref('crm\_case\_12')]**

**, {'install\_mode': True}" />**

调用crm.lead类的action\_set\_lost 方法，通过eval属性传递两个参数。第一个是运行的IDS列表，下一个是要使用的上下文。

另一种XML数据文件用来执行操作的方式是通过<workflow>元素触发Odoo工作流。例如，工作流可以更改销售订单的状态或生成发票。销售应用程序不再使用工作流，但这个示例仍然可以在演示数据中找到：

**<workflow model="sale.order"**

**ref="sale\_order\_4"**

**action="order\_confirm" />**

现在，模型属性很明显，REF标识了我们正在执行的工作流实例。“action”是发送到工作流实例的工作流信号。

## 6.5 第四章小结

您已经了解了关于数据序列化的所有要点，并对我们在前几章中看到的XML方面有了更好的理解。我们还花了一些时间来理解外部标识符，这是一般的数据处理的重要概念，特别是模块配置。详细讲解了XML数据文件，您了解了在字段中设置值以及执行操作的几个选项，比如删除记录和调用模型方法。CSV文件和数据导入/导出特性也做了解释。对于Odoo的初始设置或者大规模编辑数据来说，这些都是很有价值的工具。

# 7 第五章 模型 - 结构化应用程序数据

在前面的章节中，我们对Odoo中创建新模块有个整体的认知。第二章-创建你的第一个应用中，我们创建了一个完整的新应用，第三章-继承-扩展已存在的应用，我们探索了继承以及运用它去创建一个新的扩展我们已有应用的模块。第四章-模块数据，我们讨论了如何为我们的模块添加初始值和demo数据

整个概述中，我们了解了在Odoo中一个后台应用的整个组成。在接下来的章节中，我们会逐一介绍这一系列的应用的组成的细节：模块，视图和业务逻辑。

在这章节中，你将学习怎样设计一个应用的数据结构以及怎样实现他们之间的关联关系。

## 7.1 将应用程序特性组织到模块中

在这之前，我们会通过一个例子来更好的理解这个概念。

Odoo的继承特性是一个有效的扩展特性。通过它，你可以在不直接修改第三方应用的情况下，对它进行扩展。这种组合特性也可以运用到面向对象的开发模式中，大型的应用可以分成许多小特性，一点一点的丰富。

这就使得不管在技术层面，还是用户体验层面都有效的降低了其复杂度。在技术层面，将一个大功能分成若干个小功能，可以更好的解决并且更容易开发。在用户体验层面，我们可以在一个简单友好的操作界面中，选择性的使用真实需要的功能。所以，我们将通过插件模块化的方式，一点点的添加我们的应用，最终实现完整的应用。

### 7.1.1 todo\_ui模块介绍

在前面的章节中，我们先创建了一个个人模块，然后再扩展它，使它能被其它人使用。

现在，我们需要提升我们的模块的用户体验，其中包括看板仪表盘。看板仪表盘是一个简单的工作流节点展示工具，它是从左往右的展示整个工作流的节点。我们可以编辑任务、修改流程，比如Waiting，Ready，Started，Done

我们添加数据结构以实现这样的功能。我们需要添加流程，这样更容易展示，也可以让任务分类。这本章中，我们主要学习数据模块。用户界面的这些特性，将在第六章-视图-自定义用户界面讲解。而看板视图将在第九章-QWeb和看板视图中讲解。

首先找出如何将我们的数据结构化,这样我们就可以设计支持它的模型。我们已经拥有中心实体：To-do Task。每个任务在同一时间只能处于一个节点，而任务可以有一个或多个标签。我们需要添加两个有对应特性的模块，并且它们之间需要有以下关系：

每个任务有一个状态，而每个状态可以有多个任务

每个任务可以有多个标签，并且每个标签可以关联多个任务

也就是说，任务和状态之间的关系是many2one，与标签之间的关系是many2many。换句话说，反过来的关系应该是：状态与任务之间的关系是one2many，而标签与任务之间的关系是many2many。

我们用~/odoo-dev/custom-addons/目录作为我们的模块主目录。我们需要在其中创建一个新的todo-ui目录作为新包。我们可以使用以下shell命令：

**$ cd ~/odoo-dev/custom-addons**

**$ mkdir todo\_ui**

**$ cd todo\_ui**

添加\_\_manifest\_\_.py文件，并且添加以下内容：

**{**

**'name': 'User interface improvements to the To-Do app',**

**'description': 'User friendly features.',**

**'author': 'Daniel Reis',**

**'depends': ['todo\_user']**

**}**

我们再添加上\_\_init\_\_.py文件，一个空的完美的模块就创建好了。 现在，我们就可以在Odoo中，安装并使用该模块了。

## 7.2 创建模型

为了to\_do任务拥有看板，我们使用stage. Stages是看板上的列，每一项任务都适合其中的一列:

1，编辑todo\_ui/\_\_init\_\_.py 导入models子模块

**from . import models**

2，创建todo\_ui/models 目录，并添加 \_\_init\_\_.py 文件

**from . import todo\_model**

3、增加todo\_ui/models/todo\_model.py python文件

**# -\*- coding: utf-8 -\*-**

**from odoo import models, fields, api**

**class Tag(models.Model):**

**\_name = 'todo.task.tag'**

**\_description = 'To-do Tag'**

**name = fields.Char('Name', 40, translate=True)**

**class Stage(models.Model):**

**\_name = 'todo.task.stage' \_description = 'To-do Stage'**

**\_order = 'sequence,name'**

**name = fields.Char('Name', 40, translate=True)**

**sequence = fields.Integer('Sequence')**

在这里，我们创建了两个类，这些类将在to\_do任务中被引用。

看下任务阶段，我们有一个Python类:stage，它基于models.Model类，它定义了一个新的odoo类todo.task.stage。我们也有两个字段：名称和序列。我们可以看到一些新的模型属性（加下划线前缀），让我们进一步了解下。

### 7.2.1 模型属性

模型类可以使用其他属性来控制它们的一些行为。这些是最常用的属性。

\_name 是我们创建odoo模型的内部标识符，创建新模型时，必含项。

\_description 是当用户界面显示模型时，一个方便用户的模型记录标题。它是可选项、推荐项。

\_order ：当浏览模型记录或者显示在列表视图时，设置默认顺序。它是一个用来作为SQL的文本字符串order by子句，所以它可以是你可以使用的任何东西，尽管它拥有灵活的行为、支持可翻译和多对一的字段名称。为了完整起见，还有更多的属性在后续案例中会使用：

\_rec\_name：用来指出引用关联字描述记录的字段，例如多对一关系。 默认情况下，它使用name字段，这是模型中常见的字段。但是这个属性允许我们使用任何其他字段来实现这个目的。

\_table：用来支持模型的数据库表名。通常，它是左自动计算，是下划线取代点的模型名称，但也可以设置特定的表名。

在第3章“继承–扩展现有的应用程序”中，我们已学习了 \_inherit 和\_inherits 属性。

### 7.2.2 模型和Python类

Odoo模型由Python类表示。在前面的代码中，有一个基于models.Model模型的Python类Stage，它定义了新的odoo模型： todo.task.stage.

Odoo模型被保存在一个中央注册中心，旧的Odoo版本中也被称为池。它是一个字典，它可以引用实例中可用的所有模型类，并且它可以被一个模型名引用。具体来说，模型方法代码中可以通过 self.env['x'] 获取一个代表model x的类的引用。

您可以看到，模型名称很重要，因为它们是用于访问注册中心。模型名称的约定规则是使用包含点的小写字母列表，如todo.task.stage。核心模块的其他例子有project.project, project.task, 或者 project.task.type。我们应该用单数形式todo.task 而不是todo.tasks。由于历史原因，有些核心模型没有遵循这些，例如res.users，但是规则并非如此。

模型名称必须是全局唯一的。正因为如此，第一个词应该对应于该模块涉及到的主要应用程序。在我们的例子中是 todo，核心模型的例子有 project, crm, 或者 sale.

另一方面 Python的类 在本地的python文件声明，用于它们的标识符仅对该文件中的代码有效。基于此，类标识符并没有要求在相关主应用程序中前置声明。所以我们为

todo.task.stage 命名类stage 是没有问题的。在其他模块中，不存在与可能的类发生冲突的风险。

类标识符可以使用两种不同的约定规则:snake\_case或CamelCase。从历史上看，Odoo的代码使用了snake\_case的案例，而且仍然可以找到使用这个约定的类。但是趋势是使用 CamelCase，因为它是Python标准定义的“PEP8”编码惯例。您可能已经注意到我们使用的是后一种形式。

### 7.2.3 瞬态和抽象模型

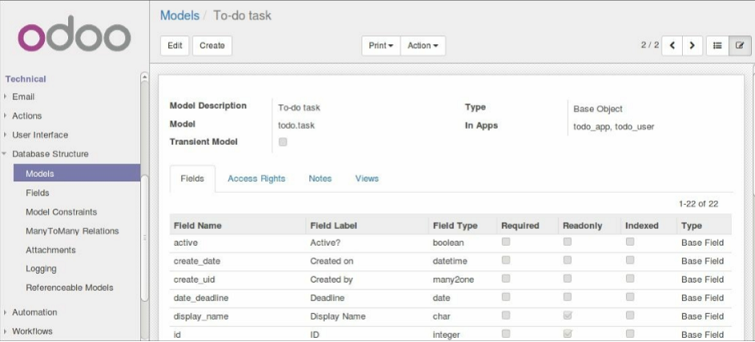
在前面的代码和绝大多数Odoo模型中，类都是基于models.Model类的。 这种类型的模型具有永久的数据库持久性：为它们创建数据库表，并且它们的记录被存储，直到被显式删除。 此外，Odoo还提供了另外两种模型类型：瞬态模型和抽象模型。

瞬态模型基于models.TransientModel类，用于向导式的用户交互。 它们的数据仍然存储在数据库中，但期望是临时的。 一个清理作业定期从这些表中清除旧数据。 例如，从设置|翻译菜单打开的加载语言对话框窗口，它使用瞬态模型来存储用户选择和实现向导逻辑。

抽象模型基于models.AbstractModel类,并且它不存储数据。 抽象模型作为可重用的功能集，利用Odoo的继承功能，混入到其他模型。 例如，mail.thread是一个抽象模型，由讨论插件提供，用于向其他模型添加消息和追踪关注功能。

### 7.2.4 检查现有模型

通过Python类创建的模型和字段本身的元数据可以通过用户界面进行访问。 在设置顶部菜单中，导航到技术| 数据库结构| 模型菜单项。 在这个菜单下，您将看到数据库中可用的所有模型的列表。 点击列表中的某一模型将打开一个包含其详细信息的表单：

这是一个检查模型结构的好工具，因为在这里，您可以看到一个模型来自不同模块的所有定制的结果。 本示例中，在右上角,您可以看到In Apps字段，它表示模型todo.task的定义来自todo\_app和todo\_user两个模块。

在图的下部区域中，有一些选项卡信息：模型的字段的快速参考，在安全组中授予的访问权限，以及列出此模型可用的视图。

我们可以从开发者菜单中的查看元数据选项，找到模型的外部标识符。 模型外部标识符或者XML IDs，由ORM自动生成但可以预简单推测出来：例如todo.task模型，外部标识符为model\_todo\_task。

小贴士

模型表单是可编辑的！ 这样就可以在这里创建和修改模型，字段和视图。 您可以在模块实现之前，使用它来构建原型。

## 7.3 创建字段

创建新模型后，下一步是向其中添加字段。 Odoo支持所有预期的基本数据类型，例如文本字符串，整数，浮点数，布尔值，日期，日期时间和图像/二进制等数据类型。

一些字段名称比较特别，因为它们由ORM为特殊目的而保留，或者因为一些默认内置功能使用一些默认字段名称。

让我们来探讨Odoo中可用的几种字段类型。

### 7.3.1 基本字段类型

我们现在有一个Stage模型，我们将扩展它添加一些额外的字段。 我们应该编辑todo\_ui / models / todo\_model.py文件并添加额外的字段定义，使它看起来像这样:

**class Stage(models.Model):**

**\_name = 'todo.task.stage'**

**\_description = 'To-do Stage'**

**\_order = 'sequence,name'**

**# String fields:**

**name = fields.Char('Name', 40)**

**desc = fields.Text('Description')**

**state = fields.Selection(**

**[('draft','New'), ('open','Started'),**

**('done','Closed')],'State')**

**docs = fields.Html('Documentation')**

**# Numeric fields:**

**sequence = fields.Integer('Sequence')**

**perc\_complete = fields.Float('% Complete', (3, 2))**

**# Date fields:**

**date\_effective = fields.Date('Effective Date')**

**date\_changed = fields.Datetime('Last Changed')**

**# Other fields:**

**fold = fields.Boolean('Folded?')**

**image = fields.Binary('Image')**

这里，我们有一个Odoo中可用的非关系字段类型的样本，每个字段都有期望的位置参数。

在大多数情况下，第一个参数是字段标题，对应于字符串字段参数; 这用作用户界面标签的默认文本。 它是可选的，如果未提供，将从字段名称自动生成标题。

对于日期字段名称，有使用日期作为前缀的约定。 例如，我们应该使用date\_effective字段而不是effective\_date。 类似的约定也适用于其他字段，如amount\_，price\_或qty\_。

这些是每个字段类型所期望的标准位置参数：

Char 需要第二个可选的参数大小，以获取最大文本大小。 建议不要使用它，除非有需要它的业务需求，例如固定长度的社会安全号码

Text 不同于Char，因为它可以保存多行文本内容，但期望相同的参数

Selection 是一个下拉选择列表。 第一个参数是可选选项的列表，第二个参数是字符串标题。 选择项是针对存储在数据库中的值和相应的用户界面描述的

（'值'，'标题'）元组的 列表。 当扩展继承时，selection\_add参数可用于将新项目附加到现有选择列表。

Html 存储为文本字段，但在用户界面上具有特定的处理，用于HTML内容呈现。 出于安全原因，默认情况下会对其进行清理，但可以覆盖此行为

Integer 只需要字段标题的字符串参数。

Float 有第二个可选参数，一个带有字段精度的（x，y）元组：x是浮点数的总位数，y是小数点位数

Date and Datetime 只需要字符串文本作为位置参数。 由于历史原因，ORM以字符串格式处理它们的值。 辅助函数应该用于将它们转换为实际的日期对象。

此外，datetime值以UTC时间存储在数据库中，但使用用户的时区首选项显示为本地时间。 这在第6章“视图 - 设计用户界面”中有更详细的讨论。

Boolean 保存True或False值，如您所料，并且只有一个字符串文本的位置参数。

Binary 存储文件状二进制数据，并且只需要字符串参数。 它们可以使用base64编码字符串由Python代码处理。

除了这些，我们还有关系字段，将在本章后面介绍。 但现在，还有更多要了解这些字段类型及其属性。

### 7.3.2 公共字段属性

字段可以在定义它们时设置属性。 根据字段类型，可以设置传递位置属性，而不必使用参数关键字，如上一节所示

例如，name = fields.Char（'Name'，40）可以使用位置参数。 如果使用关键字参数，同样可以写为name = fields.Char（size = 40，string ='Name'）。 有关关键字参数的详细信息，请参阅Python官方文档

https://docs.python.org/2/tutorial/controlflow.html#keyword-arguments

所有可用的属性都可以作为关键字参数传递。 这些是通常可用的属性和相应的参数关键字：

**string**是字段默认标签，用于在用户界面中使用。 除了选择和关系字段，它是第一个位置参数，因此大多数时候它不用作关键字参数

**default**为字段设置默认值。 它可以是一个静态值，例如一个字符串或一个可调用的引用，一个命名函数或一个匿名函数

（一个lambda表达式）。

**size**仅适用于Char字段，并且可以设置允许的最大大值。 目前的最佳做法是不使用它，除非它真的需要

**translate**仅适用于Char，Text和Html字段，并使字段内容可翻译，为不同语言保存不同的值。

**help**提供显示给用户的工具提示的文本。

**readonly = True**使该字段默认不能在用户界面上编辑。 这不是在API级别强制执行; 它只是一个用户界面设置

**index = True**将在字段上创建一个数据库索引

**copy = False**在使用重复记录功能copy（）ORM方法时，字段被忽略。 默认情况下，非关系字段是可复制的。

**groups**允许将字段的访问和可见性限制为仅某些组。 它需要以逗号分隔的安全组的XML ID列表，例如groups ='base.group\_user，base.group\_system'。

**states**期望字典映射UI属性的值取决于状态字段的值。 例如：states = {'done'：[（'readonly'，True）]}。 可以使用的属性是readonly，required和invisible。

备注

注意，**states**字段属性等同于视图中的attrs属性。 注意，视图支持states属性，但是它有不同的用法：它接受逗号分隔的状态列表，以控制元素何时可见。

了完整性，有时在Odoo主要版本之间升级时使用另外两个属性：

**deprecated = True**在使用字段时记录警告。

**oldname ='field'**用于在较新版本中重命名字段时，使旧字段中的数据自动复制到新字段中。

### 7.3.3 特殊字段名称

一些字段名保留起来是供ORM使用的。

自动编号的id字段是唯一标识每个记录，并用作数据库主键。 它会自动添加到每个模型。

除非设置了\_log\_access = False model属性，否则将在新模型上自动创建以下字段：

**create\_uid**是为创建记录的用户

**create\_date**用于创建记录的日期和时间

**write\_uid**是为最后一个用户修改记录

**write\_date**是修改记录的最后日期和时间

此信息可从Web客户机访问，导航到“开发人员模式”菜单并选择“查看元数据”选项。

一些API内置功能默认情况下需要特定的字段名称。 我们应避免将这些字段名称用于除了预期目的之外的目的。 其中一些甚至保留，不能用于其他目的：

**name**默认用作记录的显示名称。 通常它是一个Char类型，但也可以是一个Text或Many2one字段类型。 我们仍然可以使用\_rec\_name model属性设置要用于显示名称的另一个字段。

**Boolean**类型的active，允许停用记录。 具有active == False的记录将自动从查询中排除。 要访问它们，必须向搜索域添加（'active'，'='，False）条件，或

者'active\_test'：False应该添加到当前上下文

**Integer**类型的Sequence（如果存在于列表视图中）允许手动定义记录的顺序。 要正常工作，您不应该忘记使用模型的\_order属性。

**Selection**类型的状态表示记录生命周期的基本状态，并且可以由状态的字段属性用于动态修改视图：某些表单字段可以在特定记录状态中为只读，必需或不可见。

**parent\_id，parent\_left和parent\_right，Integer**类型的，有父/子分层关系特殊的意义。 我们将在下一节详细讨论它们

到目前为止，我们已经讨论了非关系领域。 但是应用程序数据结构的很大一部分是描述实体之间的关系。 让我们看看现在。

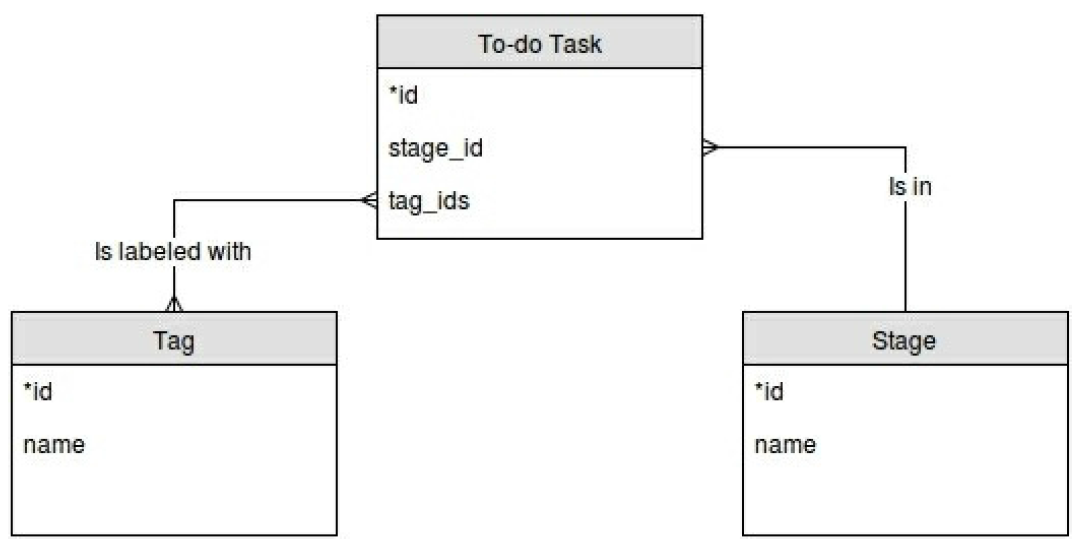
## 7.4 模型之间的关系

重新观察我们设计的模块，我们总结以下关系：

每个任务拥有一个状态。这种many2one的关系，也通常被称为外键。而反过来的关系，one2many，就意味着每个状态可以有多个任务。

每个任务有多个标签。这是many2many的关系。而反过来的关系，当然，也是many2many，一个标签也可被多个任务拥有

下面的实体关系图，可以帮助我们将创建的模块关系可视化。线条以三角形结尾的，代表关系中many的那方。

在我们的to-do任务模块的todo\_model.py文件中添加对应的关系字段：

**class TodoTask(models.Model):**

**\_inherit = 'todo.task'**

**stage\_id = fields.Many2one('todo.task.stage', 'Stage')**

**tag\_ids = fields.Many2many('todo.task.tag', string='Tags')**

代码中，字段的基本语法是，设置相关模块，并且设置该字段展示的文本。命名字段时，有以下约定：以\_id、\_ids为结尾命名的字段，分别对应的是2one和2many的关系。

作为练习，请你尝试为关联模块添加正确的反过来的对应的关系字段：

在Stages中，Many2one反过来的关系字段是One2many，这样，每个状态才能有多个任务。我们需要为Stages类添加这种字段。

在Tags中，Manymany反过来的关系字段仍然是Manymany，这样，每个标识才可以被多个任务使用。

接下来，让我们仔细看看关系字段的定义

### 7.4.1 多对一关系

**Many2one**关系需要两个参数：关联模块（设置对应模块的关键参数）和展示字符。它会在数据库表中创建一个拥有外键的字段去关联数据库表。

这种类型的字段，也可以使用一些其它的参数：

**ondelete** 在关联的记录被删除时，将会触发。默认值是null，也就是说，当关联的记录被删除时，它将被置空。其它值为restrict，当记录被关联时，禁止删除，cascade，删除关联记录的同时删除当前记录。

**context**是一个数据字典，在web客户端视图，通过关联关系访问时，会将上下文传过去。比如，设置默认值。它的详细的介绍，在第六章-视图-设计用户界面中。

**domain**是一个由多个元组组成的列表的域表达式，用来删选关联字段中有效的记录。

**auto\_join=True**使用这个参数后，将允许ORM在数据查询是使用SQL的join(拼接，级联)功能。如果使用它，那么访问权限规则将被忽略，并且权限规则不允许访问的关联记录，用户也可以访问，因此这种情况下的SQL查询将会更快更高效。

### 7.4.2 多对多关系

Many2many 最少要提供一个参数，也就是关联的模块，建议使用string参数，以便更好的标题展示。

在数据库层面，数据库表中是不会添加任何字段。它会自动的创建一个新的中间表，这个表只有两个外键ID字段，并且这两个字段分别关联对应的数据库表。这个中间表和字段的名字都是自动生成的。中间表的名字是两个表的名字用下划线拼接后在加上\_rel组成的。

有时候，我们可能需要改变这种自动的默认值。

其中的一种情况就是，关联的模块有很长的名字，那么自动生成的中间表的名称就会很长，可能会超出PostgreSQL对于表名不得多于63个字符的限制。这种情况下，我们就需要手动选择一个不超出限制的名字。

另一种情况，在相同的模块中需要两个many2many的关系。这种情况，我们也需要手动的为中间表添加名称，只有这样，才能避免数据库名称冲突。

有两种方式去手动的设置这些值：按位置传值或指定参数名传值

使用按位置传值的方式，字段的定义如下：

**# Task <-> Tag relation (positional args):**

**tag\_ids = fields.Many2many(**

**'todo.task.tag', # related model**

**'todo\_task\_tag\_rel', # relation table name**

**'task\_id', # field for "this" record**

**'tag\_id', # field for "other" record**

**string='Tags')**

注意

这些位置参数都是可选的。我们可以只设置中间表的名称，而让它的字段名称自动生成。

我们也可以使用指定参数名称的方式赋值，那样更具有可读性：

**# Task <-> Tag relation (keyword args):**

**tag\_ids = fields.Many2many(**

**comodel\_name='todo.task.tag', # related model**

**relation='todo\_task\_tag\_rel', # relation table name**

**column1='task\_id', # field for "this" record**

**column2='tag\_id', # field for "other" record**

**string='Tags')**

和many2one字段相似，many2many字段也可以定义domain和context等关键属性

注意

目前，由于ORM的局限性，对于抽象模块（Abstract models），当强制命名中间表的名称和列名称后，它们就不能被继承修改了。所以对于抽象模块，请谨慎使用。

Many2many相反的关系也是Many2many。如果我们为Tags模块添加了一个Many2many的字段，ODOO会判定，在Task模块中就会存在一个相反的many2many关系。

在Tags模块中，与Tasks相反的关系可以如下定义：

**class Tag(models.Model):**

**\_name = 'todo.task.tag'**

**# Tag class relationship to Tasks:**

**task\_ids = fields.Many2many(**

**'todo.task', # related model**

**string='Tasks')**

### 7.4.3 一对多反向关系

一种可以添加到另一端的与Many2one相反的关系。这对于数据库结构没有实质的影响，但是允许我们在one端更容易访问many端关联的记录。一个典型的使用场景就是，文档主体与它的明细行的关系。

在我们的示例中，Stages中的One2many关系能让我们更容易的列出拥有的Tasks。在Stages中添加的代码如下：

**class Stage(models.Model):**

**\_name = 'todo.task.stage'**

**# Stage class relationship with Tasks:**

**tasks = fields.One2many(**

**'todo.task', # related model**

**'stage\_id', # field for "this" on related model**

**'Tasks in this stage')**

One2many按顺序接收3个参数：关联的模块，关联模块的字段名称，标题文本。最开始的两个参数通常是模块名和相反关系对应的外键字段名。 可选的附加关键参数和Many2one相同：**context，domain，ondelete**（这个是作用在many端的）和**auto\_join**。

### 7.4.4 分层关系

通过Many2one关系可以在相同的模块中表现父子树关系，每条记录可以引用它的父类。相反的One2many让父类更容易跟踪它的子类。

odoo支持并优化了这种层级数据结构，以便更快的浏览树的兄弟节点，并且可以使用带child\_of操作符的规则表达式使查询更容易。

这种特性生效，需要设置\_parent\_store标签属性并且添加模块的辅助字段：parent\_left和parent\_right。注意，这种操作会增加额外的存储和执行效率的开销，所以最好在你确认你的读操作远远比写操作更频繁时，再使用，比如树一类的场景。

重新改写todo\_model.py文件中的Tags模块为：

**class Tags(models.Model):**

**\_name = 'todo.task.tag'**

**\_description = 'To-do Tag'**

**\_parent\_store = True**

**# \_parent\_name = 'parent\_id'**

**name = fields.Char('Name')**

**parent\_id = fields.Many2one('todo.task.tag', 'Parent Tag', ondelete='restrict')**

**parent\_left = fields.Integer('Parent Left', index=True)**

**parent\_right = fields.Integer('Parent Right', index=True)**

这里，我们又一个基本的模块，使用parent\_id字段引用父记录，并且附加上\_parent\_store属性添加层级查询。这个时候，parent\_left和parent\_right字段就必须添加。

指定父类的字段通常命名为parent\_id，但是其他的任何字段名字我们都可以自定义。

同时, 通常添加一个字段，直接访问子记录:

**child\_ids = fields.One2many('todo.task.tag', 'parent\_id', 'Child Tags')**

### 7.4.5 使用动态关系的参考字段

一个引用固定模块的常规关系字段。引用字段的类型没有限制，并且支持动态关联，所以同样的字段可以关联多个模块。

比如，我们可以使用它给To-do Tasks添加一个引用字段，这个字段可以引用User或Partner:

**# class TodoTask(models.Model):**

**refers\_to = fields.Reference([('res.user', 'User'), ('res.partner','Partner')],'Refers to')**

你可以发现，字段的定义方式和Selection字段定义的方式类似，只是这里的选项列表是可使用的模块。在用户界面，用户需要先从列表中选择模块，然后再选择该模块的一条记录。

这可以灵活的配置: Referenceable Models配置表的Reference字段可以配置哪些模块可以被引用。在Setting|Technical|Database Structure（设置|技术|数据库结构）菜单中配置。当创建一条记录时，我们可以设置任何一个已经安装注册的模块，这要得益于odoo.addons.res.res\_request模块的referenceable\_models()函数。

使用Referenceable Models，最新的版本的引用字段是这样：

**from odoo.addons.base.res.res\_request importreferenceable\_models**

**# class TodoTask(models.Model):**

**refers\_to = fields.Reference(referenceable\_models, 'Refers to')**

注意，在9.0中，这个函数的拼写有些不同，并且使用的是旧API。因此，在9.0版本中，我们在使用之前，要在Python文件头部添加一些封装代码，从而使它可以像新API一样使用：

**from openerp.addons.base.res import res\_request**

**def referenceable\_models(self):**

**return res\_request.referencable\_models(self, self.env.cr, self.env.uid, context=self.env.context)**

## 7.5 计算字段

字段可以使用函数计算值，而不是简单地读取数据库存储的值。计算字段就像常规字段一样声明，但是有额外的compute参数,定义在函数中用来计算值. 在大多数情况下，计算的字段涉及编写一些业务逻辑，因此我们将在第7章第七章 ORM应用逻辑 - 支持业务流程中学习.我们仍将在此解释它们，但将使业务逻辑方面尽可能地简单。让我们来做一个例子:阶段有一个fold字段。我们将在to\_do 任务中增加一个计算字段，对应相关阶段的“folded？”标记。

我们应该编辑TodoTask 模型 在todo\_model.py文件添加以下内容:

**# class TodoTask(models.Model):**

**stage\_fold = fields.Boolean(**

**'Stage Folded?',**

**compute='\_compute\_stage\_fold')**

**@api.depends('stage\_id.fold')**

**def \_compute\_stage\_fold(self):**

**for task in self:**

**task.stage\_fold = task.stage\_id.fold**

前面的代码添加了一个新的stage\_fold字段和\_compute\_stage\_fold方法用于计算它。通过函数名作为字符串,但它也允许它作为一个可调用的引用传递(函数标识符没有引号)。在本例中，我们应该确保在该字段之前，在Python文件中已定义函数。

通常，当计算依赖于其他字段时，需要@api.depends装饰器。它让服务器知道什么时候重新计算存储或缓存的值。一个或多个字段名被作为参数接受，点符号可以用来跟踪字段关系。

计算函数将指定一个值到字段或多个字段用来计算。如果它不能，就是错误的。既然self是一个记录对象，我们的计算仅仅是 通过stage\_id.fold获取“Folded?” 字段。通过将该值(写入)分配给计算字段stage\_fold，来实现这一结果。

我们现在还不会对这个模块的视图进行操作，但是您可以立即对任务表单进行快速编辑，以确认计算出的字段是否如预期的那样:使用开发者模式 ，选择编辑视图，并直接在XML表单中添加字段。不用担心:下一次升级时，它将被干净的模块视图所取代。

### 7.5.1 在计算字段上搜索和写入

我们刚刚创建的计算字段可以被读取，但不能被搜索和写入。为了启用这些操作，我们首先需要为它们实现专门的函数。除了compute函数，我们还可以设置search函数，实现搜索逻辑，以及inverse函数，实现写逻辑。

使用这些，我们的计算字段声明如下:

**# class TodoTask(models.Model):**

**stage\_fold = fields.Boolean(**

**string='Stage Folded?',**

**compute='\_compute\_stage\_fold',**

**# store=False, # the default**

**search='\_search\_stage\_fold',**

**inverse='\_write\_stage\_fold')**

**支持的函数是:**

**def \_search\_stage\_fold(self, operator, value):**

**return [('stage\_id.fold', operator, value)]**

**def \_write\_stage\_fold(self):**

**self.stage\_id.fold = self.stage\_fold**

每当在搜索域表达式中找到 a (field, operator, value)就会调用搜索函数。它接收到搜索的操作符和值，并期望将原来的搜索元素转换为另一个域搜索表达式。 inverse函数执行计算的反向逻辑，查找在计算源字段中写入的值。在我们的示例中,这意味着回到stage\_id.fold字段。

### 7.5.2 存储计算字段

通过设置“store= true”的定义，计算字段的值也可以存储在数据库中。他们的任何依赖关系的变化，他们将重新计算。由于当前值被存储，它们可以像常规字段一样搜索，并且不需要搜索函数。

### 7.5.3 关联字段

我们在前一节中实现的计算字段只是将一个相关记录的值复制到一个模型的字段中。然而这是一种明智的做法可以通过Odoo自动处理。

使用相关字段也可以实现同样的效果。它们可以直接在模型上使用，这些字段属于一个相关的模型，可以使用点符号链访问。在点符号不能使用的情况下，也可以使用，例如UI表单视图。 要创建一个相关的字段，我们声明一个字段所需的类型，就像与正规的计算字段，但不是计算，我们使用含有点符号字段链的属性达到所需字段。

to\_do任务是组织在可定制的阶段,这些地图变成基本状态。我们将使状态值在任务模型上可用，以便在下一章中用于一些客户端逻辑。

类似于stage\_fold,我们将在任务模型添加一个计算字段,但这一次使用更简单的相关字段:

**# class TodoTask(models.Model):**

**stage\_state = fields.Selection(**

**related='stage\_id.state',**

**string='Stage State')**

在后台，相关的字段只是计算字段，方便地实现search和inverse方法。这意味着我们可以在不需要编写任何额外代码的情况下搜索和编写这些代码。

## 7.6 模型约束

为了加强数据完整性，模型还支持两种类型的约束：SQL和Python。SQL约束添加到L数据库表定义和直接PostgreSQL执行。他们是使用sql\_constraints定义的类属性。这是一个列表的元组:约束标识符名称;约束的SQL,供使用的错误消息。一个常见的用例是为模型添加唯一的约束。假设我们不想让活动任务具有相同标题:

**# class TodoTask(models.Model):**

**\_sql\_constraints = [**

**('todo\_task\_name\_uniq',**

**'UNIQUE (name, active)',**

**'Task title must be unique!')]**

Python约束可以使用一段任意代码来检查条件。检查函数应该使用@api.constraints 修饰，说明被检查的字段的列表。当其中任何一个被修改，并且在条件失败时将会抛出一个异常，验证就会被触发。

例如，为了验证任务名称至少有五个字符长，我们可以添加以下约束：

**from odoo.exceptions import ValidationError**

**# class TodoTask(models.Model):**

**@api.constrains('name')**

**def \_check\_name\_size(self):**

**for todo in self:**

**if len(todo.name) < 5:**

**raise ValidationError('Must have 5**

**chars!')**

## 7.7 第五章 小结

我们对模型和字段进行了详细的解释，使用它们在任务中通过标记和阶段来扩展to\_do事项。您学会了如何定义模型之间的关系，包括分层的父/子关系。最后，我们了解了使用Python代码计算字段和约束的简单示例。

在下一章中，我们将研究这些后端模型特性的用户界面，使它们在与应用程序交互的视图中可用。

# 8 第六章 视图 - 设计用户界面

本章将帮助您了解如何构建图形界面以供用户与To-Do应用交互。 您将发现不同类型的视图和可用的小部件，了解什么是上下文和域，并了解如何使用它们提供良好的用户体验。 我们将继续使用todo\_ui模块。 它已经具备了模型层，现在需要用户接口的View层。

## 8.1 使用XML文件定义用户界面

用户界面的每个组件都以记录的形式存储在数据库中，就像业务记录一样。 模块从XML文件加载相应的数据时会向数据库添加UI元素。

这意味着定义UI的XML数据文件需要添加到todo\_ui模块。 我们可以通过编辑\_\_manifest\_\_.py文件来声明这些新的数据文件：

**{**

**'name': 'User interface improvements to the To-Do app',**

**'description': 'User friendly features.',**

**'author': 'Daniel Reis',**

**'depends': ['todo\_user'],**

**'data': [**

**'security/ir.model.access.csv',**

**'views/todo\_view.xml',**

**'views/todo\_menu.xml',**

**]}**

小贴士

请记住，数据文件是按您指定的顺序加载的。 这很重要，因为您只能引用已经定义的XML ID。

我们还可以创建子目录和具有最小结构的views/todo\_view.xml和views/todo\_menu.xml文件：

**<?xml version="1.0"?>**

**<odoo>**

**</odoo>**

在第3章“继承 - 扩展现有应用程序”中，向我们的应用程序添加了一个基本菜单，但我们现在想改进它。 因此，我们将添加新的菜单项和菜单被选择时被触发的相应的窗口动作。

### 8.1.1 菜单项

菜单项存储在ir.ui.menu模型中，可以通过设置下的技术| 用户界面| 菜单项菜单进行浏览。

todo\_app 插件创建了一个顶级菜单，用于打开 To-Do 应用的任务。 现在我们要将其修改为二级菜单，并在其旁边添加其他菜单选项。

为此，我们将为应用添加一个新的顶级菜单，并修改现有的 To-Do 任务菜单选项。 在 views/ todo\_menu.xml文件中，添加：

**<!-- Menu items -->**

**<!-- Modify top menu item -->**

**<menuitem id="todo\_app.menu\_todo\_task" name="To-Do" />**

**<!-- App menu items -->**

**<menuitem id="menu\_todo\_task\_view" name="Tasks" parent="todo\_app.menu\_todo\_task" sequence="10" action="todo\_app.action\_todo\_task" />**

**<menuitem id="menu\_todo\_config" name="Configuration" parent="todo\_app.menu\_todo\_task" sequence="100" groups="base.group\_system" />**

**<menuitem id="menu\_todo\_task\_stage" name="Stages" parent="menu\_todo\_config" sequence="10" action="action\_todo\_stage" />**

<menuitem>快捷元素，提供了一种缩减方式来定义要加载的记录，我们可以使用它来代替<record model =“ir.ui.menu”>元素。

第一个菜单项是To-do应用程序的顶部菜单项，只有name属性，并将用作下两个菜单项的父菜单。注意，它使用现有的XML ID：todo\_app.menu\_todo\_task，因此这会重写todo\_app模块中定义的菜单项，使其不绑定任何动作。 这是因为我们将添加子菜单项，并且将从其中一个调用打开任务视图的动作。

然后子菜单通过parent =“todo\_app.menu\_todo\_task”属性置于顶层菜单下。

第二个菜单就是那个打开任务视图的菜单，通过action =“todo\_app.action\_todo\_task”属性实现。 从它所使用的XML ID中可以看出，它正在重用由todo\_app模块已创建的动作。

第三个菜单项为我们的应用程序添加配置部分。 我们希望它只对超级用户可用，因此我们还使用groups属性使其仅对系统管理|设置安全组可见。

最后，在配置菜单下，我们添加了任务阶段菜单项。 我们将使用它来维护将要添加的待办事项看板功能所使用的阶段。

这个时候，如果我们尝试升级插件，我们应该得到错误，因为我们没有定义在action属性中使用的XML ID。 我们将在下一节中添加它们。

### 8.1.2 窗口动作

窗口动作给出GUI客户端指令，通常由视图中的菜单项或按钮使用。 它告诉GUI应该使用什么模型，以及提供哪些视图。 这些操作可以使用domain过滤器强制只有一部分记录可见。 他们还可以通过context属性设置默认值和过滤器。

我们将向 views/todo\_menu.xml数据文件添加窗口动作，这些动作将由上一节中创建的菜单项使用。 编辑文件，并确保动作定义在菜单项之前：

**<!-- Actions for the menu items -->**

**<act\_window id="action\_todo\_stage" name="To-Do Task Stages" res\_model="todo.task.stage" view\_mode="tree,form" target="current"**

**context="{'default\_state': 'open'}" domain="[]" limit="80" />**

**<act\_window id="todo\_app.action\_todo\_task" name="To-Do Tasks" res\_model="todo.task" view\_mode="tree,form,calendar,graph,pivot"**

**target="current" context="{'search\_default\_filter\_my\_tasks': True}" />**

**<!-- Add option to the "More" button -->**

**<act\_window id="action\_todo\_task\_stage" name="To-Do Task Stages" res\_model="todo.task.stage" src\_model="todo.task" multi="False" />**

窗口动作存储在ir.actions.act\_window模型中，可以使用前面代码中使用的<act\_window>快捷方式在XML文件中定义。

**name**是将在通过此操作打开的视图中显示的标题。

**res\_model**是目标模型的标识符

**view\_mode**是可用的视图类型和它们的顺序。 第一个是默认打开的。

**target，**如果设置为new，将在弹出的对话框窗口中打开视图。 默认值为current，在主内容区域中打开视图。

**context**设置关于目标视图的上下文信息，上下文中可以设置默认值或激活过滤器以及其他信息。 我们将在稍后更详细地看到它。

**domain**是一个域表达式，强制过滤在打开的视图中可浏览的记录。

**limit**是列表视图中每个页面的记录数。

XML中定义的第二个动作替换todo\_app插件的原始To-do任务动作，以便显示我们将在本章后面要讨论的其他视图类型：日历和图形。 这些更改安装后，您会在右上角列表和表单按钮的后面到其他按钮; 然而，这些按钮将不工作，直到我们创建相应的视图。

我们还添加了第三个动作，不在任何菜单项中使用。 它向我们展示了如何向列表和表单视图右上角的更多菜单添加选项。 它使用两个特定的属性：

**src\_model**指示应该在什么模型上此动作可用。

**multi，**当设置为True时，使它在列表视图中可用，以便它可以应用于多条选中的记录。 默认值为False，如在我们的示例中，它将使该选项仅在表单视图中可用，因此一次只能应用于一个记录。

## 8.2 上下文和域

我们已经断断续续的使用过几次上下文和域。知道窗口动作可以设置它们并且模块中的关联字段也可以通过访问属性的方式访问他们。

### 8.2.1 上下文数据

上下文是一个包含session数据的字典，它可以被客户端的用户接口和服务器端的ORM与业务逻辑使用。

在客户端，它可以将信息从一个视图传输到另一个视图。比如当前视图的有效的记录ID，通过一个连接或按钮跳转到另一个视图后，它可以作为默认值被使用。

在服务器端，一些记录的字段值可以通过上下文被本地设置提供。特别的lang值是影响翻译的字段。上下文可以在服务器端提供信号代码。比如，active\_test设置为False时，ORM的search()方法的将不会筛选出不活动的记录。

来自网络客户端的上下文，初始化的是以下这样的：

**{'lang': 'en\_US', 'tz': 'Europe/Brussels', 'uid': 1}**

lang是用户使用的语言，tz是时区信息，uid是当前用户ID

从当前form视图通过连接或则按钮打开一个form视图时，active\_id属性会被添加到上下文中，并且包含当前的记录ID。在list视图中，上下文中还包含一个active\_ids属性，它包含当前list视图中，已选择的记录的ID的列表。

在客户端，上下文可以通过属性default\_或default\_search\_\*来设置目标视图的默认值或默认筛选条件。例如：

**{'default\_user\_id': uid}**

在目标视图中，我们想filter\_my\_tasks筛选器默认有效，我们可以这样做：

**{'default\_search\_filter\_my\_tasks': 1}**

### 8.2.2 域表达式

域被用来筛选数据记录。它们使用特殊的语法，以便于Odoo ORM 将它们解析后生成对应的SQL WHERE数据库筛选语句。

域表达式是一个条件列表，每个条件是一个形如('field\_name', 'operator', value')的元组。比如，以下这个就是只有一个条件的有效的域表达式：

**[('is\_done','=',False)].**

以下就是每个元素的介绍：

**filed\_name** 是需要筛选的字段，它可以使用点（.）来访问关系模块的字段。

**value** 是一个Python表达式的值。它可以使用字符值，比如：字符串，数字，布尔值，或则列表、某个字段、用户在context中自定义的有效的值。这里有两种在实际环境中，可能的出现在context中的可用于domains的值：

当在客户端使用时，比如在窗口动作或字段的属性值中，传入当前视图中的有效的字段的值。但是不能使用点（.）拼接它们。

当在服务器端使用时，比如在记录规则和服务器Python代码中，当前记录是一个对象，就可以使用点（.），访问它的字段。

operator 可以为：

常用的操作符：<,>,<=,>=,=,!=。

'=like'通配符，使用下划线(\_)时，匹配一个任意字符，使用百分号(%)时，匹配多个字符。

**'like'**匹配一个’%value%’的字符串。’ilike’与此类似但不区分大小写。‘not like’和**‘not ilike**’也可以使用

**'child of'**在层级关系中，筛选子集

**'in'和’not in’**筛选是否在一个列表里面，所以，给的值应该是个list。当在’to-many’的关系字段中，‘in’的作用和contains的作用一样

domain表达式是一个包含多项内容的list，因此，可以包含多个条件表达式元组。 默认情况下（隐含），这些条件之间是用AND逻辑符连接的，也就是说，它只返回满足所有条件的记录集。

显示的逻辑连接符也可以使用：and连接符（&，默认使用），或连接符（|）。这些连接符后面跟着两项内容，以递归的方式。我们一会儿会看见具体的实现。

感叹号（!），是非操作符（NOT），作用于紧挨它的后一个项。因此，它应该出现在否定项的前面。比如，['!', ('is\_done','=',True)]，这个表达式将筛选所有的没有完成的记录。

“下一项”也可以作为操作符作用于它的下一项，也就是嵌套定义。举一个例子，可能帮我们更好的理解这句话。 我们可以在服务器端的记录规则中找到如下类似的表达式：

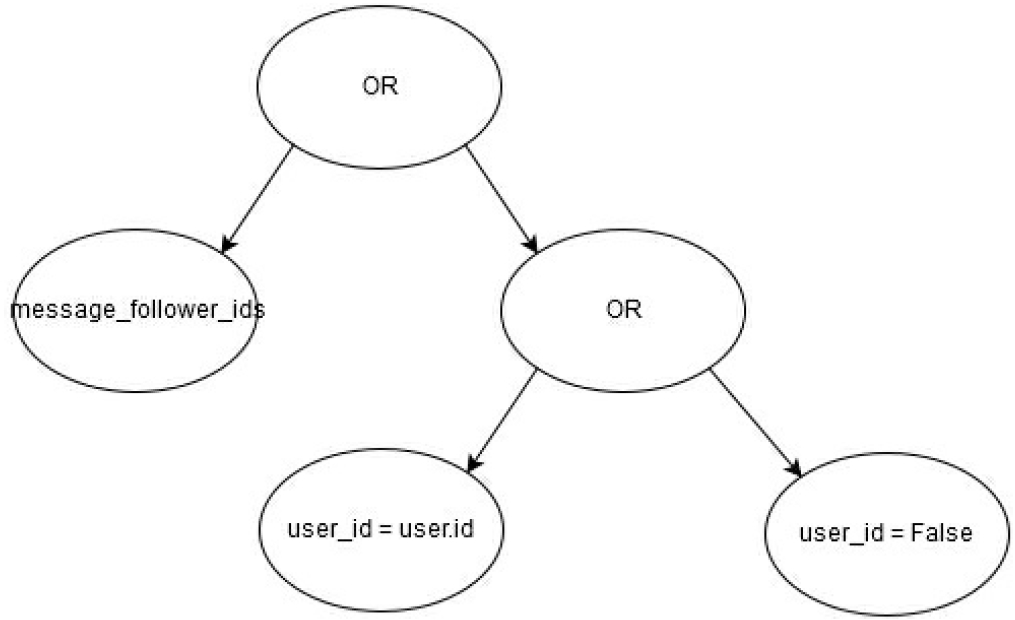
**['|', ('message\_follower\_ids', 'in', [user.partner\_id.id]),**

**'|', ('user\_id', '=', user.id), ('user\_id', '=', False)]**

这个域筛选了，当前用户在记录的关注者列表中，或当前用户是负责人，或负责人字段为空值的所有记录。

第一个‘|’（OR）操作符筛选出的结果是，满足在关注者列表中的条件的所有记录，加上满足后面条件的所有记录。而后面一个条件又是另外两个条件联合查询：用户ID是当前用户或则为空值的记录。

下面这幅图能很好解释这种操作符的原理:



## 8.3 窗体视图

正如前面的章节中介绍, 表单视图可以采用简单的布局或类似于纸张文档的业务票据布局. 我们现在将了解如何设计这些业务文档视图和如何使用相关元素和可用的小部件。 通常我们会通过继承和扩展to\_do 视图来实现, 但为了更加清晰, 我们将创建全新的视图来重写原视图。

### 8.3.1 处理同一类型的几个视图

相同模型可以有多个同类型的视图，因为一个窗口动作通过它的“XML ID ”可以指向的特定的视图。这样我们就可以灵活的根据菜单项，通过不同的视图打开相同的模型。 要实现以上功能，需要根据视图中的 XML ID，为窗口操作添加view\_id属性, 例如我们在todo\_app.action\_todo\_task 动作中曾使用类似view\_id="view\_form\_todo\_task\_ui"的做法。 但是, 如果没有特定视图的定义，将如何？ 在这种情况下, 使用的将是第一个返回的查询视图,这将是一个具有较低的优先级视图。如果我们添加新的视图，并设置它的优先级低于现有的, 它将被调用。最终的结果是：看起来像这个新视图覆盖了原来的。 因为缺省值的视图优先级是 16, 较低优先级的视图将被采纳, 所以 15 的优先级将工作。为了保持我们的例子可读性，这不是最常用的方法。 我们将在下一个例子中使用优先级方法。

### 8.3.2 业务文档视图

业务应用程序通常是关于仓库产品、财会部门发票等记录的系统。大部分这些记录信息都可用纸质文档呈现。为了更好的用户体验，表单视图都是在模仿这些纸质文档。 例如在我们的系统中，我们可以认为 “to\_do task”包含需要填写的纸制表单。 我们将提供一个遵循该设计思想的表单视图。

要增加一个具有业务文档视图基本骨架的XML视图，我们需要编辑“views/todo\_views.xml”文件，并置于文件顶部。

**<record id="view\_form\_todo\_task\_ui" model="ir.ui.view">**

**<field name="model">todo.task</field> <field name="priority">15</field>**

**<field name="arch" type="xml"> <form> <header> </header> <sheet> </sheet>**

**<field name="message\_follower\_ids" widget="mail\_followers" />**

**<field name="message\_ids" widget="mail\_thread" />**

**</form> </field> </record>**

视图名称是可选的，如果缺少将自动生成。为了简单，我们省略了视图记录中的<field name="name">元素。

#### 8.3.2.1 header头部

顶部的标题将展示文档流转的阶段和生命周期,以及动作按钮. 这些操作按钮一般为表单按钮,最重要的阶段可以使用class=“oe\_highlight”突出显示.文档生命周期通过状态栏小控件呈现当前所处的状态.这些通常是状态选择字段或者多对一的阶段字段.这些字段在很多odoo核心模块中可以找到.这些多对一的阶段字段 通过建立模型实现过程步骤.它们可以根据用户需求动态编译,并且完美支持看板. 状态是一个选择列表，呈现的是过程中有几个相当稳定的步骤阶段,例如:新建、处理中、完成。它不是由用户最终配置的，但因为它是静态的，所以在事务逻辑中更容易使用。它也支持字段展示：状态属性可以根据文档当前的属性决定一个字段是否向用户呈现。 阶段出现的比状态晚，两者并存，但是阶段取代状态将是odoo内核发展的趋势。但正如所看到的前面的解释，状态仍然提供一些阶段不具备的功能。通过状态和阶段的互相映射，可以做到两全其美。在前面的章节中，在任务阶段模型中添加一个状态字段，通过计算控制字段，使得状态字段属性可以使用。 在views/todo\_view.xml 文件中我们可以添加状态栏来扩展基本的标题。

**<header>**

**<field name="state" invisible="True" />**

**<button name="do\_toggle\_done" type="object" attrs="{'invisible':[('state','in',['draft'])]}" string="Toggle Done" class="oe\_highlight" />**

**<field name="stage\_id" widget="statusbar" clickable="True" options="{'fold\_field': 'fold'}" />**

**</header>**

这里我们增加了一个隐藏的状态字段。我们需要客户端发送给服务器的数据请求中包含该字段，否则就不可用于表达式。

提示:

非常重要的是：任何你想使用的字段，在一个域或属性表达式，必须加载到视图中，故可以将随时会使用但不需要用户看到的字段设置为不可见。下一步将一个按钮添加到状态栏，让用户切换任务完成标记。状态栏中按钮的显示，应该依据文档所处步骤阶段而变化。在草稿状态，我们通过“attrs”属性隐藏按钮。要实现这个功能，需要使用状态字段，它无须呈现在窗口，所以我们必须添加这一隐藏字段。如果我们有一个状态选择字段，我们可以使用状态属性。在这种情况下我们可以通过state=“open，done”做到同样的效果。虽然它不如‘attrs’属性灵活，它更多的是简洁.这些可见性也可以用于其他视图元素，例如字段.我们将在本章后面更详细地探讨它们。点击属性允许用户点击状态栏来改变文件的阶段。我们通常会启用这个，但也有我们不需要情况。例如当我们需要更多的控制工作流程或者用户通过可见的动作按钮来实现阶段的行进，这样方便实现阶段转换前的确认。

当使用阶段状态栏小部件时，我们可以使用在阶段族群中隐藏的很少使用的阶段。为此，阶段模型必须有一个标志来配置隐藏属性，通常命名为fold。StatusBar控件应该使用一个options属性，如前面代码所示，提供字段名的fold\_field选项。当使用含有状态字段的状态栏小部件时,通过statusbar\_visible 可以实现类似的效果。它可以呈现特殊情况下各种可见的和隐藏的异常状态。 例如：

**<field name="stage\_id" widget="statusbar" clickable="True" statusbar\_visible="draft,open" />**

#### 8.3.2.2 sheet工作表

表画布是表单实际的数据元素放置的主要区域,它看起来像一个实际的纸质文件,odoo中的记录也被作为文档。

通常,一个文档表结构将这些部分：

文档标题和副标题。 在右上角的一个按钮盒， 其他文档标题字段。 在页面和标签中提供有组织字段的笔记簿。

让我们一起来了解下这些区域.

#### 8.3.2.3 标题和副标题

<group>元素之外的字段 是不会自动呈现标签的。这同样适用于标题元素,所以<label for"..."/>元素经常被用来呈现标题。 虽然会带来一些额外的工作,但是这样做的优势是：对标签显示具有更多的控制。 常规的HTML,包括css-stylel元素,也可以用来制作闪光标题。为了实现最佳效果，标题应该通过oe\_title嵌入在 中。 下边是一个扩展的sheet元素，它的标题添加了一些额外字段作为副标题。

**<sheet><div class="oe\_title"> <label for="name" class="oe\_edit\_only"/> <h1><field name="name"/></h1><h3>**

**<span class="oe\_read\_only>By</span> <label for="user\_id" class="oe\_edit\_only"/> <field name="user\_id" class="oe\_inline" /> </h3>**

**</div> <!-- More elements will be added from here... --> </sheet>**

在这里，我们可以看到，我们使用常规的HTML元素，如div，span，H1，和H2。<label>元素允许我们控制何时何地展示。‘for’属性告诉我们获取标签文本的字段。另一种可能性是使用字符串属性来提供特定文本。我们的实例使用了class="oe\_edit\_only"，这样只有在编辑模式可见。 在某些情况下，如合作伙伴或产品，需要有代表性的图像显示在左上角。假设我们有一个my\_image二进制字段，我们可以在＜div class=“oe\_title”>行前添加： <field name="my\_image" widget="image" class="oe\_avatar"/> 。

#### 8.3.2.4 智能按钮区域

在右上角 有一个隐藏的按钮箱.8.0版本引入了智能按钮,当被点击时,可以呈现为矩形的统计指示. 我们可以 在"oe\_title" DIV 后添加按钮框,如下:

按钮的容器是一个div。div具有oe\_button\_box样式以及使其靠表右侧对齐的oe\_right样式。我们将在后边章节进一步介绍按钮，学习添加按钮到按钮箱中。

#### 8.3.2.5 对表单中的内容进行分组

表单的主要内容应该使用<group>标签组织，group标记在画布中插入两列，其中的字段标签默认会被展示。字段值、字段标签需要两列,所以在一个组添加字段将垂直叠放。如果我们将两个<group>元素置于一个顶部group中，我们能够得到两列并排的字段标签。 继续我们的表单视图，我们将在智能按钮箱后边添加主要内容。

**<group name="group\_top">**

**<group name="group\_left">**

**<field name="date\_deadline" />**

**<separator string="Reference" />**

**<field name="refers\_to" />**

**</group>**

**<group name="group\_right">**

**<field name="tag\_ids" widget="many2many\_tags"/>**

**</group>**

**</group>**

这是一个很好的做法，分配一个名称给组标签，以便它更容易后期被引用并扩展视图（由您或其他开发人员）。字符串属性也是允许的，如果设置，则用于显示节标题。在一个组内，<newline>将强制生成一个新行。下一个元素将在组的第一列中呈现。附加部分标题可以使用<separator>元素添加到组内。 提示： 从开发者菜单选项切换表单布局，它绘制了每个组的边框线，以便更好地了解当前表单布局。 使用col和colspan属性，我们可以更好的控制组元素布局。col属性可以指定组中包含的列数，默认值是2，它也可以改为任何数字。双数的效果更好，因为每一个字段加起来都要两列，用于标签加字段值。组内元素,包括<filed>元素,可以使用colspan属性设置一个特定的列数，默认都是一列。

#### 8.3.2.6 标签笔记本

组织内容的另一种方法是使用笔记本元素,包含多个标签的部分,称为页面。这样可以实现较少使用的数据给予隐藏，或者按照标题组织大量数据。我们不需要把这个添加到我们的to\_do表单中，但这里有一个例子 可以添加到任务台窗体。

**<notebook>**

**<page string="Whiteboard" name="whiteboard">**

**<field name="docs" />**

**</page>**

**<page>**

**</page>**

**</notebook>**

## 8.4 查看语义组件

我们已经看到了如何以一种形式来组织内容，使用诸如标题、组和笔记本这样的结构组件。现在，我们可以更深入地了解语义组件、字段和按钮，以及我们可以对它们做些什么。

### 8.4.1 字段

视图字段有一些属性可供他们使用。它们中的大多数都有从模型中定义的值，但这些可以在视图中重写。

属性是通用的，不依赖于字段类型，如下：

**name：**标识字段数据库名称

**string：**标签文本，如果我们想要覆盖模型定义提供的标签文本，可以使用它。

**help：**提示文本，当您将指针悬停在字段上时，将显示一个提示文本，并允许覆盖模型定义提供的帮助文本。

**placeholder：**是一个建议文本，显示在该字段内。

**widget：**允许我们覆盖用于该字段的缺省小部件。我们将在稍后探讨可用的小部件。

**options：**是一个JSON数据结构，为小部件提供了额外的选项，并且取决于每个小部件支持什么。

**class：**是用于字段HTML呈现的CSS类。

**nolabel="True"：**防止自动字段标签被呈现。仅对<group>元素中的字段有意义，并且经常和<label for="..."> 元素一起使用.

**invisible="True":**使字段不可见，但是数据是从服务器获取的，并且在表单上可用。

**readonly="True":** 使表单上的字段不可编辑。

**required="True"** 在表单上字段为必需字段。

某些字段类型的特殊属性：

**password="True"：**用于文本字段。它显示为一个密码字段，屏蔽输入的字符。

**filename：** 用于二进制字段，它是用于存储上传文件名称的模型字段名称。

**mode：**用于一对多字段。它指定用于显示记录的视图类型。默认情况下，它是树，但也可以是表单、看板或图形。

#### 8.4.1.1 字段的标签

<label>元素可以用来更好地控制字段标签的呈现。一种情况是仅在窗体处于编辑模式时才呈现该标签：

**<label for="name" class="oe\_edit\_only" />**

当这样做时,如果该字段在一个<group>元素,我们通常也要设置nolabel = " True "。

#### 8.4.1.2 关系字段

在关系字段中，我们可以对用户可以做的一些额外的控制。默认情况下,用户可以创建新的记录(也称为“快速创建”)和打开相关的记录表单。可以使用选项字段属性禁用此功能:

**options={'no\_open': True, 'no\_create': True}**

上下文和域在关系字段中也特别有用。上下文可以定义相关记录的默认值，域可以限制可选择的记录。一个常见的例子是，顶部有一个字段中可选择的记录列表，依赖当前记录的其他字段值.域可以在模型中定义，但也可以在视图中被重写。

#### 8.4.1.3 字段小部件

每个字段类型都显示在表单中，并使用适当的默认小部件。但是还可以使用其他可选的小部件。

对于文本字段，我们有以下小部件:

**email:**是用来将电子邮件文本转换为一个可操作的“邮件”地址。

**url :**用于将文本格式化为可单击的url。

**html:**用于渲染文本作为html内容;在编辑模式下,以WYSIWYG编辑器,允许内容的格式,而不需要使用HTML语法。

对于数值字段，我们有以下小部件:

**handle:**是专门为列表视图中的序列字段设计的，并显示一个句柄，允许您将行拖到一个定制的顺序。

**float\_time:**拥有小时和分钟的时间浮点字段格式.

**monetary:**作为货币金额显示浮动字段。它需要一个currency\_id伙伴字段，另一个字段名称可以设置选项=“{ 'currency\_field '：' currency\_id“}”。

**progressbar** 用一个浮点数表示为一个进度百分比，并且可以用于表示完成率字段.

对于关系和选择字段，我们有这些附加的小部件:

**many2many\_tags :** 显示按钮标签列表值.

**selection:**用于多对一字段的选择字段部件。

**radio** 使用单选按钮显示选择字段选项.

**kanban\_state\_selection**展示了看板状态选择列表的信号灯。一般状态为灰色,已经完成为绿色,其他状态为红色.

**priority** 一个可点击的星星列表的选择字段。可选项一般为数字.

### 8.4.2 按钮

按钮支持如下属性

**icon** 是图标图像使用按钮显示,不像智能按键,只有在

"addons/web/static/src/img/icons"的图标才可以为普通按钮所用.

**string**是按钮文本标签，或者是使用图标时的HTML alt文本。

**type**是动作执行类型。可能值是：

**workflow** 用于触发工作流引擎信号；

**object** 用于调用Python方法;

**action** 用于运行窗口动作。

**name** 根据所选类型标识要执行的特定操作：工作流信号名、模型方法名称或运行窗口操作的数据库ID。%(xmlid)d 公式将XML ID转换成所需的数据库ID。

**args** 当类型是对象时，将使用其他参数传递给该方法。

**context** 将值添加到上下文，可以在Windows操作运行后，或在Python代码方法中调用。

**confirm** 显示一个确认消息框，并将文本分配给此属性。

**special="cancel"**用于向导，用来取消和关闭向导表单。

### 8.4.3 智能按钮

在设计表单结构时，我们设置了一个包含智能按钮的顶级区域。现在让我们在里面添加一个按钮。

对于我们的应用程序，我们将会有一个按钮，显示当前任务的所有者的总数量，点击它将导航到这些项目的列表。

首先，我们需要将相应的计算字段添加到models/todo\_model.py.为TodoTask类添加如下代码：

**def compute\_user\_todo\_count(self):**

**for task in self:**

**task.user\_todo\_count = task.search\_count(**

**[('user\_id', '=', task.user\_id.id)])**

**user\_todo\_count = fields.Integer(**

**'User To-Do Count',**

**compute='compute\_user\_todo\_count')**

接下来，我们添加按钮框和其中的按钮。在oe\_title div之后，取代我们之前加入的按钮框占位符。如下：

**<button class="oe\_stat\_button"**

**type="action" icon="fa-tasks"**

**name="%(action\_todo\_task\_button)d"**

**context="{'default\_user\_id': user\_id}"**

**help="All to-dos for this user" >**

**<field string="To-Dos" name="user\_todo\_count"**

**widget="statinfo"/>**

**</button>**

计算user\_todo\_count 字段，按钮向负责todo任务人员显示todo任务数量。

这里有一些属性，可以添加到智能按钮：

**class="oe\_stat\_button"** 呈现一个矩形而不是一个普通的按钮。

**icon ：**设置要使用的图标,可以从图标字体中选择，可用的图标可浏览http://fontawesome.io。

**type和name**是按钮类型和要触发的动作的名称。对于智能按钮，类型通常是action，对于窗口操作，name将是要执行的操作的ID。这里需要一个实际操所的数据库ID，因此我们必须用一个公式将XML ID转换成数据库ID:“%(action-external-id)d”。该操作应该使用相关记录打开一个视图。

**string** 在按钮上添加标签文本。我们没有在这里使用它，因为包含的字段已经为它提供了一个文本。

**context** 用于在目标视图上设置默认值，点击按钮后，可以在视图上创建新记录。

**help** 添加一个帮助提示，当鼠标指针在按钮上时显示该提示。

按钮元素本身是一个容器，带有显示统计信息的字段。这些是使用小部件状态信息的常规字段。它是在底层模型中定义的计算字段。除了字段之外，在按钮中我们还可以使用静态文本，例如：

**User's To-dos**

单击按钮时，我们希望看到一个只包含当前负责用户的任务的列表。将由action\_todo\_task\_button动作完成，尚未实施。但它需要知道当前的负责用户，才能执行过滤器。

为此，我们使用按钮的上下文属性来存储该值。

使用的动作必须在窗体之前定义，所以我们应该在xml文件的顶部添加它：

**<act\_window id="action\_todo\_task\_button"**

**name="To-Do Tasks"**

**res\_model="todo.task"**

**view\_mode="tree,form,calendar,graph,pivot"**

**domain="[('user\_id','=',default\_user\_id)]" />**

请注意我们是如何使用default\_user\_id上下文是域过滤器的关键。依循按钮链接建立一个新的任务时，这个特定的关键也在user\_id字段设置默认值。

## 8.5 动态视图

视图元素也支持一些动态属性,允许动态视图根据字段值来改变他们的外观或行为。

我们可能更改事件,可以改变其他字段值形式,在编辑数据字段是强制性的或可见的只有当某些条件得到满足。

改变事件

变化机制允许我们更改在其他表单字段值改变一个特定的领域。例如,价格变化对产品领域上可以设置字段默认值只要产品的改变。

在旧版本上更改事件被定义在视图层,但自从8.0版本他们直接在模型层上定义,而不需要任何特定的标记的视图。这是通过创建方法来执行计算,并使用@api.onchange('field1', 'field2') 将其绑定到字段。这些onchange方法将更详细地讨论在第7章,ORM应用程序逻辑支持业务流程。

## 8.5.动态属性

变化机制还负责计算字段的自动重新计算,立即对用户的输入作出反应。使用相同的例子,当我们改变了产品价格字段改变,计算总金额字段也会自动更新使用新的价格信息。动态属性的属性提供了一种简单的方法来控制一个特定的用户界面元素的可见性:

根据安全组可以使一个元素可见当前用户所属的组,只有指定的组的成员会看到它。预计一组XML IDS的逗号分隔列表。

声明可以使一个元素可见根据记录的状态字段。预计一个逗号分隔的值列表。

除了这些,我们也有一个灵活的方法可用来设置一个元素的可见性取决于客户端动态评估表达式。这是特殊属性,期待一个值的字典将无形的属性的值映射到一个表达式的结果。

例如,在所有国家,除了refers\_to字段可见草案,使用以下代码:

**<field name="refers\_to" attrs="{'invisible': state=='draft'}" />**

不可见的属性可以在任何元素,不仅仅是字段。例如,我们可以使用它在笔记本页面和组元素。

attrs设置值,也可以为其他两个属性:只读的和必需的。这些数据字段意义,让他们没有可编辑或强制性的。这让我们可以实现一些基本的客户端逻辑,如强制执行一个字段根据其他记录值,如状态执行一个字段根据其他记录值,比如状态。

## 8.6 列表视图

在这一点上，列表视图应该很少介绍，但是我们仍然要讨论可以与它们一起使用的属性。下面是我们的to\_do 任务列表视图的一个例子:

**<record id="todo\_app.view\_tree\_todo\_task"**

**model="ir.ui.view">**

**<field name="model">todo.task</field>**

**<field name="arch" type="xml">**

**<tree decoration-muted="is\_done"**

**decoration-bf="state=='open'"**

**delete="false">**

**<field name="name"/>**

**<field name="user\_id"/>**

**<field name="is\_done"/>**

**<field name="state" invisible="1"/>**

**</tree>**

**</field>**

**</record>**

行文本颜色和字体可以根据Python表达式求值的结果动态变化。

这是通过decoration-NAME属性,和计算字段属性的表达式来实现的。name部分可以是BF或it，粗体和斜体字体，或任何引导文本上下文颜色：危险，信息，静音，初级，成功，或警告。 类似引导文档例子可以察看：

http://getbootstrap.com/css/#helper-classes-colors.

提示

颜色和字体属性,可以在8.0版本使用,但9.0版本中被弃用，故应该使用新的装饰属性。

记住，在表达式中使用的字段必须在一个<field>元素中声明,，这样网络客户端知道该列需要从服务器检索。如果我们不想显示给用户,我们应该使用invisible="1" 属性。

树元素的其他相关属性是：

**default\_order** 允许覆盖模型的默认排序，其值遵循在模型定义中order属性相同的格式。

**create 、delete和edit，**如果设置为false(以小写形式),则禁用列表视图中的相应操作。

editable，在列表视图中直接编辑记录 ，新记录将被添加的位置可能的值是顶部和底部。

列表视图可以包含字段和按钮，而且表单的大多数属性也在这里有效。在列表视图中，数值字段可以显示其列的汇总值。为此，可以向字段添加一个可用的聚合属性，sum、**avg、min或max，**并指定用于汇总值的标签文本。例如:

**<field name="amount" sum="Total Amount" />**

## 8.7 搜索视图

可用的搜索选项是通过<search>视图类型定义的。我们可以选择在搜索框中输入字段时自动搜索。我们还可以提供预定义的过滤器，用单击激活，并在列表视图中使用预定义的分组选项。

以下是对to\_do任务的一个搜索视图:

**<record id="todo\_app.view\_filter\_todo\_task"**

**model="ir.ui.view">**

**<field name="model">todo.task</field>**

**<field name="arch" type="xml">**

**<search>**

**<field name="name"/>**

**<field name="user\_id"/>**

**<filter name="filter\_not\_done" string="Not Done"**

**domain="[('is\_done','=',False)]"/>**

**<filter name="filter\_done" string="Done"**

**domain="[('is\_done','!=',False)]"/>**

**<separator/>**

**<filter name="group\_user" string="By User"**

**context="{'group\_by': 'user\_id'}"/>**

**</search>**

**</field>**

**</record>**

我们可以看到两个搜索字段—name和user\_id.当用户开始在搜索框上键入时，下拉框将建议搜索这些字段中的任何一个。如果用户键入“ENTER”，搜索将在第一个过滤器字段中执行。

然后我们有两个预定义的过滤器，过滤未完成和完成的任务。可以独立激活这些过滤器，并与一个or操作符连接. <separator/> 分离的块过滤器，将与“and”运算符连接起来。

第三个过滤器只根据上下文设置一个组。在这种情况下user\_id 告诉视图对纪录以此字段分组。

字段元素可以使用如下属性：

**name** 标识要使用的字段

**string** 是一个标签文本，而不是默认值。

**operator** 用于更改默认操作符(= 对应数字类型字段 ，ilike 用于其他类型字段)

**filter\_domain**设置一个特定的域表达式用于搜索,提供一个灵活的操作符属性。搜索的文本字符串在表达式中用self引用，一个简单的例子是:

**filter\_domain="[('name', 'ilike', self)]".**

**group** 只对属于某些安全组的用户进行搜索。这里需要一个逗号分隔的XML id列表。

对于过滤器元素，可用的属性:

**name** 是继承使用的标识符或通过窗口操作启用它。虽然不是强制性的，但始终提供它是一个很好的做法。

**string** 为筛选器显示的标签文本。必选项

**domain** 是要添加到当前域的域表达式。

**contex**是添加到当前上下文的上下文字典。通常为字段名称设置一个group\_id键，以对记录进行分组。

**groups** 使字段搜索仅对安全组列表（XML IDs)有效。

## 8.8 日历视图

顾名思义,这个视图类型介绍了日历记录,可以查看月,星期,或几天的时间。对于to\_do任务的日历视图如下:

**<record id="view\_calendar\_todo\_task" model="ir.ui.view">**

**<field name="model">todo.task</field>**

**<field name="arch" type="xml">**

**<calendar date\_start="date\_deadline" color="user\_id"**

**display="[name], Stage [stage\_id]" >**

**<field name="name" />**

**<field name="stage\_id" />**

**</calendar>**

**</field>**

**</record>**

日历视图属性:

**date\_start**开始日期的字段,必需字段。

**date\_end**结束日期的字段,可选字段。

**date\_delay**持续天数的字段,可以使用代替date\_end。

**all\_day**提供一个布尔字段是用来发送全天事件信号。在这些事件中，忽略了持续时间。

**color** 日历条目颜色分组的字段 。这个字段中的每个不同的值将被分配一个颜色，并且所有的条目将具有相同的颜色。

**display** 是每个日程表条目的显示文本. 它使用方括号加字段名称表示用户记录值，例如[name]。这些字段必须声明为日历元素的子元素，在前面的示例中可以看到。

**mode** 是日历的默认显示模式，无论是一天、一周还是一个月。

## 8.9 图形和枢轴视图

图表视图 ，它以图表的形式提供了一个数据的图形视图。to\_do 任务中当前可用字段不是一个图表的好候选，所以我们将在这样的视图中添加一个。

在todo\_ui/models/todo\_model.py 文件的 todotask 类中添加：

**effort\_estimate = fields.Integer('Effort Estimate')**

添加到todo任务表单中是十分必要的，以便我们可以在现有记录中添加值或添加它，并且能够检查这个新视图。

现在让我们添加todo任务图表视图:

**<record id="view\_graph\_todo\_task" model="ir.ui.view">**

**<field name="model">todo.task</field>**

**<field name="arch" type="xml">**

**<graph type="bar">**

**<field name="stage\_id" />**

**<field name="effort\_estimate" type="measure" />**

**</graph>**

**</field>**

**</record>**

图形视图元素可以有一个类型属性，可以设置为条形（默认）、饼状或线。在条形的情况下,可以使用stacked="True"呈现堆叠在一起的条形图。

数据也可以在主表中看到，这是一个动态分析矩阵。为此，我们有了在9.0版本中引入的枢轴视图。在8.0版本中，枢轴表已经可以使用了，但是在9.0中，它们拥有自己的视图类型。与此同时，它改进了枢轴表的UI特性，并极大地优化了枢轴表数据的检索。

要为todo任务添加一个枢轴表，请使用此代码：

**<record id="view\_pivot\_todo\_task" model="ir.ui.view">**

**<field name="arch" type="xml">**

**<pivot>**

**<field name="stage\_id" type="col" />**

**<field name="user\_id" />**

**<field name="date\_deadline" interval="week" />**

**<field name="effort\_estimate" type="measure" />**

**</pivot>**

**</field>**

**</record>**

图形和枢轴视图应该包含描述轴的字段元素和使用的方法。大多数可用属性在视图类型中都常看到。

**“name”**标识图中使用的字段，类似于其他视图的用法。

**type**是字段将如何使用，如行组（默认值）、度量值或列（仅用于枢轴表，用于列组）。

对于日期字段，interval是有意义的，它是用于按日、周、月、季度或年分组时间数据的时间间隔。

默认情况下，使用的聚合值是值的和。这可以通过设置在Python字段定义的**group\_operator**属性进行更改，使用的值包括平均值，最大值和最小值。

## 8.10 其他视图类型

值得注意的是，我们还没有覆盖其他三种视图类型：看板、甘特图和图表。

看板视图将在第9章、QWeb和看板视图中详细介绍。甘特视图需用到8.0版本,但这在社区版 9.0中删除,因为许可不兼容。

最后，图表视图用于非常特定的情况，而addon模块很少需要它们。为了以防万一，这两种视图类型的参考资料可以在官方文档中找到：

https://www.odoo.com/documentation/10.0/reference/views.html

## 8.11 第六章 小结

在本章中，我们学习了更多关于Odoo视图的内容以构建用户界面，涵盖了最重要的视图类型。在下一章中，我们将学到更多关于向应用程序中添加业务逻辑的内容。

# 9 第七章 ORM应用逻辑 - 支持业务流程

用 Odoo 编程 API，我们可以编写复杂的逻辑和向导，可以为我们的应用程序提供丰富的用户互动。在这一章中，我们将看到如何在我们的模型中编写代码，支持业务逻辑，我们还将学习如何激活事件和用户操作。

我们可以对事件执行计算和验证，比如创建或写入记录，或单击按钮时执行一些逻辑。例如，我们实施了待办任务，切换完成标志，并通过不激活来清除所有完成的任务。 此外，我们还可以使用向导来执行更复杂的用户交互，允许输入要求和提供反馈的互动过程。

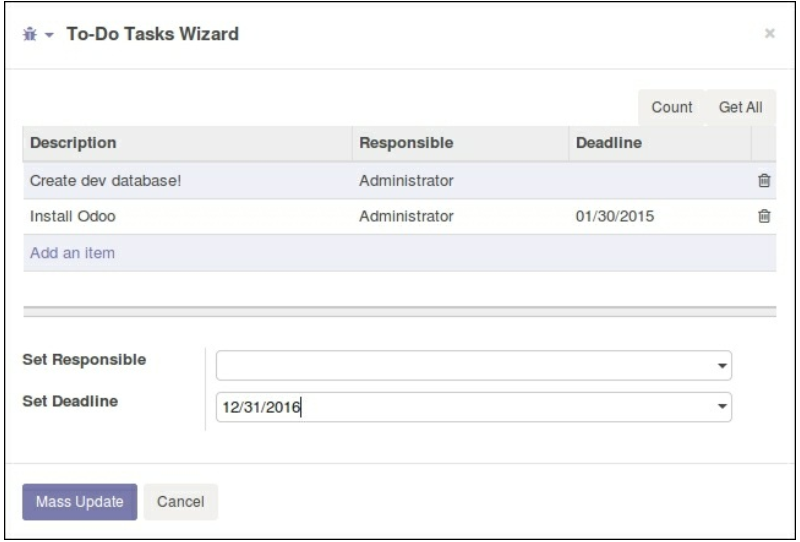
我们首先为我们的To-Do应用创建一个向导。

## 9.1 创建向导

假设我们的To-Do应用程序用户经常需要为批量任务设置最后的期限和任务的负责人。他们可以使用助手来帮助做这个。它应该允许他们选择要更新的任务，然后选择的截止日期或负责任的用户来设置它们。

向导视图，它是用来从用户获取输入的信息，然后用它来进一步处理的。他们可以用于简单的任务，如要求几个参数和运行报告，或复杂的数据操作，如前面所述的案例。

例如，这是我们的向导的外观︰



我们可以开始创建todo\_wizard新的功能插件模块。

我们的模块将有一个 Python 文件和 XML 文件，所以todo\_wizard/\_\_manifest\_\_.py描述将显示下面的代码︰

在以前的插件，在todo\_wizard/\_\_init\_\_.py文件增加一行︰

**from . import models**

接下来，我们需要描述支持我们向导的数据模型。

### 9.1.1 向导模型

向导显示窗体视图给用户，通常作为一个对话框窗口，某些字段必须填写。这些然后会利用向导逻辑。

这是使用相同的模型/视图体系结构作为常规视图的应用，但支持模型基于 models.TransientModel 代替models.Model。

这种类型的模型还具有数据库表示和存储状态存在，但是只有在向导完成工作之前，该数据才会有用。计划的作业会定期清理向导数据库表中的旧数据。 这个models/todo\_wizard\_model.py文件,我们需要定义与用户进行交互的字段︰ 要更新的任务清单，由负责的用户设置期限日期。

首先添加 models/\_\_init\_\_.py文件，写下面的代码行︰from .import todo\_wizard\_model

然后创建实际models/todo\_wizard\_model.py文件︰

**#-\*- coding︰ utf-8 -\*-**

**from odoo import models,fields,api**

**class TodoWizard(models.TransientModel):**

**\_name = 'todo.wizard'**

**\_description = 'To-do Mass Assignment'**

**task\_ids = fields.Many2many('todo.task',string='Tasks')**

**new\_deadline = fields.Date('Deadline to Set')**

**new\_user\_id = fields.Many2one('res.users',string='Responsible to Set')**

值得注意的是，与常规模型的一对多关系不应该被使用在瞬态模型。这样做的原因是它需要有规律的模型具有与瞬态模型之间的反转多对一关系，但是这是不允许的，因为有可能需要定期收集模型记录和瞬态记录。

### 9.1.2 向导窗体

向导窗体视图与普通模型是相同，除了两个特定的元素︰

 一个<footer>节点可以用于放置动作按钮  一种特殊type ="cancel"按钮可用于中断向导不执行任何操作

这是我们views/todo\_wizard\_view.xml文件的内容︰

**<odoo>**

**<record id="To-do Task Wizard" model="ir.ui.view">**

**<field name="name">To-do Task Wizard</field>**

**<field name="model">todo.wizard</field>**

**<field name="arch" type="xml">**

**<form>**

**<div class="oe\_right">**

**<button type="object" name="do\_count\_tasks"**

**string="Count"/>**

**<button type="object" name="do\_populate\_tasks"**

**string="Get All"/>**

**</div>**

**<field name="task\_ids">**

**<tree>**

**<field name="name"/>**

**<field name="user\_id"/>**

**<field name="date\_deadline"/>**

**</tree>**

**</field>**

**<group>**

**<group>**

**<field name="new\_user\_id"/>**

**</group>**

**<group>**

**<field name="new\_deadline"/>**

**</group>**

**</group>**

**<footer>**

**<button type="object" name="do\_mass\_update"**

**string="Mass Update" class="oe\_highlight"**

**attrs="{'invisible':[('new\_deadline','=',False),('new\_user\_id', '=',False)]}"/>**

**<button special="cancel" string="Cancel"/>**

**</footer>**

**</form>**

**</field>**

**</record>**

**<!-- More button Action -->**

**<act\_window id="todo\_app.action\_todo\_wizard"**

**name="To-Do Tasks Wizard"**

**src\_model="todo.task" res\_model="todo.wizard"**

**view\_mode="form" target="new" multi="True"/>**

**</odoo>**

我们在 XML 中看到<act\_window>窗口，通过使用src\_model属性将选项添加到 To-do Taskp窗体的多个按钮。target ="new" 属性将打开一个对话框窗口。

你可能也注意到批量更新模块按钮使用attr，直到选择新的截止日期或负责任的用户选择更新后才能可见。

### 9.1.3 向导业务逻辑

接下来，我们需要实现对窗体按钮执行操作。排除Cancel按钮，我们有三个操作按钮来实现，但现在我们将集中使用Mass Update按钮。

通过按钮调用的方法是do\_mass\_update和它应该被定义在models/todo\_wizard\_model.py文件，如下面的代码所示︰

**from odoo import exceptions**

**import logging**

**\_logger = logging.getLogger(\_\_name\_\_)**

**# ...**

**# class TodoWizard(models.TransientModel):**

**# ...**

**@api.multi**

**def do\_mass\_update(self):**

**self.ensure\_one()**

**if not (self.new\_deadline or self.new\_user\_id):**

**raise exceptions.ValidationError('No data to update!')**

**\_logger.debug('Mass update on Todo Tasks %s',**

**self.task\_ids.ids)**

**vals = {}**

**if self.new\_deadline:**

**vals['date\_deadline'] = self.new\_deadline**

**if self.new\_user\_id:**

**vals['user\_id'] = self.new\_user\_id**

**# Mass write values on all selected tasks**

**if vals:**

**self.task\_ids.write(vals)**

**return True**

我们的代码应该处理一次向导实例，所以我们明确指出这一点self.ensure\_one()。在这里 self表示向导窗体上数据的浏览记录。

该方法首先验证一个新的截止日期日期或负责任的用户如果没有就会出现错误。接下来，我们怎样写一个调试信息传到服务器日志的例子。

然后使用批量更新一起来设置构建“vals”字典:

新的日期，新的责任人，或者两者都有。然后在记录集上执行批量更新的平台上使用“write”方法。这比在每个记录上执行单独的操作循环更有效。

对于总是返回某些东西的方法来说，这是一种很好的做法。这就是为什么它最后返回True的值。唯一的原因是XML-RPC协议不支持任何值，因此这些方法无法使用这一协议。

在实践中，您可能没有意识到这个问题，因为web客户端使用的是JSON-RPC，而不是XML-RPC，但是这仍然是一个很好的实践。

接下来，我们将对日志进行更深入的研究，然后对顶部的两个按钮进行后面的逻辑分析︰Count和Get All

### 9.1.4 记录

这些大规模的更新可能会被滥用，所以它可能要记录一些信息时，是一个好的主意。前面的代码中在两行 TodoWizard 类前初始化 \_logger，使用 Python logging标准库。Python \_\_name\_\_ 的内部变量是标识来自这个模块的消息。

将方法代码中写入日志消息，我们可以使用︰

**\_logger.debug('A DEBUG message')**

**\_logger.info('An INFO message')**

**\_logger.warning('A WARNING message')**

**\_logger.error('An ERROR message')**

当传递值用于在日志消息中，而不是使用字符串交互值，我们应该提供他们作为附加参数。举个例子，不是

**\_logger.info('Hello %s' % 'World')**

我们应该使用

**\_logger.info('Hello %s', 'World')。**

你可能注意到，我们在做 do\_mass\_update() 方法。

注意 关于日志记录，值得注意的是，日志条目总是打上UTC时间戳。对于新管理员来说，这可能是一个惊喜，这是由于服务器内部使用UTC处理所有的日期。

### 9.1.5 抛出异常

当有什么不对时，我们会想要中断程序并显示错误消息。这是通过抛出异常来完成。Odoo 提供可用在 Python 中的几个额外的异常类。这些都是最有用的例子︰

**from odoo import exceptions**

**raise exceptions.Warning('Warning message')**

**raise exceptions.ValidationError('Not valid message')**

Warning消息也会中断执行，但听起来不像ValidationError那么严重。虽然它不是友好的用户界面，我们利用这一点对Count按钮来向用户显示一条消息︰

**@api.multi**

**def do\_count\_tasks(self):**

**Task = self.env['todo.task']**

**count = Task.search\_count([('is done', '=', False)])**

**raise exceptions.Warning('There are %d active tasks.' %count)**

作为一个侧边备注，它看起来像我们可以使用@api.model装饰器，因为此方法不能操作self的记录集。但我们在这种情况下不能从一个按钮调用方法。

### 9.1.6 向导中的助手操作

现在，假设我们想要一个按钮来自动挑选所有的任务，让用户一个接一个地挑选它们。这就是在表单中Get All按钮的要点。该按钮背后的代码将获得一个记录集，其中包含所有活动的任务，并将其分配给多对多字段中的任务。

但这里有个问题。在对话框窗口中，当按下一个按钮时，向导窗口会自动关闭。我们没有使用Count按钮来面对这个问题，因为它使用一个异常来显示它的消息;因此，该操作不成功，窗口没有关闭。

幸运的是，我们可以通过请求客户机重新打开相同的向导来解决这个问题。模型方法可以返回由web客户端执行的窗口操作，在字典对象形式下执行。该字典使用与XML文件中定义窗口操作相同的属性。

我们将为窗口操作字典定义一个helper函数，以重新打开向导窗口，以便可以在几个按钮中轻松地重复使用它:︰ @api.multi

**@api.multi**

**def \_reopen\_form(self):**

**self.ensure\_one()**

**return {**

**'type': 'ir.actions.act\_window',**

**'res\_model': self.\_name, # this model**

**'res\_id': self.id, # the current wizard record**

**'view\_type': 'form',**

**'view\_mode': 'form',**

**'target': 'new'}**

值得注意的是，窗口操作可能是其他的东西，比如跳转到不同的向导表单来请求额外的用户输入，这可以用于实现多页面向导。 现在，Get All按钮可以完成它的工作，并且仍然让用户在同一个向导中工作︰

**@api.multi**

**def do\_populate\_tasks(self):**

**self.ensure\_one()**

**Task = self.env['todo.task']**

**open\_tasks = Task.search([('is\_done', '=', False)])**

**# Fill the wizard Task list with all tasks**

**self.task\_ids = all\_tasks**

**# reopen wizard form on same wizard record**

**return self.\_reopen\_form()**

在这里，我们可以看到如何使用任何其他可用的模型:我们首先使用self.env[]获得对模型的引用，在这种情况下todo.task，然后可以对其执行操作，比如search()，以检索满足某些搜索条件的记录。 瞬态模型将值存储在向导表单字段中，可以像其他模型一样读取或写入。将all\_tasks变量分配给模型task\_ids一对多字段。正如您所看到的，这就像我们对任何其他字段类型所做的那样。

## 9.2 使用ORM API

从在上一节中，我们已经尝到使用ORM API的滋味。接下来，我们将看看能用它做些什么。

### 9.2.1 装饰器方法

在我们的使用中，我们遇到的几种方法使用 API 装修器像@api.multi。这些都是服务器知道的重要如何处理方法。现在让我们回顾一下他们的使用。

@api.multi装修器最常用来处理新的 API 记录集。在这里self是一个记录集，且方法通常会包括for 循环来循环它。

在某些情况下，编写该方法以用在单例模式︰ 包含不超过一条记录的记录集。 @api.one 装饰器在 9.0 被弃用，应该避免再使用。相反，我们应该使用 @api.multi 并在方法代码中添加一条行self.ensure\_one()，以确保它是一个单例。

提到@api.one装饰器已被弃用，但仍然支持。出于完整性的考虑，可能需要知道它封装修饰器的方法，一次为为他提供一条记录，并它为做记录集迭代。在我们的方法能保证self是单例模式。每个单独的方法调用的返回值进行聚合为一个列表并返回。

@api.model装饰器是一个类级别的静态方法，它不使用任何记录集数据。为保持一致性， self仍然是一个记录集，但它的内容无关。请注意，不能从用户界面中的按钮使用这种类型的方法。

其他一些装修器有更具体的用途，它们将与前面所描述的装饰器一起使用︰

 @api.depends(fld1,...)用于计算字段函数，用于确保(重新)计算改变被触发  @api.constrains(fld1,...)用于验证函数，用于确保哪些约束生效被触发  @api.onchange(fld1,...)用于改变函数，用于确保表单上字段的动作被触发

特别是onchange方法可以向用户界面发送一条警告消息。例如，这可以警告用户刚刚输入的产品数量在库存中是不可用的，而不会阻止用户继续使用。这通过方法返回一个描述警告消息的字典来完成的︰

**return {**

**'warning': {**

**'title': 'Warning!',**

**'message': 'You have been warned'}**

**}**

### 9.2.2 重写 ORM 的默认方法

我们已经了解了API所提供的标准方法，但是这些方法并没有结束！我们还可以扩展它们，以向我们的模型添加自定义行为。

最常见的情况是扩展create()和write()方法。这可以用于添加在执行这些操作时触发的逻辑。通过将我们的逻辑放在适当部分中的自定义方法，我们可以在执行主操作之前或之后运行代码。

使用TodoTask模型作为示例，我们创建一个定制的create()，它看起来是这样的:︰

**@api.model**

**def create(self, vals):**

**# Code before create: can use the `vals` dict**

**new\_record = super(TodoTask, self).create(vals)**

**# Code after create: can use the `new\_record` created**

**return new\_record**

自定义write()将遵循以下结构︰

**@api.multi**

**def write(self, vals):**

**# Code before write: can use `self`, with the old values**

**super(TodoTask, self).write(vals)**

**# Code after write: can use `self`, with the updated values**

**return True**

这些都是常见的扩展示例，但当然，任何用于模型的标准方法都可以以类似的方式继承，以将我们的自定义逻辑添加到该模型中。

这些技术为您提供了许多可能性，但是请记住，其他工具也可以更适合于常见的特定任务︰

 要有一个基于另一个字段值的字段值，我们应该使用计算字段。这样做的一个例子是，当改变行值时，计算一个标题总数。

 要有动态计算的字段缺省值，我们可以使用字段缺省值绑定到函数而不是固定值。

 要在字段被更改时设置其他字段的值，我们可以在更改函数上使用。例如，在选择客户时，将其货币设置为文档的货币，稍后可以由用户手动更改。请记住，更改只在表单视图交互中工作，而不是直接写调用。

 对于验证，我们应该使用修饰过的约束函数

@api.constraints(fld1,fld2,…)。这些字段类似计算字段，但它们代替计算值会引起错误。

### 9.2.3 RPC和Web客户端调用的方法

我们已经看到了用于生成记录集最重要的模型方法，以及如何对它们进行写入。但是，有更多的模型方法可以用于更具体的操作，如下所示︰

** read(fields)与browse**方法类似，但它不是一个记录集，而是返回一个包含作为其参数的字段的数据行列表。每行是一个字典。它提供了可以通过RPC协议发送表示数据的序列化，它的目的是供客户机程序使用，而不是在服务器逻辑中使用。

 **search\_read([domain], [fields], offset=0, limit=None, order=None)** 执行搜索操作，然后在结果记录列表中读取。它的目的是为RPC客户端使用，并为他们省去进行search时所需要的read结果上额外往返行程。

 **load([fields], [data])** 用于从CSV文件中导入数据。第一个参数是要导入的字段列表，它直接映射到CSV 第一行。第二个参数是一个记录列表，其中每个记录都是一个用于解析和导入的字符串值列表，它直接映射到CSV数据行和列。它实现了在第4章、模块数据中描述的CSV数据导入的特性，就像外部标识符支持一样。它被web客户端导入特性使用。它取代了已弃用的import\_data方法。

 **export\_data ([fields], raw\_data = False)** 被web客户端导出函数使用。它返回一个包含数据的数据键字典;一个行列表。字段名可以使用。在CSV文件中可以使用字段名字的id和/id后缀，数据的格式要与导入的CSV文件兼容。可选的rawdata参数允许使用它们的Python类型导出数据值，代替CSV中使用的字符串表示。

下面的方法主要用于web客户端来呈现用户界面和执行基本的交互︰

** name\_get()**︰ 这将返回每条记录(ID、name)的文本元组。它被默认用于计算displayname值，提供关联字段的表示广本。可以扩展它来实现显示自定义，例如显示记录代码和名称，而不是只显示名称。

** name\_search(name=,args=None, operator='ilike',limit=100)** 返回一个(ID、name)元组的列表，其中显示名与name参数中的文本匹配。它在UI中使用，建议同时在一个关系字段中输入文本匹配的记录。例如，它被用于在一个字段中输入一个产品时，通过名称和引用来实现产品查找。

** name\_create(name)** 创建一个新的记录，只使用标题名称来使用它。它在UI中用于“快速创建”功能，您可以通过提供它的名称来快速创建相关记录。可以通过扩展来为通过该特性创建的新记录提供特定的缺省值。。

** default\_get([fields])** 返回一个包含创建新记录的默认值字典。默认值可能取决于当前用户或会话上下文等变量。

 **fields\_get()**用于描述模型的定义字段，如在开发人员菜单的View Fields字段选项中所看到的那样。

** fields\_view\_get()**web客户端使用它来检索要呈现UI视图的结构。可以将视图的ID作为参数或我们想要使用view\_type='form'。例如，你可以试试这个︰rset.fields\_view\_get(view\_type='tree')。

### 9.2.4 shell命令

Python有一个命令行接口，这是探索语法的一种很好的方式。同样，Odoo也有一个类似的功能，在那里我们可以交互式地尝试命令，看看他们是如何工作的。这就是shell命令。 使用它，用shell命令和数据库来运行Odoo，如下所示︰

**$ ./odoo-bin shell -d todo**

您会在终端中看到通常的服务器启动顺序，在Python提示符中等待您的输入，直到它停止。在这里，self将代表Administrator用户的记录，您可以确认输入以下内容︰

**>>> self**

**res.users(1,)**

**>>> self.\_name**

**'res.users' >>>**

**self.name**

**u'Administrator'**

在前一阶段，我们对环境进行了一些检查。self代表一个res.users记录集，它只包含ID 的记录。我们还可以确认记录集的模型名检查self.name，并获取记录name字段的值，确认它是Administrator用户。

与Python一样，您可以使用Ctrl+D退出提示符。这也将关闭服务器进程，并返回到系统shell提示符。

小提示

在9.0版本中添加了shell特性。对于8.0版本，有一个社区的模块支持这个shelll特性。下载到addons路径中，就不需要进行安装了。它可以从这个地址

https://www.odoo.com/apps/modules/8.0/shell/下载 .

### 9.2.5 服务器环境

服务器shell提供了一个与用户模型方法相同的self引用res.users。

正如我们所见，self是一个记录集。记录集携带着一个环境信息，包括用户浏览数据和附加的上下文信息，比如语言和时区。这些信息很重要，也很重要。

我们可以开始检查我们当前的环境︰

**>>> self.env**

**<openerp.api.Environment object at 0xb3f4f52c>**

在self.env执行环境具有下列属性可用︰

**env.cr**被数据库游标使用 

**env.uid**是用户会话ID 

**env.user**是当前用户的记录 

**env.context**是会话上下文可变的字典

环境还提供对所有已安装模型的注册中心可使用的访问。例如self.env['res.partner']。返回是一个伙伴模型。我们可以使用search()或browse()来检索记录集︰

**>>> self.env['res.partner'].search([('name', 'like', 'Ag')])**

**res.partner(7, 51)**

在此示例中，记录集res.partner模型包含两条记录，即 IDs7和IDs51.

### 9.2.6 修改执行环境

环境是不可变的，所以它不能被修改。但是我们可以创建一个 修改后的环境，然后运行操作使用它。 可以使用这些方法:

 env.sudo(user) 提供了一个用户记录，并返回那个

 用户环境。如果没有提供用户，Administrator这个超级用户将被使用，允许运行特定的安全规则查询.

 env.with\_context(dictionary)替换为一个新的上下文。env.with\_context(key=value,...) 修改当前上下文设置一些键值的值。

此外，我们还有env.ref()函数，使用一个外部的字符串 标识符并返回一个记录，如下所示︰

**>>> self.env.ref('base.user\_root')**

**res.users(1,)**

### 9.2.7 事务和底层SQL

数据库写入操作是在数据库的上下文中执行的事务。通常情况下，我们不用担心服务器在运行模型方法时的问题。 但在某些情况下，我们可能需要更好地控制事务。这可以通过数据库游标self.env.cr完成,如下所示︰

 self.env.cr.commit()提交事务的缓冲写入操作 

self.env.savepoint()设置一个事务回滚到保存点 

self.env.rollback()取消事务最后一个保存点的写操作，因或者假设没有创建保存点

小提示

在shell会话中，除非使用self.env.cr.commit()否则你的数据操作不会使用。

使用游标execute()方法，我们可以直接在数据库中运行SQL。它使用带有SQL语句的字符串和第二个可选参数作为SQL的参数的一个元组或值列表。这些值将使用了%s占位符。

注意

小心 ！

对于cr.execute()，我们应该拒绝直接将参数值添加到查询字符串。可以通过SQL加以利用注入攻击，这是众所周知的安全风险。通常使用%s占位符和第二个参数值。 如果您使用的是SELECT查询，那么应该获取记录。fetchall()函数检索所有的行，作为一个tupples的列表和dictfetchall()将它们作为一个字典列表检索，如下面的示例所示︰

**>>> self.env.cr.execute("SELECT id, login FROM res\_users WHERE**

**login=%s OR id=%s", ('demo', 1))**

**>>> self.env.cr.fetchall()**

**[(4, u'demo'), (1, u'admin')]**

还可以运行数据操作语言(DML)指令像UPDATE和INSERT。由于服务器保存数据缓存，它们可能会与数据库中的实际数据不一致。正因为如此，在使用时原始的DML，应该在以后使用这些缓存self.env.invalidate\_all()。

注意

小心 ！

直接在数据库中执行SQL可能导致不一致的数据。只有你确信你要做什么事才使用它。

## 9.3 使用记录集

现在我们将探索ORM是如何工作的并学习最常见的操作执行。

我们将使用shell提供的提示符命令交互式地探索记录集是如何工作的。

### 9.3.1 查询模型

对于self，我们只能访问该方法的记录集。但self.env环境引用允许我们访问任何其他模型。例如，self.env['res.partner'] 返回对合作伙伴模型的引用(这是实际上一个空记录集)。然后我们可以使用search()或browse()生成记录集。

**search()**方法获取一个域表达式并返回一个记录集记录匹配这些条件。空域将返回所有记录。有关域表达式的更多细节请参阅 第6章节 视图-设计用户界面。如果模型具有**active**的特殊字段，默认情况下只有active=True的记录才会被考虑。

这里有一些可选的关键字参数，如下所示︰

order用作ORDER BY条件的数据库查询字符串。这通常是一个以逗号分隔的字段名列表。 

limit设置要检索的最大记录数量。 

offset忽略了第一个n个结果;它可以用于limit查询一段时间的记录

有时候我们只需要知道记录的数量条件。为此，我们可以使用search\_count()来返回记录 计数而不是记录集。它节省了检索要计数记录列表的成本，所以当我们还没有记录集的时候就更有效率了，只需要计算记录的数量。 browse()方法获取返回一个记录集IDs或单个ID的列表方法。这我们想要已经很方便的记录IDs。

这里展示了一些使用示例︰

**>>> self.env['res.partner'].search([('name', 'like', 'Ag')])**

**res.partner(7, 51)**

**>>> self.env['res.partner'].browse([7, 51])**

**res.partner(7, 51)**

### 9.3.2 单例模式

只有一个记录的记录集特殊情况称为单例记录集。单例仍然是一个记录集，可以在任何记录集的地方预计使用。 但是与多元素记录集不同，单例可以使用它访问它们的字段点表示法，如下所示︰

**>>> print self.name**

**Administrator**

在下一个例子中，我们可以看到self的单例记录集也有行为作为一个记录集，我们可以迭代它。它只有一个记录，所以只有一个名字打印出来︰

**>>> for rec in self:**

**print rec.name**

**Administrator**

尝试访问记录集上的字段值，其中有多个记录错误，所以这可能是一个问题我们不确定我们是否在处理一个单例记录集。对于只使用单例模式的方法，我们可以在开始时检查使用self.ensure\_one() 。如果self不是单例它会引发错误。

小提示

请注意，空的记录也是单例。

### 9.3.3 写在记录上

记录集实现了活动记录模式。这意味着我们可以分配它们的值，这些更改将在数据库中持久存储。这个这是一种直观方便操作数据的方式，如下所示︰

**>>> admin = self.env['res.users'].browse(1)**

**>>> print admin.name**

**Administrator**

**>>> admin.name = 'Superuser'**

**>>> print admin.name**

**Superuser**

记录集也有三种方法来处理他们的数据︰create(), write()，和unlink()︰

create()方法使用字典将字段映射到值并返回值创建记录。默认值是按照预期自动应用的，也就是这里显示︰

**>>> Partner = self.env['res.partner']**

**>>> new = Partner.create({'name': 'ACME', 'is\_company': True})**

**>>> print new**

**res.partner(72,)**

unlink()︰方法删除记录的记录集，如下所示︰

**>>> rec = Partner.search([('name', '=', 'ACME')])**

**>>> rec.unlink()**

**True**

Write()用使用字典将字段映射到值。这些都是更新记录集的所有元素，没有返回任何内容，如下所示︰

**>>> Partner.write({'comment': 'Hello!'})**

使用活动记录模式有一些限制;它只更新一个一段时间的字段。另一方面，write()方法可以通过一个简单的数据库指令更新几个字段。这些在性能可能是问题的情况下，应该记住差异。

同样值得一提的是copy()来复制一个已有的记录；作为一个可选的参数和一个带有值的字典记录。例如，要从演示用户创建一个新用户︰

**>>> demo = self.env.ref('base.user\_demo')**

**250>>> new = demo.copy({'name': 'Daniel', 'login': 'dr','email':''})**

注意

请记住带有copy=False属性的字段不会被复制。

### 9.3.4 使用时间和日期

由于历史原因，ORM记录集处理date和datetime时间值使用他们的字符串表示，而不是实际的Python date和Datetime对象。在数据库中，它们存储在date字段中，但是datetimes存储在UTC时间。

**odoo.tools.DEFAULT\_SERVER\_DATE\_FORMAT**

**odoo.tools.DEFAULT\_SERVER\_DATETIME\_FORMAT**

他们映射到%Y-%m-%d和%Y-%m-%d %H:%M:%S分别。

协助处理日期，fields.Date和fields.Datetime提供几个函数。例如︰

**>>> from odoo import fields**

**>>> fields.Datetime.now()**

**'2014-12-08 23:36:09'**

**>>> fields.Datetime.from\_string('2014-12-08 23:36:09')**

**datetime.datetime(2014, 12, 8, 23, 36, 9)**

日期和时间由服务器处理和存储在一个简单的UTC格式中，这可能不同于用户正在处理的时区中的服务器的日期和时间。正因为如此，我们可以使用的其他一些功能，来帮助我们处理这个问题︰

fields.Date.today() 返回当前日期的字符串格式，并使用UTC作为参考是服务器所期望的。这是符合计算机的默认值。 

fields.Datetime.now() 返回一个带有当前datetime的字符串，并使用UTC作为参考的格式是服务器所期望的。这是符合计算机的默认值。  fields.Date.context\_today(record, timestamp=None)返回一个在会话的上下文中使用当前日期。时区值是从记录的上下文中获取，并使用可选参数datetime而不是当前时间。 fields.Datetime.context\_timestamp(record, timestamp) 转换一个真实的datetime(没有时区)进入一个时区感知的datetime。那时区是从记录的上下文中提取出来的，因此做为函数的名字。

促进格式之间的转换，这两个fields.Date和fields.Datetime提供了这些功能︰

rom\_string(value)将字符串转换为日期或datetime对象 

to\_string(value)转换为一个date或datetime对象为字符串是服务器所期望的格式

### 9.3.5 对记录集的操作

记录集支持对它们进行额外的操作。我们可以检查a记录是否被包含在一个记录集中。如果x是一个单例记录集和myrecordset是一个记录集包含许多记录的记录集，我们可以使用它︰

**x in my\_recordset**

**x not in my\_recordset**

以下操作也是可用的︰

recordset.ids 返回以元素IDs记录集列表 

recordset.ensure\_one() 检查它是否为一个单例的记录(singleton);如果它不是，有一个ValueError异常抛出 

recordset.filtered(func) 返回一个经过过滤的记录集 

recordset.mapped(func) 返回一个映射值的列表 

recordset.sorted(func)返回一个有序的记录集

这里是这些函数的用法示例︰

**>>> rs0 = self.env['res.partner'].search([])**

**>>> len(rs0) # how many records?**

**40**

**>>> starts\_A = lambda r: r.name.startswith('A')**

**>>> rs1 = rs0.filtered(starts\_A)**

**>>> print rs1**

**res.partner(8, 7, 19, 30, 3)**

**>>> rs2 = rs1.filtered('is\_company')**

**>>> print rs2**

**res.partner(8, 7)**

**>>> rs2.mapped('name')**

**[u'Agrolait', u'ASUSTeK']**

**>>> rs2.mapped(lambda r: (r.id, r.name))**

**[(8, u'Agrolait'), (7, u'ASUSTeK')]**

**>> rs2.sorted(key=lambda r: r.id, reverse=True)**

**res.partner(8, 7)**

### 9.3.6 操作记录集

我们肯定想要添加、删除或替换这些相关的元素字段，因此这就引出了一个问题:如何对记录集进行操作?

记录集是不可变的，这意味着它们的值不能被直接修改。相反，修改一个现有的记录集意味着根据一个新的记录集创建一个新的记录集。

一种方法是使用受支持的设置操作︰

 rs1 | rs2是设置union操作，结果是两个记录集所有的元素。

 rs1 + rs2是设置addition操作，结果是两个记录集做为一体的元素，它可能会导致重复记录。

 rs1 & rs2是设置intersection的操作，结果只是两个记录集相同的元素。

 rs1 – rs2是设置difference的操作，结果是rs1元素不在rs2记录集的元素 也可以使用切片表示法，如这些例子所示︰

 rs[0]和rs[-1]分别检索第一个元素和最后一个元素。

 rs[1:]结果是拷贝没有第一个元素的记录集。这个生成与rs – rs[0]相同的记录，但它保存原来的顺序。

注意

在Odoo 10中，操纵记录集保存顺序。这是不同于以往Odoo的版本，尽管添加和切片是已知的，操纵记录集不能保证记录的顺序。

我们可以通过使用这些操作来删除或添加更改记录集元素。这里有一些例子︰

 self.task\_ids |= task1如果不在记录集，则添加task1记录

 self.task\_ids -= task1如果在记录集中，则删除特定的记录task1

 self.task\_ids = self.task\_ids[:-1]删除最后一条记录

关系字段包含记录集值。多对一的字段可以包含一个单例记录集，并且对多字段包含有任意数量的记录集记录。我们使用一个常规的赋值语句，或者使用create()和write()方法和一个字典的值。在这最后条件，一种特殊的语法用于修改对多字段。它是相同的XML记录为关系字段提供值，在第4章，模块数据，设置相关字段值部份中。

例如，write()语法与前面的三个任务等价的例子是。 作为一个例子，write()相当于三个前面的分配示例中的语法是︰

** self.write([(4, task1.id, None)]) 添加task1记录**

** self.write([(3, task1.id, None)]) 从记录集删除task1**

** self.write([(3, self.task\_ids[-1].id, False)]) 删除最后一条记录**

### 9.3.7 使用关系字段

正如我们前面所看到的，模型可以有关系字段:多对一、一对多和多对多。这些字段类型有记录集作为值。

在多对一的情况下，值可以是单例或空记录集。在这两种情况下，我们都可以直接访问它们的字段值。作为一个例子,以下说明是正确和安全的︰

**>>> self.company\_id**

**res.company(1,)**

**>>> self.company\_id.name**

**u'YourCompany'**

**>>> self.company\_id.currency\_id**

**res.currency(1,)**

**>>> self.company\_id.currency\_id.name**

**u'EUR'**

方便的是，空记录集也表现为单例行为，它访问字段不会返回错误，但返回False。因为这个，我们可以使用点表示法遍历记录而不用担心来自空值错误，如下所示︰

**>>> self.company\_id.country\_id**

**res.country()**

**>>> self.company\_id.country\_id.name**

**False**

### 9.3.8 处理关系字段

在使用活动记录模式时，可以分配这样的关系字段记录集。

对多对一字段，分配的值必须是单个记录(一个单例 记录集)。

对一对多字段，它们的值也可以用记录集来分配，如果有的话，用一个新的替换为连接记录列表。这里允许一个任意大小的记录集。

在使用create()或write()方法时，在其中指定使用字典的值，关系字段不能被分配给记录集值。那应该使用相应的ID或IDs列表。

例如,替换self.write({'user\_id': self.env.user})，我们应该使用 self.write({'user\_id': self.env.user.id}).

摘要

在前几章中，我们了解如何构建模型和设计视图。在这里我们更进一步，学习如何实现业务逻辑和使用用于操作模型数据的记录集。

我们还看到了业务逻辑如何与用户界面交互，并学会了创建与用户通信的向导，并作为一个平台启动高级流程的向导。

在下一章中，我们将学习如何在addon添加自动化测试模块，以及一些调试技术。

## 9.4 第七章 小结

在前面的章节中，我们看到了如何构建模型和设计视图。本章，我们更近一步，学习如何实现业务逻辑和使用记录集操作模型数据。

我们还看到了业务逻辑如何与用户界面交互，并学会了创建与用户通信的向导，并作为一个平台来启动高级流程。

在下一章中，我们将了解我们的插件模块添加的自动化测试，以及一些调试技术。

# 10 第八章 编写测试和调试代码

好的开发人员是工作的测试和调试代码。自动化的测试是一个不可估量的工具来建立和维持强健的软件。在这一章，我们将学习如何将自动的测试添加到我们的插件模块，以使它们更加可靠。服务器端调试技术也提出了，允许开发人员要检查和理解他的代码发生了什么。

## 10.1 单元测试

自动化的测试被公认的软件的最佳做法。它不单止有助我们确保我们的代码正确实施。更重要的是，它为未来代码增强或改写提供安全网。

在动态编程语言，如 Python，因为那里是没有编译步骤，语法错误可以被忽视。这使得它更重要的是要通过尽可能多的行的代码尽可能的单元测试。

编写测试时，描述了这两个目标可以提供指路的明灯。您的测试的第一个目标应该是提供了很好的测试覆盖，设计测试用例，它们在所有代码行。这独自一人通常就会进步上第二个目标 — — 以显示代码的功能正确性。

这独自将通常好在取得进展的第二个目标-显示功能代码的正确性，因为在此之后我们一定会很好的起点，打造更多测试用例为非明显的用例。

## 10.2 添加单元测试

使用，Python 测试添加到插件模块测试 /子目录。测试运行器会自动发现测试具有该特定名称的子目录中。

试验研究我们todo\_wizard插件将在tests/test\_wizard.py文件。我们将需要添加测试/来文件︰

**from . import test\_wizard**

这将是为基本骨架tests/test\_wizard.py:

**# -\*- coding: utf-8 -\*-**

**from odoo.tests.common import TransactionCase**

**class TestWizard(TransactionCase):**

**def setUp(self, \*args, \*\*kwargs):**

**super(TestWizard, self).setUp(\*args, \*\*kwargs)**

**# Add test setup code here...**

**def test\_populate\_tasks(self):**

**"Populate tasks buttons should add two tasks"**

**# Add test code**

Odoo 提供几个类，用于测试。 TransactionCase 测试使用不同的事务对于每个测试，在结束时是自动回滚。

我们还可以使用SingleTransactionCase，运行所有测试的单个事务，这是只有在回滚结束了最后的测试中。这可以很有用，当你想要的初始状态为下列测试每个测试的最终状态

setup （)方法是我们准备数据和要使用的变量的地方。我们将通常将它们存储为类的属性，以便他们可以在测试方法中使用。

然后应把测试，作为类方法实现像test\_populate\_tasks()。测试用例方法名称必须与 test\_ 前缀。他们自动发现，和此前缀是什么标识执行测试用例的方法。

方法将运行在测试函数名称的顺序。当使用

TransactionCase类，在每年年底将完成回滚。该方法的文档字符串显示当测试运行，并应为它提供一个简短的描述

这些测试类是包装单元测试测试用例。这是 Python 标准库的一部分，您可以参阅其文档以获取更多的细节，在[https://docs.python.org/2/library/unittest.html](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fdocs.python.org%2F2%2Flibrary%2Funittest.html).

为了更准确地说，Odoo 使用单元测试扩展库unittest2.

## 10.3 写测试用例

现在让我们展开test\_populate\_tasks()在我们最初的骨架中看到的方法。最简单的测试，我们可以写，从被测对象，查询的结果来验证，运行一些代码，然后使用 assert 与预期的结果进行比较。

test\_populate\_tasks()方法将测试do\_populate\_tasks()Todo 方法。因为我们确信我们有两个开放 Todos 的设置，在运行它之后我们指望向导 task\_ids 来引用这两个记录。

**# class TestWizard(TransactionCase):**

**def test\_populate\_tasks(self):**

**"Populate tasks buttons should add two tasks"**

**self.wizard.do\_populate\_tasks()**

**count = len(self.wizard.task\_ids)**

**self.assertEqual(count, 2, 'Wrong number of populated**

**tasks')**

文档字符串，在方法定义中，第一行是有用来描述测试和运行它时打印出来。

检查核实，是否测试成功，或失败是self.assertEqual的声明。最后一个参数是可选的但推荐是因为它提供了更详尽的信息时，则测试失败。

紧接是最常用的但它是可用的 assert 方法之一。我们应该使用 assert 函数适合每种情况下，因为他们会更有助于理解测试失败的原因。举个例子，而不是比较的长度 task\_ids，一个人能准备的两个预期的任务，和然后使用记录集︰

**self.assertItemsEquals(**

**self.wizard.task\_ids, expected\_tasks,**

**'Incorrect set of populated tasks')**

这会给万一失败，与预期的任务与实际完全比较最好的输出。

**注意**

单元测试文档提供了很好的参考有关所有可用在[https://docs.python.org/2/library/unittest.html#test-cases](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fdocs.python.org%2F2%2Flibrary%2Funittest.html%23test-cases)的方法。

以添加新的测试用例、 向类中添加另一种方法，它的实施。接下来我们将测试 do\_mass\_update() 向导方法。这是一个做的工作，当我们点击向导的**确定**按钮︰

**def test\_mass\_change(self):**

**"Mass change deadline date"**

**self.wizard.do\_populate\_tasks()**

**self.wizard.new\_deadline = self.todo1.date\_deadline**

**self.wizard.do\_mass\_update()**

**self.assertEqual(**

**self.todo1.date\_deadline,**

**self.todo2.date\_deadline)**

我们开始通过运行do\_populate\_tasks()再一次。请记住，与 TransactionCase测试中，每个测试尽头是完成回滚。所以前面测试中所进行的操作进行恢复，，我们需要再次填充向导的 Todo 任务列表。下一步我们模拟用户填写新的截止日期字段和执行大规模更新。在结束了我们的检查是看是否这两个 Todo 任务结束的同一日期。

## 10.4 设置测试

我们应该开始准备要测试中使用的数据。

它是方便进行测试操作根据特定的用户，还测试访问控制配置正确。这使用实现 sudo() 模型方法。记录集执行该信息与他们，所以后被创建时使用 sudo()，将使用同样的背景下执行随后的操作在同一个记录集中。

这是代码安装程序方法和几个还需要的额外的导入语句︰

**from datetime import date**

**from odoo.tests.common import TransactionCase**

**from odoo import fields**

**class TestWizard(TransactionCase):**

**def setUp(self, \*args, \*\*kwargs):**

**super(TestWizard, self).setUp(\*args, \*\*kwargs)**

**# Close any open Todo tasks**

**self.env['todo.task']\**

**.search([('is\_done', '=', False)])\**

**.write({'is\_done': True})**

**# Demo user will be used to run tests**

**demo\_user = self.env.ref('base.user\_demo')**

**# Create two Todo tasks to use in tests**

**t0 = date.today()**

**Todo = self.env['todo.task'].sudo(demo\_user)**

**self.todo1 = Todo.create({**

**'name': 'Todo1',**

**'date\_deadline': fields.Date.to\_string(t0)})**

**self.todo2 = Todo.create({**

**'name': 'Todo2'})**

**# Create Wizard instance to use in tests**

**Wizard = self.env['todo.wizard'].sudo(demo\_user)**

**self.wizard = Wizard.create({})**

来测试我们的向导，我们想要两个开放 Todos。所以我们开始通过关闭任何现有的托，不会妨碍我们的测试中，创建两个新 Todos 对于测试，使用演示用户。我们最后创建我们的向导，使用演示用户，一个新实例，并将其分配给 self.wizard，所以这是可用的测试方法.

## 10.5 测试异常

有时我们需要我们的测试来检查是否生成异常。常见的情况是当测试如果一些验证做得好。

在我们的示例中，test\_count()方法使用警告异常作为一种方式向用户提供信息。要检查是否引发了异常，我们把里面的相应代码与 self.assertRaises() 块。

我们需要导入警告在文件顶部的例外︰

from odoo.exceptions import Warning

并将添加到测试类与另一个测试用例的方法︰

**def test\_count(self):**

**"Test count button"**

**with self.assertRaises(Warning) as e:**

**self.wizard.do\_count\_tasks()**

**self.assertIn(' 2 ', str(e.exception))**

如果do\_count\_tasks()方法不会引发异常，则检查将会失败。如果它会引发该异常，检查成功而引发的异常存储在 e 变量。

我们用它来进一步检查它的。异常消息包含任务数，的数量，我们希望有两个。在最后的发言中，我们使用 assertIn 检查的异常文本包含 '2' 字符串。

## 10.6 运行测试

笔试，它是时间来运行它们。为此，我们只需要添加 -测试-启用到 Odoo 服务器选项启动命令，在安装或升级 (-i 或 -u) 插件模块。

该命令将如下所示︰

**$ ./odoo-bin -d todo --test-enable -i todo\_wizard --stop-afterinit**

**--addons-path="..."**

将测试只有模块安装或升级。如果某些依赖关系需要安装，其测试将运行太。如果你想要避免这种情况，您可以安装的模块，以便测试以惯常的方式，然后运行进行升级时测试 (-u) 的模块，以便测试。

## 10.7 关于YAML测试

Odoo 还支持第二种测试，描述了使用 YAML 的数据文件。原来所有测试都使用 YAML，直到最近介绍了基于单元测试测试。虽然两者都支持，及许多核心插件还包括 YAML 测试，官方文档目前不提到 YAML 测试。关于它的最后一个文件是可用在

[https://doc.odoo.com/v6.0/contribute/15\_guidelines/coding\_guidelines\_testing/](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fdoc.odoo.com%2Fv6.0%2Fcontribute%2F15_guidelines%2Fcoding_guidelines_testing%2F)

.

Python 背景的开发人员很可能会更多在家里与单元测试，因为它是一个标准的 Python 功能，而 YAML 测试设计与 Odoo 特定公约。这种趋势显然是倾向于单元测试 YAML，和 YAML 支持可以预计被丢弃在将来版本。

出于这些原因，我们不会做深入的 YAML 测试覆盖率。它仍然可能有用，有一些基本的了解，对他们的工作。

YAML 测试是类似于 CSV 和 XML 的数据文件。事实上 YAML 格式旨在将更紧凑的数据格式，可以用来代替 XML。与 Python 测试，在测试都必须在不同测试 / 子目录，YAML 测试文件可以在任何地方里面的插件模块。但经常他们会内测试 / 或测试 / 子目录。虽然 Python 测试自动发现，YAML 测试必须在中声明和

\_\_manifest\_\_.py清单文件。这做了测试关键，类似于数据我们已经知道关键。

在 Odoo 10 YAML 测试不使用了，但在这里是一个例子，从02\_order\_to\_invoice.yml在point\_of\_sale插件模块︰

-

**I click on the "Make Payment" wizard to pay the PoS order**

**-**

**!record {model: pos.make.payment, id: pos\_make\_payment\_2,**

**context: '{"active\_id": ref("pos\_order\_pos1"), "active\_ids":**

**[ref("pos\_order\_pos1")]}' }:**

**amount: !eval >**

**(450\*2 + 300\*3\*1.05)\*0.95**

**-**

**I click on the validate button to register the payment.**

**-**

**!python {model: pos.make.payment}: |**

**self.check(cr, uid, [ref('pos\_make\_payment\_2')], context=**

**{'active\_id': ref('pos\_order\_pos1')} )**

开头的行!是 YAML 标记，相当于我们在 XML 文件中找到的标记元素。在上述代码中我们可以看到 ！ 记录标记，相当于 XML < 记录 >，和 ！ python 标记，这使我们能够运行 Python 代码在模型上，pos.make.payment 在示例中。

正如你所看到的 YAML 测试使用 Odoo 特定的语法，需要学习。相比较而言，Python 测试使用现有的TransactionCase的单元测试框架，只添加像 Odoo 特定包装类.

## 10.8 开发工具

有开发者应该学习，帮助他们在其工作中的一些技巧。在章 1 , *Odoo 开发入门*，我们已经介绍了用户界面**开发人员模式**。我们也有提供一些开发人员友好的功能服务器选项。我们将会详细描述它更多下一步。在那之后我们将为开发人员讨论另一个相关的话题︰ 如何调试服务器端代码。

### 10.8.1 服务器开发选项

Odoo 服务器提供— — 开发选项，以便加快我们的发展周期，如一些开发人员功能︰

进入调试器时异常插件模块重新加载 Python 代码中自动发现，一旦某个 Python 文件保存，避免手动服务器重新启动

从 XML 文件，避免手动模块升级直接读取视图定义

— — 开发选项接受的逗号分隔列表的选项，虽然大部分时间时，会适用于所有选项。我们还可以指定我们更愿意使用调试器。默认情况下，Python 调试器，使用pdb。有些人可能喜欢安装和使用其他调试程序。在这里另外也支持子午线和 pudb.

**注意**

Odoo 前 10 我们反而— — 调试选项，允许打开插件模块异常调试器。

在 Python 代码上工作时，服务器需要重新启动每次代码更改，以便它重新加载。 — — Dev 命令行选项使那重装︰ 当服务器检测到 Python 文件发生更改，它自动重复服务器加载序列，使代码更改立即有效。

若要使用它只是添加选项— — dev = 所有服务器命令︰

**$ ./odoo-bin -d todo --dev=all**

这项工作看门狗Python 包是必需的并且它应该安装如下所示︰

**$ pip install watchdog**

**注意**

请注意，这是只用于 Python 代码更改和在 XML 文件中的视图体系结构。其他的变化，比如模型的数据结构，模块升级需要的和重新加载是不够的。

### 10.8.2 调试

我们都知道，好一个开发者的工作的一部分是要调试的代码。为此我们经常让使用代码编辑器，可以设置断点并运行我们的程序一步一步。

如果您使用 Microsoft Windows 作为开发工作站，设置环境能够从源运行 Odoo 代码是一个非平凡的任务。也事实上，Odoo 是一个服务器等待客户端调用，才采取行动，使它完全不同地调试相比，客户端程序。

#### 10.8.2.1 Python调试器

虽然它可能看起来有点不知所措，对于新手来说，最务实的做法来调试 Odoo 是使用 Python 集成调试器，pdb。我们亦会向它提供丰富的用户界面介绍扩展、 Ide 通常提供了类似于什么复杂。

若要使用调试器，最佳方法是插入一个断点上我们想要检查，通常一种模型方法的代码。这是通过在所需位置插入以下行︰

**导入 pdb;pdb.set\_trace()**

现在重新启动服务器以便加载修改后的代码。当程序执行到该行， (pdb) Python 提示符将显示在终端窗口中运行服务器，等待我们输入。

**小贴士**

— — 开发选项无需使用手动设置的 Python 调试器断点。

此提示工作作为一个 python 的可以在当前执行上下文中运行的任何表达式或命令。这意味着可以检查，甚至修改当前变量。这些都是最重要的快捷命令可用︰

h（帮助） 将显示可用的 pdb 命令摘要

p（打印） 计算并打印表达式

pp（漂亮打印） 打印将非常有用如字典或列表的数据结构

l（列表） 列出周围要执行下一步指令的代码

n（下） 走到下一个指令s到当前指令的 （步骤） 步骤c（继续） 继续执行正常u（向上） 执行堆栈中向上移动d（向下） 的执行堆栈中下移

Odoo 服务器还支持dev = 所有选项。如果激活，当引发异常服务器进入*验尸*模式在相应的行。这是 pdb提示，如前文所述，允许我们在时刻检查程序状态被发现错误的地方。

虽然pdb优点是可用的--预置，它可以是相当简洁，和几个更舒适的选择存在。

#### 10.8.2.2 示例调试会话

让我们看看一个简单的调试会话的样子。我们可以通过在第一行中添加调试器断点开始do\_populate\_tasks 向导方法︰

**def do\_populate\_tasks(self):**

**import pdb; pdb.set\_trace()**

**self.ensure\_one()**

**# ...**

现在重新启动服务器，打开**待办任务向导**窗体中，并点击**获得全部**按钮。这将触发do\_populate\_tasks 服务器和 web 客户端向导方法会留在**...加载**状态，等待服务器响应。看看服务器运行终端窗口，您会看到与此类似︰

**> /home/daniel/odoo-dev/customaddons/**

**todo\_wizard/models/todo\_wizard\_model.py(54)do\_populate\_t**

**asks()**

**-> self.ensure\_one()**

**(Pdb)**

这是pdb调试器提示和两个第一线给你有关你在哪里在 Python 代码执行的信息。第一行通知文件、 线数和函数名，和第二行是要运行的代码的下一行。

在调试会话期间服务器日志消息可以进来。这些不会损害我们的调试，但他们可以打扰我们。我们可以避免，通过减少冗长的日志消息。大多数情况下这些日志消息将从刀具模块。我们可以让他们使用选项保持沉默 — — --log-handler=werkzeug:CRITICAL.。如果这还不够，我们可以降低一般日志级别，使用--log-level=warn.

如果我们键入h现在，我们将看到可用的命令的快速参考。键入 l 显示当前代码行和周围的代码行。

打字n将运行当前代码行并移到下一步。如果我们只按*enter 键*，将重复上一个命令。这三次也是如此，我们应该在该方法的返回语句。

我们可以检查任何变量，内容如open\_tasks在此方法中使用和打字p open\_tasks或打印 open\_tasks将显示该变量的表示形式。允许使用任何 Python 表达式，即使变量赋值。例如，要显示友好的列表与我们可以使用的任务名称︰

(pdb) p open\_tasks.mapped('name')

运行返回行使用n再一次，我们将显示该函数的返回值。就像这样︰

**--Return--**

**> /home/daniel/odoo-dev/customaddons/**

**todo\_wizard/models/todo\_wizard\_model.py(59)do\_populate\_t**

**asks()->{'res\_id': 14, 'res\_model': 'todo.wizard', 'target':**

**'new', 'type': 'ir.actions.act\_window', ...}**

**-> return self.\_reopen\_form()**

调试会话将继续在调用方的行的代码，但我们可以完成，并继续正常执行键入c.

#### 10.8.2.3 替代Python调试器

虽然pdb的优点是可利用的开箱即用，它可以是相当简洁，和几个更舒适的选择存在。

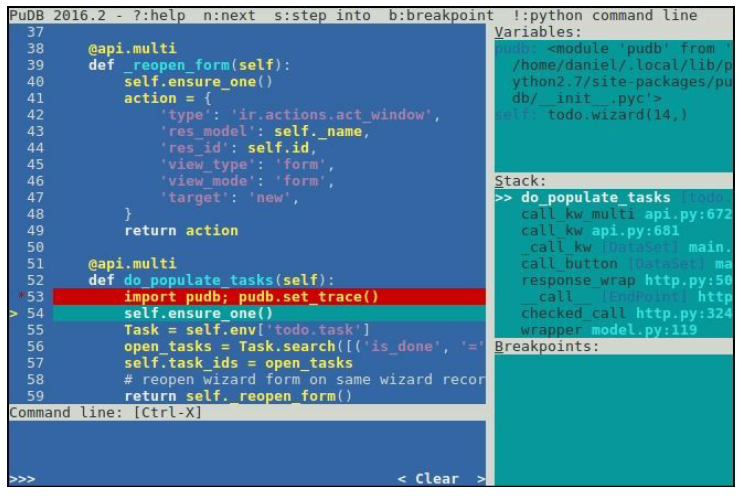
铁 Python 调试器子午线是受欢迎的选择，使用相同的命令pdb，但是添加了改进的 tab 键完成、 语法突出显示、 更舒适的使用情况等。它可以与安装︰

**$ sudo pip install ipdb**

与线添加断点︰

**import ipdb; ipdb.set\_trace()**

另一个替代调试器是pudb。它还支持相同的命令 pdb 和作品在纯文本的终端，但使用类似于你可以找到在 IDE 的调试器中的图形显示。有用的信息，例如在当前上下文和它们的值，变量是现成的屏幕中他们自己的窗口︰



它可以安装系统包管理器或通过pip，如下所示︰

**$ sudo apt-get install python-pudb # using OS packages**

**$ sudo pip install pudb # using pip, possibly in a virtualenv**

添加pudb断点做你希望的样子︰

import pudb; pudb.set\_trace()

#### 10.8.2.4 打印消息和记录

有时我们只被需要检查的一些变量的值或检查是否正在执行一些代码块。Python 打印语句可以做的工作完全没有停下来的执行流程。因为我们在一个终端窗口中运行的服务器，打印的文本将显示在标准输出中。但如果它正在写入一个文件，不会存储到服务器日志。

另一种选择，要时刻牢记是代码的在我们的敏感点设置调试级别的日志消息，如果我们觉得我们可能需要它们调查中已部署的实例的问题。它只会需要提升该服务器日志记录级别，以调试，然后检查日志文件。

### 10.8.3 检查运行过程

也有一些技巧，使我们能够检查正在运行的 Odoo 进程。

为此我们首先需要找到相应的进程 ID (PID)。要找到 PID 运行另一个终端窗口并键入︰

**$ ps ax | grep odoo-bin**

在输出中的第一列是这一进程的 PID。注意到关于进程的 PID 来视察，因为我们将会需要它下一个。

现在我们想要发送一个信号过程。该命令用来做，就是杀。默认情况下它将发送信号要终止的进程，但它也可以发送其他友好的信号。

知道我们正在运行的 Odoo 服务器进程的 PID，我们可以打印的代码目前正在痕迹使用执行︰

**$ kill -3 <PID>**

如果我们看看终端窗口或日志文件服务器的输出被写入的位置，我们将看到信息在多个线程正在运行和详细的堆栈跟踪对它们运行代码的哪一行上。

我们也可以看到使用缓存/内存统计信息转储︰

**$ kill -USR1 <PID>**

## 10.9 第八章 小结

自动化的测试是有价值的实践，既为业务应用程序在一般情况下，并确保代码动态编程语言，如 Python 中的鲁棒性。

我们学会了如何添加和运行插件模块测试的基本原则。我们还讨论了一些技术来帮助我们调试我们的代码。

在下一章中，我们将深入视图层，并将讨论板视图。

# 11 第九章 QWeb和Kanban视图

Qweb是一个Odoo使用的模板引擎。它是基于XML的，用来生成HTML片段和页面。Qweb在7.0版本第一次被引入，使看板视图更丰富，8.0以后的版本，也被用于报表设计和CMS网站页面。 您将了解Qweb语法以及如何使用它来创建自己的看板视图和自定义报告。让我们从看板学习开始。

## 11.1 关于Kanban

看板是一个日语单词，用来表示一个工作队列管理方法。它的灵感来自丰田生产系统和精益生产，其敏捷的工作方法，已经被软件业普遍采纳。 看板是一种工作队列的可视化工具。它的列代表工作流程的各个阶段，卡片则代表工作项，并被放置在适当的列中。新的工作项从看板的最左边的列开始，直到到最右边的列，最右边的列表示已经完成的工作。 看板的简单和可视的特点，使其为简单业务流程提供了非常优秀的支持作用。一个基本的看板的例子有三列，如下图所示：将要做、正在做、已做。 当然，我们可以根据需要进行扩展，以适应特定的流程：

图片来源：“一个建议使用看板的Scrum方法（译者：一种敏捷软件开发方法）” Jeff.lasovski，维基百科授权。

### 11.1.1 Kanban视图

对于许多业务场景，看板可以比典型的工作流引擎更能有效地管理相应的流程。Odoo在提供经典的列表和表单视图的同时，支持看板视图。它很容易实现这种类型的视图。让我们学习如何使用它们。

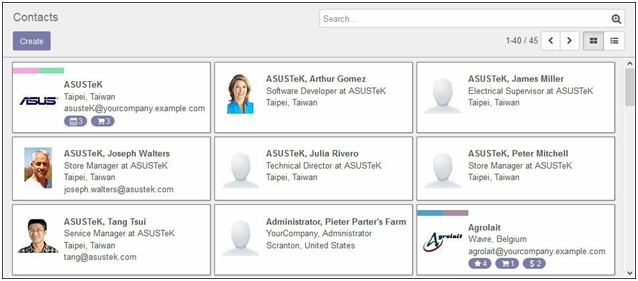
在表单视图中，我们主要使用特定的XML元素，如<field>和 <group>,以及少量的HTML元素，如<h1> 或 <div>。不同的是，看板视图是基于HTML的模板，并且仅支持2个Odoo指定的元素<field> 和<button>。

HTML是使用Qweb模板动态生成的，Qweb引擎处理特殊的XML标记和属性，产生提交给Web客户端的最终的HTML。这带来了很多如何渲染内容的控制，也使视图设计更复杂。

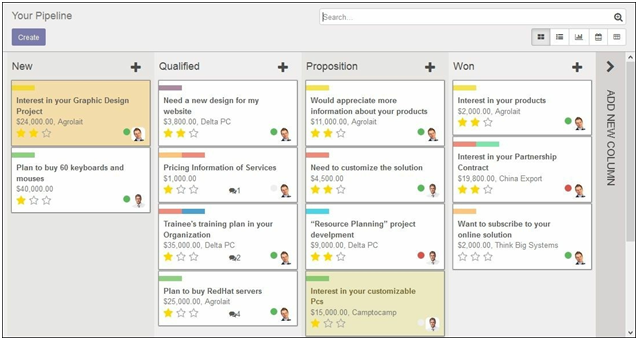
看板视图设计是非常灵活的，我们将尽我们所能给你一个简单的方法来快速建立你的看板视图。一个很好的方法是找到一个跟你的需求类似的已有的看板视图，分析它从而得知如何如何建立你的视图。

有两种不同的方法使用看板视图，一种是卡片列表。它用于诸如联系人、产品、员工通讯录或应用程序的地方。

下图为联系人看板视图：

但这并不是真正的看板。看板更应该是以列的形式加以分组的卡片，当然，看板视图也支持这种布局，例如在销售|我的漏斗或项目任务。

下图为销售|我的漏斗:

两者之间最显著的区别是卡片的看板列组织。这是按照特性进行分组来实现的，类似于列表视图提供的功能。通常是按照阶段这个字段来进行分组。看板视图的一个非常有用的特性是它支持对卡片在列之间进行拖放，从而自动对相应的分组字段进行赋值。

我们可以看到，以上两个例子中的卡片有一些差异。事实上，他们的设计是相当灵活的，并没有一个固定的方式来设计看板卡。这两个例子可以为你的设计提供一个设计参考。

联系人卡片的左侧有一个图像，主区域有一个大的标题，随后是其它的相关信息。我的销售漏斗卡片上，主区域也有一个标题，随后是相关信息列表以及页脚区域，左侧有一个优先级部件，右边是其对于的负责人的用户。这是不可见的图像，当鼠标悬停在他上面时候，卡片有一个选项菜单在右上角时。该菜单允许改变卡的背景颜色。

我们将使用更精细的结构建立我们的看板上的卡片模型

## 11.2 设计Kanban视图

我们将增加看板视图到一个新的插件模块中的待办任务中。如果直接添加到已有的待办任务的界面将是简单的，但是为了更清楚的讲解，我们将使用一个新的模块，从而避免对已经建好的系统文件造成混乱。

我们命名一个新的模块“待办看板”，产生初始化文件，编辑描述文件todo\_kanban/\_\_manifest\_\_.py如下：

**{'name': 'To-Do Kanban', </nowiki>**

**'description': 'Kanban board for to-do tasks.',**

**'author': 'Daniel Reis',**

**'depends': ['todo\_ui'],**

**'data': ['views/todo\_view.xml'] }**

同时增加空的todo\_kanban/\_\_init\_\_.py文件，使Phthon导入目录，满足Odoo模块的需要。

接下来，创建XLM文件，产生我们的新看板视图，并设置为待办任务窗口操作的默认视图。todo\_kanban/views/todo\_view.xml文件,包含下面的代码：

**<?xml version="1.0"?>**

**<odoo>**

**<act\_window id="todo\_app.action\_todo\_task" name="To-Do Tasks" res\_model="todo.task" view\_mode="kanban,tree,form,calendar,graph,pivot" context="{'search\_default\_filter\_my\_tasks': True}" />**

**<!-- Add Kanban view -->**

**<record id="To-do Task Kanban" model="ir.ui.view">**

**<field name="model">todo.task</field>**

**<field name="arch" type="xml">**

**<kanban>**

**<!-- Empty for now, but the Kanban will go here! -->**

**</kanban>**

**</field>**

**</record></odoo>**

在我们有了我们的模块的基本骨架。

在开始看板视图之前，我们需要添加两个字段到待办事项模型中。

### 11.2.1 优先级，Kanban状态和颜色

看板视图需要更多有用的字段:

priority（优先级） 让用户组织他们的工作项目,应该首先解决什么。

kanban\_state（看板状态）指明一个任务是否已经准备好进入下一个阶段或者由于某种原因被阻塞。在模型定义层，两者都是选择字段。在视图层，它们有特定的部件，可以用于窗体和看板视图。

color（颜色）用于存储看板卡显示的颜色，并且可以使用看板视图中的颜色选择器菜单来设置颜色。

为增加这些字段到我们的模块，我们要增加一个文件：models/todo\_task\_model.py

首先，我们需要使它可导入，编辑文件todo\_kanban/\_\_init\_\_.py，导入models子目录: from . import models

然后通过: from . import todo\_task产生 models/\_\_init\_\_.py文件，

现在编辑 models/todo\_task.py 文件:

**from odoo import models,**

**fields class TodoTask(models.Model): \_inherit = 'todo.task'**

**color = fields.Integer('Color Index')**

**priority = fields.Selection( [('0', 'Low'), ('1', 'Normal'), ('2', 'High')], 'Priority', default='1')**

**kanban\_state = fields.Selection([('normal', 'In Progress'), ('blocked', 'Blocked'),('done', 'Ready for next stage')], 'Kanban State', efault='normal')**

现在我们可以在看板视图上工作了。

### 11.2.2 Kanban卡元素

看板视图架构有一个<看板>顶层元素和以下基本结构：

**<kanban default\_group\_by="stage\_id" class="o\_kanban\_small\_column" >**

**<!-- Fields to use in expressions... -->**

**<field name="stage\_id" />**

**<field name="color" />**

**<field name="kanban\_state" />**

**<field name="priority" />**

**<field name="is\_done" />**

**<field name="message\_partner\_ids" /><!-- (...add other used fields). -->**

**<templates>**

**<t t-name="kanban-box">**

**<!-- HTML Qweb template... -->**

**</t>**

**</templates>**

**</kanban>**

注意default\_group\_by =“stage\_id”属性用于<看板>元素。我们使用它，默认情况下，看板卡片按看板板等阶段分组。在简单的列表看板，如联系人，我们不需要这个，只是用一个简单的<kanban>空标记。

<kanban>顶级元素支持几个有趣的属性：

default\_group\_by设置字段使用默认的列分组。

default\_order设置一个默认的看板排序项。

quick\_create =“false”禁用快速创建选项（大加号），在每一列的顶部可以通过只提供标题描述创建新项目。FALSE值是JavaScript文字，必须小写。

class向看板视图的根元素添加CSS类[编辑]

一个相关的类是o\_kanban\_small\_column，使列比默认的更加紧凑。额外的类可以由模块提供自定义CSS。

然后，我们看到模板中使用的字段列表。准确地说，只有专门用于Qweb表达式的字段才需要声明，以确保其数据从服务器获取。

接下来，我们有一个<templates>元素，包含一个或多个Qweb模板来生成HTML片段。我们必须有一个名为kanban-box的模板，这将提交看板卡片。还可以添加其他模板，通常用于定义在主模板中重用的HTML片段。

这些模板使用标准的HTML和Qweb模板语言。Qweb提供了特殊的指令处理，动态生成了最终的HTML。

小窍门[编辑]

Odoo采用推特Bootstrap 3的网站风格的库，所以无论HTML在哪提交，那些样式类一般是可用的。在https://getbootstrap.com您可以了解更多。 现在我们仔细看看在看板视图中使用的Qweb模板。

### 11.2.3 Kanban卡布局

看板卡片的主要内容区域是在kanban-box模板中定义的。此内容区域还可以有页脚子容器。

一个单一的页脚，通过oe\_kanban\_footer CSS类我们将在kanban box的底部使用<div>元素。这个类将自动分割其内部元素适应空间，实现左右对。

打开一个动作菜单的按钮也可以在卡的右上角。作为替代方案，Bootstrap提供pull-left和pullright类，可以用来在卡的任何地方添加左对齐或右对齐元素，包括在oe\_kanban\_footer页脚。

这是我们为我们的看板卡Qweb模板第一次循环：

**<!-- Define the kanban-box template -->**

**<t t-name="kanban-box">**

**<!-- Set the Kanban Card color: -->**

**<div t-attf-class="#{kanban\_color(record.color.raw\_value)} oe\_kanban\_global\_click">**

**<div class="o\_dropdown\_kanban dropdown">**

**<!-- Top-right drop down menu here... -->**

**</div>**

**<div class="oe\_kanban\_content">**

**<div class="oe\_kanban\_footer">**

**<div>**

**<!-- Left hand footer... -->**

**</div>**

**<div>**

**<!-- Right hand footer... -->**

**</div>**

**</div>**

**</div><!-- oe\_kanban\_content -->**

**<div class="oe\_clear"/>**

**</div><!-- kanban color -->**

**</t>**

这就列出了看板卡的总体结构。您可能注意到，color颜色字段正在使用在顶部

元素中，以动态设置卡片的颜色。我们将在下一个章节解释t-attf Qweb指令。

现在让我们来研究主要内容区域，并选择在那里放置什么内容：

**<!-- Content elements and fields go here... -->**

**<div>**

**<field name="tag\_ids" /></div>**

**<div>**

**<strong>**

**<a type="open"><field name="name" /></a>**

**</strong></div>**

**<ul>**

**<li><field name="user\_id" /></li>**

**<li><field name="date\_deadline" /></li></ul>**

大多数模板是常规的HTML，但我们也看到了用于渲染字段值的<field>元素，以及在常规窗体视图按钮中使用的type属性，在此用作<a>锚定标记。 在左边的页脚，我们要插入一个优先级（priority）部件:

**<div>**

**<!-- Left hand footer... -->**

**<field name="priority" widget="priority"/>**

**</div>**

这里我们可看到priority字段被增加，与在表单视图做的一样。

在右边页脚，我们放置看板状态（state）部件和代办任务的所有者的头像：

**<div>**

**<!-- Right hand footer... -->**

**<field name="kanban\_state" widget="kanban\_state\_selection"/>**

**<img t-att- t-att-src="kanban\_image(**

**'res.users', 'image\_small', record.user\_id.raw\_value)" width="24" height="24" class="oe\_kanban\_avatar pull-right" />**

**</div>**

使用<field>元素添加看板状态，就像在常规窗体视图中一样。用户头像图片插入使用HTML<img> 标记。图像的内容是使用Qweb的 t-att-指令动态生成的，我们将在稍后解释。

有时我们希望有一个小的代表性的图像显示在卡上，如同联系人的例子。作为参考，可以通过添加以下内容作为第一个内容元素：

<img t-att-src="kanban\_image( 'res.partner', 'image\_medium', record.id.value)" class="o\_kanban\_image"/>

### 11.2.4 添加Kanban卡选项菜单

看板卡片可以有一个选项菜单，放在右上角。通常的操作是编辑或删除记录，但它有可能通过一个按钮调用的任何动作。我们也有一个小部件设置卡的颜色可用。

以下是在oe\_kanban\_content元件的顶部添加选项菜单的基准HTML代码：

**<div class="o\_dropdown\_kanban dropdown">**

**<!-- Top-right drop down menu here... -->**

**<a class="dropdown-toggle btn" data-toggle="dropdown" href="#">**

**<span class="fa fa-bars fa-lg"/>**

**</a>**

**<ul class="dropdown-menu" role="menu" arialabelledby="dLabel">**

**<!-- Edit and Delete actions, if available: -->**

**<t t-if="widget.editable">**

**<li><a type="edit">Edit</a></li>**

**</t>**

**<t t-if="widget.deletable">**

**<li><a type="delete">Delete</a></li>**

**</t>**

**<!-- Call a server-side Model method: -->**

**<t t-if="!record.is\_done.value">**

**<li><a name="do\_toggle\_done" type="object">Set as**

**Done</a>**

**</li>**

**</t>**

**<!-- Color picker option: -->**

**<li>**

**<ul class="oe\_kanban\_colorpicker" data-field="color"/>**

**</li>**

**</ul></div>**

注意，如果在视图的某处没有<field name="is\_done" />，以上内容不会有效，因为它采用的是一个表达式。如果我们不需要在模板中使用它，我们可以在<templates>元素之前声明它，正如我们在定义<kanban>视图时所做的那样。

下拉菜单是基本的HTML<a> 元素的列表。某些选项，如编辑和删除，只有在满足某些条件时才可用。这用t-if Qweb指令实现。在随后的章节中，我们解释这个和其他的Qweb指令的更多细节。

Widget全局变量表示当前看板记录， JavaScript对象负责当前的看板卡片的渲染。两个特别有用的性质是widget.editable和widget.deletable，用于检测是否动作可用。 我们还可以看到如何根据记录的字段值显示或隐藏选项。只会is\_done字段没有被设置时，Set as Done选项才会显示。

最后一个选项使用color数据字段添加颜色选择特殊部件，用来选择并更改卡片的背景颜色。

### 11.2.5 Kanban视图中的动作

在Qweb模板，<a>链接标记可以有一个类型的属性。它设置了链接将执行的动作类型，这样链接就可以像常规窗体中的按钮一样运行。所以除了<button>元素以外，<a>标记也可用于运行Odoo动作。 在窗体视图中，动作类型可以是动作或对象，也应该附加一个name属性，标识要执行的特定动作。此外，下列动作类型也可用：

open 打开相应的表单视图.

edit 直接以编辑模式打开相应的表单视图.

delete 删除记录，从看板视图移除项.

## 11.3 QWeb模板语言

Qweb解析器查找模板中的特殊指令，并用动态生成的HTML替换。这些指令是XML元素属性，可用于任何有效的标记或元素，如<div>, <span>, 或 <field>.。

有时我们想用一个Qweb指令但不想放在任何在我们的模板的XML元素。在这种情况下，我们有一个特殊的元素<t>，可以有Qweb指令，如t-if或t-foreach，但是不会产生任何的XML /HTML输出。

Qweb指令将经常使用的计算表达式，来产生不同的结果，这些结果取决于当前的记录值。有两种不同的Qweb实现方法：客户端使用JavaScript、服务器端使用Python。

报表和网站页面使用服务器端Python实现，而看板视图使用客户端JavaScript实现。这意味着使用看板视图使用Qweb表达式，它使用JavaScript的语法，而不是Python。

当显示看板视图时，内部步骤大致如下：

获取用于渲染的模板的xml。

调用服务器read()方法得到模板中的字段的数据。

定位kanban-box模板，使用Qweb解析，输出最终的HTML片段。

Inject the HTML in the browser's display (the DOM). 插入在浏览器显示（the DOM）的HTML。

这并不意味着在技术上准确。这只是一个思维导图，去理解看板中的事物是如何工作的。

接下来，我们将了解Qweb表达式计算，探索可用的Qweb指令，使用的例子，增强我们的待办任务看板卡片。

### 11.3.1 QWeb JavaScript评估上下文

许多的Qweb指令使用表达式计算并产生一定的结果。当从客户端使用时，正如看板视图的情况一样，这些表达式是用JavaScript编写的。它们在上下文被计算，上下文有一些有用的变量可用。

一个记录对象可用，表示由服务器请求的字段所组成的记录被渲染。该字段的值可以使用raw\_value或value属性访问：

raw\_value 是服务器read()方法返回的值，所以它更适合用在条件表达式。

value 是根据用户设置进行格式化的值，并用于在用户界面中显示。这通常是与date/datetime and float/monetary字段相关。

Qweb上下文计算也被JavaScript的Web客户端实例引用。为了充分利用它们，我们需要对Web客户端架构有一个很好的理解，但是我们不能详细地理解它。作为参考，以下标识符在Qweb表达式求值是可用的：

widget 是参考当前kanbanrecord()部件对象，负责当前把记录渲染到看板卡片。它暴露了一些有用的辅助功能，我们可以使用。

record 是 widget.records 的快捷方式，并提供对可利用的字段的访问。

read\_only\_mode 显示当前视图是否是只读模式（非编辑模式），它是widget.view.options.read\_only\_mode的快捷方式。

instance是全WEB客户端实例的引用。

值得注意的是，一些字符是不允许内表达式。小写的符号（<）就是这样的情况。这是因为XML标准中，这样的字符具有特殊的含义，不应该被用于XML内容中，非> =是对<的替代，但通常的做法是使用下面的替代符号，可用于不等式操作：

lt 小于

lte 小于等于

gt 大于

gte大于等于

### 11.3.2 使用t-attf进行属性字符串替换

我们的看板卡片使用t-attf 的Qweb指令动态地设置上部<div>元素，卡片的颜色取决于color字段值。为此，使用t-attf –的 Qweb指令。

t-attf指令使用字符串替换动态生成标记属性。这允许动态生成较大的字符串，例如URL地址或CSS类名。

该指令寻找表达式块，这些块将被计算并被计算的结果替换。这些串被模板:And 或 #{ and }分割，块的内容可以是任何有效的JavaScript表达式，可以使用任何可用的Qweb表达式的变量，如record 和widget。

在我们的例子中，我们还使用了kanban\_color() JavaScript函数，专门提供地图颜色索引值给CSS类的颜色名称。

作为一个更详细的例子，我们可以使用这个指令来动态更改截止日期的颜色，以便过期日期以红色显示。

为此，在看板卡片中用一下内容替代

**<field name="date\_deadline"/>:**

**<li t-attf-class="oe\_kanban\_text\_{{ record.date\_deadline.raw\_value and**

**!(record.date\_deadline.raw\_value > (new Date()))**

**? 'red' : 'black' }}">**

**<field name="date\_deadline"/></li>**

结果， class="oe\_kanban\_text\_red" 或者 class="oe\_kanban\_text\_black", 这取决于最终的日期。注意，当 oe\_kanban\_text\_red CSS 类在看板视图可用时, oe\_kanban\_text\_black CSS 类不存在，被用来更好地解释这一点。

提示:

小写符号 <, 不允许在表达式出现，我们选择不大于等于；或者使用&lt来替代

### 11.3.3 使用t-att作为动态属性

t-att - Qweb指令动态计算一个表达式并生成一个属性值。我们的看板卡片使用它来动态地设置<img>标记是一些属性。

Title元素使用t-来动态渲染 。

字段.value返回它表示的应该显示在屏幕上的值，对于多对一字段，这通常是相关记录的name值。对于用户来说，这是用户名。因此，当鼠标指针悬停在图像上时，您将看到相应的用户名。

标记也能动态生成，从而提供对应用户的图像。图像数据由JavaScript辅助函数kanban\_image()提供:

**t-att-src="kanban\_image('res.users', 'image\_small', record.user\_id.raw\_value)"**

函数参数为：读取图像的模型、读取的字段名和记录的id。这里我们使用.raw\_value，得到用户的数据库ID来代替文本。

它不会停在那里，t-att-name和t-attf-name可以用于渲染任何属性，作为生成的属性名称是从NAME后缀获取的。

### 11.3.4 使用t-foreach循环

通过循环可以重复一个HTML块。我们可以用它将任务跟进者的图像添加到任务看板卡中。

让我们先渲染任务的合作伙伴ID：

**<t t-foreach="record.message\_partner\_ids.raw\_value" t-as="rec">**

**<t t-esc="rec" />; </t>**

t-foreach指令接受一个JavaScript表达式，计算一个重复的集合。多数情况下，这将只是一个多对关系字段的名称。它是用一个t-as指令来设定在循环中用到的每项的名字.

下一条使用的t-esc指令根据所所提供的表达式进行计算，本例中只是rec变量，并将在HTML转码时加以渲染。

在前面的例子中，我们按照任务的跟进者来进行循环，并存储在message\_parter\_ids字段。因为看板卡片上有限的空间，我们可以使用JavaScript函数slice()来限定显示的跟进者的数量，如下所示：

**t-foreach="record.message\_partner\_ids.raw\_value.slice(0, 3)"**

rec变量保存每次循环的值，本例中是合作伙伴ID。有了这个，我们可以重写跟随者的循环如下：

**<t t-foreach="record.message\_parter\_ids.raw\_value.slice(0, 3)" t-as="rec">**

**<img t-att-src="kanban\_image('res.partner', 'image\_small', rec)"**

**class="oe\_avatar" width="24" height="24" /></t>**

例如，这可以在右页脚添加下一位负责用户的图像。

一些辅助变量也可用，变量名被冠以t-as变量所定义的前缀。本例中，我们使用rec，所以可用的辅助变量如下：

rec\_index是0开始的循环索引

rec\_size 集合的元素数量

rec\_first 循环的第一个元素为真

rec\_even 偶数索引为真

rec\_odd奇数索引为真

rec\_parity奇数或偶数,根据当前代表被遍历对象rec\_all的索引

rec\_value当遍历一个字典时，{键：值}，保存值（rec保存键值）

例如，我们可以使用以下来避免在我们的ID列表中的逗号：

<t t-foreach="record.message\_parter\_ids.raw\_value.slice(0, 3)" t-as="rec">

<t t-esc="rec" />

<t t-if="!rec\_last">;</t>

</t>

### 11.3.5 使用t-if进行条件渲染

我们的看板视图在卡片选项菜单中使用t-if指令做一些与条件关联的可用的选项。客户端在渲染看板视图时，t-if指令计算JavaScript表达式的值，只有当条件计算为true时，标记和内容才会被渲染。

另一个例子是，为在看板卡片上显示任务工作量估算，仅当它是有值时，在date\_deadline字段后添加以下：

**<t t-if="record.effort\_estimate.raw\_value gt 0">**

**<li>Estimate <field name="effort\_estimate"/></li></t>**

我们用< T t-if =“…”>元素，如果条件为假，元素不产生输出；如果是真的，--则只包含<li>的元素被渲染。注意条件表达式用gt符号代替>，表示大于运算符。

### 11.3.6 使用t-esc和t-raw来渲染值

我们使用<field>元素来渲染字段内容，但字段值也可以直接在没有a <field>标记的情况下直接渲染。

t-esc指令计算表达式并将其作为一个html转码值，如下所示：

<t t-esc="record.message\_parter\_ids.raw\_value" />

在某些情况下，如果源数据的安全，t-raw可以用来渲染字段原值，没有任何转码，如下面的示例所示：

<t t-raw="record.message\_parter\_ids.raw\_value" />

提示:

出于安全原因，尽量避免使用t-raw。它的使用应该被严格限于输出没有任何用户数据的HTML的情况，或者是在用户数据被显式地转码为HTML特殊字符的情况。

### 11.3.7 使用t-set设置变量的值

对于更复杂的逻辑，我们可以将表达式的结果存储到变量中，以便以后在模板中使用它。要做到这点，需要使用t-set指令，命名的变量并通过t-value指令设置计算指定的值表达式。

例如，下面的代码将错过最后期限的渲染为红色，就像在上一节中，但使用red\_or\_black变量为CSS类的使用，如下所示：

**<t t-set="red\_or\_black" t-value=" record.date\_deadline.raw\_value and record.date\_deadline.raw\_value lte (new Date())**

**? 'oe\_kanban\_text\_red' : ''" />**

**<li t-att-class="red\_or\_black">**

**<field name="date\_deadline" />**

**</li>**

变量也可以将HTML内容分配给变量，如下面的示例：

**<t t-set="calendar\_sign">**

**<span class="oe\_e"></span>**

**</t>**

**<t t-raw="calendar\_sign" />**

oe\_e CSS类使用 Entypo pictogram字体。代表日历符号的HTML被存储在变量中，这个变量在模板需要时可以使用它。Font Awesome图标集也非常好用，可能已被使用。

### 11.3.8 使用t-call插入其他模板

Qweb模板可以成为可复用的HTML代码段，可以被插入其他模板。我们不需要反复重复相同的HTML块，就可以设计出复杂的用户界面视图。

可复用的模板定义在<模板>标记并且被顶层元素以t-name 而不是kanban-box标识。这些其他模板可以通过t-call指令包含。在同一个插件模块的某个地方，或在不同的插件。在同一看板视图、同一个插件模块的某个地方中或在不同的插件中，声明的模板是正确的。

跟进者头像列表有时候被隔离到一个可重用片段，让我们可以使用子模板重做它。我们应从增加另外的模板到XML文件中开始, 在 <templates> 元素内部, <t tname="kanban-box"> 节点之后,如下所示:

**<t t-name="follower\_avatars">**

**<div>**

**<t t-foreach="record.message\_parter\_ids.raw\_value.slice(0, 3)"**

**t-as="rec">**

**<img t-att-src="kanban\_image('res.partner',**

**'image\_small', rec)"**

**class="oe\_avatar" width="24" height="24" />**

**</t>**

**</div></t>**

从kanban-box调用主模板很简单。不需要<div>元素包括for each指令，我们应该使用以下内容：

<t t-call="follower\_avatars" />

为调用其他插件模块中定义的模板，我们需要使用module.name全标识符，就像调用其它视图。例如，此段代码可以使用完整标识符来引用todo\_kanban.follower\_avatars. 调用的模板运行在与调用者相同的上下文中，所以在调用者中可用的任何变量名，在处理被调用模板时也可用。

一个更简洁的选择是传递参数到调用模板。这通过设置内部变量t-call标记实现。这些将仅在子模板上下文中进行计算并可用，并且不会在调用者的上下文中存在。

我们可以使用这个获得最大数量的跟进者头像，这个头像呗调用者设置，而无需通过子模板中不易修改的编码。首先，我们需要用变量arg\_max代替固定值3，例如：

**<t t-name="follower\_avatars">**

**<div>**

**<t t-foreach="record.message\_parter\_ids.raw\_value.slice(0, arg\_max)" t-as="rec">**

**<img t-att-src="kanban\_image('res.partner',**

**'image\_small', rec)"**

**class="oe\_avatar" width="24" height="24" />**

**</t>**

**</div>**

**</t>**

**<t t-call="follower\_avatars">**

**<t t-set="arg\_max" t-value="3" /></t>**

然后，在执行子模板调用时定义该变量的值如下：

在t-call元素的整个内容也可通过改变变量值0使其对子模板可用。我们可以定义一个可以在子模板中使用的HTML代码片段，而不是通过参数变量<t traw="0" />.

### 11.3.9 更多的方法来使用t-attf

我们了解了最重要的Qweb指令，但还有一些也是我们应该知道的。我们将对做一个简短的解释。

我们已经看到t-att-name和t-attf-name风格动态标识属性。此外，固定t-att指令也可以使用。它要么接受一个键值字典映射，要么是一对（一个二元素的列）。

使用如下映射：

<p t-att="{'class': 'oe\_bold', 'name': 'test1'}" />

结果如下:

<p class="oe\_bold" name="test1" />

使用如下:

<p t-att="['class', 'oe\_bold']" />

结果如下:

<p class="oe\_bold" />

## 11.4 Kanban视图的继承

使用看板视图和报表中使用的模板扩展了使用其他视图的常规技术，例如使用XPath表达式。更多的细节见第3章，继承-扩展现有的应用程序。

一个常见的情况是使用<field>元素作为选择器，然后在它们之前或之后添加其他元素。在看板视图的情况下，同一字段可以不止一次声明，例如，模板前一次，再在模板内声明一次。这里选择器将匹配第一个字段元素，并且不会在模板中添加我们的修改。

为做到这点，我们需要使用XPath表达式来确保模板内的字段是匹配的。例如：

**<record id="res\_partner\_kanban\_inherit" model="ir.ui.view">**

**<field name="name">Contact Kanban modification</field>**

**<field name="model">res.partner</field>**

**<field name="inherit\_id" ref="base.res\_partner\_kanban\_view"**

**/>**

**<field name="arch" type="xml"><xpath expr="//t[@t-name='kanban- box']//field[@name='display\_name']" position="before"><span>Name:</span>**

**</xpath>**

**</field>**

**</record>**

在上面的例子中，XPath在<t tname="kanban-box">中查找<field name="display\_name">元素。这将排除<templates>节之外的同一字段元素。

对于这些更复杂的XPath表达式，我们可以使用一些命令行工具来查看正确的语法。xmllint命令行实用程序可能已经在你的Linux系统中使用了，它有一个—Xpath选项来执行XML文件的查询。

看提供更好的输出的另一种选择，是XPath命令从libxml-xpath-perl Debian/Ubuntu package:

**$ sudo apt-get install libxml-xpath-perl**

**$ xpath -e "//record[@id='res\_partner\_kanban\_view']" -e**

**"//field[@name='display\_name']]" /path/to/\*.xml**

## 11.5 自定义CSS和JavaScript

正如我们所看到的，看板视图大多是HTML，并且大量使用CSS类。我们已经介绍了一些标识产品提供的常用的CSS类。但是为了达到最佳效果，模块也可以添加自己的CSS。

我们不会在这里详细介绍如何编写CSS代码，但这与解释模块如何添加自己的CSS（和JavaScript）web assets有关。Odoo后台assets在assets\_backend模板声明。为添加我们的模块assets，我们应该扩展该模板。xml文件通常放在视图/模块子目录内。

如下是增加CSS和JavaScript文件到todo\_kanban模块的XML例子，它可能在todo\_kanban/views/todo\_kanban\_assets.xml:

**<?xml version="1.0" encoding="utf-8"?>**

**<odoo>**

**<template id="assets\_backend" inherit\_id="web.assets\_backend" name="Todo Kanban Assets" >**

**<xpath expr="." position="inside">**

**<link rel="stylesheet"**

**href="/todo\_kanban/static/src/css/todo\_kanban.css"/>**

**<script type="text/javascript"**

**src="/todo\_kanban/static/src/js/todo\_kanban.js">**

**</script>**

**</xpath>**

**</template></odoo>**

通常，它应该在\_\_manifest\_\_ PY描述符文件被引用。请注意，assets位于/static/src 子目录。虽然这不是必需的，但它是一个普遍使用的惯例。

## 11.6 第九章 小结

您学习了看板以及如何建立看板视图来实现它。我们还介绍了Qweb模板和如何用它来设计看板卡片。Qweb也是渲染引擎，用于驱动网站的CMS，所以它在Odoo工具集越来越重要。在下一章中，我们将继续使用Qweb，但是在服务器端创建自定义报告

# 12 第十章 创建QWeb报告

报告对管理软件是非常有价值的。自8.0版本开始，内置的 QWeb 报表引擎，是默认报表引擎。利用 QWeb 模板来生成 HTML 文件，然后可以转换为 PDF 表单设计报告。

Odoo 内置报表引擎发生了重大变化。之前的版本 7.0 的报告基于版库和使用特定标记语法，RML。在版本 7.0，Webkit 报告引擎被列入核心，允许报告旨在改为使用普通的 HTML。最后，在这一概念被远一点，8.0 版和 QWeb 模板成为了内置的报告引擎背后的主要概念。

这意味着我们可以方便地利用我们对 QWeb 的了解并运用它来创建业务报告。在这一章，我们将报表添加到我们**的做**的应用程序，并将审查最重要的技术，用 QWeb 报告，包括报告的计算，如合计、 翻译和打印纸张的格式。

但在我们开始之前，我们必须确保我们有安装推荐的版本的实用程序，用于将 HTML 转换成 PDF 文档。

## 12.1 安装wkhtmltopdf

来正确地生成报告，推荐的版本wkhtmltopdf库需要安装。其全称为**Webkit HTML 到 PDF**。Odoo 用于呈现的 HTML 网页转换为 PDF 文档。

的旧版本wkhtmltopdf库已知有问题，如不打印页面页眉和页脚，所以我们需要对要使用的版本挑剔。对于版本 9.0，当时写的推荐的版本是 0.12.1。不幸的是，打包提供内容，如Debian/Ubuntu 或其他版本，并全面。所以我们应该下载并安装包推荐给我们的操作系统和 CPU 的体系结构。下载链接可以发现在[http://wkhtmltopdf.orgo](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=http%3A%2F%2Fwkhtmltopdf.org%2F)r [http://download.gna.org/wkhtmltopdf](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=http%3A%2F%2Fdownload.gna.org%2Fwkhtmltopdf).

我们首先应该肯定，我们不是已经在我们的系统中安装的版本不正确︰

**$ wkhtmltopdf –version**

如果上述报告不是我们想要的版本，我们应该卸载它。在 Debian/Ubuntu 系统上，我们可以使用︰

**$ sudo apt-get remove --purge wkhtmltopdf**

接下来我们需要下载相应的包，我们的系统并安装它。

检查[http://download.gna.org/wkhtmltopdf/0.12/0.12.1](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=http%3A%2F%2Fdownload.gna.org%2Fwkhtmltopdf%2F0.12%2F0.12.1)的正确的文件名称。为 Ubuntu 14.04 LTS （可靠） 64 位，下载命令将会像这样︰

**$ wget**

**http://download.gna.org/wkhtmltopdf/0.12/0.12.1/wkhtmltox-0.12.1\_linux-trusty-amd64.deb -O /tmp/wkhtml.deb**

下一步我们应该安装它。安装一个本地 deb 文件不会自动安装依赖项，因此将需要第二步来做，并完成安装︰

**$ sudo dpkg -i wkhtml.deb**

**$ sudo apt-get -f install**

现在我们可以检查如果wkhtmltopdf库安装正确，并确认它的版本号是我们想要的东西︰

**$ wkhtmltopdf --version**

**wkhtmltopdf 0.12.1 (with patched qt)**

在此之后，Odoo 服务器启动序列不会显示**你需要 Wkhtmltopdf 打印 pdf 版本的报告的**信息消息。

## 12.2 创建业务报告

通常我们会在我们做的应用程序的插件模块执行报告。但出于学习的目的，我们将创建一个新的插件模块只是为了我们的报告。

我们的报告将如下所示︰

我们将命名为这个新的插件模块todo\_report。第一件事是要创建一个空的来文件和 \_\_manifest\_\_.py清单文件︰

**{**

**'name': 'To-Do Report',**

**'description': 'Report for To-Do tasks.',**

**'author': 'Daniel Reis',**

**'depends': ['todo\_kanban'],**

**'data': ['reports/todo\_report.xml'] }**

reports/todo\_report.xml文件可以开始通过声明新的报告，如下所示︰

**<?xml version="1.0"?>**

**<odoo>**

**<report id="action\_todo\_task\_report"**

**string="To-do Tasks"**

**model="todo.task"**

**report\_type="qweb-pdf"**

**name="todo\_report.report\_todo\_task\_template"**

**/>**

**</odoo>**

**<report** >标记是一个快捷方式，将数据写入ir.actions.report.xml模型，是一种特殊的客户端操作。它的数据是可用在**设置**|**技术**|**报告**菜单选项。

**小贴士**

在设计中的报告，你可能更喜欢使用report\_type ="qwebhtml"，当开发按完成后，再改回来到qweb-pdf。这将使它生成速度更快和更容易地检查来自 OWeb 模板的 HTML 结果。

在安装模块之后，待办任务窗体视图将显示**打印**按钮在顶部，左侧的**更多**按钮，包含此选项运行该报表。

它现在还不能正确工作，因为我们没有定义报告。这将是 QWeb 报告，所以它将使用 QWeb 模板。名称属性标识要使用的模板。与其他标识符引用，不同的模块中前缀名称属性是必需的。我们必须使用完整的引用 < 模块名 >.

< identifier\_name >.

## 12.3 QWeb报告模板

以下内容通常是报告的基本的骨架。

这些内容要添加到reports/todo\_report.xml文件，加到< 报告 >元素之后。

**<template id="report\_todo\_task\_template">**

**<t t-call="report.html\_container">**

**<t t-call="report.external\_layout">**

**<div class="page">**

**<!-- Report page content -->**

**</div>**

**</t>**

**</t>**

**</template>**

最重要的元素是t-call指令，表示使用标准报告结构。 Report.html\_container 模板实现支持 HTML 文档的基本设置。 Report.external\_layout 模板处理报表的页眉和页脚，按照相应的公司设置。作为替代，我们可以使用 report.internal\_layout 模板相反，它使用仅包括页面部分。

我们现在实现了模块和报表视图的基本骨架。请注意，由于报告，只是 QWeb 模板继承可以应用，就像在其他视图中。在报表中使用的 QWeb 模板可以扩展常规继承的视图使用**XPATH**表达式。

## 12.4 在报告中提供数据

看板的意见，报告中的 QWeb 模板呈现服务器端，与使用 Python QWeb 实现。我们可以看到这两个

实现相同的规格，还有我们必须要注意的一些差异。

开始时，QWeb 表达式都使用 Python 语法，不 JavaScript。对于最简单的表达式，可能很少或没有差异，但更复杂的操作可能会不同。

计算表达式的方式也是不同的。对于报表，我们有可用的以下变量︰

文档是要打印的记录可迭代集合doc\_ids是要打印的记录的 Id 的列表doc\_model标识的记录模型todo.task举个例子

时间是对 Python 的时间库的引用用户是用于运行报表的用户记录res\_company是为当前用户的公司记录

的报告内容写在 HTML 中，可以使用引用字段值t-字段属性，然后它可以配合t-fieldoptions属性，以使用特定的小部件来呈现的字段内容。

现在我们可以开始设计我们的报告的页面内容︰

**<!-- Report page content**

**<div class="row bg-primary">**

**<div class="col-xs-3">**

**<span class="glyphicon glyphicon-pushpin" />**

**What**

**</div>**

**<div class="col-xs-2">Who</div>**

**<div class="col-xs-1">When</div>**

**<div class="col-xs-3">Where</div>**

**<div class="col-xs-3">Watchers</div>**

**</div>**

**<t t-foreach="docs" t-as="o">**

**<div class="row">**

**<!-- Data Row Content -->**

**</div>**

**</t>**

内容的布局可以使用 Twitter 引导 HTML 网格系统。总括来说，引导有 12 列网格布局。可以使用添加一个新行 < div 类 ="row">。内部行，我们有细胞，每家都虽然有一定数量的列，应该负起 12 列。每个单元格可以用行定义 < div 类 ="col-xs-N">，其中 N 是它跨越的列数。

**注意**

完整的参考，引导，描述这些和其他样式元素，可以发现在[http://getbootstrap.com](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=http%3A%2F%2Fgetbootstrap.com%2Fcss%2F).

在这里我们要添加带有标题的标题行，然后我们有t-foreach循环，遍历每个记录，并用于每个渲染行。

因为渲染都是服务器端，记录对象，我们可以使用点表示法访问字段从相关的数据记录。这使得它易于遵循通过相关的字段，访问他们的数据。请注意，这是不可能在客户端呈现 Qweb，视图，例如 web 客户端看板视图。

这是 XML 内容的记录的行︰

**<div class="col-xs-3">**

**<h4><span t-field="o.name" /></h4>**

**</div>**

**<div class="col-xs-2">**

**<span t-field="o.user\_id" />**

**</div>**

**<div class="col-xs-1">**

**<span t-field="o.date\_deadline" />**

**<span t-field="o.amount\_cost"**

**t-field-options='{**

**"widget": "monetary",**

**"display\_currency": "o.currency\_id"}'/>**

**</div>**

**<div class="col-xs-3">**

**<div t-field="res\_company.partner\_id"**

**t-field-options='{**

**"widget": "contact",**

**"fields": ["address", "name", "phone", "fax"],**

**"no\_marker": true}' />**

**</div>**

**<div class="col-xs-3">**

**<!-- Render followers -->**

**</div>**

正如我们所看到的可以与其他选项一起使用字段。这些都是非常类似于选项属性在窗体视图上使用，如所示章 6 、*意见-设计的用户界面*，使用与其他小部件设置要用于呈现字段的构件.

一个例子是货币的小部件，上面使用，旁边的截止日期。

是一个更复杂的例子联系小部件，用于设置地址格式。我们使用过该公司的地址，res\_company.partner\_id，因为它有一些默认数据，我们可以立即看到呈现的地址。但它会更有意义，为使用分配的用户地址， o.user\_id.partner\_id。默认情况下联系小部件显示地址与一些象形图，如电话图标。 No\_marker ="true"我们使用的选项禁用它们。

## 12.5 渲染图像

我们的报告的最后一列将功能追随者，与角色的列表。我们将使用介质列表引导组件和遍历的追随者来呈现每个之一︰

**<!-- Render followers -->**

**<ul class="media-list">**

**<t t-foreach="o.message\_follower\_ids" t-as="f">**

**<li t-if="f.partner\_id.image\_small"**

**class="media-left">**

**<img class="media-object"**

**t-att-src="'data:image/png;base64,%s' %**

**f.partner\_id.image\_small"**

**style="max-height: 24px;" />**

**<span class="media-body"**

**t-field="f.partner\_id.name" />**

**</li>**

**</t>**

**</ul>**

中提供的二进制字段内容base64表示。 < Img > 元素可以直接接受这种类型的数据为的src属性。因此我们可以使用 t att src QWeb 的指令，以动态生成的图像。

## 12.6 总计摘要和运行总计

在报告中一个常见的需要是提供总计。这可以使用 Python 表达式来计算这些汇总。

后的结束标记< t t-> foreach，我们将添加最后一排，总计︰

**<!-- Totals -->**

**<div class="row">**

**<div class="col-xs-3">**

**Count: <t t-esc="len(docs)" />**

**</div>**

**<div class="col-xs-2" />**

**<div class="col-xs-1">**

**Total:**

**<t t-esc="sum([o.amount\_cost for o in docs])" />**

**</div>**

**<div class="col-xs-3" />**

**<div class="col-xs-3" />**

**</div>**

len()Python 语句用于计算集合中的元素数。可以使用计算总计 sum （) 值对的值的列表。在前面的示例中，我们使用列表理解产生出的值的列表文档记录集。你能想到的链表像嵌入式为循环。

有时我们想要执行某些计算，如我们去与报告一起发送。例如，运行总和，与总数达到当前记录。这可以实现与 t 套积累的变量，定义，然后更新它在每一行上。

为了说明这一点，我们可以计算追随者的累积的数目。我们应该开始通过初始化变量，只是之前 t foreach 在循环文档记录集，使用︰

**<t t-set="follower\_count" t-value="0" />**

然后，在循环中，添加的记录数目的追随者给该变量。我们会选择做这项权利提出的追随者，清单后，也将打印出当前共在每行上︰

**<!-- Running total-->**

**<t t-set="follower\_count" t-value="follower\_count + len(o.message\_follower\_ids)" />**

**Accumulated # <t t-esc="follower\_count" />**

## 12.7 定义纸张格式

在这一点上我们的报告看起来好在 HTML 中，但它在 PDF 页面上并不很好地打印。我们可能会得到一些更好的结果，使用横向页面。所以我们需要添加此文件格式。

在 XML 文件的顶部，添加此记录︰

**<record id="paperformat\_euro\_landscape"**

**model="report.paperformat">**

**<field name="name">European A4 Landscape</field>**

**<field name="default" eval="True" />**

**<field name="format">A4</field>**

**<field name="page\_height">0</field>**

**<field name="page\_width">0</field>**

**<field name="orientation">Landscape</field>**

**<field name="margin\_top">40</field>**

**<field name="margin\_bottom">23</field>**

**<field name="margin\_left">7</field>**

**<field name="margin\_right">7</field>**

**<field name="header\_line" eval="False" />**

**<field name="header\_spacing">35</field>**

**<field name="dpi">90</field>**

**</record>**

它是定义在的欧洲 A4 格式副本插件，报表，数据, report\_paperformat.xml文件，但将方向从纵向更改为景观。从 web 客户端通过**设置**菜单中，可以看出定义的纸张格式 |**技术**|**报告**|**纸张格式**。

现在我们可以使用它在我们的报告。默认纸张格式定义在公司设置中，但我们还可以指定要使用的特定报表的纸张格式。那就使用 paperfomat 属性在报告行动。

让我们编辑操作用来打开我们的报告，以便添加此属性︰

**<report id="action\_todo\_task\_report"**

**string="To-do Tasks"**

**model="todo.task"**

**report\_type="qweb-pdf"**

**name="todo\_report.report\_todo\_task\_template"**

**paperformat="paperformat\_euro\_landscape"**

**/>**

**小贴士**

paperformat属性< 报告 >在 9.0 版中添加了标记。为

我们需要使用的 8.0< 记录 >元素添加报告操作与paperformat值。

## 12.8 在报告中启用语言翻译

要启用报告的翻译，它需要从模板调用使用< t t-电话 >具有元素t-郎属性。

t-郎属性应求值为一种语言代码如es或en\_US。它需要哪里可以找到要使用的语言的字段的名称。这经常会语言的文档是发送到，存储在合作伙伴 partner\_id.lang 字段。在本例中，我们没有一个伙伴的领域，但我们可以用负责任的用户，和相应的语言首选项是在 user\_id.lang.

功能预计模板的名称，将呈现和翻译它。这意味着，我们需要定义页面内容，我们在报告中单独的模板，如以下所示︰

**<report id="action\_todo\_task\_report\_translated"**

**string="Translated To-do Tasks"**

**model="todo.task"**

**report\_type="qweb-pdf"**

**name="todo\_report.report\_todo\_task\_translated"**

**paperformat="paperformat\_euro\_landscape"**

**/>**

**<template id="report\_todo\_task\_translated">**

**<t t-call="todo\_report.report\_todo\_task\_template"**

**t-lang="user.lang" >**

**<t t-set="docs"**

**t-value="docs" />**

**</t>**

**</t>**

**</template>**

## 12.9 基于自定义SQL的报表

我们建立的报告基于定期记录集。但在某些情况下我们需要变换或聚合数据，处理数据的飞行，如同时呈现报表时并非易事。

这一种方法是编写一个 SQL 查询以生成数据集我们需要揭露这些结果通过一个特殊的模型，并有我们基于记录集的报告工作。

为此，我们将创建reports/todo\_task\_report.py文件与此代码︰

**# -\*- coding: utf-8 -\*-**

**from odoo import models, fields**

**class TodoReport(models.Model):**

**\_name = 'todo.task.report'**

**\_description = 'To-do Report'**

**\_sql = """**

**CREATE OR REPLACE VIEW todo\_task\_report AS**

**SELECT \***

**FROM todo\_task**

**WHERE active = True**

**"""**

**name = fields.Char('Description')**

**is\_done = fields.Boolean('Done?')**

**active = fields.Boolean('Active?')**

**user\_id = fields.Many2one('res.users', 'Responsible')**

**date\_deadline = fields.Date('Deadline')**

对于要加载此文件，我们需要添加from . import reports行到顶部\_\_init\_\_.py文件，和from . import todo\_task\_report至reports/\_\_init\_\_.py文件。

sql属性用于重写为此提供的 SQL 的数据库表自动创建。我们希望它来创建一个数据库视图提供报表所需的数据。我们的 SQL 查询是很简单，但问题是，我们可以使用任何有效的 SQL 查询我们的观点。

我们也映射我们需要与 ORM 字段类型的字段，以便他们可以在记录集中在此模型上生成。

接下来我们可以添加一份新的报告，基于此模型中，

reports/todo\_model\_report.xml:

**<odoo>**

**<report id="action\_todo\_model\_report"**

**string="To-do Special Report"**

**model="todo.task"**

**report\_type="qweb-html"**

**name="todo\_report.report\_todo\_task\_special"**

**/>**

**<template id="report\_todo\_task\_special">**

**<t t-call="report.html\_container">**

**<t t-call="report.external\_layout">**

**<div class="page">**

**<!-- Report page content -->**

**<table class="table table-striped">**

**<tr>**

**<th>Title</th>**

**<th>Owner</th>**

**<th>Deadline</th>**

**</tr>**

**<t t-foreach="docs" t-as="o">**

**<tr>**

**<td class="col-xs-6">**

**<span t-field="o.name" />**

**</td>**

**<td class="col-xs-3">**

**<span t-field="o.user\_id" />**

**</td>**

**<td class="col-xs-3">**

**<span t-field="o.date\_deadline" />**

**</td>**

**</tr>**

**</t>**

**</table>**

**</div>**

**</t>**

**</t>**

**</template>**

**</odoo>**

对于更复杂的情况下，我们可以使用一个不同的解决方案︰ 向导。为此我们应该有相关的线，在那里头包括报表参数，介绍了由用户和行都将具有所生成的数据用于报告所创建的瞬态模型。可以包含我们可能需要的任何逻辑的模型法生成这些行。强烈建议从现有的类似报告那里获得启示。

## 12.10 第十章 小结

在前一章我们了解 QWeb，以及如何使用它来设计一个看板视图。在这一章我们了解 QWeb 报表引擎和最重要的技术来生成报表与 QWeb 模板语言。

在下一章中，我们将继续与 QWeb，这一次来构建网站页面。我们还将学习写 web 控制器，提供更丰富的功能，到我们的 web 页。

# 13 第十一章 创建网站前端功能

Odoo 开始作为一个后端系统，但很快就感到需要一个前端界面。早期的门户功能，基于相同的接口后端，不是非常灵活，也不是移动设备友好。

为了解决这一差距，版本 8 介绍新网站功能，添加一个**内容管理系统**(**CMS**) 的产品。这将使我们能够建立美丽和有效的前端，而无需集成第三方的CMS。

在这里我们将学习如何开发我们自己的前端面向插件模块，利用由 Odoo 提供的网站功能。

## 13.1 路线图

我们将创建一个网站的页面清单我们待办任务，使我们能够导航到详细页面的每个现有的任务。我们也想要能提出新待办任务通过 web 窗体。

这一点，我们将能够覆盖网站发展的关键技术︰ 创建动态页，将参数传递给另一个页面，创建窗体和处理它们的有效性和计算逻辑。

但首先，我们将介绍一个非常简单的**Hello World**网页的基本网站概念。

## 13.2 我们的第一个网页

我们将为我们的网站功能创建插件模块。我们可以叫它 todo\_website。介绍 Odoo web 开发的基本知识，我们将实现一个简单的**Hello World** web 页面。富有想象力，是吧？

像往常一样，我们将开始创建它是清单文件。创建 todo\_website/\_\_manifest\_\_.py 与文件︰

**{**

**'name': 'To-Do Website',**

**'description': 'To-Do Tasks Website',**

**'author': 'Daniel Reis',**

**'depends': ['todo\_kanban']}**

我们正在建设的顶部todo\_kanban插件模块，以便我们有所有可用的功能添加到整本书待办任务模型。

注意到这一权利，现在我们不取决于网站插件模块。而网站提供了一个有用的框架，建立充分的特色的网站，功能内置的核心框架的基本 web。让我们来探索它们。

### 13.2.1 Hello World！

若要提供我们第一个 web 页，我们将添加一个控制器对象。首先，我们可以有其与模块导入的文件︰

首先添加todo\_website/\_\_init\_\_.py文件与下面的行︰

from . import controllers

，然后添加todo\_website/controllers/\_\_init\_\_.py文件与下面的行︰

from . import main

现在，添加控制器，实际的文件todo\_website/controllers/main.py，用下面的代码︰

# -\*- coding: utf-8 -\*-

from odoo import http

class Todo(http.Controller):

@http.route('/helloworld', auth='public')

def hello\_world(self):

return('<h1>Hello World!</h1>')

odoo.http模块提供 Odoo web 相关的功能。我们的控制器，负责页面呈现，应该是从继承的对象odoo.http.Controller 类。用于类的实际名称并不重要。在这里我们选择了使用主.

在控制器类中我们有相匹配的路线的方法，进行一些处理，然后返回的结果;要向用户显示的页。

odoo.http.route装饰是什么将方法绑定到 URL 路由。我们的例子使用 /hello 路由。导航到http://localhost:8069/你好和你将迎接**世界你好**消息。在此示例中执行该方法的处理是非常简单︰ 它只是返回一个文本字符串与**Hello World**消息的 HTML 标记。

你可能已经注意到，我们添加了auth = '公开'参数到工艺路线。这被需要页后，可以对非身份验证的用户可用。如果我们删除它，只有经过身份验证的用户才可以看到该页面。如果没有会话处于活动状态，将相反显示登录屏幕。

### 13.2.2 Qweb模板的Hello World！

使用 Python 字符串生成 HTML 将很快就变得无聊。QWeb 模板做得更好在那。因此，让我们改善我们**Hello World** web 页后，可以使用模板来代替。

QWeb 模板添加通过 XML 数据文件，并从技术上讲，他们是一种视图，与窗体或树视图。他们实际上存储在相同的模型中， ir.ui.view.

要加载的数据文件必须是像往常一样，在清单文件中声明，因此编辑todo\_website/\_\_manifest\_\_.py文件来添加密钥︰

**'data': ['views/todo\_templates.xml'],**

然后添加实际的数据文件，views/todo\_web.xml，使用以下内容︰

**<odoo>**

**<template id="hello" name="Hello Template">**

**<h1>Hello World !</h1>**

**</template>**

**</odoo>**

**注意**

< 模板 >元素是实际上的快捷方式声明<record>为ir.ui.view模型使用type="qweb"，和<t> 模板内它。

现在我们需要我们的控制器方法使用此模板︰

**from odoo.http import request**

**# ...**

**@http.route('/hello', auth='public')**

**def hello(self, \*\*kwargs):**

**return request.render('todo\_website.hello')**

模板渲染是由提供的。请求通过其render()功能。

**小贴士**

请注意，我们添加了\* \* 调用对方法参数。如果任何额外的参数提供的 HTTP 请求，如查询字符串或邮政参数，可以通过捕获调用本词典。这使我们的方法更可靠，因为提供了意外的参数不会导致其错误。

### 13.2.3 扩展Web功能

可扩展性是我们期望的所有功能的 Odoo，和 web 功能也不例外。事实上，我们可以扩展现有的控制器和模板。作为一个例子，我们将延长我们的**Hello World** web 页，它使用参数表示要打招呼的名称︰ 使用 URL /你好？ 名称 = 约翰将返回**你好约翰 ！**问候。

扩展通常是从一个不同的插件模块，但它也内相同的插件。为了保持简洁，我们会做而无需创建一个新的插件模块。

让我们添加一个新的todo\_website/controllers/extend.py文件用下面的代码︰

**# -\*- coding: utf-8 -\*-**

**from odoo import http**

**from odoo.addons.todo\_website.controllers.main import Todo**

**class TodoExtended(Todo):**

**@http.route()**

**def hello(self, name=None, \*\*kwargs):**

**response = super(TodoExtended, self).hello()**

**response.qcontext['name'] = name**

**return response**

在这里我们可以看到我们需要做延长控制器。

首先，我们使用 Python导入以获取我们想要扩展的控制器类的引用。他们有一个中央登记处，所提供的模型相比， env 对象时，可以获得对任何模型的类的引用，而不需要知道的模块和执行这些文件。与控制器并不是那样，和需要的模块和文件实现控制器我们要知道延长。

下一步我们需要 （重新） 定义来自控制器正在扩展的方法。它需要至少用装饰简单的 @http.route()为其路线来保持活跃。另外，我们可以提供的参数 route()，然后我们将替换，重新定义其路线。

扩展hello （)方法现在已名称参数。参数可以获得它们的值，从路由 URL 段、 从查询字符串参数，或从邮政参数。在这种情况下，因为我们处理和路线已经没有可萃取的变量 （我们会表明，在一个时刻），将从 URL 查询字符串中提取得到请求，不后，名称参数的值。测试的 URL 可能是

http://localhost:8069/hello?name=John.

内hello （)方法我们运行继承的方法来获得其响应，然后再进行修改，根据我们的需要。控制器方法的常见模式是为他们用一个语句来呈现模板来结束。在我们的例子︰

return request.render('todo\_website.hello')

这将生成http。响应对象，但实际呈现推迟到调度端。

，这意味着，继承方法仍然可以更改 QWeb 模板和上下文用于渲染。我们可以更改模板修改response.template，但我们不会需要它。我们宁愿要修改 response.qcontext 添加名称呈现上下文的关键。

别忘了添加到新的 Python 文件

todo\_website/controllers/\_\_init\_\_.py:

**from . import main**

**from . import extend**

现在我们需要修改 QWeb 模板，以便让它利用这额外的信息。添加todo/website/views/todo\_extend.xml:

**<odoo>**

**<template id="hello\_extended"**

**name="Extended Hello World"**

**inherit\_id="todo\_website.hello">**

**<xpath expr="//h1" position="replace">**

**<h1>**

**Hello <t t-esc="name or 'Someone'" />!**

**</h1>**

**</xpath>**

**</template>**

**</odoo>**

网页模板是 XML 文档，就像其他 Odoo 视图类型，和我们可以使用xpath定位的元素，然后操纵它们，就像我们可能与其他视图类型。继承的模板确定在<template>元由 inherited\_id 属性。

我们不应该忘了申报在我们的插件清单，此附加的数据文件

todo\_website/\_\_manifest\_\_.py:

**'data': [**

**'views/todo\_web.xml',**

**'views/todo\_extend.xml'],**

在此之后，访问http://localhost:8069/你好？ 名称 = 约翰应表明我们**你好，约翰 ！**消息。

我们还可以提供参数通过 URL 的细分。例如，我们可以获得完全相同的结果，从http://localhost:8069/hello?name=John使用此替代实现 URL:

**class TodoExtended(Todo):**

**@http.route(['/hello', '/hello/<name>])**

**def hello(self, name=None, \*\*kwargs):**

**response = super(TodoExtended, self).hello()**

**response.qcontext['name'] = name**

**return response**

正如你所看到的路线可以包含**占位符**对应参数并将其提取出来，然后传递给该方法。占位符，也可以指定一个转换器来实现特定类型映射。例如， < int:user\_id > 将提取 user\_id 作为一个整数值的参数。

转换器是一种提供的一个功能刀具使用 Odoo 和大多数的可用的库，可以发现在刀具库的文档，网址[http://werkzeug.pocoo.org/docs/routing/.](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=http%3A%2F%2Fwerkzeug.pocoo.org%2Fdocs%2Frouting%2F)

Odoo 添加特定和特别有用的转换器︰ 提取模型记录。例如 @http.route('/hello/<model("res.users"):user>) 提取用户参数作为记录的对象上为 res.users 模型。

### 13.2.4 HelloCMS！

让我们使这更有趣的是，和创建我们自己简单的 CMS。为此我们可以期望在 URL 中的模板名称 （页），然后只是使它的路线。然后，我们可以动态地创建 web 页面，并让他们由我们的 CMS。

原来，这是很容易做到︰

@http.route('/hellocms/<page>', auth='public')

def hello(self, page, \*\*kwargs):

return http.request.render(page)

现在，打开http://localhost:8069/hellocms/todo\_website.hello在您的 web 浏览器，你会看到我们的**Hello World** web 页 ！

内置网站事实上，提供 CMS 功能包括一个更可靠的实现以上所述，在/ 页端点路线。

**注意**

在刀具术语的端点是别名的路线，和由其静态的部分 （不包括占位符）。例如我们简单 CMS，终点是/hellocms.

大部分时间我们想我们要集成到 Odoo 网站的页面。因此本章的其余部分为我们的示例，我们将一道与网站插件。

## 13.3 建设网站

由前面的例子给出的页面并没有集成到 Odoo 网站︰ 我们有没有页面页脚、 菜单，等等。Odoo 网站插件模块方便地提供所有这些功能，我们不必担心他们自己。

要使用它，我们应该开始通过安装网站在我们工作中的插件模块实例，并将其作为依赖关系添加到我们的模块。 \_\_Manifest\_\_.py 关键取决于应该看起来像这样︰

'depends': ['todo\_kanban', **'website'**],

若要使用本网站，我们还需要修改控制器和模板。

控制器需要额外网站 = True参数路线上︰

**@http.route('/hello', auth='public', website=True)**

**def hello(self, \*\*kwargs):**

**return request.render('todo\_website.hello')**

模板需要插入到的网站的总体布局︰

**<template id="hello" name="Hello World">**

**<t t-call="website.layout">**

**<h1>Hello World!</h1>**

**</t>**

**</template>**

与此， **Hello World ！**现在我们以前使用的示例 Odoo 网站内页所示。

### 13.3.1 添加CSS和JavaScript

我们网站的页面可能需要一些额外的 CSS 或 JavaScript 的资产。所以我们需要一种方式来告诉它也使用我们的文件，由网站管理这方面的 web 页。

我们将添加一些 css 代码添加简单的删除线效果为完成的任务。为此，创建todo\_website/static/src/css/index.css 与此内容的文件︰

**.todo-app-done {**

**text-decoration: line-through;**

**}**

下一步我们需要把它包括在网站的页面。这通过将它们在添加 website.assets\_frontend 模板负责加载网站特定资产。编辑 todo\_website/views/todo\_templates.xml 数据文件，来扩展该模板︰

**<odoo>**

**<template id="assets\_frontend"**

**name="todo\_website\_assets"**

**inherit\_id="website.assets\_frontend">**

**<xpath expr="." position="inside">**

**<link rel="stylesheet" type="text/css"**

**href="/todo\_website/static/src/css/index.css"/>**

**</xpath>**

**</template>**

**</odoo>**

我们很快将使用这项新todo 应用程序完成样式类。当然，JavaScript 资产还可以使用类似的方法。

### 13.3.2 待办事项列表控制器

现在，我们经历了基本知识，让我们努力在我们的 Todo 任务列表。我们将有 /todo 向我们展示的 Todo 任务列表的 web 页的 URL。

为此，我们需要准备数据，目前，一个控制器方法和 QWeb 模板，以向用户显示该列表。

编辑todo\_website/controllers/main.py文件，添加此方法︰

**#class Main(http.Controller):**

**@http.route('/todo', auth='user' , website=True)**

**def index(self, \*\*kwargs):**

**TodoTask = request.env['todo.task']**

**tasks = TodoTask.search([])**

**return request.render(**

**'todo\_website.index', {'tasks': tasks})**

控制器检索要使用的数据，并使其可用于呈现模板。在这种情况下控制器需要身份验证的会话，因为其路由 auth = '用户' 属性。即使这是默认值，它是明确说明用户会话是需要很好的做法。

与此，Todo 任务search （)语句将运行与当前会话用户。

公共用户可以访问的数据是非常有限的使用该类型的路线时，我们经常需要使用sudo()提升访问并使页面数据可用，否则将无法访问。

这可还会有安全风险，所以要小心的输入参数的验证和执行的行动。此外保持 sudo() 记录集使用限制到最低的操作可能。

request.render()方法预计的 QWeb 模板渲染中和一本字典，可用模板评价的上下文标识符。

### 13.3.3 待办事项列表模板

应由一个数据文件，添加 QWeb 模板，我们可以将其添加到现有的todo\_website/views/todo\_templates.xml数据文件︰

**<template id="index" name="Todo List">**

**<t t-call="website.layout">**

**<div id="wrap" class="container">**

**<h1>Todo Tasks</h1>**

**<!-- List of Tasks -->**

**<t t-foreach="tasks" t-as="task">**

**<div class="row">**

**<input type="checkbox" disabled="True"**

**t-att-checked=" 'checked' if task.is\_done else {}"**

**/>**

**<a t-attf-href="/todo/{{slug(task)}}">**

**<span t-field="task.name"**

**t-att-class="'todo-app-done' if task.is\_done**

**else ''" />**

**</a>**

**</div>**

**</t>**

**<!-- Add a new Task -->**

**<div class="row">**

**<a href="/todo/add" class="btn btn-primary btn-lg">**

**Add**

**</a>**

**</div>**

**</div>**

**</t>**

**</template>**

前面的代码中使用t-foreach指令，以便呈现的任务列表。 T att 指令用于输入复选框允许我们添加，或不， 检查属性取决于 is\_done 价值。

我们有一个复选框，输入，和想要检查如果任务完成。在 HTML 中，根据它具有或不选中复选框检查属性。为此，我们使用 t-att-名称指令的动态渲染检查属性根据表达式。在这种情况下，表达式计算结果为无，QWeb 将省略的属性，对于这种情况下方便。

的任务名称，在呈现时t-attf指令用来动态创建的 URL 打开详细信息窗体为每个特定的任务。我们使用的特殊功能 slug() 来生成每个记录人类可读的 URL。该链接不会现在工作的因为我们还将创建相应的控制器.

对每个任务我们也使用t-att指令来设置todo 应用程序完成样式只完成的任务。

最后，我们有一个**Add**按钮以使用窗体来创建新的 Todo 任务打开网页。我们将用它来介绍下一步处理的 web 窗体。

### 13.3.4 待办事宜任务详细信息页面

每个 Todo 列表中的项是指向详细页的链接。我们应该执行这些链接，控制器和 QWeb 模板的演示文稿。在这一点上，这应该是一项简单的工作。

在todo\_website/controllers/main.py文件添加方法︰

**#class Main(http.Controller):**

**@http.route('/todo/<model("todo.task"):task>',**

**website=True)**

**def index(self, task, \*\*kwargs):**

**return http.request.render(**

**'todo\_website.detail',**

**{'task': task})**

注意，路线使用带有占位符model("todo.task")转换器，将映射到任务变量。它捕获任务标识符从 URL，一个简单的 ID 号或蛞蝓表示形式，并将它转换成相应的浏览记录对象。

和 QWeb 模板，添加下列代码为todo\_website/views/todo\_web.xml数据文件︰

**<template id="detail" name="Todo Task Detail">**

**<t t-call="website.layout">**

**<div id="wrap" class="container">**

**<h1 t-field="task.name" />**

**<p>Responsible: <span t-field="task.user\_id" /></p>**

**<p>Deadline: <span t-field="task.date\_deadline" /></p>**

**</div>**

**</t>**

**</template>**

在这里值得注意的是使用的< t t 领域 >元。它处理的适当的表示形式的字段值，就像在后端。它正确地提出了日期值和多对一的值，例如。

## 13.4 网站表单

窗体是一个共同的特点，在网站上找到。我们已经实现一个所需的所有工具︰ QWeb 模板可以为窗体提供的 HTML，相应提交行动可以是一个 URL，由一个控制器，可以运行所有的验证逻辑，并最后将数据存储在模型中适当处理。

但对于非平凡的形式，这可能是项艰巨的任务。它不是那么简单，执行所有所需的验证，并向有关什么是错误的用户提供反馈。

因为这是一个共同的需要，website\_form插件是可用来帮助我们与此。让我们看看如何使用它。

回望 Todo 任务列表中的**添加**按钮，我们可以看到它打开/todo/addURL。这将会呈现窗体提交一个新的 Todo 任务，和可用的字段将任务名称，负责任务和文件附件的人 （用户）。

我们应该开始通过添加website\_form依赖于我们的插件模块。我们可以替换网站，因为保持它显式地将冗余。对 todo\_website/\_\_manifest\_\_.py 编辑取决于关键字︰

'depends': ['todo\_kanban', **'website\_form'**],

现在我们将添加包含表单的页面。

### 13.4.1 表单页面

我们可以开始通过执行控制器方法以支持的形式呈现，todo\_website/controllers/main.py文件︰

**@http.route('/todo/add', website=True)**

**def add(self, \*\*kwargs):**

**users = request.env['res.users'].search([])**

**return request.render(**

**'todo\_website.add', {'users': users})**

这是一个简单的控制器，呈现todo\_website.add模板，并向其提供的用户，列表，以便它可以用来建立一个选择框。

现在为相应的 QWeb 模板。我们可以添加它入 todo\_website/views/todo\_web.xml 数据文件︰

**<template id="add" name="Add Todo Task">**

**<t t-call="website.layout">**

**<t t-set="additional\_title">Add Todo</t>**

**<div id="wrap" class="container">**

**<div class="row">**

**<section id="forms">**

**<form method="post"**

**class="s\_website\_form**

**container-fluid form-horizontal"**

**action="/website\_form/"**

**data-model\_name="todo.task"**

**data-success\_page="/todo"**

**enctype="multipart/form-data" >**

**<!-- Form fields will go here! -->**

**<!-- Submit button -->**

**<div class="form-group">**

**<div class="col-md-offset-3 col-md-7**

**col-sm-offset-4 col-sm-8">**

**<a class="o\_website\_form\_send**

**btn btn-primary btn-lg">**

**Save**

**</a>**

**<span id="o\_website\_form\_result"></span>**

**</div>**

**</div>**

**</form>**

**</section>**

**</div> <!-- rows -->**

**</div> <!-- container -->**

**</t> <!-- website.layout -->**

**</template>**

不出所料，我们可以找到特定于 Odoo 的< t t-call ="website.layout">元素，负责插入模板内部网站的布局，和

< t t-set ="additional\_title">，设置额外的标题，预期的网站布局。

对于内容，我们可以看到在此模板中的大多数可以发现典型引导 CSS 窗体上。但我们也有几个属性和特定于网站形式的 CSS 类。我们纪念他们中的粗体代码，这样就便于您识别它们。

CSS 类需要的 JavaScript 代码，以便能够正确执行它处理逻辑的形式。然后我们有几个特定属性 < 窗体 > 元素︰

行动是一个标准窗体的属性，但必须有""/website\_form/值。尾部的反斜杠是必需。

data-model\_name标识模型要写入，并将传递给/website\_form控制器。

data-success\_page是要成功的表单提交后重定向到的 URL。在这种情况下，我们将送回 /todo 列表。

我们也不需要提供我们自己的控制器方法来处理表单提交的。 /Website\_form 路线将为我们做的。它需要它需要从窗体，包括刚才描述的特定属性和基本验证对输入数据执行，然后在目标模型上创建一个新的记录的所有信息。

先进的用例，我们可以强制使用自定义控制器方法。

为此我们应添加数据-force\_action归因于< 窗体 >元，与要使用的目标控制器的关键字。例如，dataforce\_action ="todo-自定义" 会有表单提交来电

/website\_form/todo-customURL。然后，我们应该提供一个附加到该工艺路线的控制器方法。然而，这样做将超出我们的范围这里。

我们仍然需要完成我们的窗体，添加要从用户获得输入的字段。

内< 窗体 >元素添加︰

**<div class="col-md-3 col-sm-4 text-right">**

**<label class="control-label" for="name">To do\*</label>**

**</div>**

**<div class="col-md-7 col-sm-8">**

**<input name="name" type="text" required="True"**

**class="o\_website\_from\_input form-control" />**

**</div>**

**</div>**

**<!-- Add an attachment field -->**

**<div class="form-group form-field">**

**<div class="col-md-3 col-sm-4 text-right">**

**<label class="control-label" for="file\_upload">**

**Attach file**

**</label>**

**</div>**

**<div class="col-md-7 col-sm-8">**

**<input name="file\_upload" type="file"**

**class="o\_website\_from\_input form-control" />**

**</div>**

**</div>**

在这里我们添加两个字段、 描述常规文本字段和一个文件字段，要上传的附件。所有的标记可以发现经常引导的形式，除 o\_website\_from\_input 类，需要为网站表单逻辑准备数据提交。

用户选择列表不一样只是它需要使用tforeachQWeb 指令，以便呈现的可选的用户列表。我们可以这样做是因为控制器检索该记录集并使其可用于该模板在名称下用户:

**<!-- Select User -->**

**<div class="form-group form-field">**

**<div class="col-md-3 col-sm-4 text-right">**

**<label class="control-label" for="user\_id">**

**For Person**

**</label>**

**</div>**

**<div class="col-md-7 col-sm-8">**

**<select name="user\_id"**

**class="o\_website\_from\_input form-control" >**

**<t t-foreach="users" t-as="user">**

**<option t-att-value="user.id">**

**<t t-esc="user.name" />**

**</option>**

**</t>**

**</select>**

**</div>**

**</div>**

然而，我们仍然无法工作直到我们做一些访问安全设置。

### 13.4.2 访问安全和菜单项

因为此泛型形式处理是非常开放的和依赖于不受信任客户端发送的数据，出于安全原因需要一些服务器端套上对客户端允许做什么。尤其是，可以编写模型字段基于表单数据应列入白名单。

提供了将字段添加到此白名单中，一个 helper 函数，我们可以使用它从一个 XML 数据文件。我们应该创建 todo\_website/data/config\_data.xml 与文件︰

**<?xml version="1.0" encoding="utf-8"?>**

**<odoo>**

**<data>**

**<record id="todo\_app.model\_todo\_task" model="ir.model">**

**<field name="website\_form\_access">True</field>**

**</record>**

**<function model="ir.model.fields"**

**name="formbuilder\_whitelist">**

**<value>todo.task</value>**

**<value eval="['name', 'user\_id', 'date\_deadline']"/>**

**</function>**

**</data>**

**</odoo>**

要能够使用的形式模型，我们必须做两件事︰ 启用上的模型和白名单中的字段，可以使用的标志。这些都是在前面的数据文件所做的两项行动。

别忘了，为我们的插件模块了解此数据文件，它需要被添加到数据的清单文件的关键。

它也是很好的我们的 Todo 页面，可从网站菜单。

让我们添加使用此相同的数据文件。添加另一个 < 数据 > 这样的元素︰

**<data noupdate="1">**

**<record id="menu\_todo" model="website.menu">**

**<field name="name">Todo</field>**

**<field name="url">/todo</field>**

**<field name="parent\_id" ref="website.main\_menu"/>**

**<field name="sequence" type="int">50</field>**

**</record>**

**</data>**

正如你所看到的添加网站菜单项我们只需要在中创建记录website.menu模型，用一个名称、 URL 和父菜单项的标识符。此菜单的顶层有的父结点; website.main\_menu项目。

### 13.4.3 添加自定义逻辑

网站形式让我们插上我们自己的验证和计算到表格处理。这通过执行 website\_form\_input\_filter()与目标模型的逻辑方法。它接受值字典，验证和进行更改，然后返回可能修改值字典。

我们将使用它来执行两个功能︰ 从任务标题，删除任何前导空格和尾随空格和执行任务标题必须至少三个字符长。

添加todo\_website/models/todo\_task.py文件包含以下代码︰

**# -\*- coding: utf-8 -\*-**

**from odoo import api, models**

**from odoo.exceptions import ValidationError**

**class TodoTask(models.Model):**

**\_inherit = 'todo.task'**

**@api.model**

**def website\_form\_input\_filter(self, request, values):**

**if 'name' in values:**

**values['name'] = values['name'].strip()**

**if len(values.['name']) < 3:**

**raise ValidationError(**

**'Text must be at least 3 characters long')**

**return values**

website\_form\_input\_filter方法实际上需要两个参数︰

请求对象和值字典。防止表单提交的错误应该提高 ValidationError 例外。

大多数情况下此扩展点的形式应使我们避免自定义表单提交处理程序。

像往常一样，我们必须使 Python 导入，通过添加此新文件from

.import models在todo\_website/\_\_init\_\_.py文件，并添加todo\_website/models/\_\_init\_\_.py文件与from . import todo\_task线。

## 13.5 第十一章 小结

你现在应该很好了解的网站功能要点。我们已经看到如何使用 web 控制器和 QWeb 模板来呈现动态的 web 页。然后，我们学会了如何使用网站插件并为其创建我们自己的页面。最后，我们介绍了网站的形式插件，帮助我们创建一个 web 窗体。这些应该提供我们创建网站功能所需的核心技能。

接下来，我们将学习如何与我们的 Odoo 应用程序进行交互的外部应用程序。

# 14 第十二章 外部API - 与其他系统集成

Odoo 服务器还提供一个外部的 API，这是其 web 客户端使用的也可供其他客户端应用程序。

在这一章，我们将学习如何使用 Odoo 外部 API 从我们自己的客户端程序。可以使用任何编程语言，只要它具有对 XML-RPC 或 JSON RPC 协议支持。作为一个例子，官方文档为四个流行的编程语言提供了代码示例︰ Python、 PHP、 红宝石和 Java。

以避免引入其他读者可能不熟悉的语言，在这里我们将专注于基于 Python 的客户，虽然处理 RPC 调用的技术也适用于其他编程语言。

我们将描述如何使用 Odoo RPC 调用，然后再使用，要构建简单到做桌面应用程序使用 Python。

最后，我们将介绍 ERPPeek 客户端。它是一个 Odoo 客户端库，可用于作为一个方便的抽象层 Odoo RPC 调用，并且也是 Odoo，允许远程管理 Odoo 实例的命令行客户端。

## 14.1 设置一个Python客户端

可以从外部使用两种不同协议访问 Odoo API: XMLRPC 和 JSON RPC。任何外部的程序能够执行其中一种协议的客户端将能够与 Odoo 服务器进行交互。若要避免引入额外的编程语言，我们将继续使用 Python 探索的外部 API。

到目前为止，我们一直只在服务器上运行 Python 代码。这次，我们将使用 Python 在客户端，所以很可能你可能需要做一些额外的设置您的工作站上。

要按照本章中的示例，需要将无法运行 Python 文件在您的工作计算机上。Odoo 服务器需要 Python 2，但我们 RPC 客户端可以在任何语言中，所以 Python 3 会好起来。然而，由于一些读者可能他们正在 （你好 Ubuntu 用户 ！） 的同一台计算机上运行服务器，它将每个人都效法如果我们坚持 Python 2 简单。

如果你使用的 Ubuntu 或 Mac 的 Python 可能已经安装了。打开一个终端类型 python，和你应该会看到类似下面的内容︰

**Python 2.7.12 (default, Jul 1 2016, 15:12:24)**

**[GCC 5.4.0 20160609] on linux2**

**Type "help", "copyright",", "credits" or "license" for more**

**information.**

**>>>**

**注意**

Windows 用户可以找到安装程序为 Python 和也迅速得到加速。官方的安装包，可以发现在[https://www.python.org/downloads/](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fwww.python.org%2Fdownloads%2F).

如果您是 Windows 用户，并有 Odoo 安装在您的计算机，您可能会想，为什么你不已经有一个 Python 解释器，并需要额外的安装。简短的答案是 Odoo 安装了嵌入式的 Python 解释器外面不是很容易使用。

## 14.2 使用XML-RPC调用Odoo API

最简单的方法来访问服务器使用 XML-RPC。我们可以使用 xmlrpclib 从 Python 的标准库为此库。请记住，我们编程客户端才能连接到服务器，所以我们需要连接到 Odoo 服务器实例运行。在我们的例子中，我们将假设 Odoo 服务器实例在同一台机器上运行 (本地主机)，但您可以使用任何可访问的 IP 地址或服务器名称，如果服务器运行在不同的机器。

### 14.2.1 打开XML-RPC连接

让我们与 Odoo 外部 API 的第一次接触。启动一个 Python 控制台和类型在以下︰

**>>> import xmlrpclib**

**>>> srv = 'http://localhost:8069'**

**>>> common = xmlrpclib.ServerProxy('%s/xmlrpc/2/common' % srv)**

**>>> common.version()**

**{'server\_version\_info': [10, 0, 0, 'final', 0, ''],**

**'server\_serie': '10.0', 'server\_version': '10.0',**

**'protocol\_version': 1}**

在这里，我们导入xmlrpclib库，然后设置一个变量与服务器地址和侦听端口的信息。感觉自由地适应这些你具体研讨

下一步，我们设置了访问服务器的公共服务 （不需要登录），暴露在/xmlrpc/2/common端点。可用的方法之一就是 version()，它负责检查服务器版本。我们用它来确认我们可以与服务器通信。

另一个公共方法是authenticate()。事实上，这不创建一个会话，因为你可能不得不相信。此方法只是确认用户名和密码都接受和返回应该用于请求而不是用户名，用户 ID，如下所示︰

**>>> db = 'todo'**

**>>> user, pwd = 'admin', 'admin'**

**>>> uid = common.authenticate(db, user, pwd, {})**

**>>> print uid**

如果登录凭据不正确，假返回值，而不是用户 id。

### 14.2.2 从服务器读取数据

与 XML-RPC，没有会话进行维护并随每个请求一起发送的身份验证凭据。这会增加一些开销，议定书 》，但使它更易于使用。

接下来，我们设置了访问需要登录才能访问服务器方法。

这些暴露在/xmlrpc/2/object端点，如以下所示︰

**>>> api = xmlrpclib.ServerProxy('%s/xmlrpc/2/object' % srv)**

**>>> api.execute\_kw(db, uid, pwd, 'res.partner',**

**'search\_count'[[]])**

**40**

在这里，我们正在做我们的第一次访问到服务器 API，对**合作伙伴**记录进行计数。调用的方法使用execute\_kw() 使用以下参数的方法︰

要连接到的数据库的名称

连接的用户 ID

用户密码

目标模型标识符名称

要调用的方法

位置参数列表

使用关键字参数可选字典

前面的示例调用search\_count的方法res.partner具有一个位置参数，模型[]和没有关键字参数。位置参数是一个搜索域;因为我们正在提供一个空的列表，它计数所有伙伴。

频繁的操作是搜索和阅读。当从 RPC 调用搜索方法返回的 Id 列表的域匹配。浏览方法不可用从 RPC 和读应该用于在它的地方给出的记录 Id 列表和检索他们的数据，如下面的代码所示︰

**>>> api.execute\_kw(db, uid, pwd, 'res.partner', 'search',**

**[[('country\_id', '=', 'be'), ('parent\_id', '!=', False)]])**

**[18, 33, 23, 22]**

**>>> api.execute\_kw(db, uid, pwd, 'res.partner', 'read',**

**[[18]], {'fields': ['id', 'name', 'parent\_id']})**

**[{'parent\_id': [8, 'Agrolait'], 'id': 18, 'name': 'Edward**

**Foster'}]**

指出，阅读方法，我们使用的一个位置参数列表的 Id，[] 18，和一个关键字参数，字段。我们还可以注意到多对一关系的字段作为搭配相关的记录 ID 检索和显示名称。这是在处理您的代码中的数据时，请牢记。

搜索和阅读组合是如此频繁，search\_read方法可用于单步执行这两种操作。相同的结果前两个步骤可以获得以下︰

**>>> api.execute\_kw(db, uid, pwd, 'res.partner', 'search\_read',**

**[[('country\_id', '=', 'be'), ('parent\_id', '!=', False)]],**

**{'fields': ['id', 'name', 'parent\_id']})**

search\_read方法类似于阅读，但希望域作为第一的位置参数，而不是 Id 列表。值得一提的字段参数对阅读和 search\_read不是强制性的。如果未提供，将检索所有字段。这可能会导致昂贵计算的函数字段和大量数据的检索，但可能从未使用过，所以它通常建议提供显式的字段列表。

### 14.2.3 调用其他方法

所有其他模型方法受到通过 RPC，除了带有下划线前缀被认为是私事。这意味着我们可以使用创建, 写，和取消链接修改服务器上的数据，如下所示︰

**>>> api.execute\_kw(db, uid, pwd, 'res.partner', 'create',**

**[{'name': 'Packt Pub'}])**

**45**

**>>> api.execute\_kw(db, uid, pwd, 'res.partner', 'write', [[45],**

**{'name': 'Packt Publishing'}])**

**True**

**>>> api.execute\_kw(db, uid, pwd, 'res.partner', 'read', [[45],**

**['id', 'name']])**

**[{'id': 45, 'name': 'Packt Publishing'}]**

**>>> api.execute\_kw(db, uid, pwd, 'res.partner', 'unlink',**

**[[45]])**

**True**

XML-RPC 协议的一个限制是，它不支持没有一个值。言下之意是，不返回任何值的方法将不会通过 XML-RPC，可用，因为他们都隐式地返回无。这就是为什么方法应该总是完成与至少返回 True 语句。

值得重复 Odoo 外部 API 可以用于大多数的编程语言。在正式文件中我们可以找到红宝石、 PHP 和 Java 实例。它是发售[https://www.odoo.com/documentation/10.0/api\_integration.html](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fwww.odoo.com%2Fdocumentation%2F10.0%2Fapi_integration.html).

## 14.3 编写Notes桌面应用程序

让我们做一些有意义的 RPC API。Odoo 为笔记提供了一个简单的应用程序。如果用户可以直接从他们的计算机桌面管理他们个人的笔记吗？让我们写一个简单的 Python 应用程序，做到这一点，如下面的屏幕快照中所示︰

为清楚起见，我们会将它拆分成两个文件︰ 一个处理与交互服务器后端，note\_api.py，和另一种使用图形用户界面，note\_gui.py.

### 14.3.1 Odoo的通信层

我们将创建一个类来设置了连接和存储其信息。它应该暴露两种方法︰ get （) 检索任务数据和 set()来创建或更新任务。

选择主机应用程序文件和创建的目录note\_api.py文件。我们可以通过添加类构造函数，如下︰

**import xmlrpclib**

**class NoteAPI():**

**def \_\_init\_\_(self, srv, db, user, pwd):**

**common = xmlrpclib.ServerProxy(**

**'%s/xmlrpc/2/common' % srv)**

**self.api = xmlrpclib.ServerProxy(**

**'%s/xmlrpc/2/object' % srv)**

**self.uid = common.authenticate(db, user, pwd, {})**

**self.pwd = pwd**

**self.db = db**

**self.model = 'note.note'**

在这里我们将存储在所创建的对象需要在模型上执行调用的所有信息︰ API 参考，uid，密码、 数据库名称和要使用的模型。

接下来，我们将定义一个帮助器方法来执行调用。它利用了存储对象数据提供更小的函数签名，如下所示︰

**def execute(self, method, arg\_list, kwarg\_dict=None):**

**return self.api.execute\_kw(**

**self.db, self.uid, self.pwd, self.model,**

**method, arg\_list, kwarg\_dict or {})**

现在我们可以使用它来实现更高的级别get （)和set()方法。

get （)方法将接受一个可选的 Id 来检索列表。如果没有列出，将返回所有的记录︰

**def get(self, ids=None):**

**domain = [('id',' in', ids)] if ids else []**

**fields = ['id', 'name']**

**return self.execute('search\_read', [domain, fields])**

set()方法将有任务文本写和一个可选的 ID 作为参数。如果不提供 ID，则会创建新的记录。它返回的 ID 的记录写入或创建，如下所示︰

**def set(self, text, id=None):**

**if id:**

**self.execute('write', [[id], {'name': text}])**

**else:**

**vals = {'name': text, 'user\_id': self.uid}**

**id = self.execute('create', [vals])**

**return id**

让我们结束与一块小小的测试代码，如果我们运行 Python 文件，将执行文件︰

**if \_\_name\_\_ == '\_\_main\_\_':**

**srv, db = 'http://localhost:8069', 'todo'**

**user, pwd = 'admin', 'admin'**

**api = NoteAPI(srv, db, user, pwd)**

**from pprint import pprint**

**pprint(api.get())**

如果我们运行的 Python 脚本，我们应该看到的内容打印出来我们待办任务。现在，我们有一个简单的包装在我们 Odoo 后端附近，让我们处理桌面的用户界面。

### 14.3.2 创建GUI

这里我们的目标是要学会写 Odoo 服务器，与外部应用程序的接口，这样做是在前一节。但它将是一个遗憾不到额外的步骤并实际上将其提供给最终用户。

继续安装程序尽可能简单，我们将使用 Tkinter 来实现图形用户界面。因为它是标准库的一部分，它不需要任何额外的安装。它不是我们的目标是解释 Tkinter 是如何工作的所以我们一定会短上对它的解释。

每个**任务**应该有一个小的黄色窗口在桌面上。这些窗口会有一个单一的**文本**小部件。按住***ctrl 键***+ ***N***将打开新的注释，并按***Ctrl***+ ***S***会将当前便笺的内容写到 Odoo 服务器。

现在，旁边note\_api.py创建一个新的文件note\_gui.py文件。它将首先导入 Tkinter 模块和小部件，我们将使用，然后 NoteAPI 类，如下所示︰

**from Tkinter import Text, Tk**

**import tkMessageBox**

**from note\_api import NoteAPI**

如果前面的代码错误

ImportError: No module named

\_tkinter, please install the python-tk package

这意味着您的系统上需要额外的库。在 Ubuntu 上你将需要运行以下命令︰

**$ sudo apt-get install python-tk**

下一步我们创建 Tkinter 之一来自我们自己的文本小部件。当创建一个实例，它将指望 API 参考，用于保存行动，以及也该任务的文本和 ID，如以下所示︰

**class NoteText(Text):**

**def \_\_init\_\_(self, api, text='', id=None):**

**self.master = Tk()**

**self.id = id**

**self.api = api**

**Text.\_\_init\_\_(self, self.master, bg='#f9f3a9',**

**wrap='word', undo=True)**

**self.bind('<Control-n>', self.create)**

**self.bind('<Control-s>', self.save)**

**if id:**

**self.master.title('#%d' % id)**

**self.delete('1.0', 'end')**

**self.insert('1.0', text)**

**self.master.geometry('220x235')**

**self.pack(fill='both', expand=1)**

Tk()构造函数创建一个新的用户界面窗口和文本小部件的地方本身在它里面，以便创建新的NoteText实例会自动打开一个桌面窗口。

下一步，我们将实施创建和保存行动。创建行动打开一个新的空窗口，但它将存储在服务器中只有当保存执行的操作。下面是相应的代码︰

**def create(self, event=None):**

**NoteText(self.api, '')**

**def save(self, event=None):**

**text = self.get('1.0', 'end')**

**self.id = self.api.set(text, self.id)**

**tkMessageBox.showinfo('Info', 'Note %d Saved.' % self.id)**

保存可以执行操作，对现有的或新的任务，却不需要担心，在这里，因为这些案件已由set()方法NoteAPI.

最后，我们将添加代码，以检索并创建所有注意 windows 时启动该程序，如下面的代码所示︰

**if \_\_name\_\_ == '\_\_main\_\_':**

**srv, db = 'http://localhost:8069', 'todo'**

**user, pwd = 'admin', 'admin'**

**api = NoteAPI(srv, db, user, pwd)**

**for note in api.get():**

**x = NoteText(api, note['name'], note['id'])**

**x.master.mainloop()**

最后一个命令运行mainloop()上创建，开始等待窗口事件的最后一注窗口。

这是一个非常基本的应用程序，但这里的关键是要有趣的方式来利用 Odoo RPC API 示例。

## 14.4 介绍ERPpeek客户端

ERPpeek 是一种多功能的工具，可以使用作为一个交互式**命令行界面**(**CLI**) 以及**Python 库**，与比提供的一个更方便的 APIxmlrpclib。它是从 PyPi 索引可用和可以用以下安装︰

**$ pip install -U erppeek**

在 Unix 系统中，如果您正在安装它系统范围内，你可能需要追加sudo到的命令。

### 14.4.1 ERPpeek API

ERPpeek库提供了一个编程接口，环绕在周围xmlrpclib，这是类似于我们对服务器端代码的编程接口。

我们这里的重点是提供一瞥的什么ERPpeek图书馆已经提供，并不提供其所有功能充分解释。

我们可以开始复制了我们第一步xmlrpclib使用erppeek，如下所示︰

**>>> import erppeek**

**>>> api = erppeek.Client('http://localhost:8069',**

**'todo','admin', 'admin')**

**>>> api.common.version()**

**>>> api.count('res.partner', [])**

**>>> api.search('res.partner', [('country\_id', '=', 'be'),**

**('parent\_id', '!=', False)])**

**>>> api.read('res.partner', [44], ['id', 'name', 'parent\_id'])**

正如你所看到的 API 调用使用更少的参数和类似服务端对应。

但ERPpeek不在这儿停，并且还提供模型的表示法。我们有以下两种替代方法来获取实例模型，可以使用 model() 方法或访问作为一个骆驼案例属性名称︰

**>>> m = api.model('res.partner')**

**>>> m = api.ResPartner**

现在我们可以执行操作的这一模式，如下所示︰

**>>> m.count([('name', 'like', 'Packt%')])**

**1>**

**>> m.search([('name', 'like', 'Packt%')])**

**[44]**

它还提供客户端的对象表示形式为记录，如下所示︰

**>>> recs = m.browse([('name', 'like', 'Packt%')])**

**>>> recs**

**<RecordList 'res.partner,[44]'>**

**>>> recs.name**

**['Packt Publishing']**

正如你所看到的erppeek图书馆走很长的路，从平原xmlrpclib并使它可能写代码，可以重复使用服务器端很少或没有修改。

### 14.4.2 ERPpeek CLI

不仅可以erppeek使用库作为 Python 库，它也是可以用来在服务器上执行管理操作的 CLI。在 odoo 壳提供本地的交互式会话主机服务器上的命令 erppeek 图书馆提供远程交互式会话的客户端上整个网络。

打开命令行，我们可以在聚醚醚酮可选项，如以下所示︰

**$ erppeek –help**

让我们看一个示例会话，如下所示︰

**$ erppeek --server='http://localhost:8069' -d todo -u admin**

**Usage (some commands):**

**models(name) # List models matching pattern**

**model(name) # Return a Model instance**

**(...)**

**Password for 'admin':**

**Logged in as 'admin'**

**todo >>> model('res.users').count()**

**3t**

**odo >>> rec = model('res.partner').browse(43)**

**todo >>> rec.name**

**'Packt Publishing'**

正如你所看到的连接到服务器，并执行上下文提供参考model()方法以获取模型实例，并对它们执行操作。

erppeek。客户端用于连接的实例也是可通过客户端变量。

值得注意的是，它提供对 web 客户端来管理安装的加载项模块替代︰

client.modules()︰ 列出可用或已安装的模块

client.install()︰ 执行模块安装

client.upgrade()︰ 执行模块升级

client.uninstall()︰ 卸载模块

所以，erppeek也可以提供良好的服务作为一种远程管理工具 Odoo 服务器。

## 14.5 第十二章 小结

这一章我们的目标是要了解外部 API 是如何工作，它是有能力。我们开始探索它使用一个简单的 Python XML-RPC 客户端，但是可以从任何编程语言使用外部 API。事实上，官方文档提供了代码示例 Java，PHP，和红宝石。

有大量要处理 XML-RPC 或 JSON RPC、 一些泛型和 Odoo 用一些特定的库。我们不试图指出任何图书馆特别，除了 erppeek，因为它不是只有一个证明的包装为 Odoo/OpenERP XML-RPC，而是因为它也是远程服务器管理和检验的宝贵工具。

直到现在，我们使用我们的 Odoo 服务器实例为开发和测试。但是，有一个生产级服务器，有额外的安全性和需要做的优化配置。在下一章中，我们将专注他们。

# 15 第十三章 实战部署

在本章中，您将学习如何准备您的 Odoo 服务器在生产环境中使用。

有许多可能的策略和工具，可以用来部署和管理 Odoo 生产服务器。我们将指导您通过一种方法做这件事。

这是服务器设置清单，我们将遵循︰

1.安装依赖项和专用的用户运行服务器

2.从源代码安装 Odoo

3.设置 Odoo 配置文件

4.设置多重处理工作人员

5.建立 Odoo 系统服务

6.设置与 SSL 支持的反向代理

我们开始吧。

## 15.1 可用的预构建软件包

Odoo 有 Debian/Ubuntu 软件包可供安装。使用它，你得到系统开机自动启动的工作服务器进程。此安装过程非常简单，你可以找到所有你需要在[https://nightly.odoo.com](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fnightly.odoo.com%2F)。还可以找到 rpm ( CentOS构建)和 .exe 安装程序。

虽然这是简单、方便的方式来安装 Odoo，大多数集成商倾向于部署和运行受版本控制的源代码。这提供了更好地控制什么部署，并且可以轻松地管理更改和修复生产中的一次。

## 15.2 安装依赖关系

当使用 Debian 的分布，在默认情况下您的登录名需要管理员的权力，与你的命令提示符下显示#。当使用 Ubuntu，root帐户被禁用，并在安装过程中配置的初始用户是*sudoer*，用户需要具备 sudo 命令使用根权限运行命令。

首先，我们应该更新软件包的索引，然后执行升级，以确保所有已安装的程序是最新︰

**$ sudo apt-get update**

**$ sudo apt-get upgrade -y**

接下来，我们将安装 PostgreSQL 数据库，并使我们的用户是数据库超级用户︰

**$ sudo apt-get install postgresql y**

**$ sudo su-c"createuser-s $(whoami)"postgres**

我们将运行 Odoo 来源，但在之前，我们需要安装所需的依赖关系。这些都是需要的 Debian 软件包︰

**$ sudo apt-get install git python-pip python2.7-dev -y**

**$ sudo apt-get install libxml2-dev libxslt1-dev**

**libevent-dev \**

**libsasl2-dev libldap2-dev libpq-dev**

**libpng12-dev libjpeg-dev \**

**poppler-utils**

**node-less node-clean-css -y**

我们不应忘记安装wkhtmltox，这需要打印报表︰

**$ wget http://nightly.odoo.com/extra/wkhtmltox-0.12.1.2\_linuxjessie-**

**amd64.deb**

**$ sudo dpkg -i wkhtmltox-0.12.1.2\_linux-jessie-amd64.deb**

**$ sudo apt-get -fy install**

安装说明将报告缺少的依赖项错误，但最后一个命令强制安装这些依赖项和正确完成安装。

现在我们唯一缺少所需的 Odoo 的 Python 程序包。其中许多人也有 Debian/Ubuntu 系统软件包。官方的 Debian 安装包使用它们，和你可以包名称 Odoo 在源代码中查找，在 debian/控制文件。

然而，直接从也可以安装这些 Python 依赖项

**Python 包索引**(**PyPI**)。这是友好对于那些喜欢安装在 Odoo 。所需的软件包列表是在 Odoorequirements.txt 文件，和往常一样对于基于 Python 的项目。我们可以用这些命令来安装它们︰

**$ sudo -H pip install --upgrade pip # Ensure pip latest version**

**$ wget https://raw.githubusercontent.com/odoo/odoo/10.0/requirements.txt**

**$ sudo -H pip install -r requirements.txt**

现在，我们有安装的所有依赖文件，数据库服务器、 系统的软件包，以及 Python 包，我们可以安装 Odoo。

### 15.2.1 准备专用系统用户

良好的安全实践是运行 Odoo 使用专用的用户，无任何特殊权限在系统上。

我们需要为所创建的系统和数据库的用户。我们可以为它们命名 odoo 产品，例如︰

**$ sudo adduser --disabled-password --gecos "Odoo" odoo**

**$ sudo su -c "createuser odoo" postgres**

**$ createdb --owner=odoo odoo-prod**

在这里，odoo是用户名和odoo 产品是支持我们 Odoo 实例的数据库的名称。

注意这些普通用户没有任何管理权限。为新的系统用户自动创建一个主目录。在此示例中，它是/home/odoo，和用户可以引用其自身主目录 ~ 快捷方式符号。我们将用于该用户的 Odoo 具体配置和文件。

我们可以打开一个会话作为此用户使用下面的命令︰

**$ sudo su odoo**

退出命令终止该会话并返回到我们的原始用户。

### 15.2.2 从源代码安装

或早或晚，您的服务器需要升级和修补程序。在时机成熟时，版本控制存储库可以很大的帮助。我们使用 git 从存储库中，就像我们一样要安装开发环境要我们的代码。

下一步，我们将模拟odoo用户和代码下载到其主目录︰

**$ sudo su odoo**

**$ git clone** [**https://github.com/odoo/odoo.git /home/odoo/odoo-10.0 -b 10**](https://github.com/odoo/odoo.git%20/home/odoo/odoo-10.0%20-b%2010)**.0 --depth=1**

-b选项可以确保我们获得正确的分支，和--depth=1选项忽略更改历史记录和检索只最新代码的版本，使下载更小和更快。

**小贴士**

Git 肯定会宝贵的工具来管理您的 Odoo 部署的版本。我们只是一点皮毛的所作所为来管理代码版本。如果你不已经熟悉 Git，值得更多地了解它。一个好的起点是[http://git-scm.com/doc .](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=http%3A%2F%2Fgit-scm.com%2Fdoc)

现在我们应该有一切需要从源运行 Odoo。我们可以检查它可以正常启动，然后退出从专用的用户会话︰

**$ /home/odoo/odoo-10.0/odoo-bin --help**

**$ exit**

接下来，我们将设置一些系统级文件和目录使用的系统服务。

### 15.2.3 设置配置文件

添加— — 保存选项启动 Odoo 服务器保存到使用的配置时~/.odoorc文件。我们可以使用该文件作为起点对于我们服务器配置，将存储在 /etc/odoo，如下面的代码所示︰

**$ sudo su -c "~/odoo-10.0/odoo-bin -d odoo-prod --save --stopafter-init" odoo**

这会要我们服务器实例所用的配置参数。

**小贴士**

前者.openerp\_serverrc仍然支持配置文件，并使用的如果发现。在一种机器，也被用来运行以前的 Odoo 版本设置 Odoo 10 时，这可能造成一些混乱。在这种情况下 — — 保存可能更新选项.openerp\_serverrc 文件而不是 .odoorc.

接下来，我们需要将配置文件放在预期的位置︰

**$ sudo mkdir /etc/odoo**

**$ sudo cp /home/odoo/.odoorc /etc/odoo/odoo.conf**

**$ sudo chown -R odoo /etc/odoo**

我们还应该创建 Odoo 服务将在哪里存储其日志文件的目录。这预期要去某个地方里面 /var/log:

**$ sudo mkdir /var/log/odoo**

**$ sudo chown odoo /var/log/odoo**

现在我们应该确保几个重要参数的配置。这里是最重要的建议的值︰

**[options]**

**addons\_path = /home/odoo/odoo-10.0/odoo/addons,**

**/home/odoo/odoo-10.0/addons**

**admin\_passwd = False**

**db\_user = odoo-prod**

**dbfilter = ^odoo-prod$**

**logfile = /var/log/odoo/odoo-prod.log**

**proxy\_mode = True**

**without\_demo = True**

**workers = 3**

**xmlrpc\_port = 8069**

让我们解释他们︰

addons\_path是一个逗号分隔的列表，需要查附加模块的路径。它读从左到右，和具有较高优先级的最左边目录。

admin\_passwd是主密码来访问 web 客户端数据库管理功能。它是关键设置此用一个强密码，或甚至更好，要将它设置为虚假要停用的功能。

db\_user要在服务器启动序列的过程中初始化的数据库实例。

dbfilter是筛选器要访问的数据库。它是一个 Pythoninterpreted 的正则表达式表达式。用户不会提示您选择一个数据库，以及为未经身份验证的 Url 才能正常工作，应将其设置与 ^ 数据库名称 $，例如， dbfilter = ^ odoo prod$。它支持 %h 和 占位符，取而代之的是 HTTP 请求的主机名和子域名称。

日志文件是应在其中写入服务器日志。系统服务预期的位置在什么地方是里面 /var/log。如果留空，或设置为虚假，日志打印到标准输出相反。

proxy\_mode应设置为真正Odoo 访问在反向代理后面我们会做的事时

without\_demo应设置为真正在生产环境中，新的数据库并没有对他们演示数据。

工人具有两个或多个启用多处理器模式的值。我们将这更详细地讨论一会儿。

xmlrpc\_port是，服务器将侦听的端口号。默认情况下，端口 8069 使用。

以下参数也可以是有益的︰

data\_dir是存储会话数据和附件文件的路径。记得要对它有备份 xmlrpc 接口设置将听的地址。默认情况下，它听所有 0.0.0.0，但是当使用反向代理服务器，可以将其设置为

127.0.0.1为了只响应本地请求

我们可以检查由运行与服务器设置的效果-c或— — 配置选项，如下所示︰

**$ sudo su-c"~/odoo-10.0/odoo-bin-c /etc/odoo/odoo.conf"odoo**

通过以上设置运行 Odoo 不会显示任何输出到控制台，因为它正在写入配置文件中定义的日志文件。要遵循什么我们需要打开另一个终端窗口，并那里运行的服务器︰

**$ sudo tail -f /var/log/odoo/odoo-prod.log**

如果是还可能要使用配置文件和仍然强制日志输出要打印到控制台，通过添加— — 日志文件 = False选项，像这样︰

**$ sudo su -c "~/odoo-10.0/odoo-bin -c /etc/odoo/odoo.conf --logfile=False" odoo**

## 15.3 多线程操作

A 生产实例预计处理重大的工作负荷。默认情况下，服务器运行一个进程，并且可以使用只有一个 CPU 核心处理，因为 Python 语言吉尔。然而，多进程模式是可用，以便可以处理的并发请求。选项工人 = N 设置的要使用的工作进程数。作为一个准则，你可以试着将它设置为 1 + 2 \* P、 在哪里 P ，是处理器的数目。最好要使用需要进行调整以每个案件的设置，因为它取决于服务器的负载和其他负荷密集服务正在运行在服务器上，如 PostgreSQL。

它是更好地设置太高了，比太低负载的工人。至少应该 6 由于大多数浏览器所使用的并行连接和最大值通常是通过在计算机上的 RAM 量有限。

有一些极限-\*要调整工人的配置参数。工人被回收的当他们达到这些限制 — — 相应进程已停止，开始一个新。这可以保护服务器从内存泄漏和超载的服务器资源的特定过程。

官方文档已经对参数整定的工人，提供好的建议，你可能提到它为更多的细节，在[https://www.odoo.com/documentation/10.0/setup/deploy.html .](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fwww.odoo.com%2Fdocumentation%2F10.0%2Fsetup%2Fdeploy.html)

### 15.3.1 设置系统服务

接下来，我们将想要设置 Odoo 作为一种系统服务，它在系统引导时自动启动。

在 Ubuntu/Debian，初始化系统是负责任的启动服务。

从历史上看，Debian （和派生的作业系统） 已使用sysvinit并用 Ubuntu 兼容的系统调用暴发户。最近，这已改变，和 init 系统使用最新版本是现在舰碰撞.

这意味着有两种不同的方法来安装系统服务，您需要选取正确的一个取决于您的操作系统的版本。

上的最后一个 Ubuntu 的稳定版本 160.4，我们应该使用舰碰撞。但如 14.04 旧版本仍使用多云提供商，有很好的机会，你可能需要使用它。

要检查是否舰碰撞用于在您的系统中试用此命令︰

$ man init

这将打开当前使用的文档初始化系统，和你将能够检查干什么用。

### 15.3.2 创建systemd服务

如果你正在使用的操作系统是最近的例如 Debian 8 或 Ubuntu

160.4，您应该使用舰碰撞为初始化系统。

要向系统添加新的服务，我们只需要创建一个描述它的文件。

创建/lib/systemd/system/odoo.service文件与下面的内容︰

**[Unit]**

**Description=Odoo**

**After=postgresql.service**

**[Service]**

**Type=simple**

**User=odoo**

**Group=odoo**

**ExecStart=/home/odoo/odoo-10.0/odoo-bin -c /etc/odoo/odoo.conf**

**[Install]**

**WantedBy=multi-user.target**

接下来，我们需要注册新服务︰

**$ sudo systemctl enable odoo.service**

要启动这个新的服务使用下面的命令︰

**$ sudo**

**systemctl**

**odoo start**

并检查其状态运行此查询︰

**$ sudo systemctl odoo status**

最后，如果你想要停止它，使用此命令︰

**$ sudo systemctl odoo stop**

### 15.3.3 创建Upstart/sysvinit服务

如果您使用较旧的操作系统，如 Debian 7，Ubuntu 15.04，或甚至; 14.04，很有可能是您的系统sysvinit关于暴发户。为此目的，两者应该相同的行为方式。多云的 VPS 服务仍然基于 Ubuntu 14.04 图像，所以这可能是一个场景你可能会遇到当部署您的 Odoo 服务器。

多云的 VPS 服务仍然基于 Ubuntu 14.04 图像，所以这可能是一个部署您的 Odoo 服务器时，您可能会遇到的场景。

Odoo 源代码包含初始化脚本用于 Debian 打包分发。我们可以使用它作为我们的服务 init 脚本与一些微小的修改，如下所示︰

**$ sudo cp /home/odoo/odoo-10.0/debian/init /etc/init.d/odoo**

**$ sudo chmod +x /etc/init.d/odoo**

在这一点上，你可能想要检查的内容初始化脚本。关键参数分配给变量在文件的顶部。示例如下︰

**PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin**

**DAEMON=/usr/bin/odoo**

**NAME=odoo**

**DESC=odoo**

**CONFIG=/etc/odoo/odoo.conf**

**LOGFILE=/var/log/odoo/odoo-server.log**

**PIDFILE=/var/run/${NAME}.pid**

**USER=odoo**

这些变量应该是足够和我们将准备集的其余部分了牢记其默认值。但当然，你可以将它们更改为更好地满足您的需求。

用户变量是服务器将在其下运行的系统用户。我们已经创建了预期 odoo 用户。

守护进程变量是服务器可执行文件的路径。我们实际的可执行文件以启动 Odoo 是在一个不同的位置，但我们可以创建指向它的符号链接︰

**$ sudo ln -s /home/odoo/odoo-10.0/odoo-bin /usr/bin/odoo**

**$ sudo chown -h odoo /usr/bin/odoo**

配置变量是要使用的配置文件。在前一节中，我们创建的预期的默认位置中的配置文件︰

**/etc/odoo/odoo.conf.**

最后，日志文件变量是存储日志文件的目录。预期的目录是 /var/log/odoo 我们创建我们被定义的配置文件时。

现在，我们应该能够启动和停止我们的 Odoo 服务，如下所示︰

**$ sudo /etc/init.d/odoo start**

**Starting odoo: ok**

停止该服务完成类似的方式，如以下所示︰

**$ sudo /etc/init.d/odoo stop**

**Stopping odoo: ok**

在 Ubuntu 中，服务也可以使用命令︰

**$ sudo service odoo start**

**$ sudo service odoo status**

**$ sudo service odoo config**

现在我们只需要让系统开机自动启动此服务︰

**$ sudo update-rc.d odoo defaults**

在此之后，当我们重新启动我们的服务器上的 Odoo 服务应该开始自动并没有错误。它是一个好时间检查所有，按预期方式工作。

### 15.3.4 用命令行检查Odoo服务

在这一点上，我们可以确认，是否我们 Odoo 实例正在和对请求的响应。

如果 Odoo 正常运行的我们现在应该能够从它那里得到的反应，看看在日志文件中的没有错误。如果 Odoo 响应 HTTP 请求使用下面的命令，我们可以在服务器内部检查︰

**$ curl http://localhost:8069**

**<html><head><script>window.location = '/web' + location.hash;**

**</script></head></html>**

要看到什么是在日志文件中我们可以使用以下方法︰

**$ sudo less /var/log/odoo/odoo-server.log**

在你刚开始使用 Linux 的情况下，你会想知道，你可以遵循在日志文件中使用什么尾巴 — — f:

**$ sudo tail -f /var/log/odoo/odoo-server.log**

## 15.4 使用反向代理

虽然 Odoo 本身可以提供 web 页面，它是强烈建议有反向代理。反向代理充当中介管理客户端发出的请求和响应他们的 Odoo 服务器之间的通信。使用反向代理服务器有几个好处。

在安全方面，它可以执行以下操作︰

处理 （和执行） 的通信进行加密的 HTTPS 协议

隐藏内部网络特性

作为*应用程序防火墙*限制接受进行处理的 Url

并在性能方面，它可以提供重大改进︰

高速缓存静态内容，从而减少 Odoo 服务器上的负载

压缩内容以加快加载时间

作为分配几个服务器之间的负载的负载平衡器

Apache 是受欢迎的选择要用作反向代理。Nginx 是一个最近的替代与良好的技术参数。在这里，我们将选择使用 Nginx 作为反向代理和显示如何使用它来执行这里提到的安全性和性能侧功能。

### 15.4.1 设置Nginx反向代理

首先，我们应该安装 Nginx。我们希望它侦听默认 HTTP 端口，所以我们应该确保他们不已经采取的一些其他服务。执行此命令应该导致错误，如下︰

**$ curl http://localhost**

**curl: (7) Failed to connect to localhost port 80: Connection**

**refused**

如果不是这样，你应该禁用或删除该服务允许 Nginx 使用这些端口。例如，若要停止现有的 Apache 服务器应使用此命令︰

$ sudo service apache2 stop

或更好的是，你应该考虑将其删除从您的系统，或重新配置为侦听另一个端口，所以的 HTTP 和 HTTPS 端口 (80 和 443) 都是免费使用的 Nginx。

现在，我们可以安装 Nginx，是以预期的方式︰

**$ sudo apt-get install nginx**

以确认它是否正确工作，访问服务器与浏览器的地址时，我们应该看到**欢迎光临 nginx**的网页或使用卷曲 http://localhost在我们的服务器内。

Nginx 配置文件遵循 Apache 相同的方法︰ 它们存储在**/etc/nginx/available-sites** /并通过添加一个符号链接在激活

/etc/nginx/enabled-sites /。我们还应该禁用默认配置中提供的 Nginx 安装，如下︰

**$ sudo rm /etc/nginx/sites-enabled/default**

**$ sudo touch /etc/nginx/sites-available/odoo**

**$ sudo ln -s /etc/nginx/sites-available/odoo /etc/nginx/sites-enabled/odoo**

使用编辑器如纳米或第六，我们应该编辑我们 Nginx 配置文件如下︰

$ sudo nano /etc/nginx/sites-available/odoo

首先，我们添加的上游，和后端服务器 Nginx 将重定向到在我们的例子，其中正在侦听端口 Odoo 服务器的通信8069，具体如下︰

**upstream backend-odoo {**

**server 127.0.0.1:8069;**

**}**

**server {**

**location / {**

**proxy\_pass http://backend-odoo;**

**}**

**}**

若要测试已编辑的配置是否正确，请使用下面的命令︰

$ sudo nginx t

如果您发现错误，请确认正确键入配置文件。同时，一个常见的问题是默认 HTTP 采取的其他服务，比如 Apache 或默认 Nginx 网站。仔细检查之前的指示以确保并非如此，然后重新启动 Nginx。在此之后，我们可以有 Nginx 重新加载新的配置，如下所示︰

$ sudo /etc/init.d/nginx reload

我们现在可以确认，Nginx 把通信重定向到后端 Odoo 服务器︰

**$ curl http://localhost**

**<html><head><script>window.location = '/web' + location.hash;**

**</script></head></html>**

### 15.4.2 执行HTTPS

接下来，我们应该安装证书，以便能够使用 SSL。若要创建自签名证书的证书，请按照以下步骤︰

**$ sudo mkdir /etc/nginx/ssl && cd /etc/nginx/ssl**

**$ sudo openssl req -x509 -newkey rsa:2048 -keyout key.pem -out**

**cert.pem -days 365 -nodes**

**$ sudo chmod a-wx \* # make files read only**

**$ sudo chown www-data:root \* # access only to www-data group**

这将创建ssl /目录内/etc/nginx /目录并创建一个无密码的自行签署式 SSL 证书。当运行 openssl 命令，会问一些额外的信息，而且生成的证书和密钥的文件。最后，这些文件的所有权给用户 www 数据用于运行 web 服务器。

**注意**

使用自签名的证书可以的构成一些安全风险，如人在中东的攻击，甚至可能不允许某些浏览器。为一个健壮的解决方案，您应该使用由一个公认的证书颁发机构签名的证书。这是特别重要，如果你正在运行一个商业或电子商务网站。

现在，我们有一个 SSL 证书，我们准备配置 Nginx 来使用它。

执行 HTTPS，我们将所有的 HTTP 流量重定向到它。取代服务器指令，我们以前定义如下︰

**server {**

**listen 80;**

**add\_header Strict-Transport-Security max-age=2592000;**

**rewrite ^/.\*$ https://$host$request\_uri? permanent;**

}

如果我们重新加载现在 Nginx 配置和访问服务器与 web 浏览器，我们将看到，http://地址将被转换为https://地址。

但在我们通过添加以下服务器配置正确，配置 HTTPS 服务之前，它不会返回任何内容︰

**server {**

**listen 443 default;**

**# ssl settings**

**ssl on;**

**ssl\_certificate /etc/nginx/ssl/cert.pem;**

**ssl\_certificate\_key /etc/nginx/ssl/key.pem;**

**keepalive\_timeout 60;**

**# proxy header and settings**

**proxy\_set\_header Host $host;**

**proxy\_set\_header X-Real-IP $remote\_addr;**

**proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;**

**proxy\_set\_header X-Forwarded-Proto $scheme;**

**proxy\_redirect off;**

**location / {**

**proxy\_pass http://backend-odoo;**

**}**

**}**

这将侦听 HTTPS 端口和使用/ 等/nginx/ssl /证书文件的通信进行加密。我们还将一些信息添加到要让知道它正在代理 Odoo 后端服务的请求标头。

出于安全方面的考虑，它是重要的 Odoo，以确保proxy\_mode参数设置为真正。原因是，当 Nginx 充当代理，所有请求将都显示来自服务器本身而不是远程 IP 地址。设置 X 转发为代理中的标头，并使 -代理-模式还解决了。但有利 — — 代理模式而不强制此标头在代理级别会允许任何人欺骗他们的远程地址。

在结束了，位置指令定义所有的请求被传递给后端 odoo上游。

重新加载配置，和我们应该有我们的 Odoo 服务工作通过 HTTPS，如以下命令所示︰

**$ sudo nginx -t**

**nginx: the configuration file /etc/nginx/nginx.conf syntax is ok nginx: configuration file /etc/nginx/nginx.conf test is successful**

**$ sudo service nginx reload \* Reloading nginx configuration nginx ...done.**

**$ curl -k https://localhost**

**<html><head><script>window.location = '/web' + location.hash;**

**</script></head></html>**

最后输出确认，Odoo web 客户端是通过 HTTPS。

**小贴士**

Odoo POSBox 正在使用的位置特定的情况下，我们需要添加为例外/pos/URL 必须能够访问它在 HTTP 模式。POSBox 位于本地网络，但并没有启用 SSL。如果在 HTTPS 加载 POS 接口，则它不能根本联系 POSBox。

### 15.4.3 Nginx优化

现在，它是为一些微调 Nginx 设置时间。他们建议启用响应缓冲和数据压缩，应提高网站的速度。我们还将设置日志的特定位置。

以下配置应添加内部服务器侦听端口443，例如，只是代理定义之后︰

**# odoo log files**

**access\_log /var/log/nginx/odoo-access.log;**

**error\_log /var/log/nginx/odoo-error.log;**

**# increase proxy buffer size**

**proxy\_buffers 16 64k;**

**proxy\_buffer\_size 128k;**

**# force timeouts if the backend dies**

**proxy\_next\_upstream error timeout invalid\_header http\_500**

**http\_502 http\_503;**

**# enable data compression**

**gzip on;**

**gzip\_min\_length 1100;**

**gzip\_buffers 4 32k;**

**gzip\_types text/plain text/xml text/css text/less**

**application/x-javascript application/xml application/json**

**application/javascript;**

**gzip\_vary on;**

我们也可以激活缓存更快响应的要求提到前面的代码示例中，避免他们 Odoo 服务器上的负载的类型的静态内容。后位置 / 节，添加以下位置第二︰

**location ~\* /web/static/ {**

**# cache static data**

**proxy\_cache\_valid 200 60m;**

**proxy\_buffering on;**

**expires 864000;**

**proxy\_pass http://backend-odoo;**

**}**

与此，静态数据缓存为 60 分钟。缓存将由 Nginx 直接回应这些请求在该区间上的进一步请求。

### 15.4.4 Long polling长轮循

长轮询用于支持即时消息应用程序，并使用多处理器工人时它在一个单独的端口，这是处理8072默认。

为我们反向代理服务器，这意味着，长轮询请求应传递到此端口。要支持此功能，我们需要添加新上游到我们 Nginx 配置中，如下面的代码所示︰

**upstream backend-odoo-im { server 127.0.0.1:8072; }**

下一步，我们应该另一个位置向服务器添加处理 HTTPS 请求，如下面的代码所示︰

**location /longpolling { proxy\_pass http://backend-odoo-im;}**

使用这些设置，Nginx 应将这些请求传递给正确的 Odoo 服务器端口。

## 15.5 服务器和模块更新

一旦 Odoo 服务器已准备好并且正在运行，则会那里当你需要在 Odoo 上安装更新的时间。这包括两个步骤︰ 第一，以获得新版本的源代码 （服务器或模块） 和第二，要安装它们。

如果您已经遵循*从源代码安装*一节中描述的方法，我们可以去取，然后在暂存库中测试最新版本。强烈建议您使生产数据库的副本和测试上它的升级。如果 odoo 产品是生产数据库，这可以通过下面的命令︰

**$ dropdb odoo-stage; createdb odoo-stage**

**$ pg\_dump odoo-prod | psql -d odoo-stage $ sudo su odoo**

**$ cd ~/.local/share/Odoo/filestore/**

**$ cp -al odoo-prod odoo-stage**

**$ ~/odoo-10.0/odoo-bin -d odoo-stage --xmlrpc-port=8080 -c**

**/etc/odoo/odoo.conf**

**$ exit**

如果一切顺利的话，它应该是安全生产服务执行升级。记住要记下当前版本 Git 参考以便能够通过再次签出此版本回滚。也强烈建议在执行升级之前保持数据库的备份。

**小贴士**

数据库副本可以更快的方式，使用以下显式命令︰

**$ createdb --template odoo-prod odoo-stage**

在这里需要说明的是，它的运行不能有任何打开的连接到 odoo 产品数据库，所以 Odoo 服务器需要停下来，要进行复制。

在此之后，我们可以拉到生产使用 Git 的存储库中的新版本，并完成升级，如下所示︰

**$ sudo su odoo**

**$ cd ~/odoo-10.0**

**$ git pull**

**$ exit**

**$ sudo service odoo restart**

关于 Odoo 释放政策，不再发布没有轻微的版本。GitHub 分支机构预计将代表最新的稳定版本。每夜构建被认为是最新的官方稳定版本。

更新的频率，就没有在更新太频繁，但也不用等了一年两次更新之间的点。执行更新每几个月应该是好的。重新启动服务器通常足以使代码更新和模块的升级应该有必要。

当然，如果你需要一个具体的 bug 修复，应该可能作出较早的更新。此外记得小心，安全 bug 披露在公众频道 — — GitHub 问题或社区邮件列表。作为服务的一部分，企业合同客户可以期望早期的电子邮件通知的这种类型的问题。

## 15.6 第十三章 小结

在这一章，你了解了设置和基于 Debian 的生产服务器中运行 Odoo 的额外步骤。最重要的设置在配置文件中被访问，和您学习了如何利用多处理器模式。

为提高的安全性和可扩展性，您还学习了如何使用 Nginx 作为我们 Odoo 服务器进程反向代理。

这应包括的要点什么需要运行 Odoo 服务器并向您的用户提供一个稳定和安全的服务。

了解更多有关 Odoo 你还应该看的官方的文件，在[https://www.odoo.com/documentation](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fwww.odoo.com%2Fdocumentation)。一些主题在那里，更多的细节，你还会发现这本书中不涉及的主题。

你也可能发现有用的 Odoo 还有其他出版的书。包

出版有几个在其目录中，和特别是*Odoo 发展食谱*提供更先进的材料，覆盖更多的主题不在这里讨论。

和 Odoo 最后，是一个充满活力的社区开源产品。介入、 提问和作出贡献是一个伟大的方式，不仅是学习，也是为了建立一个业务网络。对此我们不能帮助一提**Odoo 社区协会**(**OCA**)，促进协作和质量开放源码的代码。你可以了解更多关于它在[odoo community.org](http://www.microsofttranslator.com/bv.aspx?from=en&to=zh-CHS&a=https%3A%2F%2Fodoo-community.org%2F).