开源智造（OSCG）内部开发培训系列文档

Odoo开发指南

快速更新你的开发技能，构建强大的Odoo业务应用程序

老杨（杨浔波）

版权申明：本丛书一律免费共享，未经授权同意切勿商业应用，违者追究法律责任！

网站：www.oscg.cn 邮箱：sales@oscg.cn 电话：400 900 4680

前言

Odoo是一个强大的商业应用开源平台。在此基础上，构建了一套紧密集成的应用程序，涵盖了从CRM到销售到股票和会计的所有业务领域。Odoo有一个动态和不断增长的社区，不断增加功能、连接器和其他商业应用。

Odoo 10开发要点提供了一个逐步指导Odoo开发的指南，让读者能够快速的爬上学习曲线，并在Odoo应用平台上变得富有成效。

前两章的目的是让读者熟悉Odoo，学习建立开发环境的基本技术，熟悉模块开发方法和工作流。

以下各章节详细解释了Odoo addon模块开发所需的关键开发主题，如继承和扩展、数据文件、模型、视图、业务逻辑等等。

最后，最后一章解释了在部署Odoo实例时应该考虑什么。

本书教学大纲

第1章，开始了Odoo开发，从开发环境的设置开始，从源代码安装Odoo，并学习如何管理Odoo服务器实例。

第2章，构建您的第一个Odoo应用程序，指导我们创建第一个Odoo模块，涵盖涉及的所有不同层:模型、视图和业务逻辑。

第3章，继承——扩展现有的应用程序，解释现有的继承机制，以及如何使用它们创建扩展模块，在其他现有模块上添加或修改功能。

第4章，模块数据，包括最常用的Odoo数据文件格式(XML和CSV)，外部标识符概念，以及如何在模块和数据导入/导出中使用数据文件。

第5章，模型构建应用程序数据，详细讨论模型层，使用模型和字段的类型，包括关系和计算字段。

第6章，视图——设计用户界面，包括视图层，详细解释了几种类型的视图以及可以用来创建动态和直观的用户界面的所有元素。

第7章，ORM应用程序逻辑——支持业务流程，在服务器端引入编程业务逻辑，探索ORM概念和特性，并解释如何使用向导进行更复杂的用户交互。

第8章，编写测试和调试代码，讨论如何向addon模块添加自动化测试，以及调试模块业务逻辑的技术。

第9章，QWeb和看板视图，通过Odoo QWeb模板，使用它创建丰富的看板。

第10章，创建QWeb报告，讨论使用基于QWeb的报告引擎，以及生成友好的PDF报告所需要的一切。

第11章，创建网站前端功能，介绍了Odoo网站开发，包括web控制器实现和使用QWeb模板构建前端web页面。

第l2章，外部API——与其他系统集成，解释了如何从外部应用程序中使用Odoo服务器逻辑，并引入了一个受欢迎的客户端编程库，也可以作为命令行客户端使用。

第l3章，部署清单——现场直播，向我们展示了如何为生产黄金时间准备一个服务器，解释应该注意哪些配置，以及如何配置Nginx反向代理以提高安全性和可伸缩性。

本书的环境基础

我们将在Ubuntu或Debian系统上安装我们的Odoo服务器，但我们希望您使用您的操作系统和编程工具，无论是Windows、Mac还是其他。

我们将提供一些关于在Ubuntu服务器上设置虚拟机的指导。您应该选择使用的虚拟化软件，

例如VirtualBox或VMWare Player;两者都是免费的。如果您使用的是Ubuntu或Debian工作站，则不需要虚拟机。

正如您已经指出的，我们的Odoo安装将使用Linux，因此我们将不可避免地使用命令行。但是，你应该能够按照所给的指令行事，即使不熟悉它。

预期Python编程语言的基本知识。如果你不喜欢它，我们建议你学习快速教程，让你开始。我们还将使用XML，因此我们希望熟悉标记语法。

本书面向对象

这本书的目标是开发人员，他们有开发商业应用程序的经验，他们愿意快速地用Odoo来生产。

读者应该了解MVC应用程序设计和Python编程语言的知识。熟悉web技术、HTML、CSS和JavaScript也会有所帮助。

示例

在这本书中，你会发现许多不同种类的信息的文本样式。以下是这些风格的一些例子，以及它们的含义的解释。

文本中的代码单词、数据库表名、文件夹名称、文件名、文件扩展名、路径名、虚拟url、用户输入和Twitter句柄如下:文本中的代码字如下所示:“创建一个新的数据库，使用createdb命令。”

代码块设置如下:

```
@api.multi
def do_toggle_done(self):
    for task in self:
        task.is_done = not task.is_done
    return True
```

当我们希望将您的注意力吸引到代码块的某个特定部分时，相关的行或项以粗体设置:

```
@api.multi
def do_toggle_done(self):
    for task in self:
        task.is_done = not task.is_done
    return True
```

任何命令行输入或输出如下:

```
$ ~/odoo-dev/odoo/odoo-bin.py -d demo
```

新的术语和**重要的词**用粗体显示。例如，在屏幕上、菜单或对话框中看到的单词出现在这样的文本中:“在登录时，你会看到**Apps**菜单，显示可用的应用程序。”

警告或重要的音符出现在这样的盒子里。

提示和技巧就像这样。

1

开始使用Odoo开发

在进入Odoo开发之前，我们需要建立我们的开发环境，并学习它的基本管理任务。

在本章中，我们将学习如何设置工作环境，在这里我们将构建我们的Odoo应用程序。我们将学习如何设置Debian或Ubuntu系统来托管开发服务器实例，以及如何从GitHub源代码中安装Odoo。然后，我们将学习如何设置与Samba的文件共享，这将允许我们从运行Windows或任何其他操作系统的工作站运行Odoo文件。

Odoo是使用Python编程语言构建的，它使用PostgreSQL数据库进行数据存储;这些是Odoo主机的两个主要需求。要从源代码运行Odoo，我们首先需要安装它依赖的Python库。然后可以从GitHub下载Odoo源代码。虽然我们可以下载ZIP文件或tarball，但我们会看到，如果我们使用Git版本控制应用程序获取源代码会更好;它也会帮助我们把它安装在我们的Odoo主机上。

为Odoo服务器设置一个主机

一个Debian / Ubuntu系统被推荐用于Odoo服务器。你仍然可以在你最喜欢的桌面系统中工作，无论是Windows、Mac还是Linux。

Odoo可以在各种操作系统上运行，那么为什么要以牺牲其他操作系统为代价来选择Debian呢?因为Debian被认为是Odoo团队的参考部署平台;它有最好的支持。如果我们使用Debian / Ubuntu，它将更容易找到帮助和额外的资源。

它也是大多数开发人员工作的平台，大多数部署都是在这个平台上进行的。因此，不可避免的是，Odoo开发人员将会对Debian / Ubuntu平台感到满意。即使你是Windows背景的，你也要对它有所了解，这一点很重要。

在本章中，您将学习如何在基于debianbased的系统上设置和处理Odoo，只使用命令行。对于那些有Windows系统的家庭，我们将介绍如何设置虚拟机来托管Odoo服务器。作为一个额外的奖励，您将在这里学到的技术也将允许您在云服务器中管理Odoo，在那里您唯一的访问将通过**Secure Shell** (**SSH**)来访问。

请记住，这些指示是为了建立一个新的发展系统。如果您想在现有的系统中尝试其中的一些，总是提前进行备份，以便在出现问题时恢复它。

为Debian主机提供的服务

如前所述，我们需要一个基于debian-based的Odoo服务器主机。如果这是您第一次使用Linux，您可能会注意到Ubuntu是基于debianbased的Linux发行版，所以它们非常相似。

Odoo可以保证使用当前稳定版本的Debian或Ubuntu。在写作的时候，这些是Debian 8“Jessie”和Ubuntu 16.04.1 LTS(Xenial Xerus)。这两环境都有Python 2.7，这是运行Odoo的必要条件。值得一提的是，Odoo并不支持Python 3，因此需要Python 2。

如果你已经在运行Ubuntu或另一个基于debian-based的发行版，你就可以设置;这也可以作为Odoo的主机。
对于Windows和Mac操作系统，安装Python、PostgreSQL和所有依赖项;接下来，直接从源程序运行Odoo。然而，这可能是一个挑战，所以我们的建议是使用运行Debian或Ubuntu服务器的虚拟机。您可以选择您喜欢的虚拟化软件，以在虚拟机中获得一个工作的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。有几个选项，比如Microsoft hyper - v(在某些版本的Windows系统中可用)、Oracle VirtualBox和VMWare工作站播放器(Mac

的VMWare Fusion)。VMWare工作站的球员可能是更容易使用,并且免费下载可以在https://my.vmware.com/web/vmware/downloads上找到。

对于使用的Linux映像，安装Ubuntu服务器要比Debian更加友好。如果您从Linux开始，我建议您尝试使用现成的映像。TurnKey Linux提供了多种格式的易于使用的预安装映像，包括ISO。ISO格式将与您所选择的任何虚拟化软件一起工作，即使是在您可能拥有的裸金属机器上。一个很好的选择可能是第三方LAPP镜像,包括Python和PostgreSQL,可以在`http://www.turnkeylinux.org/lapp`找到。

一旦安装并启动，您应该能够登录到命令行shell。

为Odoo创建一个用户帐户

如果您正在登录使用超级用户`root`帐户，那么您的第一个任务应该是创建一个正常的用户帐户来使用您的工作，因为它被认为是不好的工作实践作为`root`。特别是，如果您将其作为`root`来启动，那么Odoo服务器将拒绝运行。

如果您正在使用Ubuntu，那么您可能不需要这个，因为安装过程必须已经通过创建一个用户来指导您。

首先，确保安装sudo。我们的工作用户将需要它。如果作为root登录，执行以下命令:

```
# apt-get update && apt-get upgrade  # 安装系统更新
# apt-get install sudo  # 确保安装“sudo”
```

下一组命令将创建一个odoo用户:

```
# useradd -m -g sudo -s /bin/bash odoo  # 创建一个具有sudo能力的“odoo”用户
# passwd odoo # 请求并为新用户设置密码
```

你可以将`odoo`转换为你想要的任何用户名。`-m`选项确保创建其主目录。`-g sudo`选项将它添加到*sudoers*列表中，以便它可以作为`root`运行命令。`-s /bin/bash`选项将默认的shell设置为bash，这比默认的`sh`要好。

现在我们可以作为新用户登录，并设置Odoo。

从源程序中安装Odoo

可以在`nightly.odoo.com`上找到现成的Odoo软件包，如Windows(`.exe`)、Debian(`.deb`)、CentOS(`.rpm`)和源代码tarballs(`.tar.gz`)。

作为开发人员，我们希望直接从GitHub存储库中安装它们。这将使我们对版本和更新有更多的控制。

为了保持整洁，在我们的主目录home内建立一个 `/odoo-dev`子目录以便进行工作。

在整本书中，我们假设`/odoo-dev` 是您的Odoo服务器安装的目录。

首先，确保您已经登录为我们现在或在安装过程中创建的用户，而不是作为`root`用户。假设您的用户是`odoo`，请使用以下命令确认:

```
$ whoami
odoo
$ echo $HOME
/home/odoo
```

现在我们可以使用这个脚本了。它向我们展示了如何将Odoo从源代码安装到Debian / Ubuntu系统中。

首先，安装基本的依赖项，以使我们开始:

```
$ 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及其包管理器
$ sudo ln -s /usr/bin/nodejs /usr/bin/node  # 调用节点运行nodejs
$ sudo npm install -g less less-plugin-clean-css  #安装less编译器
```

从版本9.0开始，Odoo web客户端需要在系统中安装`less` CSS预处理器，以便正确地呈现web页面。要安装这个，我们需要节点。Node.js和npm。

接下来，我们需要获得Odoo源代码并安装它的所有依赖项。Odoo源代码包括一个实用脚本，在`odoo/setup/`目录中，帮助我们在Debian / Ubuntu系统中安装所需的依赖项:

```
$ mkdir ~/odoo-dev  # Create a directory to work in
$ cd ~/odoo-dev  # Go into our work directory
$ git clone https://github.com/odoo/odoo.git -b 10.0 --depth=1  # Get Odoo
source code
$ ./odoo/setup/setup_dev.py setup_deps  # Installs Odoo system dependencies
$ ./odoo/setup/setup_dev.py setup_pg  # Installs PostgreSQL & db superuser
for unix user
```

最后，Odoo应该准备好使用。~符号是我们的主目录(例如，`/home/odoo`)的快捷方式。`git -b 10.0`选项告诉Git明确下载Odoo的10.0分支。在写的时候，这是多余的，因为10.0是默认的分支;然而，这可能会改变，因此它可能使脚本成为未来的证明。`--depth=1`选项告诉Git只下载最后一个版本，而不是完整的变更历史，使下载变得更小更快。

要启动一个Odoo服务器实例，只需运行:

```
$ ~/odoo-dev/odoo/odoo-bin
```

在Odoo 10中，在以前的版本中`odoo.py`脚本被`odoo-bin`替换，用于启动服务器。

在默认情况下，Odoo实例侦听端口8069，因此如果我们将浏览器指向`http://<server-address>:8069`，我们将到达这些实例。当我们第一次访问它时，它向我们展示了一个创建新数据库的助手，如下面的截图所示:

作为开发人员，我们需要使用几个数据库，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。现在在终端按*Ctrl + C*停止Odoo服务器并返回命令提示符。

初始化一个新的Odoo数据库

为了能够创建一个新的数据库，您的用户必须是一个PostgreSQL超级用户。下面的命令为当前的Unix用户创建一个PostgreSQL超级用户:

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

要创建一个新的数据库，请使用createdb命令。让我们创建一个demo数据库:

```
$ createdb demo
```

要使用Odoo数据模式初始化该数据库，我们应该使用-d选项在空数据库上运行Odoo:

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

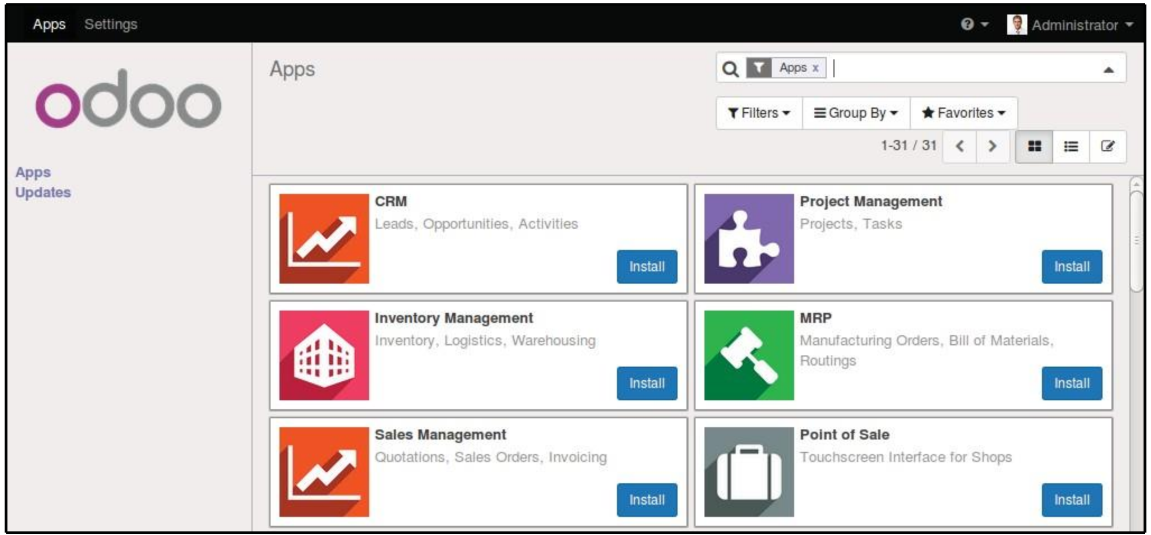
这将花费几分钟来初始化一个demo数据库，它将以一个信息日志消息结束，**Modules loaded**。

注意，它可能不是最后一个日志消息，它可以在最后三到四行。这样，服务器就可以准备好侦听客户端请求了。

默认情况下，这将用演示数据初始化数据库，这通常对开发数据库非常有用。若要初始化没有演示数据的数据库--without-demo-data=all。

现在打开http://<server-name>:8069，你的浏览器会被显示在登录屏幕上。如果您不知道您的服务器名称，在终端中键入hostname命令，以便找到它或ifconfig命令来查找IP地址。如果您在虚拟机中托管Odoo，您可能需要设置一些网络配置，以便能够从主机系统访问它。最简单的解决方案是将虚拟机网络类型从NAT改为桥接。这样，客户虚拟机就不会共享主机IP地址，而是拥有自己的IP地址。也可以使用NAT，但这需要您配置端口转发，这样您的系统就知道一些端口，比如8069，应该由虚拟机来处理。如果您遇到麻烦，希望这些细节将帮助您在您所选择的虚拟化软件的文档中找到相关信息。

默认管理员帐户是`admin`，其密码`admin`。登录后，你会看到**Apps**菜单，显示可用的应用程序:



当您想要停止Odoo服务器实例并返回到命令行时，请在bash提示符中按*Ctrl* + *C*。按下向上箭头键会给你带来先前的shell命令，所以这是一个快速启动Odoo的方法，同样的选项。按下*Ctrl* + *C*键和向上的箭头键，*Enter*是一个经常使用的组合，在开发期间重新启动Odoo服务器。

管理数据库

我们已经了解了如何从命令行创建和初始化新的Odoo数据库。有更多的命令值得管理数据库。

我们已经知道如何使用`createdb`命令创建空的数据库，但是它也可以通过复制现有的数据库创建一个新的数据库`--template`选项
确保您的Odoo实例被停止，并且您没有打开的其他连接我们刚刚创建的`demo`数据库，然后运行这个：

```
$ createdb --template=demo demo-test
```

实际上，每次创建数据库时，都会使用模板。如果没有指定，则使用预定义的`templatel`。

要列出系统中的现有数据库，可以使用`-l`选项使用PostgreSQL psql实用程序:

```
$ psql -l
```

运行它将列出我们迄今为止创建的两个数据库: `demo`和`demo-test`。该列表还将显示每个数据库中使用的编码。默认值是UTF-8，这是Odoo数据库所需的编码。
要删除不再需要的数据库(或者需要重新创建)来使用`dropdb`命令:

```
$ dropdb demo-test
```

现在您知道了使用数据库的基础知识。了解更多关于PostgreSQL,请参考官方文档:`http://www.postgresql.org/docs/`.

WARNING:
删除数据库命令将不可挽回地破坏您的数据。使用此命令时要小心，并且在使用此命令之前，总是要对重要的数据库进行备份。

一个关于Odoo产品版本

在写作的时候，Odoo的最新稳定版本是10版本，在GitHub上以10.0的形式标注。这是我们将在本书中使用的版本。

值得注意的是，Odoo数据库在Odoo主要版本之间不兼容。这意味着，如果您在一个以前的主要版本的Odoo创建的数据库上运行一个Odoo 10服务器，那么它就不会起作用。在使用该产品的后续版本之前，需要在数据库中使用非琐碎的迁移工作。

对于addon模块也是如此:作为一个通用规则，为Odoo主版本开发的addon模块将不会与其他版本一起工作。当从web下载一个社区模块时，确保它针对您正在使用的Odoo版本。

另一方面，主要的发行版(9.0,10.0)预计会收到频繁的更新，但这些更新大部分应该是bug修

复。它们被保证为“API稳定”，意味着模型数据结构和视图元素标识符将保持稳定。这很重要，因为它意味着上游核心模块中不兼容的更改将不会导致自定义模块的破坏。

请注意，`master`分支中的版本将导致下一个主要的稳定版本，但在此之前，它不是“API稳定”，您不应该使用它来构建自定义模块。这样做就像在流沙上移动:你不能确定什么时候会引入一些改变，这会破坏你的定制模块。

更多的服务器配置选项

Odoo服务器支持相当多的其他选项。我们可以查看所有可用选项`--help`:

```
$ ./odoo-bin --help
```

我们将在以下部分回顾一些最重要的选项。让我们从查看当前活动选项如何保存在配置文件中开始。

Odoo 服务器配置文件

大多数选项都可以保存在配置文件中。默认情况下，Odoo将使用`.odoorc`文件在您的主目录。在Linux系统中，它的默认位置是在`home`(`$HOME`)中，在Windows发行版中，它与用于启动Odoo的可执行文件处于同一目录。

在以前的Odoo / OpenERP版本中，缺省配置文件的名称为`.openerp-serverrc`。对于向后兼容性，Odoo 10仍然会使用它，如果它现在没有发现`.odoorc`文件。

在一个干净的安装上`.odoorc`配置文件不是自动创建的。我们应该使用`--save`选项创建默认配置文件，如果它还不存在，并将当前的实例配置存储到其中:

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

在这里，我们还使用了--stop-after-init选项，在服务器完成其操作后停止服务器。在运行测试或要求运行模块升级以检查安装是否正确时，通常使用此选项。

现在我们可以检查在这个默认配置文件中保存的内容:

```
$ more ~/.odoorc  # 显示配置文件
```

这将显示所有可使用其默认值的配置选项。在下一次启动Odoo实例时，编辑它们将是有效的。键入q退出并返回到提示符。

我们还可以选择使用一个特定的配置文件，使用--conf=<filepath>选项。配置文件不需要您刚才看到的所有选项。只有那些真正改变了默认值的才需要在那里。

改变监听端口

--xmlrpc-port=<port>命令选项允许我们更改服务器实例的监听端口，从默认的8069。这可以用于在同一台机器上同时运行多个实例。

让我们试试这个。打开两个终端窗口。首先，运行这个:

```
$ ~/odoo-dev/odoo/odoo-bin --xmlrpc-port=8070
```

在第二个终端上运行以下命令:

```
$ ~/odoo-dev/odoo/odoo-bin --xmlrpc-port=8071
```

在这里，两个Odoo实例在同一个服务器上监听不同的端口!这两个实例可以使用相同或不同的数据库，这取决于所使用的配置参数。这两个版本可以运行相同或不同版本的Odoo。

数据库过滤选项

与Odoo一起开发时，它经常与几个数据库一起工作，有时甚至使用不同的Odoo版本。在同

一个端口上停止和启动不同的服务器实例，并在不同的数据库之间切换，会导致web客户端会话的行为不正确。

使用在私有模式下运行的浏览器窗口访问我们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启用数据库过滤器,以确保它只允许我们想要处理的数据库的请求，而忽略所有其他的。这是用`--db-filter`选项完成的。它接受一个正则表达式作为有效数据库名称的过滤器。匹配一个确切的名字,表达应该开始`^`和结束`$`。

例如，为了只允许`demo`数据库，我们将使用这个命令:

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

管理服务器日志消息

`--log-level` 选项允许我们设置log verbosity。这对于了解服务器上正在发生的事情非常有用。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` 详细说明服务器接收到的请求

`debug_rpc_answer`详细说明服务器发送的响应

默认情况下，日志输出被定向到标准输出(您的控制台屏幕)，但是它可以直接指向一个日志文件，其中包含`--logfile=<filepath>`选项。

最后，当一个异常被提起时，`--dev=all`选项将打开Python调试器(`pdb`)。对服务器错误进行事后分析是很有用的。注意，它对logger verbosity没有任何影响。更多细节在Python调试器命令可以在`https://docs.python.org/2/library/pdb.html#debugger-commands`找到调试器命令。

从您的工作站

您可能正在使用一个Debian / Ubuntu系统在本地虚拟机或网络服务器上运行Odoo。但您可能更喜欢在您的个人工作站使用您喜欢的文本编辑器或IDE进行开发工作。对于Windows工作站的开发人员来说，这可能是常见的情况。但对于那些需要在本地网络上使用Odoo服务器的Linux用户来说，情况也是如此。

解决这个问题的方法是在Odoo主机中启用文件共享，这样就可以方便地从我们的工作站编辑文件。对于Odoo服务器操作，比如服务器重启，我们可以在我们最喜欢的编辑器旁边使用SSH shell(比如在Windows上的PuTTY)。

使用Linux文本编辑器

迟早，我们需要从shell命令行编辑文件。在许多Debian系统中，默认的文本编辑器是vi .如果您对它不满意，您可能可以使用更友好的选择。在Ubuntu系统中，默认的文本编辑器是nano。你可能更喜欢它，因为它更容易使用。如果您的服务器上没有可用的，可以安装:

```
$ sudo apt-get install nano
```

在下面的章节中，我们将假设nano是首选的编辑器。如果您喜欢其他的编辑器，请随意调整相应的命令。

安装和配置Samba

Samba服务帮助使Linux文件共享服务与Microsoft Windows系统兼容。我们可以在Debian / Ubuntu服务器上安装这个命令:

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

samba包安装了文件共享服务，`smbpasswd`工具需要使用`samba-common-bin`包。默认情况下，允许访问共享文件的用户需要注册。我们需要注册我们的用户`odoo`，并为其文件共享访问

设置密码:

```
$ sudo smbpasswd -a odoo
```

在此之后，我们将被要求使用一个密码来访问共享目录，而odoo用户将能够访问其主目录的共享文件，尽管它只会被读取。我们想要有写访问权限，因此我们需要编辑Samba配置文件，以更改以下内容:

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

在配置文件中，查找[homes]部分。编辑其配置行，使其与以下设置匹配:

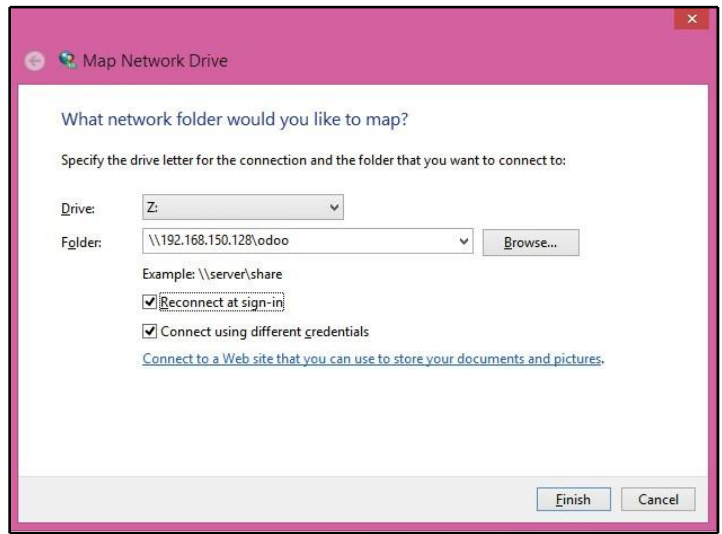
```
[homes]
    comment = Home Directories
    browseable = yes
    read only = no
    create mask = 0640
    directory mask = 0750
```

对于配置更改生效，重新启动服务:

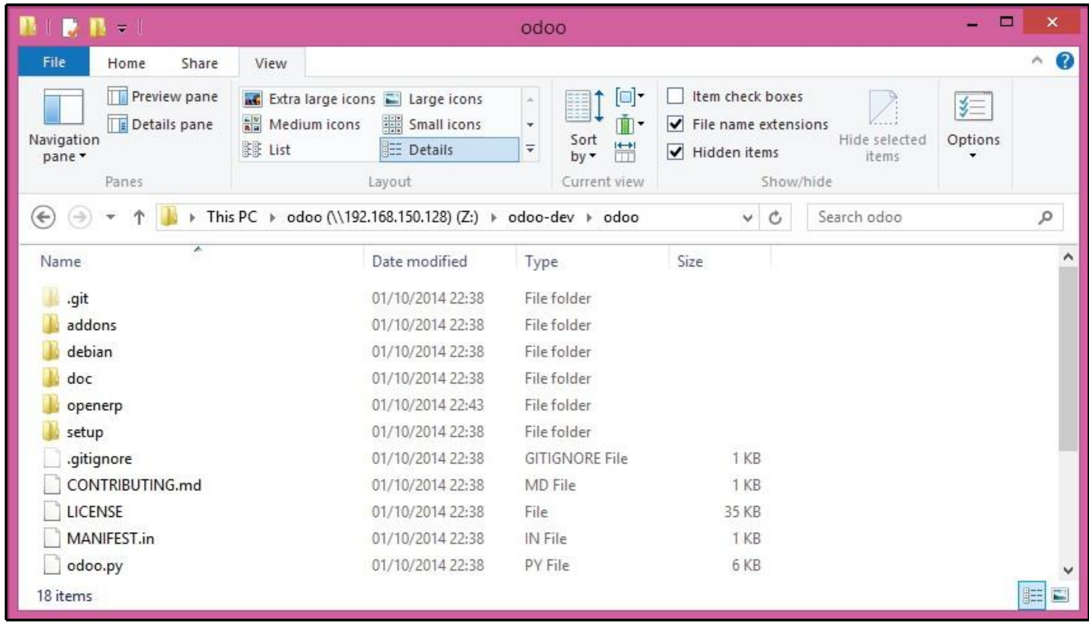
```
$ sudo /etc/init.d/smbd restart
```

要从Windows中访问这些文件，我们可以使用与smbpasswd定义的特定的用户名和密码，将网络驱动器映射为\\<my- server-name>\odoo。当试图与odoo用户登录时，您可能会遇到Windows向用户名(例如MYPC\odoo)添加计算机域的问题。为了避免这种情况，请在登录时使用一个空域(例如，\odoo):

OSCG
開源智造


Map Network Drive
What network folder would you like to map?
Specify the drive letter for the connection and the folder that you want to connect to:
Drive:
Z:
Folder:
\\192.168.150.128\odoo
Browse...
Example: \\server\share
Reconnect at sign-in
Connect using different credentials
Connect to a Web site that you can use to store your documents and pictures.
Finish
Cancel

如果我们现在使用Windows资源管理器打开映射驱动器，我们将能够访问和编辑odoo用户的主目录的内容:

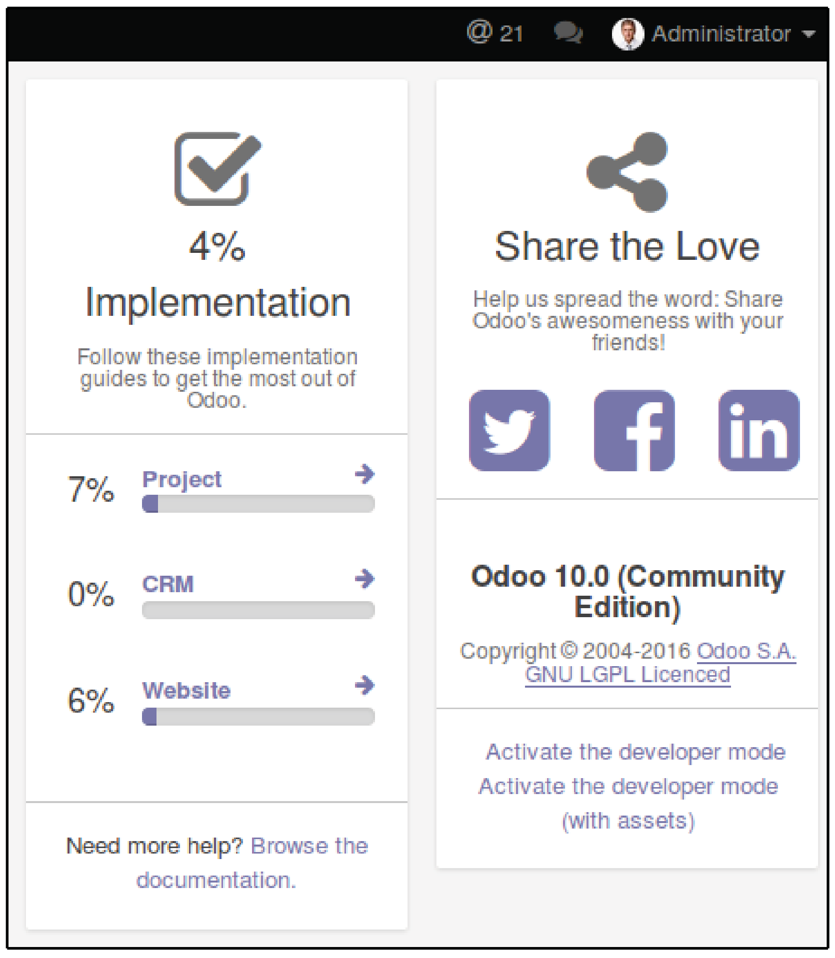


Odoo包括一些对开发人员非常有用的工具，我们将在本书中使用它们。它们是技术特性和开发模式。默认情况下，这些都是禁用的，所以这是一个学习如何启用它们的好时机。

激活开发工具

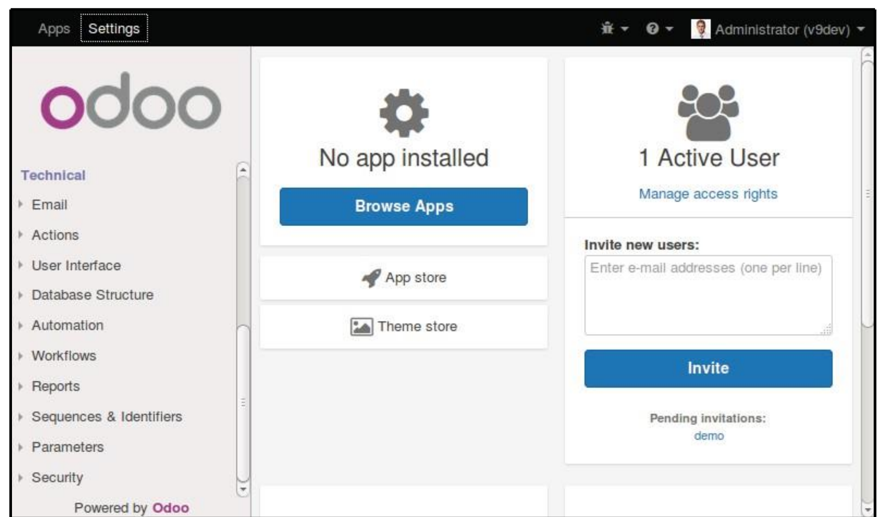
开发工具提供高级服务器配置和特性。其中包括顶部菜单栏中的一个调试菜单，以及**Settings**菜单中的其他菜单选项，特别是**Technical**菜单。

这些工具在默认情况下是禁用的，并且为了启用它们，我们需要以admin身份登录。在顶部菜单栏中，选择**Settings**菜单。在底部右侧，在Odoo版本之下，您将找到两个选项来启用开发者模式;它们中的任何一个都可以Debug和Technical。第二个选项，**Activate the developer mode (with assets)**，也禁用了web客户端使用的JavaScript和CSS，这样可以更容易地调试客户端行为:



在此之后，页面被重新加载，您应该在顶部菜单栏中看到一个bug图标，就在会话用户头像和提

供调试模式选项的名称之前。在顶部菜单的**Settings**选项中，我们应该看到一个新的**Technical**菜单部分，可以访问许多Odoo实例内部:



Technical菜单选项允许我们检查和编辑存储在数据库中的所有Odoo配置，从用户界面到安全性和其他系统参数。在这本书中，你将会学到更多。

安装第三方模块

在一个Odoo实例中提供新的模块，这样它们就可以安装，这是一个新手经常会感到困惑的东西。但不一定非得如此，让我们来揭开它的神秘面纱吧。

Finding community modules

互联网上有很多的Odoo模块。`apps.odoo.com`是一个可以在你的系统上下载和安装的模块目录。**Odoo Community Association** (**OCA**)协调社区贡献，并在`https://github.com/OCA/`上维护了GitHub上的一些模块存储库。

要在Odoo安装中添加一个模块，我们可以将它复制到addons目录中，并与官方模块一起使用。在我们的案例中，addons目录位于~/odoo- dev/odoo/addons/。对于我们来说，这可能不是最好的选择，因为我们的Odoo安装是基于版本控制的代码存储库，我们将希望它与GitHub存储库保持同步。

幸运的是，我们可以为模块使用额外的位置，这样我们就可以将自定义模块保存在不同的目录中，而不需要将它们与官方的模块混合在一起。

作为一个例子，我们将从这本书中下载代码，在GitHub中提供，并使那些addon模块在我们的Odoo安装中可用。

要从GitHub获得源代码，运行以下命令:

```
$ cd ~/odoo-dev
$ git clone https://github.com/dreispt/todo_app.git -b 10.0
```

我们使用-b选项确保我们正在下载10.0版本的模块。

在此之后，我们将在/odoo目录旁边有一个新的/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。

配置插件的路径

Odoo服务器有一个名为addons_path的配置选项，用于设置服务器应该在哪里查找模块。默认情况下，这指向/addons目录，在那里，Odoo服务器正在运行。

我们不仅可以提供一个目录，还可以提供一个目录列表，其中可以找到模块。这允许我们将自定义模块保存在一个不同的目录中，而不让它们与正式的addons混合。

让我们以一个包含我们的新模块目录的addons路径启动服务器:

```
$ cd ~/odoo-dev/odoo
$ ./odoo-bin -d demo --addons-path="../todo_app,./addons"
```

如果您仔细查看服务器日志，您将注意到一条正在使用的addons路径的行: INFO? odoo:

addons paths: [...]. 确认它包含我们的todo_app目录。

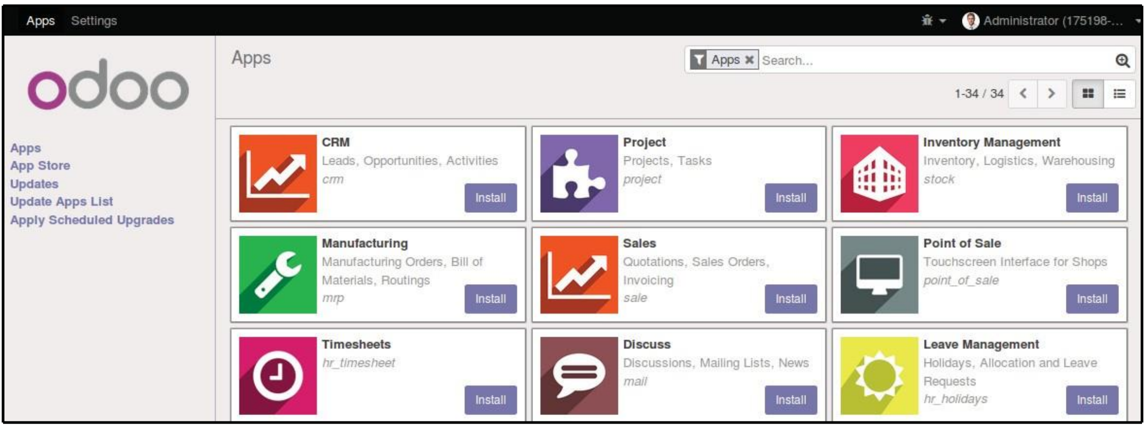
更新应用程序列表

我们还需要让Odoo在这些新模块提供安装之前更新它的模块列表。

为此，我们需要启用开发者模式，因为它提供了**Update Apps List**菜单选项。它可以在**Apps**头部菜单中找到。

更新模块列表后，我们可以确认新的模块可供安装。使用**Apps**菜单选项查看本地模块列表。搜索todo，您应该看到可以使用的新模块。

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



摘要

在本章中，我们学习了如何设置一个Debian系统来托管Odoo并将其安装到GitHub源代码中。我们还学习了如何创建Odoo数据库并运行Odoo实例。为了允许开发人员在其个人工作站上使用他们最喜欢的工具，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良好的Odoo环境，并对管理数据库和实例感到满意。

有了这个，我们就可以直接开始行动了。在下一章中，我们将从头开始创建我们的第一个Odoo模块，并理解它所涉及的主要元素。

所以让我们开始吧!

2

构建您的第一个Odoo应用程序

在Odoo中开发大部分时间意味着创建我们自己的模块。在本章中，我们将创建我们的第一个Odoo应用程序，并学习使它可用于Odoo并安装它所需的步骤。

在著名的`http://todomvc.com/` 项目的启发下，我们将构建一个简单的To-Do应用程序。它应该允许我们添加新任务，标记它们完成，最后清除所有已完成任务的任务列表。

我们将开始学习开发工作流程的基础:为您的工作建立一个新实例，创建并安装一个新模块，并更新它以应用您在开发迭代中所做的更改。

Odoo遵循一个类似于mvc的架构，我们将在实现To-Do应用程序的过程中仔细检查这些层。

model层，定义应用程序数据的结构

view层，描述用户界面

controller层，支持应用程序的业务逻辑

接下来，我们将学习如何设置访问控制安全性，最后，我们将向模块添加一些描述和品牌信息。

注意这里提到的术语控制器的概念不同于Odoo web开发控制器。这些是web页面可以调用来执行操作的程序端点。

通过这种方法，您将能够逐步了解构成应用程序的基本构建块，并体验从头构建一个Odoo模块的迭代过程。

基本概念

您可能刚刚开始使用Odoo，所以现在很明显是解释Odoo模块的好时机，以及它们是如何在Odoo开发中使用的。

理解应用程序和模块

关于Odoo模块和应用程序很常见。但它们之间的区别到底是什么呢?

Module addons是Odoo应用程序的构建块。一个模块可以向Odoo添加新功能，或者修改现有的功能。它是一个目录，包含一个名为`_manifest_.py`文件，加上实现其特性的其余文件。

Applications是将主要特性添加到Odoo的方式。它们提供了功能领域的核心元素，如会计或HR，基于附加的addon模块修改或扩展特性。因此，它们在Odoo **Apps**菜单中突出显示。

如果您的模块是复杂的，并向Odoo添加新的或主要功能，您可以考虑将其作为应用程序创建。如果您的模块只是对Odoo中的现有功能进行了更改，那么它很可能不是一个应用程序。

模块是否为应用程序，在清单中定义。从技术上讲，它对addon模块的行为没有任何特别的影响。它只是**Apps**列表中的一个亮点。

修改和扩展模块

在我们将要学习的示例中，我们将创建一个新的模块，它的依赖关系尽可能少。

然而，这不是典型的情况。大多数情况下，我们将修改或扩展已经存在的模块。

一般来说，通过直接修改源代码来修改现有模块是一种很糟糕的做法。这对于Odoo提供的官方模块尤其如此。这样做不允许您在原始的模块代码和修改之间有清晰的分离，这使得应用升级变得困难，因为它们将覆盖修改。

相反，我们应该创建在我们想要修改的模块旁边安装的扩展模块，实现我们需要的更改。事实上，Odoo的主要优势之一是**inheritance**机制，它允许定制模块扩展现有模块，无论是正式的还是社区的。继承在所有级别都是可能的：数据模型、业务逻辑和用户界面层。

在本章中，我们将创建一个全新的模块，不扩展任何现有的模块，只关注模块创建中涉及的不同部分和步骤。我们将对每个部分进行简单的研究，因为每个部分将在后面的章节中详细研究。

一旦我们熟悉了创建一个新模块，我们就可以深入到继承机制，这将在第3章中介绍，继承-扩展现有的应用程序。

为了使Odoo得到有效的开发，我们应该对开发工作流程感到满意:管理开发环境，应用代码更改，并检查结果。本节将介绍这些基础知识。

创建模块基本骨架

按照第一章的说明，开始使用Odoo开发，我们应该有Odoo服务器在~/odoo-dev/odoo/。为了使事情保持整洁，我们将在它旁边创建一个新的目录来托管我们的自定义模块，在~/odoo-dev/custom-addons中。

Odoo包括一个scaffold命令来自动创建一个新的模块目录，它的基本结构已经就绪。您可以通过以下命令了解更多信息:

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

当您开始编写下一个模块时，您可能想要记住这一点，但我们现在不会使用它，因为我们更喜欢手动创建模块的所有结构。

一个Odoo addon模块是一个包含_manifest_.py描述文件。

在以前的版本中，这个描述符文件被命名为_openerp_.py。这个名称仍然被支持，但已被弃用。

它还需要是Python可输入的，所以它也必须有一个_init_.py文件。

模块的目录名是它的技术名称。我们将使用todo_app来完成它。技术名称必须是一个有效的Python标识符:它应该以字母开头，只能包含字母、数字和下划线字符。
下面的命令将创建模块目录并创建一个空_init_.py文件在~/odoo-dev/custom-addons/todo_app/_init_.py里面

如果你想直接从命令行做到这一点，这就是你要使用的:

```
$ mkdir ~/odoo-dev/custom-addons/todo_app
$ touch ~/odoo-dev/custom-addons/todo_app/_init__.py
```

接下来，我们需要创建清单文件。它应该只包含一个包含十几个可能属性的Python字典;其中，只有name属性是必需的。description描述属性，用于更长的描述，并且author

属性提供更好的可视性和建议。

我们现在应该添加一个_manifest_.py文件在这个文件边上是_init_.py文件，这个_manifest_.py文件内容如下:

```
(
    'name': 'To-Do Application', 'description':
    'Manage your personal To-Do tasks.',
    'author': 'Daniel Reis',
    'depends': ['base'],
    'application': True,
}
```

depends属性可以拥有其他需要的模块列表。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，但建议总是拥有它。如果不需要特殊的依赖关系，那么我们应该依赖于核心base模块。
您应该注意确保在这里显式地设置所有依赖项;否则，该模块可能无法在干净的数据库中安装(由于缺少依赖关系)，或者在随后加载其他必需的模块时加载错误。

对于我们的应用程序，我们不需要任何特定的依赖项，因此我们依赖于base模块。

为了简洁，我们选择使用非常少的描述符键，但是，在一个真实的应用场景中，我们建议您也使用额外的键，因为它们与Odoo应用程序商店相关:

- summary 显示为模块的副标题。
- version 默认情况下，是1.0。它应该遵循语义版本规则(参见 http://semver.org/ 以获得详细信息)。
- license 标识符默认是LGPL-3。
- website 是一个URL，用于查找关于模块的更多信息。这可以帮助人们找到更多的文档或问题跟踪器来归档bug和建议。
- category 是模块的功能类别，默认为未分类。在**Application**字段下拉列表中，可以在安全组表单(**Settings** | **User** | **Groups**)中找到现有类别的列表。

这些其他描述符键也可用:

- installable 默认为True，但可以设置为False来禁用模块。

`auto_install` 如果设置为`True`，该模块将自动安装，提供所有依赖项已经安装。它是用来做胶水的模块。

由于Odoo 8.0，而不是`description`键，我们可以使用`README.rst`或`README.md`文件在模块的顶部目录。

对许可证

为你的工作选择一个许可是非常重要的，你应该仔细考虑什么是你最好的选择，以及它的含义。Odoo模块最常用的许可证是**GNU Lesser General Public License(LGLP)**和**Affero General Public License(AGPL)**。LGPL更加宽容，允许商业派生工作，而不需要共享相应的源代码。AGPL是一个更强的开源许可，它需要派生的工作和服务托管来共享它们的源代码。在https://www.gnu.org/licenses/了解更多关于GNU许可证。

添加到addons路径

现在我们有一个简约的新模块,我们想让它可用Odoo实例。

为此，我们需要确保包含模块的目录位于addons路径中，然后更新Odoo模块列表。

这两个动作在前一章都有详细的解释，但是在这里，我们将继续简要概述一下需要做什么。

我们将在工作目录中定位，并以适当的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。每次重新启动服务器时，都要避免重复它们:
运行`./odoo-bin`和最后一个保存的选项将被使用。
仔细查看服务器日志。它应该有一个信息`INFO ? odoo: addons paths:[...]`行，它应该包括我们`custom-addons`目录。
记住，也要包含您可能正在使用的其他addon目录。例如，如果您还有一个`~/odoo-dev/extra`的目录，其中包含了额外的模块 您可能希望将它们包括在`--addons-path`选项中:

```
--addons-path="custom-addons,extra,odoo/addons"
```

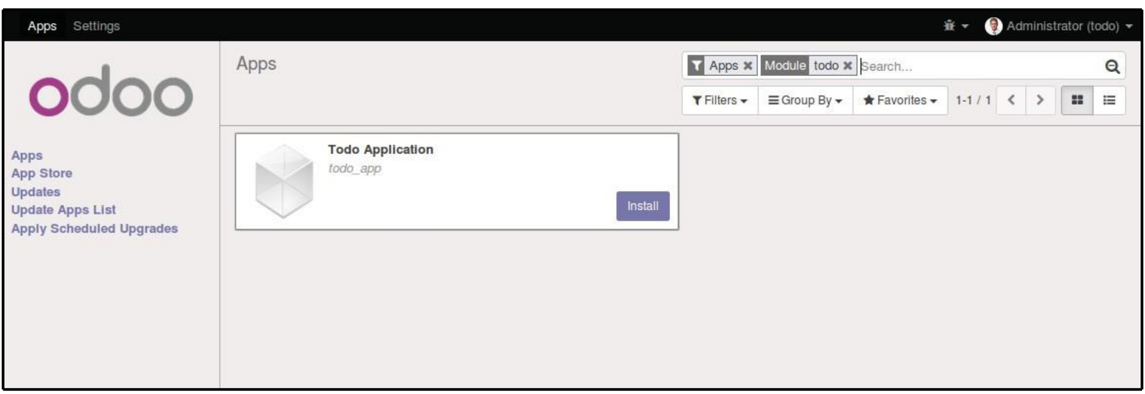
现在我们需要Odoo实例来确认刚刚添加的新模块。

安装新模块

在**Apps**菜单中，选择**Update Apps List**选项。这将更新模块列表，添加自上次更新后可能添加的任何模块。请记住，我们需要为该选项启用的开发模式是可见的。这是完成的**Settings**指示板，在链接的底部右侧，在Odoo版本号信息下面。

确保您的web客户端会话与正确的数据库一起工作。您可以在右上角检查:数据库名称在括号中显示，在用户名之后。使用正确的数据库执行的一种方法是使用附加选项启动服务器实例 `--db-filter=^MYDB$`.

Apps选项显示了可用模块的列表。默认情况下，它只显示应用程序模块。由于我们已经创建了一个应用程序模块，所以我们不需要删除该过滤器来查看它。键入`todo`在搜索中，您应该看到我们的新模块，准备安装:



现在点击模块的**Install**按钮，我们准备好了!

升级模块

开发一个模块是一个迭代的过程，您需要对源文件进行修改，并在Odoo中进行可见。

在许多情况下，这是通过升级模块来完成的:在**Apps**列表中查找模块，一旦它已经安装，您将有一个可用的**Upgrade**按钮。

然而，当更改仅在Python代码中进行时，升级可能没有效果。不需要进行模块升级，需要重新启动应用程序服务器。由于Odoo只装载一次Python代码，因此，任何后来的代码更改都需要重新启动服务器。

在某些情况下，如果模块更改在数据文件和Python代码中，那么您可能需要两个操作。对于新的Odoo开发人员来说，这是一个常见的混淆来源。

但幸运的是，还有更好的方法。使我们对模块有效的所有更改的最安全、最快的方法是停止并重新启动服务器实例，请求将我们的模块升级到我们的工作数据库。

在服务器实例正在运行的终端中，使用*Ctrl + C*来停止它。然后，使用以下命令启动服务器并升级`todo_app`模块:

```
$ ./odoo-bin -d todo -u todo_app
```

`-u` 选项(或者，从`--update`的更新)需要`-d`选项，并接受一个逗号分隔的模块列表进行更新。例如，我们可以使用`-u todo_app,mail`。当一个模块被更新时，所有其他安装的模块也会被更新。这对于维护继承机制的完整性是至关重要的。

在本书中，当您需要应用模块中的工作时，最安全的方法是用前面的命令重新启动Odoo实例。按下向上箭头键会给你带来以前使用过的命令。所以，大多数时候，你会发现自己使用*Ctrl + C*，然后输入键组合。

不幸的是，两个更新模块列表和卸载模块都是无法通过命令行获得的操作。这些需要通过**Apps**菜单中的web界面完成。

服务器开发模式

在Odoo 10中，一个新的选项被引入提供开发友好的特性。要使用它，可以使用附加选项启动服务器实例`--dev=all`。

这使得一些方便的特性能够加速我们的开发周期。最重要的是:

自动重新加载Python代码，一旦保存了Python文件，避免手动服务器重新启动

直接从XML文件读取视图定义，避免手动模块升级

`--dev`选项接受一个以逗号分隔的选项列表，尽管`all`选项在大多数情况下都是合适的。我们还可以指定我们喜欢使用的调试器。默认情况下，使用Python调试器`pdb`。有些人可能更喜欢安装和使用其他调试器。这里也支持`ipdb`和`pudb`。

model层

现在，Odoo知道了我们的新模块，让我们从添加一个简单的模型开始。

模型描述业务对象，例如机会、销售订单或合作伙伴(客户、供应商等)。模型有一个属性列表，也可以定义它的特定业务。

模型是使用从一个Odoo模板类派生的Python类实现的。它们直接转换到数据库对象，Odoo在安装或升级模块时自动处理这些问题。对此负责的机制是**Object Relational Model** (**ORM**)。

我们的模块将是一个非常简单的应用程序，以保存待办事项。这些任务将有一个单独的文本字段用于描述和一个复选框来标记它们。我们应该稍后添加一个按钮来清理已完成任务的待办事项列表。

创建数据模型

Odoo开发准则规定，模型的Python文件应该放在一个模型子目录中。为了简单起见，我们不会在这里跟踪它，所以我们创建一个`todo_model.py`文件位于`todo_app`模块的主目录中。

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，它告诉Python解释器这个文件有UTF-8，这样它就可以预期和处理non-ASCII字符。我们不会使用任何东西，但无论如何这是一个很好的实践。
第二行是Python代码导入语句，可以从Odoo内核中获取`models`和`fields`对象。

第三行声明了我们的新模型。这是一个来自`models.Model`的类。

下一行设置了定义标识符的`_name`属性,该属性将在整个Odoo中使用,以引用该模型。注意,在本例中，实际的Python类名称`TodoTask`对其他Odoo模块没有意义。`_name`值将用作标识符。

注意，这个和下面的行是缩进的。如果您不熟悉Python，您应该知道这一点很重要:缩进定义了一个嵌套的代码块，因此这四行应该都是同样的缩进。

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，可以用于更好的用户消息。

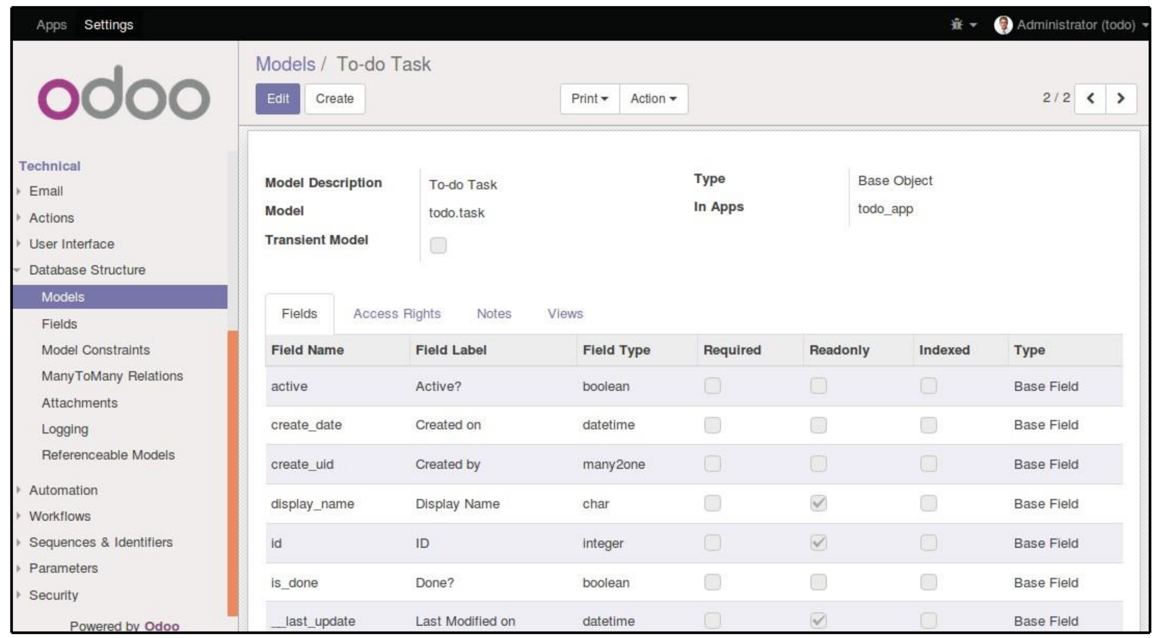
最后三行定义了模型的字段。值得注意的是，`name`和`active`是特殊的字段名。在默认情况下，Odoo在引用其他模型时将使用`name`字段作为记录的标题。`active`字段用于激活记录，默认情况下只显示活动记录。我们将使用它清除完成的任务，而不实际从数据库中删除它们。

现在，这个文件还没有被模块使用。我们必须告诉Python在`_init_.py`文件中使用模块加载它。让我们编辑它来添加以下一行:

```
from . import todo_model
```

就是这样!对于我们的Python代码更改生效，服务器实例需要重新启动(除非使用`--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。不过，我们可以使用**Technical**菜单来检查新创建的模型。在**Settings**顶部菜单，去**Technical | Database Structure | Models**，搜索`todo.task`模型在列表上，点击它看它的定义：



如果一切顺利，模型和字段就会被创建。如果您在这里看不到它们，可以尝试使用一个模块升级来重启服务器，如前所述。

我们还可以看到一些我们没有声明的字段。这些是保留的字段Odoo自动添加到每个新模型。他们如下：

`id`是模型中每个记录的惟一数字标识符。

`create_date`和`create_uid`分别指定创建记录和创建记录的时间。

`write_date`和`write_uid`在记录最后修改和修改时确认。

`____last_update`是一个没有实际存储在数据库中的助手。它用于并发检查。

添加自动化测试

编程最佳实践包括对代码进行自动化测试。对于像Python这样的动态语言来说，这一点更为重要。因为没有编译步骤，所以您不能确定没有语法错误，直到代码由解释器运行。一个好的编辑器可以帮助我们提前发现这些问题，但是不能帮助我们确保代码执行得像自动化测试一样。

Odoo支持两种方法来描述测试:使用YAML数据文件或使用Python代码，基于Unittest2库。YAML测试是旧版本的遗留问题，不推荐使用。我们更喜欢使用Python测试，并将一个基本的测试用例添加到我们的模块中。

测试代码文件应该以test_开头，并且应该从tests/_init_.py导入。但是tests目录(或Python子模块)不应该从模块的顶部_init_.py导入，因为它将在测试执行时自动被发现和加载。

测试必须放置在一个tests/子目录中。添加一个tests/_init_.py文件如下:

```
from . import test_todo
```

现在添加实际的测试代码，在tests/test_todo.py文件:

```
# -*- 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, False)
```

这就添加了一个简单的测试用例来创建一个新to-do task，并验证所做的工作是否**Is Done?**字段具有正确的默认值。

现在我们要运行我们的测试。这是通过在安装模块时添加--test-enable选项来实现的:

```
$ ./odoo-bin -d todo -i todo_app --test-enable
```

Odoo服务器将在升级后的模块中查`tests/`子目录，并运行它们。如果其中任何一个测试失败，服务器日志将显示这一点。

view层

视图层描述用户界面。视图是使用XML定义的，它由web客户端框架使用，以生成具有数据感知的HTML视图。
我们有菜单项可以激活可以呈现视图的动作。例如, **Users**菜单项处理一个名为**Users**的操作，依次呈现一系列视图。有一些可用的视图类型，例如列表和表单视图，并且提供的筛选器选项也由特定类型的视图(search视图)定义。

Odoo开发指南指出定义用户界面的XML文件应该放在一个`views/`子目录中。

让我们开始为To-Do应用程序创建用户界面。

添加菜单项

现在我们有了存储数据的模型，我们应该在用户界面上提供它。

为此，我们应该添加一个菜单选项来打开要做的`To-do Task`模型，这样它就可以被使用。

创建`views/todo_menu.xml`文件定义一个菜单项和由它执行的操作:

```
<?xml version="1.0"?>
<odoo>
  <!-- Action to open To-do Task list -->
  <act_window id="action_todo_task"
    name="To-do Task"
    res_model="todo.task"
    view_mode="tree,form" />
  <!-- Menu item to open To-do Task list -->
  <menuitem id="menu_todo_task"
    name="Todos"
```

```
        action="action_todo_task" />
</odoo>
```

用户界面(包括菜单选项和操作)存储在数据库表中。XML文件是用于在安装或升级模块时将这些定义加载到数据库中的数据文件。前面的代码是一个Odoo数据文件，描述两个记录添加到Odoo:

- `<act_window>`元素定义了一个客户端窗口操作，可以打开`todo.task`模型，并在这个顺序中启用了`tree`和`form`视图。
- `<menuitem>`定义了一个顶级菜单项，调用`action_todo_task`操作，这是以前定义的。

这两个元素都包含`id`属性。这个id属性也被称为**XML ID**，它非常重要:它被用来唯一标识模块内部的每个数据元素，并可以由其他元素来引用它。在这种情况下，`<menuitem>`元素需要引用操作来处理，并且需要使用`<act_window>`ID。在第4章，模块数据中，XML id被更详细地讨论。

我们的模块还不知道新的XML数据文件。这是通过将其添加到`_manifest_.py`文件中的`data`属性完成的。它保存由模块加载的文件列表。将此属性添加到manifest的字典:

```
'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。去**Todos**顶部菜单，你应该看到我们的新菜单选项:

即使我们没有定义我们的用户界面视图，点击**Todos**菜单会为我们的模型打开一个自动生成的表单，允许我们添加和编辑记录。

Odoo很好，可以自动生成它们，这样我们就可以立即开始使用我们的模型了。

到目前为止还好!现在让我们改进我们的用户界面。试着逐步改进，如下一节所示，做频繁的模块升级，不要害怕尝试。您可能还想尝试`--dev=all` 服务选项。使用它，视图定义可以直接从XML文件中读取，这样您的更改就可以在不需要模块升级的情况下立即可用到Odoo。

如果因为XML错误而升级失败，请不要惊慌!请注释掉最后编辑的XML部分，或者从`__manifest__.py`中删除XML文件并重复升级。服务器应该正确启动。现在仔细阅读服务器日志中的错误消息:它应该指出问题所在。

Odoo支持多种类型的视图，但最重要的三个视图是: `tree` (通常称为列表视图)、`form`和`search`视图。我们将为我们的模块添加一个示例。

创建form视图

所有视图都存储在数据库中，在`ir.ui.view`模型中。为了向模块添加视图，我们声明了一个`<record>`元素，描述了XML文件中的视图，当模块安装时，它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>
```

请记住将这个新文件添加到清单文件中的数据键，否则，我们的模块将不知道它，它将不会被加载。

这将向`ir.ui.view`模型添加一个记录，并使用标识符`view_form_todo_task`。视图是`todo.task`模型，命名为`To-do Task Form`。这个名字只是为了获取信息;它不必是唯一的，但它应该允许一个人很容易地识别它所指的记录。实际上，这个名称可以完全省略，在这种情况下，它将由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>`标记定义了视图类型，在本例中，它包含三个字段。我们还向活动字段添加了一个属性，使其仅读取。

业务form文档视图

前面的部分提供了一个基本的表单视图，但是我们可以对它进行一些改进。对于文档模型，Odoo有一个模仿纸质页面的演示样式。该表单包含两个元素：`<header>`包含操作按钮和`<sheet>`以包含数据字段。

我们现在可以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>
```

添加action按钮

表单可以有执行动作的按钮。这些按钮可以运行窗口操作，例如打开另一个表单或运行在模型中定义的Python函数。

它们可以放在表单的任何位置，但是对于文档样式表单，推荐的位置是<header>部分。

对于我们的应用程序，我们将添加两个按钮来运行todo.task模型的方法:

```
<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>
```

按钮的基本属性包括以下内容:

- string 文本显示在按钮上
- type 它执行的动作
- name 这个动作的标识符是什么
- class 是否可以使用CSS样式的可选属性，比如在普通HTML中

使用组来组织表单

<group>标记允许您组织表单内容。在<group>元素内放置<group>元素，在外部组内创建一个两列布局。建议组元素有一个name属性，以便其他模块可以更容易地扩展它们。

我们将使用这个来更好地组织我们的内容。让我们改变表格的<sheet>内容来匹配这个:

```
<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>
```

完整的form视图

在这一点上，`todo.task`表单视图应该是这样的:

```
<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>
```

请记住，对于要加载到Odoo数据库的更改，需要进行模块升级。要查看web客户端的变化，需要重新加载表单:要么单击打开它的菜单选项，要么重新加载浏览器页面(大多数浏览器中的*F5*)。

操作按钮还不能工作，因为我们还需要添加它们的业务逻辑。

添加列表和search视图

当在列表模式中查看模型时，使用<tree>视图。树视图能够显示在层次结构中组织的行，但大多数时候，它们被用来显示普通列表。

我们可以将以下tree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>
```

这定义了一个只有两列的列表：name和is_done。我们还添加了一个漂亮的触摸:完成任务的行(is_done==True)显示为灰色。这是通过使用Bootstrap类muted来完成的。检查http://getbootstrap.com/css/#helper-classes-colors更多信息引导和它的背景颜色。

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由<search>视图定义。

如前所述，我们将将其添加到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>元素定义在搜索框中输入的字段。<filter>元素添加预定义的筛选条件，可以通过使用特定语法定义的用户单击进行切换。

业务逻辑层

现在我们将在按钮上添加一些逻辑。这是用Python代码完成的，在模型的Python类中使用方法。

添加业务逻辑

我们应该编辑`todo_model.py`Python文件以向类添加按按钮调用的方法。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按钮的作用将非常简单:只需切换**Is Done**?标记。对于记录的逻辑，使用`@api.multi`装饰器。在这里，`self`将代表一个记录集，然后我们应该对每个记录进行循环。

在`TodoTask`类内，添加以下内容:

```
@api.multi
def do_toggle_done(self):
    for task in self:
        task.is_done = not task.is_done
    return True
```

代码循环遍历所有的to-do记录，并为每个任务记录修改`is_done`字段，从而改变其值。这个方法不需要返回任何东西，但是我们应该至少返回一个`True`值。原因是，客户机可以使用XML-RPC调用这些方法，而这个协议不支持只返回`None`值的服务器函数。

对于**Clear All Done**按钮，我们想更进一步。它应该查找已经完成的所有活动记录，并使它们不活动。通常情况下，表单按钮只会在选定的记录上执行，但在这种情况下，我们希望它也能根据当前的记录采取行动:

```
@api.model
def do_clear_done(self):
    dones = self.search([('is_done', '=', True)])
    dones.write({'active': False})
    return True
```

在以`@api.model`为装饰的方法中，`self`变量代表了没有特别记录的模型。我们将构建一个包含所有被标记的任务的`dones`记录集。然后，我们将`active`标记设置为`False`。

`search`方法是一种返回满足某些条件的记录的API方法。这些条件是在一个域中编写的，它是三胞胎的列表。我们将在第6章中详细讨论域，视图——设计用户界面。

write方法在记录集的所有元素上同时设置值。使用字典描述要写入的值。在这里使用write比遍历记录集更有效地将值逐个分配给每一个。

添加测试

现在我们应该为业务逻辑添加测试。理想情况下，我们希望每一行代码都能被至少一个测试用例覆盖。在tests/test_todo.py中，再添加几行代码到test_create()方法:

```
# 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，并且正确地编写了模型方法，那么在服务器日志中应该不会看到错误消息:

```
$ ./odoo-bin -d todo -i todo_app --test-enable
```

设置访问安全

您可能已经注意到，在加载时，我们的模块在服务器日志中得到一个警告消息:

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

消息非常清楚:我们的新模型没有访问规则，因此除了管理超级用户之外，其他任何人都不能使用它。作为一个超级用户，admin忽略了数据访问规则，这就是为什么我们能够在没有错误的情况下使用表单。但我们必须在其他用户使用我们的模型之前解决这个问题。

我们还需要解决的另一个问题是，我们希望to-do任务对每个用户都是私有的。Odoo支持行级访

问规则，我们将使用它来实现这一点。

测试访问安全

事实上，由于缺少访问规则，我们的测试现在应该失败了。他们不是因为他们和管理用户做的。因此，我们应该更改它们，以便它们使用演示用户。

为此，我们应该编辑tests/test_todo.py文件添加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
```

第一个指令调用父类的setUp代码。下一个改变环境，用于运行测试，self.env，到一个新的使用演示用户。我们已经编写的测试不需要进一步的更改。

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

```
from odoo.exceptions import AccessError
```

接下来，向测试类添加一个额外的方法:

```
    def test_record_rule(self): "Test
        per user record rules" Todo =
        self.env['todo.task']
        task = Todo.sudo().create(('name': 'Admin Task'})
        with self.assertRaises(AccessError):
            Todo.browse([task.id]).name
```

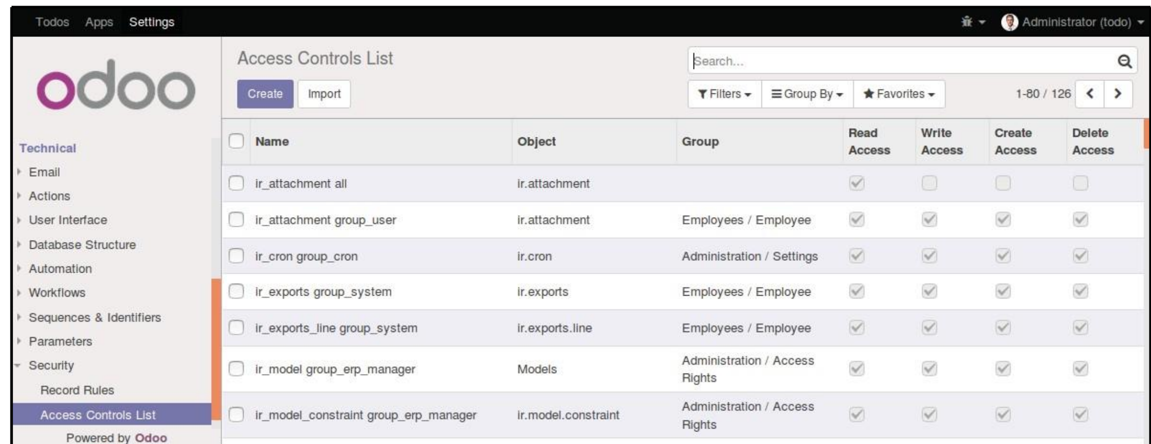
由于我们的env方法现在使用了演示用户，所以我们使用sudo()方法将上下文更改为admin用户。然后我们使用它来创建一个不应该被演示用户访问的任务。

当尝试访问这个任务数据时，我们期望提高`AccessError`异常。如果我们现在运行测试，

它们应该失败，所以让我们来处理它。

添加访问控制安全

要获取向模型添加访问规则所需的信息的图片，请使用web客户端并前往**Settings** | **Technical** | **Security** | **Access Controls List**:



这里我们可以看到一些模型的ACL。它表示，每个安全组允许记录哪些操作。

此信息必须由模块提供，使用数据文件将行加载到`ir.model.access`模型中。我们将在模型上添加完全访问员工组。员工是基本的准入群体，几乎每个人都属于。

这是使用一个名为`security/ir.model.access.csv`的CSV文件完成的。让我们添加以下内容:

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlin
k
acess_todo_task_group_user,todo.task.user,model_todo_task,base.group_user,1
```

```
,1,1,1
```

文件名对应于将数据加载到的模型，文件的第一行有列名。这些是CSV文件中提供的列:

id 是记录外部标识符(也称为XML ID)。在我们的模块中应该是唯一的。

name 是一个描述标题。它只提供信息，最好是保持独特。官方模块通常使用与模型名称和组相隔离的字符串。按照这个惯例，我们使用了todo.task.user。

model_id 是我们提供访问的模型的外部标识符。模型具有由ORM自动生成的XML ID:对于todo.task，标识符是model_todo_task。

group_id 标识要授予权限的安全组。最重要的是由基本模块提供的。Employee组就是这样一个例子，它有一个标识符base.group_user。

Perm 字段标记对授予read, write, create,或unlink (删除)访问的权限。

我们不能忘记在_manifest_.py描述符的data属性中添加对这个新文件的引用。它应该是这样的:

```
'data': [
    'security/ir.model.access.csv',
    'views/todo_view.xml',
    'views/todo_menu.xml',
],
```

和以前一样，升级这些添加的模块以生效。警告消息应该消失，我们可以通过登录用户demo (密码也为demo)来确认权限是否OK。如果我们现在运行我们的测试，他们应该只会失败test_record_rule测试用例。

行级访问规则

我们可以Technical菜单中找到Record Rules选项，以及Access Control List。

记录规则在ir.rule模型中定义。像往常一样，我们需要提供一个独特的名字。我们还需要它们操作的模型以及用于访问限制的域过滤器。域筛选器使用在Odoo中使用的通常的元组语法列表。

通常，规则适用于某些特定的安全组。在我们的案例中，我们将把它应用于Employees组。如果它适用于没有安全组，特别是它被认为是全局的(global字段自动被设置为True)。全球规则是不同的，因为它们强加了一些非全球规则无法推翻的限制。

要添加记录规则，我们应该创建一个security/todo_access_rules.xml文件包含以下内容:

```
<?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>
```

注意到noupdate="1"属性。这意味着该数据将不会在模块升级中更新。这将允许它在以后定制，因为模块升级不会破坏用户做出的改变。但是请注意，在开发过程中也会出现这种情况，因此在开发过程中，您可能需要设置noupdate="0"，直到您对数据文件感到满意为止。

在groups字段中，您还将找到一个特殊的表达式。它是一对多关系字段，它们有一个特殊的语

法来操作。在本例中，(4、x)元组指示将x附加到记录，这里x是对Employees组的引用，由base.group_user标识。在第四章，模块数据中，详细讨论了这种一对多的书写特殊语法。

如前所述，我们必须将文件添加到_manifest_.py，然后才能加载到模块中:

```
'data':
  [ 'security/ir.model.access.csv',
  'security/todo_access_rules.xml',
  'todo_view.xml',
  'todo_menu.xml',
],
```

如果我们做对了，我们可以运行模块测试，现在它们应该通过了。

更好地描述模块

我们的模块看起来不错。为什么不添加一个图标来让它看起来更好呢?为此，我们只需要向模块添加一个带有该图标的static/description/icon.png文件。

我们将重新使用现有**Notes**应用程序的图标，因此我们应该将odoo/addons/static/description/icon.png文件复制到custom-addons/todo_app/static/description目录中。

下面的命令应该为我们提供这样的技巧:

```
$ 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，我们的模块应该显示为新的图标。我们还可以为它添加一个更好的描述来解释它的作用，以及它有多伟大。这可以在_manifest_.py文件的description键中完成。但是，首选的方法是将一个README.rst文件添加到模块根目录。

摘要

我们从一开始就创建了一个新的模块，涵盖了模块中最常用的元素:模型、三种基本视图类型(表单、列表和搜索)、模型方法中的业务逻辑以及访问安全性。

在此过程中，我们熟悉了模块的开发过程，包括模块升级和应用服务器重新启动，以使逐步的变化在Odoo中有效。

记住，在添加模型字段时，需要进行升级。在改变Python代码(包括清单文件)时，需要重新启动。在更改XML或CSV文件时，需要进行升级;同样，当有疑问时，两者都要做:重新启动服务器并升级模块。

在下一章中，您将学习如何构建在现有的模块上叠加以添加功能的模块。

3

继承-扩展现有的应用程序

Odoo最强大的功能之一是在不直接修改底层对象的情况下添加功能。

这是通过继承机制实现的，它是在现有对象之上的修改层。这些修改可以在所有级别进行:模型、视图和业务逻辑。我们不直接修改现有的模块，而是创建一个新的模块来添加预期的修改。

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。作为一个相关的示例，您将学习如何向您自己的模块中添加Odoo的社交和消息传递特性。

向To-Do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他们自己的待办事项。通过添加协作和社交网络功能，将应用程序提升到另一个层次难道不是很好吗?我们将能够分享任务并与他人讨论。

我们将使用一个新模块来扩展之前创建的To-Do应用程序，并使用继承机制添加这些新特性。以下是我们在本章末尾期望达到的目标:

这将是我们为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，只在当前用户的任务上运行，而不是用户能够看到的所有任务

扩展视图，将必要的字段添加到视图中

添加社交网络功能:消息墙和追随者

我们将开始在todo_app模块旁边创建一个新的todo_user模块的基本框架。在第1章的安装示例中，开始使用Odoo开发，我们将在~/odoo-dev/custom-addons/托管我们的模块。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，包含以下代码:

```
(  'name': 'Multiuser To-Do',
   'description': 'Extend the To-Do app to multiuser.',
   'author': 'Daniel Reis',
   'depends': ['todo_app'], }
```

我们在这里没有做过这个，但是包括summary和category键在发布模块到Odoo在线应用商店时很重要。

注意，我们添加了对todo_app模块的显式依赖。这对于继承机制的正常工作是必要和重要的。从现在开始，当todo_app模块被更新时，所有依赖于它的模块，如todo_user模块，也将被更新。

接下来,安装它。在**Apps**下使用**Update Apps List**菜单选项更新模块列表就足够了;在**Apps**列表中找到新模块并单击它的**Install**按钮。注意，这次您需要删除默认的**Apps**过滤器，以便在列表中看到新模块，因为它没有被标记为应用程序。有关发现和安装模块的详细说明，请参阅第1章，开始使用Odoo开发。

现在，让我们开始添加新特性。

扩展模型

新模型通过Python类定义。扩展它们也可以通过Python类进行，但是在odoo特有的继承机制的帮助下。

为了扩展现有模型，我们使用带有_inherit属性的Python类。这标识了要扩展的模型。

这个新类继承了父Odoo模型的所有特性，我们只需要声明我们想要引入的修改。

事实上，Odoo模型存在于我们特定的Python模块之外，在一个中央注册表中。这个注册表可以从使用self.env[<model name>]的模型方法访问。例如，为了获得对代表res.partner模型的对象的引用，我们将编写self.env['res.partner']。

为了修改一个Odoo模型，我们得到一个对它的注册表类的引用，然后对其进行就地修改。这意味着这些修改也将在其他地方可用，在这个新模型被使用的地方。

在Odoo服务器启动时，加载该序列的模块是相关的:一个附加模块所做的修改只会在随后加载的附加组件上可见。因此，正确设置模块依赖关系非常重要，确保提供我们使用的模型的模块包含在我们的依赖树中。

在模型中添加字段

我们将扩展todo.task模型，为它添加几个字段:负责任务的用户和截止日期。

编码风格指南建议拥有一个每个Odoo模型的models/子目录。因此，我们应该从创建模型子目录开始，使其成为Python- importable。

编辑todo_user/__init__.py文件的内容:

```
from .import models
```

用以下代码创建todo_user/models/__init__.py:

```
from . import todo_task
```

前一行指示Python在同一目录中查找名为`odoo_task.py`的文件并导入它。您通常在目录中的每个Python文件都有一个`from`行:

现在创建`todo_user/models/todo_task.py`文件来扩展原来的模型:

```
# -*- 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')
```

类名`TodoTask`是这个Python文件的本地名称，通常与其他模块无关。`_inherit`类属性是这里的关键:它告诉Odoo这个类继承并因此修改了`todo.task`模型。

注意，`_name`属性没有出现。不需要它，因为它已经继承自父模型。

接下来的两行是常规字段声明。`user_id`字段表示来自用户模型`res.users`的用户。它是一个`Many2one`字段，相当于一个数据库术语的外键。`date_deadline`是一个简单的日期字段。在第5章中，模型——构造应用程序数据，我们将更详细地解释Odoo中可用的字段类型。

为了将新字段添加到模型的支持数据库表中，我们需要执行模块升级。如果一切按照预期进行，那么在**Technical | Database Structure | Models**菜单选项中检查`todo.task`模型时，您应该看到新的字段。

修改现有的字段

如您所见，将新字段添加到现有模型非常简单。从Odoo8开始，修改现有继承字段的属性也是可能的。它是通过添加一个具有相同名称和设置值的字段来实现的，只需要更改属性。

例如，为name字段添加一个帮助工具提示，我们将把这一行添加到todo_task.py中，前面描述过:

```
name = fields.Char(help="What needs to be done?")
```

这将使用指定的属性修改字段，将未修改的所有其他属性保留在这里。如果我们升级模块，进入to-do任务表单并在**Description**字段上暂停鼠标指针;将显示工具提示文本。

修改模型方法

继承也适用于业务逻辑级别。添加新方法很简单:在继承类中声明它们的函数。

要扩展或更改现有逻辑，可以通过使用完全相同的名称声明方法来覆盖相应的方法。新方法将取代以前的方法，而且它还可以扩展继承类的代码，使用Python的super()方法调用父方法。它可以在调用super()方法之前和之后，在原有的逻辑上添加新的逻辑。

最好避免更改方法的函数签名(也就是说，保留相同的参数)，以确保现有的调用将继续正常工作。如果需要添加额外的参数，请将它们设置为可选的关键字参数(带有默认值)。

最初的**Clear All Done**操作不适合我们的任务共享模块，因为它清除了所有的任务，而不考虑它们的用户。我们需要修改它，使它只清除当前的用户任务。

为此，我们将重写(或替换)原来的方法，使用一个新版本，它首先找到当前用户完成的任务列表，然后激活它们:

```
@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
```

为了清晰起见，我们首先构建过滤器表达式，用于查找要清除的记录。

这个过滤器表达式遵循odoo特有的语法，称为`domain`:它是一个条件列表，其中每个条件都是一个元组。
这些条件隐式地与AND (`&`)操作符连接。对于OR操作，一个管道，`|`，被用在一个元组的地方，并且它加入下两个条件。我们将在第6章中详细介绍一些域，视图-设计用户界面。

这里使用的域筛选了所有已完成的任务(`'is_done', '=', True`)，它们要么有当前用户作为负责的(`'user_id', '=', self.env.uid`)，要么没有当前用户设置(`'user_id', '=', False`)。

然后，我们使用`search`方法将记录集与所做的记录进行操作，最后，在它们上做一个批量写入，将`active`字段设置为`False`。这里的Python `False`值表示数据库`NULL`值。

在这种情况下，我们完全重写了父方法，用新的实现替换它，但这不是我们通常想要做的。相反，我们应该将现有的逻辑扩展到一些额外的操作。否则，我们可能会破坏已经存在的特性。
为了让重写方法保留已经存在的逻辑，我们使用Python的`super()`构造调用该方法的父版本。让我们来看一个例子。

我们可以改进`do_toggle_done()`方法，使它只对分配给当前用户的任务执行操作。这是实现这一目标的代码:

```
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()
```

继承类中的方法从一个for循环开始，检查是否要切换到另一个用户的任务。如果这些检查通过，它将使用super()调用父类方法。如果没有提出错误，我们应该使用这个Odoo构建-在异常中。最相关的是ValidationError，这里使用，UserError。

这些是用于覆盖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。

扩展的观点

表单、列表和搜索视图都是使用arch XML结构定义的。为了扩展视图，我们需要一种方法来修改此XML。这意味着定位XML元素，然后在这些点上进行修改。

继承的视图允许这样做。继承的视图声明如下:

```
<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">
    <!-- ...match and extend elements here! ... -->
  </field
</record>
```

inherit_id字段通过使用特殊的ref属性来表示其外部标识符，从而将视图扩展。外部标识符将在第四章模块数据中详细讨论。

作为XML，定位XML元素的最佳方法是使用XPath表达式。例如，使用前一章中定义的表单视图，一个用于定位`<field name="is_done">`元素的XPath表达式是`//field[@name]='is_done'`。这个表达式可以找到任何带有`name`属性的`field`元素，它等于`is_done`。你可以找到关于XPath的更多信息：https://docs.python.org/2/library/xml.etree.elementtree.html#xpath-support.

如果XPath表达式匹配多个元素，则只有第一个元素会被修改。因此，它们应该尽可能地具体化，使用惟一的属性。使用`name`属性是确保我们找到要使用扩展点的准确元素的最简单方法。因此，在我们的视图XML元素中设置它们是很重要的。

一旦扩展点被定位，您可以修改它或者在它附近添加XML元素。作为一个实际示例，在`is_done`字段之前添加`date_deadline`字段，我们将在`arch`中编写以下代码:

```
<xpath expr="//field[@name]='is_done'" position="before">
  <field name="date_deadline" />
</xpath>
```

幸运的是，Odoo为这个提供了快捷方式，因此大多数时候我们完全可以避免使用XPath语法。与前面的XPath元素相反，我们可以使用与元素类型类型相关的信息来定位和它的独特属性，而不是前面的XPath，我们这样写:

```
<field name="is_done" position="before">
  <field name="date_deadline" />
</field>
```

请注意，如果字段在同一视图中出现不止一次，那么您应该始终使用XPath语法。这是因为Odoo将在字段的第一次出现时停止，它可能会将您的更改应用到错误的字段。

通常，我们希望在现有的字段旁边添加新字段，因此`<field>`标记通常会被用作定位器。但是任何其他标记都可以使用: `<sheet>`，`<group>`，`<div>`，等等。名称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 `class`元素。Odoo将会找到第一个元素，它至少包含了所有指定的属性。

在9.0版本之前，字符串属性(用于显示的标签文本)也可以用作扩展定位器。从9.0开始，这是不允许的。这个限制与在这些字符串上运行的语言转换机制有关。

定位器元素使用的`position`属性是可选的，可以具有以下值:

`after`将内容添加到父元素，在匹配的节点之后。

`before`在匹配的节点之前添加内容。

`inside` (默认值)附加在匹配节点内的内容。

`replace` 替换匹配的节点。如果使用空内容，则删除一个元素。由于Odoo 10还允许将一个元素与其他标记包在一起，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。这是在元素内容`<attribute name="attr-name">`元素和新的属性值设置中使用的。

例如，在任务表单中，我们有`active`字段，但是让它可见是没有用的。我们可以从用户那里隐藏它。这可以通过设置`invisible`属性来实现:

```
<field name="active" position="attributes">
  <attribute name="invisible">1</attribute>
</field>
```

设置`invisible`属性以隐藏元素是使用`replace`定位器删除节点的好方法。应该避免删除节点，因为它可以根据被删除的节点作为一个占位符来添加其他元素。

扩展form视图

将所有以前的表单元素组合在一起，我们可以添加新字段并隐藏`active`。扩展to-do任务表单的完整继承视图如下:

```
<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>
```

这应该被添加到我们模块中的`views/todo_task.xml`文件中，在`<odoo>`元素中，如前一章所示。

继承的视图也可以继承，但是由于这会产生更复杂的依赖关系，所以应该避免。只要可能，您应该倾向于从最初的视图继承。

另外，我们不应该忘记将数据属性添加到`_manifest_.py`描述符文件中:

```
'data': ['views/todo_task.xml'],
```

扩展tree和search视图

Tree和search视图扩展也被定义为使用`arch` XML结构，它们可以以与form视图相同的方式扩展。我们将通过扩展list和search视图来继续我们的示例。

对于list视图，我们想要将`user`字段添加到它:

```
<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search视图，我们将为用户自己的任务和未分配给任何人的任务添加用户和预定义过滤器的搜索:

```
    <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>
```

不要过于担心这些视图的特定语法。我们将在第6章中详细介绍它们，视图——设计用户界面。

更多的模型继承机制

我们已经看到了模型的基本扩展，即官方文档中的**class inheritance**。这是最常见的继承，最

容易想到的是**in-place extension**。你取一个模型并扩展它。当您添加新特性时，它们被添加到现有模型中。一个新的模型没有被创建。我们还可以继承多个父模型，将值列表设置为`_inherit`属性。有了这个，我们可以使用**mixin classes**。Mixin classes是实现泛型特性的模型，我们可以添加到其他模型中。它们不被期望直接使用，而且就像一个准备被添加到其他模型的特性容器。

如果我们还使用与父模型不同的`_name`属性，我们将得到一个新的模型，该模型重用继承的特性，但使用它自己的数据库表和数据。官方文件称此为**prototype inheritance**。在这里，你拿一个模型，创建一个全新的，它是旧版本的副本。当您添加新特性时，它们被添加到新模型中。现有的模型没有改变。

还有**delegation inheritance**方法，使用`_inherits`属性。它允许模型以透明的方式包含其他模型，而在幕后，每个模型都处理自己的数据。你取一个模型并扩展它。当您添加新特性时，它们被添加到新模型中。现有模块没有更改。新模型中的记录与原始模型中的记录有关联，原始模型的字段被公开，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。

复制具有原型继承的特性

我们在扩展模型之前使用的方法仅使用`_inherit`属性。我们定义了一个继承了`todo.task`模型并添加了一些特性的类。类属性`_name`没有显式设置;隐式,这是`todo.task`。

但是，使用`_name`属性允许我们创建一个新的模型，复制继承的特性。这是一个例子:

```
from odoo import models
class TodoTask(models.Model):
    _name = 'todo.task'
    _inherit = 'mail.thread'
```

这扩展了`todo.task`模型，将其复制到`mail.thread`模型的特性中。`mail.thread`模型实现了Odoo消息和追随者特性，并且是可重用的，因此可以很容易地将这些特性添加到任何模型中。

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(继承的)和新(继承)模型的数据记录是不相关的。只有定义是共享的。

稍后，我们将详细讨论如何使用此功能将`mail.thread`及其社交网络特性添加到模块中。在实践中，当使用mixin时，我们很少从常规模型中继承，因为这会导致相同数据结构的重复。

Odoo还提供了一种委托继承机制，避免了数据结构的重复，因此在继承常规模型时，它通常是首选的。让我们更详细地看一下。

使用委托继承嵌入模型

委托继承的使用较少，但它可以提供非常方便的解决方案。它通过使用字典映射继承的模型和连接到它们的字段来使用`_inherits`属性(注意附加的 `s`)。

一个很好的例子就是标准用户模型，`res.users`;它有一个嵌入的合作伙伴模型:

```
from odoo import models, fields
class User(models.Model):
    _name = 'res.users'
    _inherits = {'res.partner': 'partner_id'}
    partner_id = fields.Many2one('res.partner')
```

有了授权继承，`res.users`模型嵌入了继承的模型`res.partner`，这样当创建一个新的`User`类时，也会创建一个合作伙伴，并将它保存在`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。

通过委托机制，从继承的模型和伙伴的所有字段都可用，就像它们是`User`字段一样。例如，伙伴`name`和`address`字段被公开为`User`字段，但实际上，它们被存储在链接的合作伙伴模型中，并且没有发生数据重复。

与原型继承相比，这一点的优点是，无需重复数据结构，比如多个表中的地址。任何需要包含地址的新模型都可以将其委托给嵌入的伙伴模型。如果在伙伴地址字段中引入了修改，那么这些都可以立即应用到所有嵌入它的模型中!

请注意，带授权继承的字段是继承的，但方法不是。

添加社交网络功能

社交网络模块(技术名称`mail`)提供了在许多表单底部和**Followers**功能底部发现的消息板，以及关于消息和通知的逻辑。这是我们经常想要添加到我们的模型中，所以让我们学习如何去做。

社交网络信息功能是由`mail`模块的`mail.thread`模型提供的。要将其添加到自定义模型，我们需要执行以下操作:

- 模块依赖于`mail`吗
- 有从`mail.thread`继承的类吗
- 在窗体视图中添加了追随者和线程小部件吗
- 我们还需要为关注者建立记录规则。

让我们按照这个清单。

关于第一点，我们的扩展模块需要对模块`_manifest_.py`清单文件额外的`mail`依赖性:

```
'depends': ['todo_app', 'mail'],
```

关于第二点，`mail.thread`的继承是使用我们以前使用的`_inherit`属性来完成的。但是我们的to-do任务扩展类已经使用了`_inherit`属性。幸运的是，它可以接受一个模型列表来继承，因此我们可以使用它来使它也包括`mail.thread`的继承:

```
_name = 'todo.task'
_inherit = ['todo.task', 'mail.thread']
```

`mail.thread`是一个抽象模型。**Abstract models**和常规模型一样，只不过它们没有数据库表示;没有为它们创建实际的表。抽象模型并不意味着直接使用。相反，它们将被用作mixin类，

正如我们刚才所做的那样。我们可以把它们看作是具有现成功能的模板。为了创建一个抽象类，我们只需要使用`models.AbstractModel`而不是`models.Model`来定义它们。

对于第三点，我们希望将社交网络小部件添加到表单的底部。这是通过扩展窗体视图定义来完成的。我们可以重用已经创建的继承视图`view_form_todo_task_inherited`，并将其添加到它的`arch`数据:

```
<sheet position="after">
  <div class="oe_chatter">
    <field name="message_follower_ids"
      widget="mail_followers" />
    <field name="message_ids" widget="mail_thread" />
  </div>
</sheet>
```

这里添加的两个字段并没有被我们明确声明，但是它们是由`mail.thread`模型提供的。

最后一步，即第4步，是为追随者设置记录规则:行级访问控制。只有当我们的模型被要求限制其他用户访问记录时，才需要这样做。在这种情况下，我们希望每个to-do任务记录也能被它的任何追随者看到。

我们已经在to-do任务模型上定义了一个记录规则，因此我们需要修改它来添加新的需求。这是我们下一节要做的事情之一。

修改数据

与视图不同，常规数据记录没有XML `arch`结构，不能使用XPath表达式进行扩展。但它们仍然可以被修改，以替换字段中的值。

`<record id="x" model="y">`数据加载元素实际上在模型y上执行`insert`或`update`操作:如果model x不存在，则创建;否则，它将被更新/写入。

由于其他模块中的记录可以使用`<model>.<identifier>`全局标识符访问，所以我们的模块可以覆盖其他模块之前编写的内容。

请注意，由于点是预留的，以将模块名与对象标识符分隔开来，因此不能在标识符名称中使用它。相反，使用下划线选项。

修改菜单和动作记录

例如，让我们将todo_app模块创建的菜单选项改为My To-Do。为此，我们可以将以下内容添加到todo_user/views/todo_task.xml文件:

```
<!-- Modify menu item -->
<record id="todo_app.menu_todo_task" model="ir.ui.menu">
  <field name="name">My To-Do</field>
</record>
```

我们还可以修改菜单项中使用的操作。动作有一个可选的上下文属性。它可以为视图字段和过滤器提供默认值。我们将使用它在默认情况下启用**My Tasks**过滤器，在本章前面定义:

```
<!-- Action to open To-Do Task list -->
<record model="ir.actions.act_window"
  id="todo_app.action_todo_task">
  <field name="context">
    ('search_default_filter_my_tasks': True}
  </field>
</record>
```

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。但是现在，随着社会功能的增加，我们也需要任务的追随者来访问它们。社交网络模块本身并没有处理这个问题。

另外，现在任务可以让用户分配给他们，因此，让访问规则来处理负责任的用户而不是创建任务的用户更有意义。
该计划与我们为菜单项所做的一样:覆盖todo_app.todo_task_user_rule，将

domain_force字段修改为一个新值。

该公约是在安全security中保留与安全相关的文件，因此我们将创建一个security/todo_access_rules.xml文件，内容如下:

```
<?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">ToDo Tasks for owner and
        followers</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>
```

这覆盖todo_task_per_user_rule记录规则，从todo_app模块。新的域筛选器现在可以让负责用户、user_id或对每个人都可见的任务，如果负责的用户没有设置(等于False);所有的任务追随者都可以看到它。

记录规则运行在可用的user变量的上下文中，并表示当前会话用户的记录。因为追随者是合作伙伴,不是users，user.id,相反,我们需要使用user.partner_id。

groups字段是一个to-many关系。在这些字段中编辑数据使用一种特殊的符号。这里使用的代码4是附加到相关记录的列表。通常使用的是代码6，取而代之的是将相关记录完全替换为一个新列表。我们将在第四章，模块数据中详细讨论这个符号。

记录元素的noupdate="1"属性意味着这个记录数据只会写在安装操作上，并且在模块升级时将被忽略。这允许它是自定义的，不需要冒着过度编写定制的风险，并且在将来某个时候进行模块升级时丢失它们。

在开发时使用<data noupdate="1">处理数据文件可能会很麻烦，因为以后对XML定义的编辑会被模块升级所忽略。为了避免这种情况，可以重新安装模块。 这更容易通过使用 -i 的命令行完成

像往常一样，我们不能忘记将新文件添加到__manifest__.py的数据属性中:

```
'data': ['views/todo_task.xml', 'security/todo_access_rules.xml'],
```

摘要

现在，您应该能够通过扩展现有模块来创建自己的模块。

为了演示如何做到这一点，我们扩展了在前一章中创建的To-Do模块，并向构成应用程序的几个层添加新特性。

我们扩展了一个Odoo模型来添加新字段并扩展其业务逻辑方法。接下来，我们修改视图，使新字段可用。最后，我们学习了如何从其他模型继承特性，并使用它们将社交网络功能添加到To-Do应用程序。

前三章概述了Odoo开发的常见活动，从Odoo安装到模块创建和扩展。

下一章将重点讨论Odoo发展的一个具体领域，我们在第一章中简要介绍了其中的大部分内容。在接下来的章节中，我们将更详细地讨论数据序列化以及XML和CSV数据文件的使用。

4

模块数据

大多数Odoo模块定义，比如用户接口和安全规则，实际上都是存储在特定数据库表中的数据记录。在运行时，在模块中发现的XML和CSV文件不被Odoo应用程序使用。相反，它们是将这些定义加载到数据库表中的一种方法。

因此，Odoo模块的一个重要部分是关于使用文件来表示(序列化)数据，以便以后可以加载到数据库中。

模块也可以有默认和演示数据。数据表示允许将其添加到我们的模块中。此外，了解Odoo数据表示格式对于在项目实现的上下文中导出和导入业务数据非常重要。

在我们进入实际案例之前，首先探索外部标识符概念，这是Odoo数据表示的关键。

了解外部标识符

external identifier (也称为XML ID)是一个可读的字符串标识符，它唯一地标识了Odoo中的特定记录。当将数据加载到Odoo时，它们非常重要。

其中一个原因是当升级一个模块时，它的数据文件将再次被加载到数据库中，我们需要检测

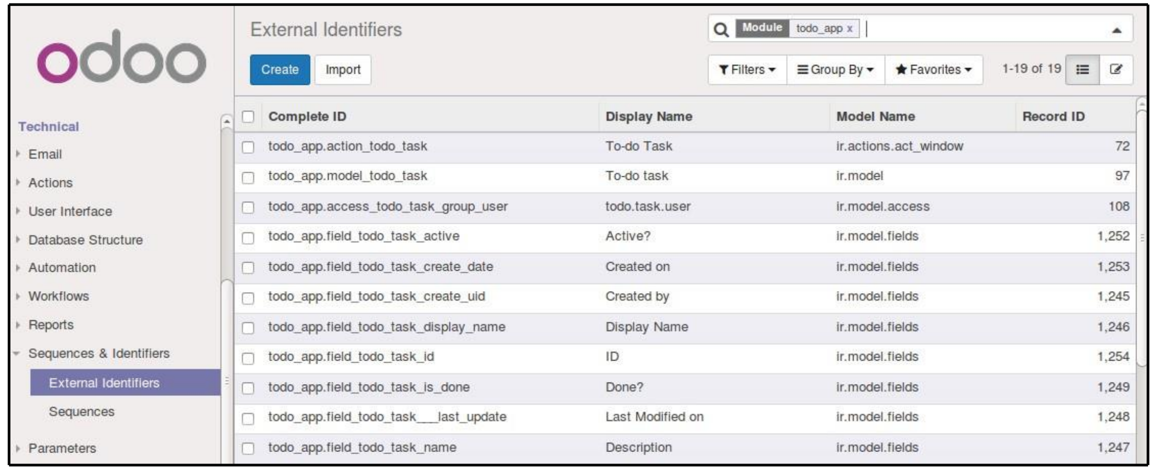
已经存在的记录，以便更新它们，而不是创建新的重复记录。

支持相关数据的另一个原因:数据记录必须能够引用其他数据记录。实际的数据库标识符是在模块安装期间由数据库自动分配的序号。外部标识符提供了一种方法来引用相关的记录，而不需要事先知道它将被分配的数据库ID，从而允许我们定义Odoo数据文件中的数据关系。

Odoo负责将外部标识符名称转换为分配给它们的实际数据库id。这背后的机制很简单:Odoo保存了一个表，表中有命名的外部标识符与其相应的数字数据库id之间的映射: `ir.model.data`模型。

要检查现有的映射，请访问**Settings**顶部菜单的**Technical**部分，并在**Sequences & Identifiers**下选择“**External Identifiers**”菜单项。

例如，如果我们访问**External Identifiers**列表并通过`todo_app`模块过滤它，我们将看到前面创建的模块生成的外部标识符:

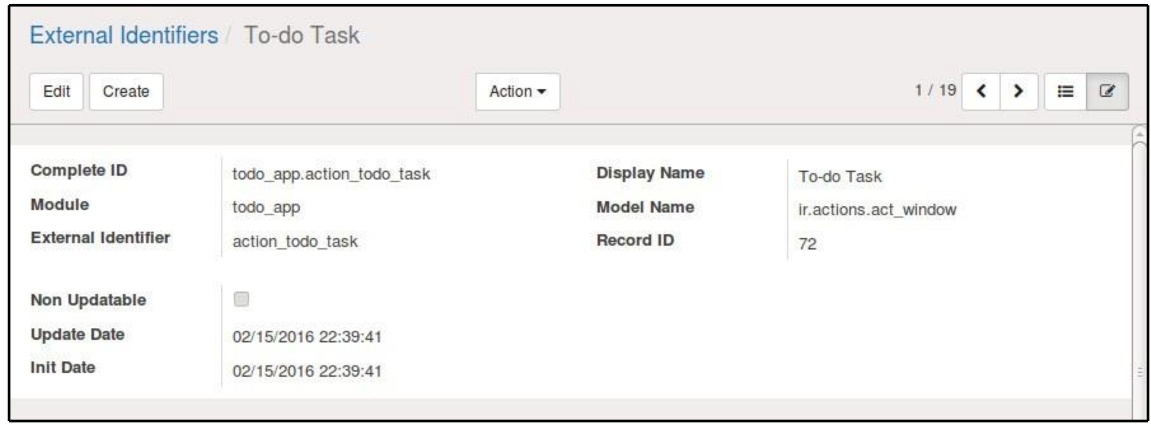


我们可以看到外部标识符有一个**Complete ID**标签。注意它是如何由模块名和标识符名称组成的，例如，`todo_app.action_todo_task`。

外部标识符只需要在一个Odoo模块内惟一，这样就不会有两个模块相互冲突的风险，因为它们不小心选择了相同的标识符。要构建一个全局惟一标识符，Odoo将模块名称与实际的外部标识符名称连接在一起。这是在**Complete ID**字段中可以看到的。

当在数据文件中使用外部标识符时，您可以选择使用完整的标识符或仅使用外部标识符名。通常使用外部标识符名称比较简单，但是完整的标识符使我们能够从其他模块引用数据记录。这样做时，请确保这些模块包含在模块依赖项中，以确保这些记录在我们之前已经加载。

在列表的顶部，我们有`todo_app.action_todo_task`完整的标识符。这是我们为模块创建的菜单操作，它也在相应的菜单项中引用。点击它，我们就会看到它的细节；`todo_app`模块中的`action_todo_task`外部标识符映射到`ir.actions.act_window`模型中的特定记录ID，在本例中为`72`：



除了为记录提供一种方便地引用其他记录的方法之外，外部标识符还允许您避免重复导入的数据重复。如果外部标识符已经存在，则将更新现有记录;您不需要创建一个新的记录。这就是为什么在后续的模块升级中，先前加载的记录是更新的而不是复制的。

发现外部标识符

在为模块准备定义和演示数据文件时，我们经常需要查找引用中需要的现有外部标识符。

我们可以使用前面显示的**External Identifiers**菜单，但是**Developer**菜单可以提供更方便的方法。正如您可能在第1章中看到的，开始使用Odoo开发时，**Developer**菜单在**Settings**仪表板中被激活，在右下角的选项中。

要查找数据记录的外部标识符，在相应的表单视图中，从**Developer**菜单中选择**View Metadata**选项。这将显示一个与记录的数据库ID和外部标识符(也称为XML ID)的对话框。

举个例子，查看演示用户ID，我们可以导航到表单视图，**Settings** | **Users**，选择**View Metadata**选项，这将显示:

Metadata (res.users) ×

ID:	4
XML ID:	base.user_demo
No Update:	true
Creation User:	Administrator
Creation Date:	02/15/2016 22:38:35
Latest Modification by:	Administrator
Latest Modification Date:	02/15/2016 22:38:35

Ok

为了找到视图元素的外部标识符，例如表单、树、搜索或操作，**Developer**菜单也是一个很好的帮助来源。为此，我们可以使用它的**Manage Views**选项，或者使用**Edit <view type>**选项打开所需视图的信息。然后，选择他们的**View Metadata**选项。

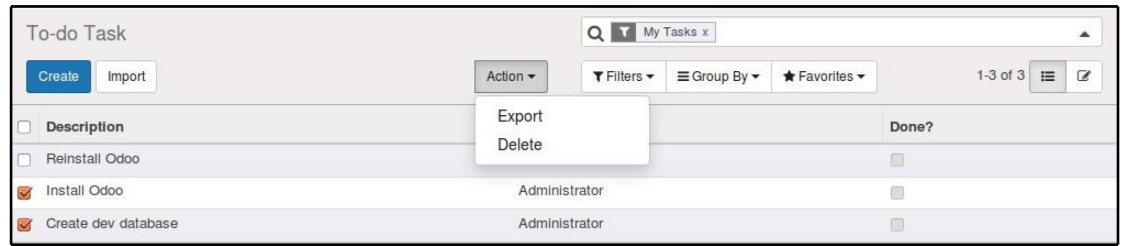
导出和导入数据

我们将开始探索如何从Odoo的用户界面导出和导入数据，然后我们将讨论如何在addon模块中使用数据文件的技术细节。

导出数据

数据导出是任何列表视图中可用的标准特性。要使用它，我们必须首先通过选择最左边的对应的复选框来选择要导出的行，然后从**More**按钮中选择**Export**选项。

这里有一个例子，使用最近创建的to-do任务:

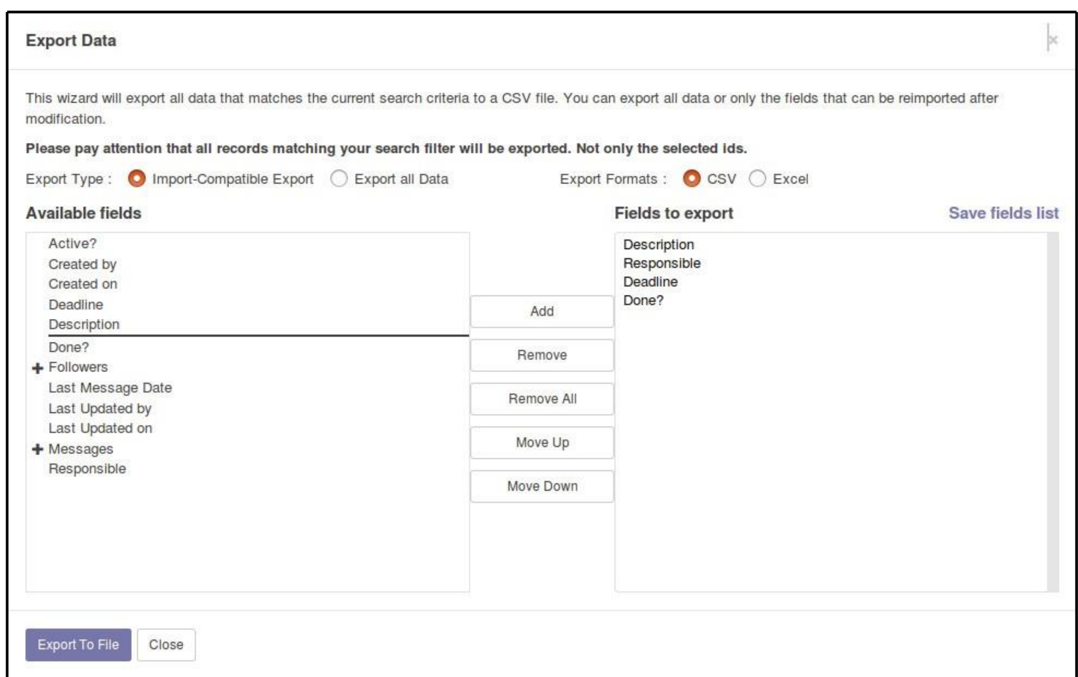


我们也可以勾选列标题中的复选框。它将同时检查所有的记录，并将导出符合当前搜索条件的所有记录。

在以前的Odoo版本中，只有在屏幕上看到的记录(当前页面)才能实际导出。从Odoo 9开始，在头文件中修改和勾勾的复选框将输出与当前筛选器匹配的所有记录，而不仅仅是当前显示的那些。这对于输出不适合屏幕的大量记录非常有用。

Export选项把我们带到了一个对话框，在那里我们可以选择输出什么。**Import-Compatible Export**选项确保导出的文件可以导入到Odoo。我们需要使用这个。

导出格式可以是CSV或Excel。我们希望CSV文件能更好地理解导出格式。接下来，选择要导出的列，并单击**Export To File**按钮。这将开始下载一个带有导出数据的文件:



如果我们遵循这些说明并选择前面截图中显示的字段，那么我们应该得到一个类似于此的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,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。如果没有指定的模块，则会自动生成使用`_export_`代替实际模块名称的新功能。新的标识符只被分配到没有一个的记录，而且从那里开始，它们被绑定到相同的记录。这意味着随后的导出将保留相同的外部标识符。

导入数据

首先，我们必须确保启用了导入特性。因为Odoo 9是默认启用的。如果不是，这个选项是可以从**Settings**级菜单，**General Settings**选项。在**Import | Export**部分，有一个应该启用的**a Allow users to import data from CSV/XLS/XLSX/ODS files**复选框。

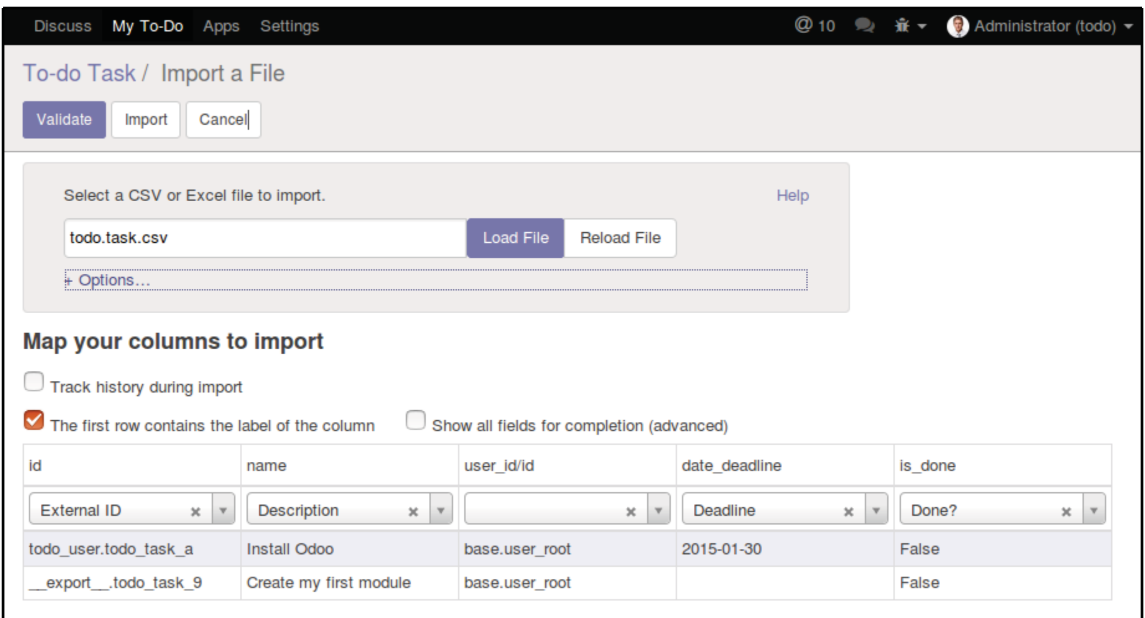
这个特性是由**Initial Setup Tools** addon提供的(`base_setup`是技术名称)。的实际效果bbb...复选框是安装或卸载`base_setup`。

有了这个选项，列表视图就显示了在列表顶部的**Create**按钮旁边的**Import**选项。

让我们先对我们的待办事项数据进行批量编辑。打开我们刚刚在电子表格或文本编辑器中下载的CSV文件，并更改一些值。另外，添加一些新行，将`id`列空出来。

如前所述，第一列`id`为每一行提供惟一标识符。这使得已经存在的记录可以更新，而不是在我们将数据导入Odoo时复制它们。对于添加到CSV文件的新行，我们可以选择提供我们选择的外部标识符，或者将空白`id`列留空，将为它们创建一个新的记录。
在将更改保存到CSV文件之后，单击**Import**选项(在**Create**按钮旁边)，我们将会看到导入助手。

在那里，我们应该选择磁盘上的CSV文件位置并单击**Validate**以检查其正确性的格式。因为要导入的文件是基于Odoo导出的，所以很有可能它是有效的:



现在我们可以点击**Import**，就这样;我们的修改和新记录本应载入Odoo。

CSV数据文件中的相关记录

在前面的示例中，负责每个任务的用户是用户模型中的一个相关记录，使用many-to-one (或一个外键)关系。它的列名是`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`。

只建议在导出和导入相同的数据库时使用数据库id。通常，您倾向于使用外部标识符。

如果使用外部标识符或 `/.id` 使用数据库(数值)id，关系列应该有 `/id` 附加到它们的名称。或者，冒号(:)可以在相同效果的地方使用。

类似地，也支持many-to-many关系。many-to-many关系的一个例子是用户和组之间的关系:每个用户可以有许多组，每个组可以有许多用户。这种类型的字段的列名应该有`/id`附加。字段值接受以逗号分隔的外部标识符列表，并使用双引号包围。

例如，to-do任务关注者在待办事项和合作伙伴之间有many-to-many的关系。它的列名应该是`follower_ids/id`和一个字段值，有两个追随者可以是这样的:

```
" export .res_partner_1, export  .res_partner_2"
```

最后，还可以通过CSV导入一对多关系。这种类型关系的典型示例是带有几行代码的文档头。注意，one-to-many关系始终是many-to-one关系的逆。每个文档头可以有许多行。而且每条直线都有一个头。

我们可以在公司模型中看到这样一个关系的例子(在**Settings**菜单中有表单视图):每个公司都可以拥有多个银行帐户，每个帐户都有自己的详细信息;相反，每个银行账户记录都属于和只有一个公司的many-to-one关系。

我们可以把公司的银行账户连同他们的银行账户一起放在一个文件里。这里有一个例子，我们将一个公司和三家银行一起装入:

```
id,name,bank_ids/id,bank_ids/acc_number,bank_ids/state
base.main_company,YourCompany,_export__.res_partner_bank_4,123456789,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，在接下来的两行中是空的。他们有记录头记录的数据，也就是公司的数据。

其他三列都是带有`bank_ids/`的前缀，在这三行中都有值。他们为公司的银行账户提供了三个相关的数据。第一行有公司和第一家银行的数据，接下来的两行数据只针对额外的公司和银行。

这些都是与导出和从GUI导入的过程中的基本要素。在新的Odoo实例中设置数据或准备将数据文件包含在Odoo模块中是很有用的。接下来，我们将学习更多关于使用模块中的数据文件。

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。这可以使用CSV和XML文件完成。对于完整性，也可以使用YAML文件格式，但它很少用于加载数据;因此，我们不会讨论这个问题。

模块使用的CSV文件与我们看到的和用于导入特性的CSV文件完全相同。当在模块中使用它们时，一个额外的限制是文件名必须与加载数据的模型的名称相匹配，这样系统就可以推断出应该导入数据的模型。

数据CSV文件的一个常见用法是访问安全定义，加载到`ir.model.access`模型中。他们通常使用命名为`ir.model.access.csv`的CSV文件。

演示数据

Odoo addon模块可以安装演示数据，这样做被认为是很好的实践。这有助于为测试中使用的模块和数据集提供使用示例。使用`__manifest__.py`清单文件的`demo`属性声明模块的演示数据。就像`data`属性一样，它是一个文件名列表，包含模块内相应的相对路径。
现在是向`todo_user`模块添加一些演示数据的时候了。我们可以先从to-do任务中导出一些数据，如前一节所解释的那样。该约定是将数据文件放置在`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`。由于该数据将由我们的模块拥有，所以我们应该编辑id值以删除标识符中的`_export_`前缀。

例如，我们的`todo.task.csv`数据文件可能是这样的:

```
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",""
```

我们不能忘记将这个数据文件添加到_`manifest_.py`清单的`demo`属性:

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

下次我们更新模块时，只要它安装了已启用的演示数据，文件的内容将被导入。注意，每当执行模块升级时，这些数据将被重新导入。

XML文件也用于加载模块数据。让我们进一步了解XML数据文件可以做哪些CSV文件不能做的事情。

XML数据文件

虽然CSV文件提供了一种简单紧凑的格式来表示数据，但XML文件更强大，并对加载过程提供了更多的控制。它们的文件名不需要匹配要加载的模型。这是因为XML格式要丰富得多，而且该信息由文件中的XML元素提供。

我们已经在前几章中使用了XML数据文件。用户界面组件，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文件是用来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`模块添加第二个数据文件，`data/todo_data.xml`，有以下内容:

```
<?xml version="1.0"?>
<odoo>
  <!-- Data to load -->
  <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>
```

这个XML等价于我们刚才看到的CSV数据文件。

XML数据文件有一个`<odoo>` top元素，其中我们可以有几个与CSV数据行对应的`<record>`元素。

在版本9.0中引入了数据文件中的<odoo> top元素，并取代前者<openerp>标签。top元素中的<data>部分仍然被支持，但现在是可选的。事实上，现在是<odoo>和 <data>是等价的，因此我们可以使用其中的一个作为XML数据文件的top元素。

一个<record>元素有两个强制属性，即model和id (记录的外部标识符)，并包含一个<field>标记，用于每个字段的写入。
注意，字段名中的斜杠符号在这里是不可用的;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。我们稍后将讨论关系到多个域的值。

数据noupdate属性

在重复数据加载时，从上一次运行中载入的记录将被重写。重要的是要记住，这意味着升级一个模块将覆盖数据库内部可能发生的任何手工更改。值得注意的是，如果视图被自定义修改，那么这些更改将随着下一个模块的升级而丢失。正确的方法是为我们需要的变更创建继承的视图，如第3章所述，继承-扩展现有的应用程序。

这种重新导入行为是默认的，但是它可以被更改，这样当一个模块升级时，一些数据文件记录就被保留了。这是由<odoo>或<data>元素的noupdate="1"属性完成的。这些记录将在安装addon模块时创建，但在随后的模块升级中，将不会对它们进行任何操作。

这允许您确保手动定制的自定义在模块升级中是安全的。它通常与记录访问规则一起使用，允许它们适应特定于实现的需求。

在同一个XML文件中可以有多个<data>部分。我们可以利用这一点来分离数据，只导入一个，使用noupdate="1"，并在每次升级中重新导入数据，使用noupdate="0"。

noupdate标志存储在每个记录的**External Identifier**信息中。使用**Non Updatable**复选框可以在**Technical**菜单中直接使用**External Identifier**表单手动编辑它。

在开发模块时，noupdate属性可能很棘手，因为稍后对数据做出的更改将被忽略。一个解决方案是，不再使用-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，忽略数据记录上的noupdate标志。

在XML中定义记录

每个<record>元素都有两个基本属性、id和model，并包含为每个列赋值的<field>元素。如前所述，id属性对应于记录的外部标识符，而模型属性对应于写入记录的目标模型。<field>元素有几种不同的赋值方法。让我们详细地看一下它们。

设置字段值

<record>元素定义了一个数据记录，并包含了在每个字段上设置值的<field>元素。

field元素的name属性标识要写入的字段。

写入的值是元素内容:字段的开头和结束标记之间的文本。对于日期和日期，字符串有"YYYY-mm-dd"和"YYYY-mm-dd HH:MM:SS"将被正确转换。但是对于布尔字段，任何非空值都将被转换为True，而"0"和"False"值被转换为False。

从数据文件中读取布尔False值的方法在Odoo 10中得到了改进。在以前的版本中，任何非空值，包括"0"和"False"都被转换为True。对于使用下面讨论的eval属性的布尔，建议使用。

使用表达式设置值

定义字段值的一种更为复杂的方法是`eval`属性。它计算Python表达式并将结果值赋给该字段。

这个表达式是在一个context中被评估的，除了Python内置的外，还有一些额外的标识符可用。让我们看一看。

为了处理日期，下列模块可用: `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，因此使用的日期接近于模块安装日期。例如，要设置一个值到昨天，我们将使用这个:

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

在评估上下文中还有`ref()`函数，它用于将外部标识符转换为相应的数据库ID，可用于为关系字段设置值。例如，我们以前使用它来为`user_id`设置值:

```
<field name="user_id" eval="ref('base.group_user')" />
```

为关系字段设置值

我们刚刚看到了如何在一个many-to-one关系字段(比如`user_id`)中设置一个值，使用带有`ref()`功能的`eval`属性。但还有一种更简单的方法。

`<field>`元素还具有一个`ref`属性，可以使用外部标识符来设置many-to-one字段的值。有了这个，我们就可以为`user_id`设置值:

```
<field name="user_id" ref="base.user_demo" />
```

对于one-to-many和many-to-many的字段，需要一个相关id列表，因此需要不同的语法;Odoo在这类字段上提供了一种特殊的语法。

下面的示例从官方的Fleet应用程序中，替换了一个`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')]
  )]" />
```

要在一个多字段上写，我们使用一个三元组列表。每个三重是一个写命令，根据使用的代码做不同的事情:

`(0,_ ,{'field': value})` 创建一个新的记录并将其链接到这个记录

`(1,id,{'field': value})` 更新已链接的记录上的值

`(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,_)` 链接一个已经存在的记录

`(5,_,_)` 取消链接，但不会删除所有链接的记录

`(6,_,[ids])` 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代表无关的值，通常填充为`0`或`False`。

常用模型的快捷方式

如果我们回到第2章，构建您的第一个Odoo应用程序，我们将在XML文件中找到除`<record>`之外的元素，比如`<act_window>`和`<menuitem>`。

对于经常使用的经常使用`<record>`元素的模型，这些都是方便快捷的快捷方式。他们将数据加载到支持用户界面的基础模型中，稍后将在第6章中详细讨论——设计用户界面。

作为参考，下列快捷方式可以使用相应的模型加载数据到:

`<act_window>`是窗口操作模型`ir.actions.act_window`

`<menuitem>`是菜单项的模型，`ir.ui.menu`

`<report>`是报告操作模型，`ir.actions.report.xml`

`<template>`是用于存储在模型`ir.ui.view`中的QWeb模板

`<url>`是URL操作模型`ir.actions.act_url`

XML数据文件中的其他操作

到目前为止，我们已经了解了如何使用XML文件添加或更新数据。但是XML文件还允许执行其他类型的操作，这些操作有时需要设置数据。特别是，他们可以删除数据，执行任意的模型方法，并触发工作流事件。

删除记录

要删除数据记录，我们使用`<delete>`元素，为它提供ID或搜索域以查找目标记录。例如，使用搜索域来查找记录的删除如下:

```
<delete
  model="ir.rule"
  search="
  [('id','=',ref('todo_app.todo_task_user_rule'))]"
/>
```

因为在这种情况下，我们知道要删除的特定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" />
```

触发功能和工作流程

XML文件还可以通过<function>元素在其加载过程中执行方法。这可以用来设置演示和测试数据。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}" />
```

这调用了crm.lead模型的action_set_lost方法，通过eval属性传递了两个参数。第一个是要使用的id列表，接下来是要使用的上下文。

XML数据文件可以执行操作的另一种方式是通过<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，或者将其转换为发票。sale应用不再使用工作流，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，model属性是不言自明的，ref标识了我们正在执行的工作流实例。action是发送到这个工作流实例的工作流信号。

摘要

您已经了解了关于数据序列化的所有要点，并更好地理解了前面章节中看到的XML方面。我们还花了一些时间来理解外部标识符，特别是在一般的数据处理和模块配置中的核心概念。详细解释了XML数据文件。您了解了用于在字段上设置值和执行操作的几个选项，比如删除记录和调用模型方法。还解释了CSV文件和数据导入/导出特性。这些都是Odoo初始设置或

对数据进行大规模编辑的宝贵工具。

在下一章中，我们将详细介绍如何构建Odoo模型，并了解更多关于构建用户界面的知识。

5

模型—构造应用程序数据

在前面的章节中，我们有一个关于为Odoo创建新模块的端到端概述。在第2章，构建您的第一个Odoo应用程序，我们构建了一个全新的应用程序，在第3章，继承—扩展现有的应用程序，我们探索了继承，以及如何使用它为我们的应用程序创建一个扩展模块。在第4章模块数据中，我们讨论了如何向模块中添加初始和演示数据。

在这些超视图中，我们涉及到构建Odoo的后端应用程序的所有层。现在，在接下来的章节中，是时候解释这些层，它们组成了更详细的应用程序:模型、视图和业务逻辑。

在本章中，您将了解如何设计支持应用程序的数据结构，以及如何表示它们之间的关系。

将应用程序特性组织成模块

和以前一样，我们将使用一个示例来帮助解释概念。

Odoo的继承特性提供了一个有效的扩展机制。它允许你扩展现有的第三方应用而不直接改变它们。这种可组合性还允许以模块为导向的开发模式，大型应用程序可以分成更小的功能，足够丰富，可以独立运行。

这有助于限制复杂性，无论是在技术层面还是在用户体验层面上。从技术的角度来说，将一个大问题分解成更小的部分可以更容易地解决问题，并且对增量特性的开发更加友好。从用户体验的角度来看，我们可以选择只激活他们真正需要的功能，以简化用户界面。因此，我们将通过附加的addon模块改进我们的To-Do应用程序，以最终形成一个完整的应用程序。

引入todo_ui模块

在前面的章节中,我们首先创建了一个应用程序为个人行动计划,然后扩展它的行动计划可以与他人共享。

现在我们想通过改进它的用户界面，包括一个看板，把我们的应用程序带到下一个层次。看板是一个简单的工作流工具，它组织列中的项目，这些项目从左栏向右流动，直到完成。我们将把我们的任务组织成列，根据他们的阶段，例如**Waiting**，**Ready**，**Started**，或者**Done**。

我们将从添加数据结构开始，以实现这一愿景。我们需要添加阶段，也很好地添加对标记的支持，让任务按主题分类。在本章中，我们将只关注数据模型。这些特性的用户界面将在第6章中讨论，视图——设计用户界面，以及在第9章，QWeb和看板视图中的看板视图。

首先要弄清楚的是我们的数据是如何构建的，这样我们就能设计出支持模型。我们已经有了中心实体:待办事项。每个任务将一次处于一个阶段，任务也可以有一个或多个标记。我们将需要添加这两个额外的模型，它们将拥有这些关系:

- 每个任务都有一个阶段，每个阶段都有很多任务
- 每个任务可以有多个标记，每个标记可以附加到许多任务

这意味着任务具有与多个阶段的many-to-one关系，以及与标签的many-to-many关系。另一方面，反向关系是:阶段与任务和标记之间的关系是一个many-to-many关系。

我们将从创建新的todo_ui模块开始，并添加任务阶段和待办事项标签模型。

我们已经使用~/odoo-dev/custom-addons/目录来托管我们的模块。我们应该为新的addons创建一个新的todo_ui目录。从shell中，我们可以使用以下命令:

```
$ cd ~/odoo-dev/custom-addons
$ mkdir todo_ui
$ cd todo_ui
```

我们开始添加_manifest_.py清单文件，其中包含以下内容:

```
(
  'name': 'User interface improvements to the To-Do app',
  'description': 'User friendly features.',
  'author': 'Daniel Reis',
  'depends': ['todo_user'] }
```

我们还应该添加一个_init_.py文件。现在空着是完全可以的。

现在我们可以在Odoo工作数据库中安装这个模块，并开始使用这些模型。

创建模型

对于有看板的to-do级任务，我们需要阶段。阶段是板列，每一个任务都适合于其中一个列:

编辑todo_ui/ init .py导入models子模块:

```
from . import models
```

创建todo_ui/models目录并添加一个_init_.py文件:

```
from . import todo_model
```

现在让我们添加`todo_ui/models/todo_model.py` Python代码文件:

```
# -*- 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，我们创建了两个新模型，这些模型将在待办事项中被引用。

专注于任务阶段，我们有一个Python类, `Stage`，基于`models.Model`类，它定义了一个叫做`todo.task.stage`的新的Odoo模型。我们还有两个字段: `name`和`sequence`。我们可以看到一些新到我们的模型属性(前缀加上下划线)。让我们仔细看看。

模型属性

模型类可以使用其他属性来控制它们的某些行为。这些是最常用的属性:

`_name` 我们正在创建的Odoo模型的内部标识符。在创建新模型时必须强制执行。

`_description` 为模型的记录提供一个用户友好的标题，当模型在用户界面中被查看时显示。可选的,但是建议您这样做。

`_order` 设置当模型的记录被浏览或在列表视图中显示时使用的默认顺序。它是一个作为SQL order by子句的文本字符串，因此它可以是您可以在那里使用的任何东西，尽管它有一个聪明的行为并支持可翻译的和many-to-one个字段名。

为了完整性起见，在高级案例中还可以使用更多的属性:

> `_rec_name` 指示字段作为记录描述在相关字段引用时使用，例如many-to-one关系。默认情况下，它使用`name`字段，该字段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`is 支持模型的数据库表的名称。通常情况下，它是自动计算的，并且是模型名称，用点替换的点。但是可以设置指定一个特定的表名。

我们也可以有`_inherit`和`_inherits`属性，如第3章中解释的，继承-扩展现有的应用程序。

模型和Python类

Odoo模型由Python类表示。在前面的代码中，我们有一个基于`models.Model`类的Python类`Stage`，它定义了一个叫做`todo.task.stage`的新的Odoo模型。

Odoo模型被保存在一个中央注册中心，也被称为`pool`在旧的Odoo版本。它是一个字典，它可以引用实例中所有可用的模型类，并且可以通过模型名称引用它。具体地说，模型方法中的代码可以使用`self.env['x']`获得一个代表model `x`的类的引用。

您可以看到模型名称很重要，因为它们是访问注册表所用的键。模型名称的约定是使用与点相连的小写字母的列表，例如`todo.task.stage`。核心模块的其他例子是`project.project`, `project.task`,或`project.task.type`。我们应该使用单数形式的`todo.task`模型，而不是`todo.tasks`。出于历史原因，可以找到一些不遵循这一原则的核心模型，比如`res.users`，但它不是规则。

模型名称必须是全局惟一的。因此，第一个单词应该与模块涉及的主要应用程序相对应。在我们的例子中，它是`todo`。核心模块的其他例子是`project`, `crm`,或`sale`。

另一方面，Python类是在声明它们的Python文件的本地。用于它们的标识符只对该文件中的代码有意义。因此，类标识符不需要被它们所关联的主要应用程序预先确定。例如，为`todo.task.stage`模型命名我们的类阶段是没有问题的。在其他模块上，没有可能与可能的类具有相同的名称。

可以使用两种不同的类标识符约定：`snake_case`或`CamelCase`。从历史上看，Odoo代码使用了snake案例，而且仍然可以找到使用这个约定的类。但趋势是使用骆驼案例，因为它是由PEP8编码规范定义的Python标准。您可能已经注意到，我们使用后一种形式。

瞬态和抽象模型

在前面的代码中，在绝大多数的Odoo模型中，类是基于`models.Model`类的。这些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，并存储它们的记录，直到被显式删除。

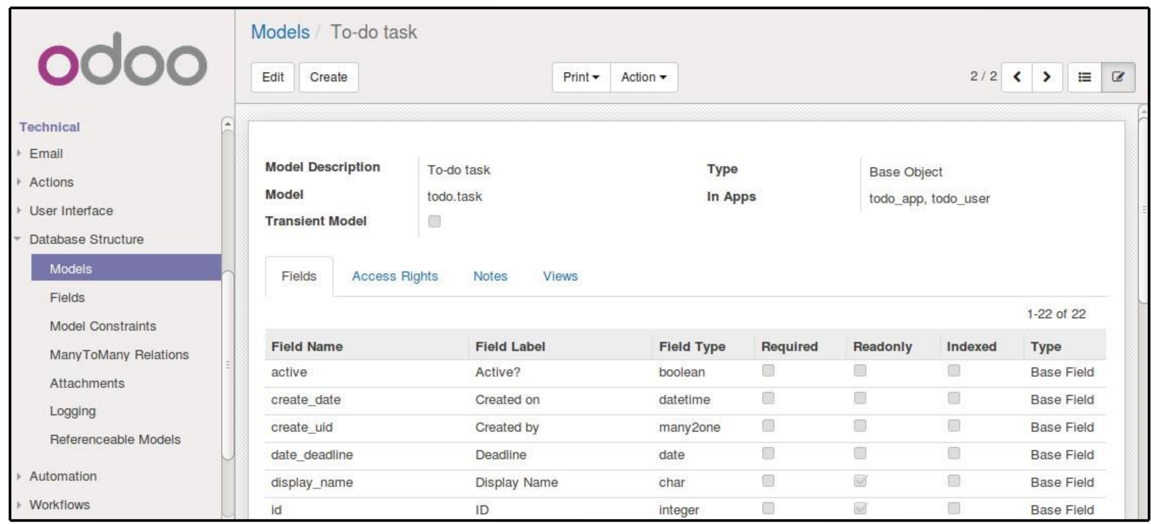
但是Odoo还提供了另外两种模型类型：瞬态和抽象模型。

> **Transient models**基于`models.TransientModel`类，并用于向导式的用户交互。他们的数据仍然存储在数据库中，但预计将是临时的。真空作业定期从这些表中清除旧数据。例如，在**Settings** | **Translations**菜单中发现的“**Load a Language**”对话框窗口，使用一个临时模型来存储用户选择并实现向导逻辑。
>
> **Abstract models**基于`models.AbstractModel`类，没有任何数据存储。它们充当可重用特性集，与其他模型混合使用，使用Odoo继承功能。例如，`mail.thread`是一个抽象模型，由`Discuss` addon提供，用于向其他模型添加消息和追随者特性。

检查现有的模型

通过Python类创建的模型和字段可以通过用户界面获得它们的元数据。在**Settings**顶部菜单中，导航到**Technical** | **Database Structure** | **Models**菜单项。在这里，您将找到数据库中可

用的所有模型的列表。点击列表中的模型将会打开一个包含其细节的表单:



这是一个很好的工具来检查模型的结构，因为在一个地方，您可以看到来自不同模块的所有自定义的结果。在这种情况下，正如您在**In Apps**字段的右上角看到的，这个模型的`todo.task`定义来自`todo_app`和`todo_user`模块。

在较低的区域，我们有一些信息标签可用:对模型的**Fields**的快速引用，对安全组授予的**Access Rights**，以及对该模型可用的**Views**。

我们可以从**Developer**菜单中找到模型的**External Identifier, View Metadata**选项。模型外部标识符(或XML id)由ORM自动生成，但很容易预测:对于`todo.task`模型，外部标识符是`model_todo_task`。

Models格式是可编辑的!可以在这里创建和修改模型、字段和视图。您可以使用它来构建原型，然后在模块中持久化它们。

创建字段

创建新模型后，下一步是向它添加字段。Odoo支持预期的所有基本数据类型，例如文本字符串、整数、浮点数、布尔值、日期、数据时间和图像/二进制数据。

一些字段名称是特殊的，因为它们是由ORM保留的特殊目的，或者因为默认使用一些默认字段名而内置了一些特性。

让我们来探索一下Odoo中可用的几种类型的字段。

基本的字段类型

我们现在有一个`Stage`模型，我们将扩展它来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，使其看起来像这样:

```
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')
```

在这里，我们有一个在Odoo中可用的非关系字段类型的示例，它们的位置参数是每个人所期望的。

在大多数情况下，第一个参数是字段标题，对应于`string`字段参数;这被用作用户界面标签的默认文本。它是可选的，如果没有提供，标题将自动从字段名中生成。

对于日期字段名，有一种约定，使用日期作为前缀。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领域，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` 期待第二个、可选、参数大小，以获得最大文本大小。建议不要使用它，除非有需要它的业务需求，比如具有固定长度的社会安全号码。

`Text` 与`Char`不同，它可以持有多行文本内容，但期望相同的参数。

`Selection` 是一个下拉选择列表。第一个参数是可选择选项的列表，第二个参数是字符串标题。选择项是(`'value'`, `'Title'`)元组的列表，用于存储在数据库中的值和相应的用户界面描述。当通过继承扩展时，`selection_add`参数可以将新项目追加到现有的选择列表中。

`Html` 存储为文本字段，但在用户界面上有特定的处理，用于HTML内容表示。出于安全原因，默认情况下它们是经过清理的，但是这种行为可以被重写。

`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`(x,y)`元组和字段的精度：`x`是数字的总数;其中`y`是小数。

`Date` 和 `Datetime` 字段只希望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。同时，

datetime值在UTC时间内存储在数据库中，但在本地时间显示，使用用户的时区首选项。在第6章中，我们将更详细地讨论这个问题——设计用户界面。

`Boolean` 如您所料，持有`True`或`False`值，并且只对字符串文本有一个位置参数。

`Binary` 存储文件如二进制数据，并且只期望字符串参数。可以使用`base64`编码的字符串来处理它们。

除了这些之外，我们还拥有关系字段，这将在本章后面介绍。但是现在，还有更多关于这些字段类型及其属性的知识。

常见的字段属性

字段具有可以在定义它们时设置的属性。根据字段类型，一些属性可以通过位置传递，没有参数关键字，如前一节所示。

例如，example, `name=fields.Char('Name', 40)`可以使用位置参数。使用关键字参数，同样可以写成`name=fields.Char(size=40, string='Name')`。关键字参数的更多信息可以在Python官方文档中找到: `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，用于用户界面。除了选择和关系字段之外，它是第一个位置参数，所以大部分时间它不被用作关键字参数。

`Default` 为字段设置默认值。它可以是一个静态值，例如字符串，或可调用引用，一个命名函数或一个匿名函数(lambda表达式)。

`size` 只适用于`Char`字段，可以设置允许的最大大小。当前的最佳实践是，除非

真的需要，否则不要使用它。

translate 只适用于Char、Text和Html字段，并使字段内容可以翻译，对不同的语言持有不同的值。

help 为显示给用户的工具提示提供文本。

readonly=True 默认情况下，在用户界面上不编辑字段。这不是在API级别执行的;它只是一个用户界面设置。

required=True 在用户界面中默认设置字段。这是通过在列上添加NOT NULL约束来在数据库级别执行的。

index=True 将在字段上创建数据库索引。

copy=False 当使用重复记录功能，copy()ORM方法时，字段被忽略。默认情况下，非关系型字段是copyable。

groups 允许将字段的访问和可见性限制为一些组。它期望一个逗号分隔的安全组XML id列表，比如groups='base.group_user,base.group_system'

states 根据state字段的值，期望一个字典映射的UI属性值。例如:可以使用的states=('done':[('readonly',True)]}属性是readonly、required和invisible。

请注意，states字段属性与视图中的attrs属性相当。请注意，视图支持states属性，但它有不同的用法:它接受一个逗号分隔的状态列表，以控制元素何时应该可见。

为了完整性，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'` 当一个字段重命名为新版本时使用，使旧字段中的数据自动复制到新字段中。

特殊字段名称

ORM保留了一些字段名。

`id`字段是一个自动编号，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中。

在新模型上自动创建以下字段，除非设置`_log_access=False`模型属性:

`create_uid` 是为创建记录的用户吗

`create_date` 当记录创建为`write_uid`的日期和时间时，是否为最后一个用户修改记录

`write_date` 记录修改后的最后一次日期和时间

此信息可从web客户端获得，导航到**Developer Mode**菜单并选择**View Metadata**选项。

默认情况下，一些API内置的特性期望特定的字段名。我们应该避免使用这些字段名，而不是用于预期目的。有些甚至是保留的，不能用于其他目的:

`name`被默认用作记录的显示名称。通常它是一个`Char`，但也可以是一个`Text`或`Many2one`字段类型。我们仍然可以使用`_rec_name`模型属性设置一个用于显示名称的字段。

`Active`级的`Boolean`，允许有未激活的记录。与`active==False`的记录将自动被排

除在查询之外。要访问它们，必须将`('active','=',False)`条件添加到搜索域，或者`'active_test': False`应该添加到当前上下文。

`Sequence`类型的`Integer`，如果出现在列表视图中，允许手动定义记录的顺序。为了正常工作，您不应该忘记使用它与模型的`_order`属性。

类型为`Selection`的`State`，表示记录生命周期的基本状态，并可由state的字段属性使用，以动态修改视图:某些表单字段可以在特定的记录状态中生成`readonly`，`required`，或`invisible`。

`parent_id`，`parent_left`，和`parent_right`是`Integer`类型的，对父/子层次关系有特殊的意义。我们将在下一节详细讨论它们。

到目前为止，我们讨论了非关系字段。但是应用程序数据结构的一个重要部分是描述实体之间的关系。现在我们来看看这个。

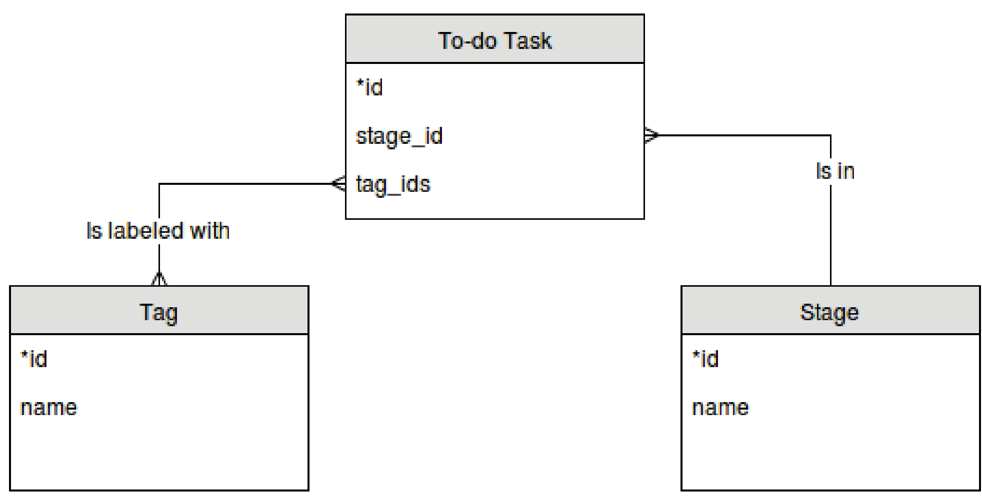
之间的关系模型

再看一下我们的模块设计，我们有这样的关系:

每个任务都有一个阶段。这是一个many-to-one的关系，也称为外键。逆是one-to-many关系，意思是每个阶段都可以有很多任务。

每个任务可以有多个标记。这是一个many-to-many关系。当然，反向关系也是many-to-many的，因为每个标记都可以有很多任务。

下面的实体关系图可以帮助可视化我们将要在模型上创建的关系。以三角形结尾的线代表了关系的许多方面:



让我们将相应的关系字段添加到`todo_model.py`文件中的待办事项:

```
class TodoTask(models.Model):
    _inherit = 'todo.task'
    stage_id = fields.Many2one('todo.task.stage', 'Stage')
    tag_ids = fields.Many2many('todo.task.tag', string='Tags')
```

前面的代码显示了这些字段的基本语法，设置了相关的模型和字段的标题`string`。关系字段名的约定分别是对字段名的`_id`或`_ids`，分别是to-one和to-many关系。

作为练习，您可以尝试将相应的反向关系添加到相关模型:

> `Many2one`关系的逆是一个分阶段的`One2many`字段，因为每个阶段都可以有很多任务。我们应该把这个字段添加到Stage类。
>
> `Many2many`关系的逆也是标签上的`Many2many`字段，因为每个标签也可以用于许多任务。

让我们仔细研究一下关系字段定义。

Many-to-one的关系

`Many2one`关系接受两个位置参数:相关模型(对应于`comodel`关键字参数)和标题`string`。它在数据库表中创建一个带外键的字段。

还有一些命名参数可以用于这类字段:

- `ondelete` 定义删除相关记录时会发生什么。它的默认值为`null`，这意味着在删除相关记录时将设置空值。其他可能的值是`restrict`，提高了防止删除的错误，并且`cascade`也删除了这个记录。
- `context` 是数据字典，对web客户端视图有意义，在处理关系时携带信息。例如，设置默认值。在第六章中可以更好地解释，视图——设计用户界面。
- `domain` 是一个域表达式，一个元组列表，用来筛选关系字段可用的记录。
- `auto_join=True` 允许ORM在使用此关系进行搜索时使用SQL连接。如果使用，访问安全规则将被绕过，用户可以访问安全规则不允许的相关记录，但SQL查询将会更高效，运行更快。

Many-to-many的关系

`Many2many`最小签名接受与相关模型的一个参数，并推荐使用字段标题提供`string`参数。

在数据库级别，它不向现有表添加任何列。相反，它会自动创建一个新的关系表，只有两个ID字段与相关表的外键。关系表名和字段名是自动生成的。关系表名是两个表名，其中添加了一个下划线，后面加上了`_rel`。

在某些情况下，我们可能需要重写这些自动默认值。

其中一个例子是相关模型有长名称，而自动生成的关系表的名称太长，超过了63个字符的PostgreSQL限制。在这些情况下，我们需要手动选择关系表的名称，以符合表名称的大小限制。

另一个例子是，我们需要在同一模型之间建立第二个many-to-many关系。在这些情况下，我们需要手动为关系表提供一个名称，以便它不会与已经用于第一个关系的表名发生冲突。

有两种方法可以手动覆盖这些值:要么使用位置参数，要么使用关键字参数。

使用位置参数来定义字段定义:

```
# 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，附加参数是可选的。我们可以为关系表设置名称，并让字段名使用自动默认值。

我们可以使用关键字参数，有些人喜欢可读性:

```
# 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')
```

就像many-to-one字段一样，many-to-many字段也支持`domain`和`context`关键字属性。

在ORM设计中有一个限制，关于抽象模型，当您强制关系表和列的名称时，它们就不能被干净地继承了。在抽象模型中不应该这样做。

`Many2many`关系的倒数也是`Many2many`级。如果我们还在`Tags`模型中加入`Many2many`域，那么Odoo就会发现这种many-to-many关系与`Task`模型中的一个相反。

任务和标记之间的反向关系可以像这样实现:

```
class Tag(models.Model):
    _name = 'todo.task.tag'
    # Tag class relationship to Tasks:
    task_ids = fields.Many2many(
        'todo.task',    # related model
        string='Tasks')
```

One-to-many的逆关系

`Many2one`的逆可以添加到关系的另一端。这对实际的数据库结构没有影响，但是允许我们轻松地从**many**相关记录的**one**端浏览。一个典型的用例是文档头和它的行之间的关系。

在我们的示例中，阶段上的`One2many`反向关系允许我们轻松地列出该阶段的所有任务。将这种逆关系添加到阶段的代码是:

```
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`接受三个位置参数:相关模型、该模型中的字段名称，以及标题字符串。前两个位置参数对应于`comodel_name`和`inverse_name`关键字参数。

附加的关键字参数与`Many2one`: `context`、`domain`、`ondelete` (在此关系的**many**方面)和`auto_join`相同。

分级的关系

父-子树关系用与相同模型的`Many2one`关系表示，因此每个记录都引用其父节点。而反向`One2many`可以让父母很容易地跟踪孩子。

Odoo为这些层次结构的数据结构提供了改进的支持，用于更快地浏览树的兄弟姐妹，以及在域表达式中使用额外的`child_of`运算符进行更简单的搜索。

为了启用这些特性，我们需要设置`_parent_store`标志属性，并将其添加到模型的帮助字段: `parent_left`和`parent_right`。注意，这个额外的操作是在存储和执行时间上进行的，所以最好在你期望阅读的时候比写作的时候更频繁，比如一棵分类树的例子。

重新访问todo_model.py文件中定义的Tags模型，我们现在应该像这样编辑它:

```
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)
```

在这里，我们有一个基本模型，有一个parent_id字段来引用父记录，另一个_parent_store属性添加层次搜索支持。这样做时，还必须添加parent_left和parent_right字段。

引用父类的字段预计将被命名为parent_id，但只要我们在_parent_name属性中声明，任何其他字段名称都可以使用。

此外，在记录的直接子项中添加一个字段通常很方便:

```
child_ids = fields.One2many(
    'todo.task.tag', 'parent_id', 'Child Tags')
```

使用动态关系的参考字段

常规关系字段引用一个固定的模型。参考字段类型没有这种限制并支持动态关系，因此同一个字段可以引用多个模型。

例如，我们可以使用它在To-do Tasks添加Refers to字段，它可以引用一个User或Partner:

```
# class TodoTask(models.Model):
    refers_to = fields.Reference(
```

```
        [('res.user', 'User'), ('res.partner', 'Partner')],
        'Refers to')
```

如您所见，字段定义类似于选择字段，但这里的选择列表保留了可以使用的模型。在用户界面上，用户将首先从av可用列表中选择一个模型，然后从该模型中选择一个记录。

可以采用另一种级别的灵活性: **Referenceable Models**配置表用于配置在**Reference**字段中使用的模型。它可以在**Settings | Technical | Database Structure**菜单中使用。在创建这样一个字段时，我们可以在`referenceable_models()`模块中使用`odoo.addons.res.res_request`函数的帮助，将其设置为在那里注册的任何模型。

使用**Referenceable Models**配置，`Refers to`字段的改进版本如下所示:

```
from odoo.addons.base.res.res_request import referenceable_models
# class TodoTask(models.Model):
    refers_to =
        fields.Reference( referenceable_mo
        dels, 'Refers to')
```

注意，在Odoo 9.0中，该函数使用了略微不同的拼写，并且仍然使用旧的API。在第9.0版本中，在使用之前显示的代码之前，我们必须在Python文件的顶部添加一些代码来包装它，以便它使用新的API:

```
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)
```

计算字段

字段可以具有由函数计算的值，而不是简单地读取数据库存储的值。一个计算字段就像一个常规字段一样声明，但是它具有定义用于计算它的函数的额外的`compute`参数。

在大多数情况下，计算字段涉及到编写一些业务逻辑，因此在第7章，ORM应用程序逻辑支持业务流程中，我们将进一步开发这个主题。我们仍然会在这里解释它们，但是将保持业务逻辑的尽可能简单。

让我们来看一个例子: Stage有一个`fold`字段。我们会在To-do Tasks的基础上加上**Folded?**标记对应的阶段。

我们应该在`todo_model.py`文件中编辑`TodoTask`模型来添加以下内容:

```
# 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`字段和用于计算它的`_compute_stage_fold`方法。函数名作为字符串传递，但是它也可以作为可调用引用传递给它(没有引号的函数标识符)。在这种情况下，我们应该确保在字段之前，在Python文件中定义函数。

当计算依赖于其他字段时，需要`@api.depends` decorator，因为它通常这样做。它让服务器知道何时重新计算存储或缓存的值。一个或多个字段名被接受为参数，并使用点符号来跟踪字段关系。

计算函数将赋值给字段或字段以进行计算。如果没有，就会出错。由于`self`是一个记录对象，我们这里的计算只是为了获得**Folded?**使用`stage_id.fold`字段。结果是通过将该值(写入)赋给计算字段，即`stage_fold`。

对于这个模块，我们目前还没有工作，但是您可以在任务表单上快速编辑，以确认计算字段是否按预期工作:使用**Developer Mode**选择**Edit View**选项，并直接在表单XML中添加字段。别担心:它会被下一次升级的干净模块视图所取代。

在计算字段中搜索和写入

我们刚刚创建的计算字段可以读取，但不能搜索或写入。为了启用这些操作，我们首先需要为它们实现专门的功能。除了compute函数，我们还可以设置一个search函数，实现搜索逻辑，以及inverse函数，实现写逻辑。

使用这些，我们计算的字段声明变成这样:

```
# 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')
```

支持功能是:

```
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
```

当在搜索域表达式中找到该字段的(field, operator, value)条件时，调用search函数。它接收了operator和value的搜索，并期望将原始的搜索元素转换成另一个域搜索表达式。

inverse函数执行计算的反向逻辑，找到在计算的源字段上写的值。在我们的例子中，这意味着在stage_id.fold字段上写回。

存储计算字段

计算字段的值也可以存储在数据库中，根据它们的定义设置`store = True`。当它们的依赖项发生变化时，它们将被重新计算。由于现在存储了这些值，因此可以像普通字段一样搜索它们，而不需要搜索函数。

相关领域

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自己的字段中。然而，这是一种常见的用法，可以由Odoo自动处理。

使用相关字段也可以实现同样的效果。它们直接在一个模型上提供，这些字段属于一个相关的模型，可以使用点符号链访问。这使得它们在不能使用点符号的情况下可用，比如UI表单视图。

要创建一个相关的字段，我们声明一个需要类型的字段，就像使用常规计算的字段一样，但是我们不需要计算，而是使用点符号字段链的相关属性来达到期望的字段。

To-do Tasks是在可定制的阶段组织的，这些阶段将转换成基本状态。我们将使状态值直接在任务模型上可用，以便在下一章中可以用于一些客户端逻辑。

与`stage_fold`类似，我们将在任务模型中添加一个计算字段，但是这次使用了更简单的相关字段:

```
# class TodoTask(models.Model):
    stage_state = fields.Selection(
        related='stage_id.state',
        string='Stage State')
```

在幕后，相关字段只是计算字段，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附加代码的情况下搜索和写入它们。

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约束被添加到数据库表定义中，并由PostgreSQL直接执行。它们是使用`_sql_constraints`级属性定义的。它是一个元组列表:约束标识符名;约束的SQL;以及要使用的错误消息。

一个常见的用例是为模型添加惟一的约束。假设我们不想允许两个具有相同标题的活动任务:

```
# class TodoTask(models.Model):
    _sql_constraints =
        [ ('todo_task_name_uniq
        ',
         'UNIQUE (name, active)',
         'Task title must be unique!')]
```

Python约束可以使用任意代码来检查条件。检查功能应该用`@api.constraints`来装饰，表示检查中涉及的字段的列表。当其中任何一个被修改时，验证被触发，如果条件失败，将引发一个异常。

例如，为了验证一个任务名至少有5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!')
```

摘要

我们对模型和字段进行了详细的解释，使用它们来扩展To- Do应用程序，并在任务上进行标记和阶段。您学习了如何定义模型之间的关系，包括分层的父/子关系。最后，我们看到了使用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。

6

视图—设计用户界面

本章将帮助你学习创建供用户与To-Do应用程序交互的图形化界面。您将发现许多可用的不同类型的视图和小部件，了解上下文(context)和域(domain)是什么，并学习如何使用它们来提供良好的用户体验。

我们将继续使用`todo_ui`模块。它已经准备好了模型层，现在它需要用户界面的视图层。

使用XML文件定义用户界面

用户界面的每个组件都存储在数据库记录中，就像业务记录一样。模块通过从XML文件中加载相应的数据，把UI元素添加到数据库中

这意味着需要将新的XML数据文件添加到`todo_ui`模块中。第一步，我们可以编辑`manifest_.py`文件来声明这些新数据文件:

```
{
  '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',
  ]}
```

请记住，数据文件是按照您指定的顺序装载的。这很重要，因为您只能引用在之前定义的XML id。

我们还可以创建子目录和views/todo_view.xml和 views/todo_menu.xml文件，文件的最小结构如下:

```
<?xml version="1.0"?>
<odoo>
  <!-- Content will go here... -->
</odoo>
```

在第三章,继承-扩展现有的应用程序*(Inheritance – Extending Existing Applications)*中，我们给应用创建了一个基本的菜单，但是我们现在想要改进它。因此，我们将添加新的菜单项和相应的窗口操作，当它们被选中时将被触发。

菜单项目

菜单项目存储在ir.ui.menu模型，可以通过**Settings**菜单下的**Technical** | **User Interface** | **Menu Items**来浏览。

todo_app addon创建了一个顶级菜单来打开To-Do应用。现在我们想要去把它修改成一个二级菜单并且在它旁边有其他菜单选项。

To do this, we will add a new top-level menu for the app and modify the existing To-Do task menu option. To views/todo_menu.xml, add:

为此，我们将为应用程序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" />
```

我们可以使用更为方便的`<menuitem>`快捷元素来代替`<record model="ir.ui.menu">`元素，它提供了一种简化的方式来定义载入记录。

我们的第一个菜单项是To-do应用的顶级菜单项，只使用`name`属性，并将作为接下来两个菜单的父菜单项。

注意，它使用了现有的XML ID `todo_app.menu_todo_task`，因此它可以重写`todo_app`模块中定义的菜单项，而不附加任何动作(action)。这是因为我们将添加子菜单项，而打开Task视图的动作现在将从其中一个调用。

下一个菜单项目位于顶级菜单项目下，通过`parent="todo_app.menu_todo_task"`属性来实现。

打开Task视图的正是我们要创建的第二个菜单，通过`action="todo_app.action_todo_task"`来实现。通过调用的XML ID，可以了解到我们再次使用了之前已经在`todo_app`模块创建过的动作(action)。

第三个菜单项添加了我们的应用程序的**Configuration**功能，我们希望它只对超级用户可用，所以我们同时使用groups属性使它只对**Administration | Settings**安全组可见。

最后，在**Configuration**菜单下，我们为任务阶段(Stage)添加选项。我们将使用它来维护（之

后会添加到模块中的）看板特性所使用的阶段。

现在，如果我们试图升级我们的`addon`，会提示错误，因为我们还没有定义`action`属性中使用的XML ID。我们将在下一节中添加它们。

窗口动作

window action向GUI客户机提供指令，通常用菜单项或视图中的按钮来调用。它告诉GUI要处理什么模型(model)，以及提供什么视图(view)。使用`domain`筛选器，这些动作可以只让记录集的一个子集可见。
它们还可以通过`context`属性设置默认值和默认过滤器

我们将向`views/todo_menu.xml`文件中添加窗口动作，它将被上一节中创建的菜单项使用。编辑文件，并确保它们添加在菜单项之前:

```
<!-- 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，并且可以通过使用以上代码中展示的`<act_window>`标签在XML文件中定义窗口动作。

第一个动作将打开任务阶段(Task Stages)模型，并包含窗口动作的最相关属性:

- `name` 是通过此操作打开的视图的标题.
- `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`，将在弹出对话框窗口中打开视图。默认情况下，它是`current`，在主内容区域内以内联方式打开视图。
- `context` 设置目标视图的上下文信息，可以设置默认值或激活过滤器等。稍后我们将详细讨论这个问题。
- `domain`是域表达式，一个强制筛选在打开的视图中可浏览的记录的过滤器。
- `limit`是列表视图中每页的记录数量。

`XML`中定义的第二个动作将替换`todo_app addon`的初始待办任务`(To-do Tasks)`操作，以便它显示在我们在本章后面将探讨的其他视图类型中：日历`(calendar)`和图表`(graph)`。安装这些更改后，您将在右上角看到额外的按钮，在列表和表单按钮之后；但是，在创建相应的视图之前，这些方法是无效的。

我们还添加了第三个动作，不用于任何菜单项。它向我们展示了如何为**More**菜单添加选项，在列表的右上角，并形成视图。为此，它使用了两个特定的属性:

- `src_model`表明可以使用该动作的模型。
- `multi`，当设置为 `True`，使它在列表视图中可用，而且可以应用于多个记录的选择。其默认值为`False`，正如我们的例子中展示的，它将使该选项仅在表单视图中可用，因此只能一次应用于一个记录。

上下文和域

我们已经好几次偶然发现了上下文和域。我们已经可以在窗口动作中设置它们，并且模型中的关系型字段也可以将它们作为属性。

上下文数据

context是一个字典，它承载会话数据，可以在客户端用户界面和服务器端ORM和业务逻辑中使用。

在客户端，它可以将信息从一个视图传递到下一个视图，比如在点击一个链接或一个按钮之后，前序视图中的活跃记录ID，或者提供后序视图中使用的默认值。

在服务器端，一些记录集的字段值可以依赖于上下文提供的地区设置。特别是`lang`键会影响可翻译字段的值。上下文还可以为服务器端代码提供信号。例如，当`active_test`键设置为`False`时，这将更改ORM的`search()`方法的行为，它就不会过滤掉非活动记录。

web客户机的初始上下文看起来是这样的:

```
('lang': 'en_US', 'tz': 'Europe/Brussels', 'uid': 1}
```

您可以看到使用用户语言的lang键，时区信息的tz和当前用户ID的`uid`。

当在上一个视图中用一个链接或一个按钮打开表单时，会将一个`active_id`键添加到上下文中，键值为我们之前所打开的记录的ID。在特定情况下，如列表视图，我们的上下文中有一个`active_ids`键，其中包含在前一个列表中选择的记录id的列表。

在客户端，可以在目标视图用上下文设置默认值或激活默认过滤器，使用`default_`或`default_search_`前缀的键。下面是一些例子:

要将当前用户设置为`user_id`字段的默认值，我们将使用以下语句:

```
{'default_user_id': uid}
```

要在目标视图上默认激活的`filter_my_tasks`过滤器，我们将使用以下语句:

```
{'default_search_filter_my_tasks': 1}
```

域表达式

domain用于筛选数据记录。它们使用一种特定的语法，Odoo ORM 可以解析生成用于查询数据库的SQL WHERE表达式。

域表达式是一个条件列表。每个条件都是一个`('field_name', 'operator', value')`元组。例如，这是一个有效的域表达式，只有一个条件：`[('is_done','=',False)]`。

以下是对于这些元素的解释：

field name是被过滤的字段，可以在关系模型的字段中使用点标记。

value被视作Python表达式。它可以使用文字值，例如数字、布尔值、字符串或列表，并且可以使用在赋值上下文中的字段和标识符。实际上有两种可能的域的赋值上下文:

- 当在客户端使用时，例如在窗口动作或字段属性中，可在用于渲染当前视图的原始字段值上使用域表达式，但是我们不能在它们上使用点标记
- 当在服务器端使用时，例如在安全记录规则和服务器Python代码中，点标记可以在字段上使用，因为当前记录是一个对象。

operator可以是：

- 一般的比较运算符如 `<`, `>`, `<=` , `>=`, `=`, `!=`.
- `'=like'`与一种表达式匹配，其中下划线符号, `_`，匹配任意一个符号，同时百分比符号, `%`,匹配任意字符序列.
- `'like'`匹配一种 `'%value%'`表达式。而 `'ilike'`类似但是不分大小写。同时
-

也可以使用`'not like'` 和 `'not ilike'` 运算符。

`'child_of'`在继承关系中查找子值。

- `'in'`和 `'not in'`用于检查是否包含在给定列表中，因此该值应该是一个值列表。当在“to-many”关系字段中使用时，`in`运算符就像一个`contains`运算符。

域表达式是条件的列表，可以包含多个条件元组。默认情况下，这些条件将隐式地使用AND逻辑运算符进行结合。这意味着它只返回满足所有这些条件的记录。

显式逻辑运算符也可以使用：`'&'`符号，表示与(AND)运算(默认)；管道符号`'|'`用于或(OR)运算。这些将在接下来的两个条件上运行，以递归的方式工作。我们稍后会详细讨论这个问题。

感叹号`'!'`也可使用，表示否(NOT)运算符，作用于下一个项目。所以，它应该放在条件被否定之前。例如，`['!', ('is_done','=',True)]`表达式将过滤所有未完成的记录。

“下一个条件”也可以是另一个运算符，定义嵌套的条件。一个例子可以帮助我们更好地理解这一点。

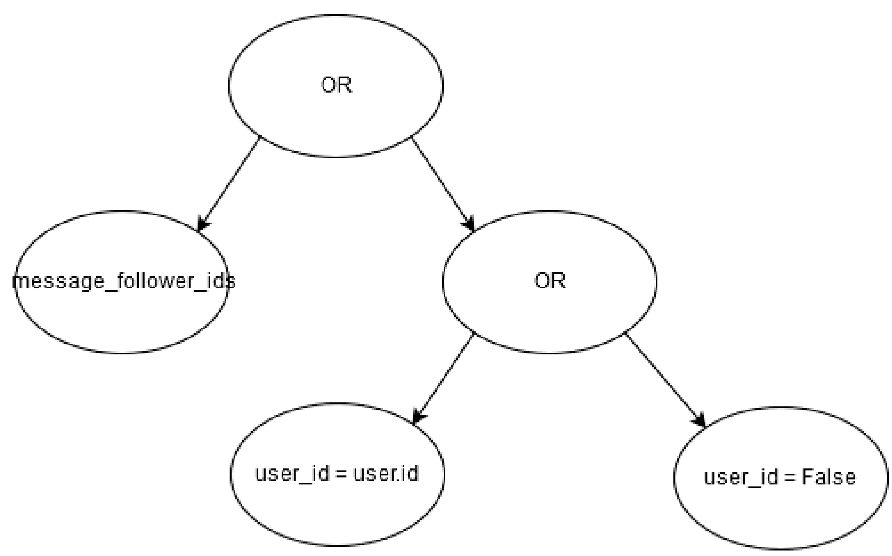
在服务器端记录规则中，我们可以找到类似于这个的域表达式：

```
['|', ('message_follower_ids', 'in',
       [user.partner_id.id]),
       '|', ('user_id', '=', user.id),
       ('user_id', '=', False)
]
```

这个域会对所有记录进行过滤，条件为：当前用户在这条记录的关注者列表中（即为相关用户），或者不存在相关用户列表。符合条件时显示记录。

第一个`'|'`(或)运算符作用于关注者条件加上下一个条件的结果。下一个条件是另外两个条件的结合:记录上的用户ID是当前会话用户，或者没有设置。

下图演示了这个嵌套运算符的解析过程:



表单视图

正如我们在前面几章中看到的，表单视图可以遵循一个简单的布局或业务文档布局规则，类似于一个纸质文档。

现在，我们将了解如何设计这些业务文档视图以及如何使用可用的元素和小部件。我们通常通过继承和扩展`todo_app`的视图来实现这一点。但是为了清晰起见，我们将创建完全新的视图来覆盖原始视图。

处理相同类型的多个视图

同一个模型可以有多个相同类型的视图。这是很有用的，因为窗口动作可以通过它的XML ID告知应该使用的特定视图，因此我们可以灵活地使用两个不同的菜单项，以使用不同的视图打开同一个模型。这是通过给窗口动作添加`view_id`属性，并将该属性赋值为视图的XML ID来实现的。例如，我们可以在`todo_app.action_todo_task`动作中使用它，其内容类似于:`view_id="view_form_todo_task_ui"`.。

但是如果没有特定的视图定义会发生什么呢?在这种情况下，使用的将是查询视图时返回的第一个实例。这将调用优先级较低的视图。如果我们添加一个新视图并将其设置为低于现有的优先级，那么将优先使用它。最后的效果是，看起来这个新视图覆盖了原来的视图。

因为视图优先级的默认值是16，所以任何较低的值都可以产生这个效果，所以15优先级将会起作用。

它并不是最常用的方法，但为了尽可能地使我们的示例具备可读性，我们将在下一个示例中使用优先级方法。

业务文档视图

业务应用程序通常是记录系统——在仓库中的产品，在会计部门的发票，以及更多的(记录)。大多数记录的数据都可以用纸质文档表示。为了获得更好的用户体验，表单视图可以模拟这些文档。例如，在我们的应用程序中，我们可以把To-Do任务(Task)看作是一个需要填写的简单纸质表单。我们将提供一个遵循此设计的表单视图。

要添加具备业务文档视图的基本框架的视图XML，我们应该编辑`views/todo_views.xml`文件并将其添加到顶部:

```
<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>
        <!-- To add buttons and status widget -->
      </header>
      <sheet>
        <!-- To add form content -->
      </sheet>
      <!-- Discuss widgets for history and
      communication: -->
      <div class="oe_chatter">
        <field name="message_follower_ids"
          widget="mail_followers" />
        <field name="message_ids" widget="mail_thread" />
      </div>
    </form>
  </field>
</record>
```

视图名是可选的，如果未设置，则自动生成。为了简单起见，我们利用了这一点，并从视图记录中省略了`<field name="name">`元素。

我们可以看到，业务文档视图通常使用三个主要区域: **header**状态栏、主要内容的**sheet**、底部**history and communication**部分，也称为**chatter**。

在底部，历史和通信部分使用了mail addon模块提供的社交网络小部件。为了能够使用它们，我们的模型应该继承泛型模型 `mail.threa`，正如我们在第三章,继承-扩展现有的应用程序*(Inheritance – Extending Existing Applications)*中所见的。

标题

顶部的标题通常包含文档将会经过的生命周期或步骤，以及操作按钮。

这些操作按钮是常规的表单按钮，最重要的下一步可以用`class="oe_highlight"`来高亮显示。

文档生命周期通过在某个字段上使用`statusbar`小部件来实现，它代表当前文档所在的生命周期中的点。这通常是**State**选择字段或**Stage** many-to-one字段。这两个字段可以在多个Odoo核心模块中找到。

阶段(stage)是many-to-one字段，它用一个辅助模型来设置业务流程中的步骤。由于这一点，终端用户可以动态地配置它以适应其特定的业务流程，并且非常适合看板。

状态是一个选择列表，其中有一些在业务过程中非常稳定的步骤，例如`New`, `In Progress`和`Done`。它不能由终端用户配置，但是，由于它是静态的，所以在业务逻辑中更容易使用它。视图字段甚至对其有特殊的支持:可以依据文档状态允许或不允许用户使用某些字段。

从历史上看，阶段是晚于状态引入的。两者目前共存，但Odoo核心的趋势是阶段取代状态。但正如前面的解释所示，状态仍然提供一些阶段不具备的特征。

通过将各个阶段映射到状态，仍然可以从两个世界的精华中获益。这是我们在前一章中所做的，通过在To-do Task模型中添加一个状态字段，并通过一个计算字段使它可以在待办任务(To-do Task)文档中使用，从而启用state字段属性。

在views/todo_view.xml 文件中，我们现在可以添加状态栏来扩展基本标题:

```
<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>
```

这里我们将state添加为隐藏字段。我们需要这一点来强制客户端在发送到服务器的数据请求中包含该字段。否则它将无法用于表达式。

重要的是要记住，您希望在域(domain)中或attrs表达式中使用的任何字段都必须加载到视图中，因此在您需要它们的时候，您将使字段不可见，但不需要用户看到它们。

接下来在状态栏中添加一个按钮，让用户可以切换到任务的**Done**标记

状态栏显示的按钮应该根据当前文档在命周期中的位置进行更改。

当文档处于draft状态时，我们使用attrs属性隐藏按钮。这样做的条件是使用不用在表单上显示的状态字段，这就是为什么我们必须将它添加为一个隐藏字段。

如果我们有一个state选择字段，我们可以使用states属性。在这种情况下，我们可以使用states="open,done"来实现相同的效果。虽然它不像attrs属性那样灵活，但更简洁。

这些可见性特性也可以用于其他视图元素，比如字段。我们将在本章后面更详细地讨论它们。

`clickable`属性允许用户通过直接点击状态栏来更改文档阶段。我们通常希望启用这个功能，但是也有一些我们不需要的情况，比如当我们需要对工作流进行更多的控制，并且要求用户只使用可用的操作按钮来完成阶段的工作时，这样这些操作就可以在文档所处阶段更改前进行验证。

当使用状态栏小部件与阶段时，我们可以将很少使用的阶段隐藏在一个更大的阶段组中。为此，阶段模型必须有一个标记来配置要隐藏的阶段，这个标记通常命名为`fold`。同时`statusbar`小部件应该使用`options`属性，如前面的代码所示，并将该字段名提供给`fold_field`选项。

当使用带有状态字段的状态栏小部件时，可以使用`statusbar_visible`属性实现类似的效果，该属性用于列出应该总是可见的状态，并为不常见的情况隐藏异常状态。例如:

```
<field name="stage_id" widget="statusbar"
  clickable="True"
  statusbar_visible="draft,open" />
```

表单

表单画布是表单中放置实际数据元素的主要区域。它被设计成看起来像一个实际的纸质文档，并且通常可以把Odoo中的记录被称为**documents**。

通常，文档表单的结构具有这些区域:

- 顶部的文档标题和副标题
- 在右上角的一个按钮框
- 其他文档标题字段
- 一个笔记本，用于选项卡或页面中组织的其他字段。明细行也会在这里，通常在第一个笔记本页面.

让我们看一下每个区域。

标题和副标题

在<group>元素之外的字段不会有自动生成的标签。这将是标题元素的情况，因此要使用<label for"..."/>元素来渲染它。这样需要多一些额外的工作，但好处是对标签显示拥有更多的控制。

常规的HTML，包括css样式的元素，也可以用来使标题更多彩。为了获得最好的结果，标题应该在一个oe_title 类的<div>标签内。

这里是<sheet>元素扩展到包含标题加上一些附加字段作为副标题:

```
<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处显示它。for属性标识我们应该从该字段中获取标签文本。另一种可能性是使用string属性为标签提供指定的文本。我们的示例还使用class="oe_edit_only"属性，这样它只能在编辑模式下可见。

在某些情况下，例如合作伙伴或产品，一个头像会显示在左上角。假设我们有一个my_image二进制字段，我们可以在<div class="oe_title">行之前添加:

```
<field name="my_image" widget="image" class="oe_avatar"/>
```

智能按钮区

右上方的区域可以有一个可以放置按钮的隐形框。8.0版本引入了智能按钮，显示为矩形，有一个统计指标，可以在单击时进行跟踪。

我们可以在oe_title DIV 结束后添加按钮框，如下:

```
<div name="buttons" class="oe_right oe_button_box">
  <!-- Smart buttons here … -->
</div>
```

按钮的容器是一个带有oe_button_box和oe_right类的div，以使其与表单的右边对齐。我们将在后面的小节中更详细地讨论按钮，因此我们将等到在这个框中添加实际的按钮时介绍相关知识。

表单中的内容分组

表单的主要内容应该使用<group>标签进行组织。group标签在画布中插入两列，在默认情况下，字段将和自己的标签一齐显示

字段值和字段标签包含两列，因此在组中添加字段将使它们垂直堆叠。如果我们将两个

<group>元素嵌套在一个顶级组中，我们将能够得到两列有标签的字段，并排显示。

继续修改我们的表单视图，现在我们可以添加智能按钮框后的主要内容:

```
<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>
```

给组标签加上name是一种很好的做法，以后可以更容易引用它们来扩展视图(由您或其他开发人员)。string属性也是可用的，如果设置，则用于显示章节标题。

在组内，<newline>元素将强制产生一个新行，从而下一个元素将在组的第一列中呈现。可以在一个组中使用<separator>元素添加额外的章节标题。

从 **Developer**菜单中尝试**Toggle Form Layout Outline**:它在每个组部分绘制线条，以便更好地理解当前表单布局。.

我们可以使用col和colspan属性对组元素的布局进行更好的控制。

可以将col属性用于<group>元素，以设置它将包含的列数。默认值是2，但可以更改为任何其他数字。数值在在这种设置下显示的效果更好，因为即便默认情况下，每个字段也添加了两个列，标签加上字段值。

组内的元素，包括<field>元素，可以使用colspan属性来设置它们应该使用的特定列数。默认情况下，元素占用一列。

选项卡式的笔记本

另一种组织内容的方法是使用`notebook`元素，其中包含多个选项卡部分，称为页面。这些方法可以将某些数据隐藏等到需要的时候再显示，或者根据主题组织大量的字段。

我们不需要将这个添加到我们的待办任务(To-do Task)表单中，但是这里有一个可以添加到任务阶段(Task Stages)表单的示例:

```
<notebook>
  <page string="Whiteboard" name="whiteboard">
    <field name="docs" />
  </page>
  <page>
    <!-- Second page content -->
  </page>
</notebook>
```

视图语义组件

我们已经了解了如何使用结构组件(如header、group和notebook)来组织表单中的内容。现在，我们可以仔细查看语义组件、字段和按钮，以及我们可以用它们来做什么。

字段

视图字段有一些属性可供使用。它们中的大多数都有从模型中定义的值，但是这些值可以在视图中被重写。

通用的，并不依赖于字段类型的属性：

- `name`标识字段的数据库名字
- `string` 是标签文本，在我们想要重写模型定义中提供的标签文本时使用
- `help`是当您将指针悬停在字段上时显示的工具提示文本，并允许覆盖模型定义提供的帮助文本。
- `placeholder`是在字段内部显示的提示文本。
- `widget`允许我们覆盖用于该字段的默认小部件。我们将在稍后讨论可用的小部件。
- `options`是一个JSON数据结构，它为小部件提供了更多的选项，范围则依赖于每个小部件所支持的内容。
- `class`是字段在HTML渲染时使用的CSS类。
- `nolabel="True"`防止自动显示字段标签。它只对`<group>`元素中的字段有意义，并且常常与`<label for="...">`元素一起使用。
- `invisible="True"`会隐藏字段，但是它的数据将从服务器获取，并且在表单上可用。
- `readonly="True"`让字段在表单上不可编辑。
- `required="True"`让字段强制必填。

只有特定类型的字段可使用的属性:

- `password="True"`用于文本(text)字段。它作为一个密码字段显示，以掩蔽键入的字符。
- `filename` 使用于二进制字段，它是用于存储上传文件名称的模型字段的名称
- `mode`用于`one-to-many`字段，它指定用于显示记录的视图类型。默认情况下，它是`tree`，但也可以是`form`，`kanban`，或`graph`。

字段标签

可以使用`<label>`元素更好地控制字段标签的表示。使用这种方法的一个例子是，只有当表单处于编辑模式时才显示标签:

```
<label for="name" class="oe_edit_only" />
```

这样做时，如果字段位于一个`<group>`元素内，我们通常也需要设置`nolabel="True"`。

关系型字段

在关系型字段上，我们可以对用户的功能进行一些额外的控制。默认情况下，用户可以从这些字段创建新的记录(也称为“快速创建”)并打开相关的记录表单。可以使用`options`字段属性禁用此属性:

```
options=('no_open': True, 'no_create': True}
```

上下文和域在关系字段上也特别有用。上下文可以为相关记录定义默认值，域可以限制可选择的记录。一个常见的例子是top有一个字段中可选择的记录列表，它依赖于当前记录的另一个字段的当前值。域可以在模型中定义，但是在视图中也可以被覆盖。

字段小部件

每个字段类型都搭配以的默认小部件显示在表单中。但是还可以使用额外的的小部件。

对于文本字段，我们有以下小部件:

- `email`用于让电子邮件文本地址，成为可操作的”发邮件到”地址。
- `url`用于格式化文本成为可点击的URL。
- `html` 用于将文本呈现为HTML内容;在编辑模式中，它以所见即所得的编辑器为特征，使内容的格式不需要使用HTML语法。

对于数值型字段，我们有以下小部件：

- `handle`专门为列表视图中的序列字段设计，并显示一个允许您将行拖到制定顺序的“把手”。
- `float_time` 将浮点字段格式化为小时和分钟。
- `monetary` 显示一个浮点字段作为货币数量。它需要一个`currency_id`作为伴随字段，但是另一个字段名也可以通过`options="('currency_field': 'currency_id'}"`来提供。
- `progressbar` 展示一个浮点数作为进度百分比，对于表示完成率的字段非常有用。

对于关系和选择字段，我们有这些额外的小部件:

- many2many_tags 用若干个按钮形状的标签来展示多个数据。
- selection 可在many-to-one字段上可以使用 selection 字段小部件。
- radio用单选按钮展示 selection 字段的选项。
- kanban_state_selection 显示看板状态选择列表的信号灯。正常状态用灰色表示，用绿色表示，任何其他状态用红色表示
- priority 将 selection字段呈现为一个可点星标的列表。选项通常是数字。

按钮

按钮支持这些属性:

- icon用于在按钮中显示图标图像;与智能按钮不同，普通按钮可用的图标仅限于在addons/web/static/src/img/icons中的图标。
- string是按钮的标签文本， 或者当使用图标时，是HTML 的alt文本.
- type 是将要使用的动作类型。有下列可能的取值 :
    - workflow被用来启动工作流引擎信号 ;
    - object被用来调用Python方法;
    - action被用来运行一个窗口动作。
- name标识执行的具体操作，根据所选的类型:要么是工作流信号名，要么是模型方法名，要么是运行窗口操作的数据库ID。%(xmlid)d公式可用于将XML ID转换为所需的数据库ID。
- args在 type是 object的时候使用，将附加的参数传递给方法。
- context向上下文添加值，在运行窗口动作或调用Python代码方法后，会产生影响。
- confirm显示一个确认消息框，并将文本分配给该属性。

- `special="cancel"`用于向导(wizard)，可以取消(操作)和关闭向导(wizard)界面。

智能按钮

在设计表单结构时，我们包含了一个可以放置智能按钮的右上角区域。现在让我们在里面添加一个按钮。

对于我们的应用程序，我们将有一个按钮，显示当前待办任务所有者拥有的待办任务总数，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模型models/todo_model.py中。在TodoTask类中添加以下内容：

```
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，我们在里面添加按钮框和按钮。在oe_title结束后DIV，替换我们之前添加的按钮框占位符，如下：

```
<div name="buttons" class="oe_right oe_button_box">
  <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>
</div>
```

这个按钮显示负责这个任务的人的所有待办任务的总数，由user_todo_count字段计算。

这些是我们在添加智能按钮时可以使用的属性：

- class="oe_stat_button" 渲染一个矩形按钮而不是普通的按钮 。
- icon用来设置要使用的图标，可从Font Awesome组合中选择。可用的图标可以在http://fontawesome.io上浏览。
- type 和 name是按钮类型和触发动作的名称。对于智能按钮，类型通常是action，若是窗口动作(window action)，name将是将要执行动作的ID。它需要一个实际的数据库ID，因此我们必须使用一个公式将XML ID转换为数据库

ID: "%(action- external-id)d"。此操作会打开展示相关记录的视图。

- string 在按钮上添加标签文本。我们在这里没有使用它，因为包含的字段已经为它提供了文本。
- context会用于在目标视图上设置默认值，以便在单击按钮后在视图上创建的新记录上使用。
- help添加一个帮助工具提示，当鼠标指针放置在按钮上时显示。

button元素本身是一个容器，其中有显示统计信息的字段。这些字段都是使用小部件statinfo的常规字段。字段应该是在底层模型中定义的一个计算字段。除了字段，在一个按钮中我们还可以使用静态文本，例如：

```
<div>User's To-dos</div>
```

单击按钮时，我们希望看到一个列表，其中只有当前用户的任务。这将由action_todo_task_button动作完成，但这个动作尚未实现。它需要了解当前用户，才能执行过滤器。为此，我们使用按钮的context属性来存储该值。

使用的动作(Action)必须在表单(Form)之前定义，所以我们应该将其添加到XML文件的顶部：

```
<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)]" />
```

请在这里留心我们是怎么使用在域筛选器中使用上下文键default_user_id。这个特定的键还会在按下按钮创建新任务(Task)时被设定为user_id字段的默认值。

动态视图

视图元素还支持一些动态属性，允许视图根据字段值动态更改外观或行为。我们可以在编辑表单数据触发`on change`事件时，更改其他字段上的值，或者只有在满足某些条件时才将字段变为强制填写或可见。

On change事件

当特定字段发生更改时，**on change**机制允许我们更改表单中其他字段中的值。例如，当产品更改时，产品字段上的on change可以设置`Price`字段的默认值。

在旧版本中，`on change`事件是在视图层定义的，但是自从`8.0`版本以后，它们是直接在模型层上定义的，而不需要对视图上的任何特定标记。这是通过创建方法来执行计算和使用`@api.onchange('fieldl', 'field2')`将其绑定到字段。这些`onchange`方法在第`7`章，`ORM`应用程序逻辑—支持业务流程（*ORM Application Logic – Supporting Business Processes*）中有更详细的讨论。

动态属性

`on change`机制还负责计算字段的自动重新计算，以立即对用户输入作出反应。使用与之前相同的示例，如果在更改产品时更改了`Price`字段，则使用新的`Price`信息自动更新计算的`Total Amount`字段。两个属性提供了一种简单的方法来控制特定用户界面元素的可见性：

- `groups`可以根据当前用户所属的安全组使元素可见。只有指定组的成员才能看到它。使用时需要一个逗号来分隔的不同组`XML id`。
- `states`可以根据记录的状态字段使元素可见。它需要一个逗号分隔不同的状态值。

除了这些之外，我们还提供了一种灵活的方法，可以根据客户端动态赋值的表达式设置元素的可见性。这是特殊属性`attrs`，需要有一个值字典将`invisible`属性的值映射到一个表达式的

结果。

例如，要让`refers_to`字段在除草稿之外的所有状态中可见，请使用以下代码：

```
<field name="refers_to" attrs="('invisible':
  state=='draft'}" />
```

`invisible`属性在任何元素中都可用，而不仅仅是字段。例如，我们可以在笔记本页面和组元素中使用它。

`attrs`还可以为其他两个属性设置值：`readonly`和`required`。这些只对数据字段有意义，使它们不可编辑或强制填写。这允许我们实现一些基本的客户端逻辑，例如根据其他记录值（比如状态）使某个字段强制填写。

列表视图

现在，列表视图应该不需要怎么介绍了，但我们仍将讨论可以与它们一起使用的属性。下面的例子，是我们待办任务一个列表视图：

```
<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属性来完成的，该表达式基于字段属性进行求值。名称部分可以是bf或it，可实现粗体和斜体字体，或任何Bootstrap的文本上下文颜色：danger, info, muted, primary, success, 或 warning。Bootstrap文档中有例子来说明这些效果：

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

8.0版本中提供的colors和fonts属性在9.0版本中被弃用。目前应该使用新的装饰属性。

请记住，表达式中使用的字段必须在<field>元素中声明，以便web客户端知道该列需要从服务器检索。如果我们不想让它对用户可见，我们应该使用它上面的invisible="1"属性。

树图的其他相关属性包括：

- default_order允许覆盖模型的默认排序顺序，其值遵循与模型定义中使用的order属性相同的格式。
- create, delete, 和 edit, 如果设置为 false （小写） 禁用列表视图上相应的操作。
- editable可以直接在列表视图上编辑元素。 可能的取值为 top和 bottom, 代表了新的记录将会被添加在底部或者顶部。

列表视图可以包含字段和按钮，它们的大部分属性在这里也是有效的。

在列表视图中，数字字段可以显示其列的合计值。为此，在字段中添加一个可用的聚合属性，sum、avg、min或max，而其等号后的字符串则会显示为合计值的标签。例如：

```
<field name="amount" sum="Total Amount" />
```

搜索视图

可用的搜索选项需要通过`<search>`类型视图定义。我们可以选择在输入搜索框时自动搜索字段。我们还可以提供预定义的过滤器，通过单击激活，以及在列表视图中使用预定义的分组选项。

下面是一个可行的任务搜索视图：

```
<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>
```

我们可以看到要搜索的两个字段- `name`和`user_id`。当用户开始在搜索框上键入时，下拉菜单将建议在这些字段中的任意一个搜索。在用户敲击`ENTER`后，将在第一个筛选字段中执行搜索。

然后我们有两个预定义的过滤器，过滤未完成和已完成任务。这些过滤器可以独立激活，并将用一个或运算符连接。与<separator/>元素分隔的过滤器将用一个和运算符连接。

第三个过滤器只设置了一个group by上下文。这告诉视图根据这个字段将记录分组，在本例中为user_id。

字段元素可以使用以下属性：

- name 标识要使用的字段。
- string设置后将会取代默认标签文本。
- operator 用于从更改默认的运算符(数值字段的=，其他字段为ilike).
- filter_domain 设置用于搜索的特定域表达式，为运算符属性提供一个灵活的选择。搜索的文本字符串在表达式中引用为self。一个简单的例子是：filter_domain="[('name', 'ilike', self)]".
- groups只对属于某些安全组的用户进行搜索。需要使用由逗号分隔XML ID列表。

对于筛选器元素，这些是可用的属性：

- name是继承筛选器时或通过窗口动作启用它时的标识符。不是强制性的，但这是一个很好的习惯。
- string 是筛选器的标签文本，必需。
- domain是需要加入当前域的域表达式。
- Context 是一个上下文字典，将被添加到当前上下文。通常设置一个group_id键，并赋值为作为分组依据的字段的名字。
- groups 使搜索只提供给在安全组列表中的用户(XML IDs)。.

日历视图

顾名思义，这个视图类型在日历中显示记录，可以查看的范围有一个月、一周或几天等。待办任务(To-Do Tasks)的日历视图可以是这样的:

```
<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]" >
      <!-- Fields used for the display text -->
      <field name="name" />
      <field name="stage_id" />
    </calendar>
  </field>
</record>
```

The calendar attributes are:

- `date_start` is the field for the start date. Mandatory.
- `date_end` is the field for the end date. Optional.
- `date_delay` is the field with the duration in days, that can be used instead of `date_end`.
- `all_day` provides the name of a Boolean field that is to be used to signal full day events. In these events, the duration is ignored.
- `color` is the field used to group color the calendar entries. Each distinct value in this field will be assigned a color, and all its entries will have the same color.
- `display` is the display text for each calendar entry. It can user record values using the field names between square brackets, such as `[name]`. These fields must be declared as child of the calendar element, an in the preceding example.
- `mode` is the default display mode for the calendar, either `day`, `week`, or `month`.

日历视图的属性有:

- `date_start` 用来表示开始日期。必填。
- `date_end` 表示结束日期， 选填。
- `date_delay` 表示持续时间是几天，可以用来替代`date_end`.

- `all_day`提供用于表示全天事件的布尔字段的名称。在这些事件中，持续时间被忽略。
- `color` 是用来对日历条目进行颜色分组的字段。这个字段中的每个不同的值将被分配一个颜色，并且具有相同值的条目都将具有相同的颜色。
- `Display` 是每个日历条目的显示文本。它可以使用方括号之间的字段名称来使用记录的值，例如`[name]`。这些字段必须声明为日历元素的子元素。
- `mode` 是日历的默认显示模式，可以设置为`day`， `week`， 或 `month`.

图表和透视视图

图表视图，以图表的形式提供数据的图形化视图。当前在待办任务中可用的字段不适合用于图表，因此我们将先添加一个字段以供图标视图使用。

在`TodoTask`类中，在`todo_ui/models/todo_model.py`文件中，添加：

```
effort_estimate = fields.Integer('Effort Estimate')
```

它还需要添加到待办任务(`To-do Task`)表单中，这样我们可以在已有的记录中添加值，之后就能够检查这个新视图。

现在让我们来添加待办任务(To-Do Tasks)的图表视图：

```
<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>
```

The `graph` view element can have a `type` attribute that can be set to `bar` (the default), `pie`, or `line`. In the case of `bar`, the additional `stacked="True"` can be used to make it a stacked bar chart.
`graph`视图元素有一个`type`属性，可以设置为`bar`(默认)、`pie`或`line`。在`bar`的情况下，附加的`stacked="True"`可以用来做堆叠的条形图。

数据也可以在透视表中查看，形式为一个动态分析矩阵。为此，我们在9.0版本中引入了透视视图。虽然在8.0版本中已经有了透视视图，但是在9.0中，才独立成为一种视图类型。在此基础上，改进了数据透视表的UI特性，极大地优化了数据透视表数据的检索。

还需要将一个pivot表添加到要待办任务(To-Do Tasks)中，请使用以下代码：

```
<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>
```

图表和透视视图应该包含描述纵轴和测度的字段元素。两种视图的大多数的属性都是通用的：

- name表示将要使用在图表中的字段，类似其他类型视图
- type是字段使用的方式，作为row组(默认)、measure或者col（仅用于透视表，用于列组)
- interval对日期字段有意义，并且用于按日、周、月、季度或年(day, week, month, quarter, 或 year)分组时间数据。

默认情况下，使用的聚合是数值的总计量。这可以通过在Python字段定义上设置group_operator属性来更改。可以使用的值包括avg、max和min。

其他视图类型

值得注意的是，我们没有涉及另外三种视图类型：看板(kanban)，甘特图(gantt)和图形(diagram)。

我们将在第九章，QWeb与看板视图(QWeb and Kanban Views)中详细介绍看板视图。

甘特视图在8.0之前是可用的，但是由于许可证不兼容，它在社区版本9.0中被删除了。

最后，图形视图用于特定的案例，而addon模块则很少需要它们。以防万一你可能想了解这两种视图类型，参考资料中可以在官方文档中找到，https://www.odoo.com/documentation/10.0/reference/views.html。

小结

在本章的总结中，我们学习了更多关于`Odoo`视图的知识，涵盖了最重要的几种视图类型，并用之构建用户界面。在下一章中，我们将进一步了解如何将业务逻辑添加到应用程序中。

7

ORM应用逻辑—支撑业务流程

使用`Odoo`编程`API`，我们可以编写复杂的逻辑和向导(`wizard`)来为我们的应用程序提供丰富的用户交互。在本章中，我们将看到如何编写模型中的代码来支撑业务逻辑，我们还将学习如何在事件和用户操作中激活它。

我们可以在事件上执行计算和验证，比如对记录执行创建或写入操作，或者在单击按钮时执行一些逻辑。例如，我们之前为待办任务添加的按钮动作，用于切换**Is Done**的标志，并用归档来清理所有已完成的任务。

此外，我们还可以使用向导来实现与用户的更复杂的交互，允许在交互过程中请求输入并提供反馈。

我们从为待办程序构建一个这样的向导开始。

创建向导

假设我们的待办任务应用程序的用户，经常需要设定最后期限和负责大量任务的人。他们可以让助手来帮忙。程序应该允许他们选择要更新的任务，然后在任务上设置最后期限和/或负责用户。

向导是用来从用户那里获取输入信息的表单，然后使用这些信息进行进一步的处理。它们可以用于简单的任务，比如请求一些参数和生成报表，或者用于复杂的数据操作，比如前面描述的用例。

我们的向导外观应该看起来如下：

To-Do Tasks Wizard

Count Get All

Description	Responsible	Deadline
Create dev database!	Administrator	
Install Odoo	Administrator	01/30/2015
Add an item		

Set Responsible

Set Deadline 12/31/2016

Mass Update Cancel

我们可以从为todo_wizard特性创建一个新的addon模块开始。

我们的模块将有一个Python文件和一个XML文件，所以todo_wizard/_manifest_.py描述如下代码所示:

```
(  'name': 'To-do Tasks Management Assistant',
   'description': 'Mass edit your To-Do backlog.',
   'author': 'Daniel Reis',
   'depends': ['todo_user'],
   'data': ['views/todo_wizard_view.xml'], }
```

和以前的许多addon一样，todo_wizard/_init_.py文件只有一行：

```
from . import models
```

接下来，我们需要描述支持向导的数据模型。

向导模型

向导会向用户显示一个表单视图，通常是一个对话框窗口，其中包含一些字段。这些将被向导逻辑所使用。

这是使用与常规视图相同的模型/视图架构实现的，但支持模型是基于models.TransientModel而不是models.Model。

这种类型的模型也会在数据库有一张表存储，但是这些数据只在向导完成工作结束前有用。一个定时的任务会定期清理向导数据库表中的旧数据。

models/todo_wizard_model.py文件将定义我们需要与用户交互的字段:要更新的任务列表、负责用户和设置它们的截止日期。

首先添加models/_init_.py，代码如下:

```
from . import todo_wizard_model
```

之后创建真正的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，与普通模型之间的one-to-many关系不适用于这种暂态模型。原因是，它将要求普通模型具有与暂态模型相反的many-to-one关系，但这是不允许的，因为可能导致清理暂态模型记录时，将普通模型的记录一并清理了。

向导表单

除两个特定元素外，向导表单视图与普通模型相同：

- 一个 `<footer>`区域可以用来放置操作按钮
- 一个特殊的 `type= "cancel"`按钮可以跳出向导而不执行任何操作

以下是我们`views/todo_wizard_view.xml`文件中的内容：

```
<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>
```

通过使用`src_model`属性，我们在XML中看到的`<act_window>`窗口动作将一个选项添加到要做的任务表单的**More**按钮中。`target="new"`属性使其打开为对话窗口。

您可能还注意到，在**Mass Update**按钮中使用的`attrs`，为该按钮添加了一个很棒的特性：直到选择一个新的截止日期或负责的用户，按钮才变为可见。

向导业务逻辑

接下来，我们需要实现在表单按钮上执行的操作。不包括**Cancel**按钮，我们有三个操作按钮需要实现，但是现在我们将重点关注**Mass Update**按钮。
按钮调用的方法是`do_mass_update`，它应该在`models/todo_wizard_model.py`文件中，代码如下所示：

```
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
```

我们的代码应该一次处理一个向导实例，所以我们使用`self.ensure_one()`确保这一点。这里`self`表示向导表单数据的记录集。

该方法首先验证是否填写了一个新的截止日期或负责的用户，如果没有通过则抛出一个错误。接下来，我们有一个如何将调试消息写入服务器日志的示例。

然后，`vals`字典由提供给批量更新的值构建:新日期、新负责人或两者都有。接着，在记录集上使用`write`方法来执行批量更新。这比在每个记录上重复执行写操作的循环结构更有效。

让方法总是返回一定结果，这是一个好的习惯。这就是为什么它在最后返回`True`的原因。这样做的惟一原因是XML-RPC协议不支持`None`值，因此使用该协议时将无法使用没有返回值的方法。实际上，您可能没有意识到这个问题，因为`web`客户机使用的是JSON-RPC，而不是XML-RPC，但它仍然是一个很好的习惯。

接下来，我们将进一步了解日志，然后将研究顶部两个按钮的逻辑：**Count**和**Get All**。

日志

这些批量更新可能会被误用，所以在使用时将一些信息记录到日志可能是个好主意。前面的代码在`TodoWizard`类两行之前，使用`Python logging`标准库初始化了`_logger`。`Python`

name的内部变量被用来识别来自这个模块的消息。

我们可以使用如下代码在方法中写日志信息:

```
_logger.debug('A DEBUG message')
_logger.info('An INFO message')
_logger.warning('A WARNING message')
_logger.error('An ERROR message')
```

当传递在日志消息中使用值时，我们应该将它们作为额外的参数提供给它们，而不是使用字符串插值。例如，而不是我们应该使用`_logger.info('Hello %s', 'World')`而不是`_logger.info('Hello %s' % 'World')`。您可能注意到，我们在`do_mass_update()`方法中这样做了。

关于日志记录的一个有趣的事情是，日志条目总是使用UTC格式来打印时间戳。对于新管理员来说这可能是一个惊喜，但这是由于服务器内部一直是使用UTC格式来处理所有日期。

抛出异常

当出现某些问题时，我们会想中断程序并发出一个错误消息。这是通过抛出异常来实现的。Odoo为Python提供了一些额外的异常类。以下是最常用的例子:

```
from odoo import exceptions
raise exceptions.Warning('Warning message')
raise exceptions.ValidationError('Not valid message')
```

`Warning`消息也可以中断执行，但可能听起来不像`ValidationError`那样严重。虽然它不是最好的用户界面，但是我们在**Count**按钮上利用它来显给用户显示一条(错误)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)
```

顺便说一下，因为此方法不操作`self`记录集，看起来我们可以使用 `@api.model` 修饰器。但在这种情况下，我们不能这样做因为这个方法需要从一个按钮调用。

向导中的帮助动作

现在假设我们想要一个按钮，自动地选择所有要做的任务，用户就不用逐个选择它们。这就是表单中**Get All**按钮的目的。此按钮背后的代码将获得一个具有所有活跃任务的记录集，并将其分配给`many-to-many`字段中的任务。

但这里有个问题。在对话框窗口中，当按下某个按钮时，向导窗口将自动关闭。我们没有处理使用**Count**按钮时会遇到的这个问题，因为它会通过抛出异常来显示它的消息；如果动作不成功，窗口就不会关闭。

幸运的是，我们可以通过让客户机重新打开同一个向导来处理这种操作。模型方法可以用字典对象的形式返回窗口动作，由`web`客户端执行。这个字典使用的属性和`XML`文件中用于定义窗口动作的属性是相同的。

我们将为窗口动作字典定义一个`helper`函数重新打开向导窗口，以便在几个按钮中轻松地重用它：

```
@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，窗口动作可能有其他作用，比如跳转到不同的向导表单来请求额外的用户输入，并且可以用来实现多页向导。

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']
    all_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可用的模型:我们首先使用`self.env[]`以获得对模型的引用,在本例中为`todo.task`。然后可以在其上执行操作，例如`search()`检索满足某些搜索条件的记录。

暂态模型存储了向导表单字段中的值，而且可以像其他模型一样被读取或写入。`all_tasks`变量被赋值给模型的`task_ids`（`one-to-many`）字段。正如您所看到的，这就像我们对任意其他字段类型所做的一样。

使用ORM API

从上一节中，我们已经了解了使用`ORM API`的感觉。接下来，我们来看看我们还能做些什么。

方法修饰器

在我们的旅程中，我们遇到几个使用了`API`修饰器的方法，如`@api.multi`。这对服务器了解如何处理方法非常重要。让我们概括一下有哪些可用的修饰器和什么时候该使用它们。

`@api.multi`修饰器用于处理带有新`API`的记录集，它是最常用的。这里`self`是一个记录集，方法通常使用一个`for`循环来迭代它。

在某些情况下，方法会只需要一个单例：一个包含不超过一个记录的记录集。`@api.one` 修饰符在`9.0`版本被弃用，应该避免使用它。相反，我们仍然应该使用`@api.multi`添加到方法代码中，与`self.ensure_one()`保持一致，以确保它是一个单例。

如前所述，`@api.one` 修饰符被弃用，但仍然被支持。为了完整起见，可能需要知道它封装了其修饰的方法，并一次为它提供一个记录，执行记录集迭代。在我们的方法中，`self`保证是一个单元素。每个方法调用的返回值被聚合为一个列表并返回。

类级别的静态方法使用`@api.model`修饰，它不使用任何记录集数据。为了保持一致性，`self`仍然是一个记录集，但是它的内容是无关的。注意，这种方法不能被用户界面的按钮所使用。

另外有一些目的更明确的修饰符，可与前面描述的修饰符一起使用：

- @api.depends(fldl,...) 用于计算字段函数，以确定哪些（字段值）更改应该触发（重新）计算。
- @api.constrains(fldl,...) 用于验证函数，以确定哪些更改应该触发验证检查。
- @api.onchange(fldl,...) 用于`on change`函数，以标识表单上将触发该操作的字段。

特别是，`onchange`方法可以向用户界面发送警告消息。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，但不会阻止用户继续使用。方法通过返回描述警告消息的字典来实现：

```
return (
         'warning': ( 'title':
         'Warning!',
         'message': 'You have been warned'}
```

```
}
```

重写ORM默认方法

我们已经了解了API提供的标准方法，但现在可以继续深入！我们还可以扩展它们来给模型添加自定义行为。

最常见的情况是扩展create()和write()方法。这可以用于向其中添加一些逻辑，在执行这些操作时触发。通过将我们的逻辑放置在重写方法的适当位置，我们可以在执行主操作之前或之后运行逻辑代码。

以TodoTask模型为例，我们可以创建一个自定义的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
```

一个自定义的write()同样遵循这种结构：

```
@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，不过对于模型可用的任何标准方法，都可用类似的方式继承并添加自定义逻辑。

这些技术提供了许多可能性，但请记住，其他工具可能更适合于常见的特定任务：

- 需要一个字段值基于另一个字段的值，我们应该使用计算字段。其中的一个示例是在更改某一明细行值时计算总数量。
- 为了动态地计算字段默认值，我们可以使用一个依赖函数返回值的默认值，而不是

一个固定值。

- 当字段值被更改时，其他字段上的值也要更改，我们可以使用`on change`函数。例如，选择客户时，将其货币设置为文档的货币，然后由用户手动更改。记住，`on change`只在表单视图交互中起作用，而不是直接调用`write`方法。
- 对于验证操作，我们应该使用`@api.constraints(fldl,fld2,...)`的约束函数。它们类似于计算字段（方法），但是，它们不是计算值，而是会（验证不通过时）抛出异常。

RPC和web客户端的方法

我们已经看到了用于生成记录集的最重要的模型方法以及如何在编写它们。但是还有一些应用于更具体操作的模型方法，如下所示:

- `read([fields])`与`browse`方法类似，但它不返回记录集，而是返回一个数据行列表，数据字段来源于方法的参数。每行都是一本字典。它提供了可以通过RPC协议发送的序列化数据，并用于客户机程序，而不用于服务器逻辑。
- `search_read([domain], [fields], offset = 0,limit = None,order = None)`执行搜索操作(search)，然后在结果列表上执行读取(read)操作。它用来给RPC客户机使用，并为节省了先search，再read所需的额外的时间。
- `load([fields],[data])`用于从CSV文件中导入数据。第一个参数是要导入odoo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，每条记录都是字符串的列表，而字符串都会被解析和导入。这个参数直接映射到CSV数据行和列。这个函数实现了第4章模块数据(*Module Data*)中描述的CSV数据导入的特性，包括外部标识符支持等。web客户端**Import**功能使用了此函数。它替换了已弃用的`import_data`方法。
- `export_data([fields], raw_data=False)`被web客户端**Export**功能使用。它使用包含数据的数据键返回字典;一个列表的行。字段名称可以使用在CSV文件中使用的`.id`和`/id`后缀，数据的格式与可输入的CSV文件兼容。可选的`raw_data`参数允许用Python类型导出数据值，而不是CSV中使用的字符串表示。

web客户端主要使用以下方法呈渲染用户界面并进行基本交互：

- `name_get()`:方法将返回一个(ID、name)元组的列表，其中包含每个记录的显示名称。默认情况下，它用于计算`display_name`值，或提供关系型字段的显示名称。可以扩展它来实现记录的自定义显示名，例如同时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)` name参数是用户在字段中的输入文本，方法默认会将该模型所有记录的显示名称和name参数进行匹配查找，最后返回一个(ID, name)元组的列表。它在UI中使用，在关系型字段中输入文本时，生成与输入文本相匹配的建议记录。例如，产品查找使用并扩展此函数，字段中键入文本时，在名称和引用中都进行匹配查找，来返回满足条件的产品记录(的选择列表)。
- `name_create(name)` 创建一条新记录，并仅使用标题名称来创建(其他数据为空)。它在UI中用于"快速创建"(quick create)功能，您可以通过在关系型字段中仅输入名称然后立即创建一条记录。可以通过扩展此方法，将通过该特性创建的新记录的某些字段，设置指定的默认值。
- `default_get([fields])` 返回一个带有新的默认值的字典。要创建记录。默认值可能取决于诸如当前用户或会话上下文等变量。
- `fields_get()`用于描述模型的字段定义，会显示在开发人员菜单的**View Fields**选项中。
- `fields_view_get()`用来检索在web客户端渲染UI视图的结构。可以将视图的ID作为参数，或者我们想使用的视图类型，如`view_type='form'`等。例如，您可以尝试以下方法：`rset.fields_view_get(view_type='tree')`。

shell命令

Python有一个命令行界面，这是探索其语法的很好方法。类似地，Odoo也有一个类似的特性，我们可以交互地尝试输入命令来查看它们是如何工作的。这就是shell命令。

要使用它，可以使用shell命令和数据库运行Odoo，如下所示：

```
$ ./odoo-bin shell -d todo
```

您应该在终端中看到和通常一样的服务器启动行，直到它停在>>>的Python提示符，并等待您的输入。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'
```

我们将对环境进行一些检查。self表示一个res.users记录集，它只包含有ID 1的记录，我们还可以通过检查self._name确认recordset的模型名称，并获取记录的name字段的值，用来确认它是Administrator用户。

与Python一样，您可以使用*Ctrl* + *D*退出，这也将关闭服务器进程并返回系统shell界面。

shell特性是在9.0版本中增加的。对于8.0版本，有一个社区反向移植模块来添加它。一旦下载并包含在addons路径中，就不需要进一步的安装。下载地址：https://www.odoo.com/apps/modules/8.0/shell/。

服务器环境

服务器shell提供了与您在用户模型res.users中的方法中所发现的相同的self引用。

正如我们所见，`self`是一个记录集。**Recordsets**携带一个环境信息，包括用户浏览数据和其他上下文信息，如语言和时区。这个信息很重要。

我们可以使用以下命令开始检查当前的环境：

```
>>> self.env
<openerp.api.Environment object at 0xb3f4f52c>
```

`self.env`中的执行环境具有以下属性：

- `env.cr` 数据库游标
- `env.uid` 当前会话用户的`ID`
- `env.user` 当前用户的记录
- `env.context` 包含会话上下文的不可变字典

环境还提供对注册表的访问，注册表所有已安装的模型都可被调用。例如，`self.env['res.partner']`返回对合作伙伴模型的引用。然后，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)
```

在本例中，`res.partner`模型的一个记录集包含两个记录，`ID`为`7`和`51`。

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，然后使用它运行操作。

These methods can be used for that:

这些方法可用于此目的：

- `env.sudo(user)`需要向方法提供一个用户记录，然后返回一个包含该用户的环境。如果没有提供用户，则将默认使用`Administrator`超级用户，这样可以绕过安全规则来运行指定的查询。
- `env.with_context(dictionary)`用一个新的上下文替换原有的。
- `env.with_context(key=value,...)`在当前上下文中修改（或增加）一些键值对。

此外，我们还有`env.ref()`函数，在传入一个外部标识符后返回一个对应的记录，如下所示：

```
>>> self.env.ref('base.user_root')
res.users(1,)
```

事务和底层SQL

数据库写入操作是在数据库事务的上下文中执行的。通常，我们不需要担心这个问题，因为服务器在运行模型方法的时候会处理这个问题。

但在某些情况下，我们可能需要对事务进行更好的控制。可以通过数据库游标 `self.env.cr`来完成这个目的，如下所示：

- `self.env.cr.commit()` 提交事务的缓冲写入操作
- `self.env.savepoint()` 设置一个事务保存点来回滚
- `self.env.rollback()`取消自上个保存点以来的事务的写操作，若没有创建保存点，则全部取消

`shell`会话中，在使用`self.env.cr.commit()`之前，数据操作不会在数据库中生效。

使用游标的`execute()`方法，我们可以直接在数据库中运行`SQL`语句。它需要一个带`SQL`语句的字符串才能运行，另外还可以选择传入格式化包含一个数组或一个值列表作为`SQL`语句的参数。这些值将替换`SQL`字符串中的对应 `%s` 占位符（`Python` 格式化字符串语法）。

请注意!

对于`cr.execute()`，我们应该避免直接将参数值添加到查询字符串中。众所周知，这样会引发`SQL`注入攻击的安全风险。应该总是使用`%s`占位符和第二个参数来传递值。

如果您使用的是`SELECT`查询，那么会得到若干记录。`fetchall()`函数会检索所有的行（即

记录）并返回一个元组的列表，而`dictfetchall()`检索并返回一个字典的列表，如下例所示：

```
>>> 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')]
```

运行**Data Manipulation Language (DML)**指令也是允许的，如`UPDATE`和`INSERT`。由于服务器会保存数据缓存，导致它们可能与数据库中的实际数据不一致。因此，在使用原始`DML`时，应该使用`self.env.invalidate_all()`先清除缓存。

请注意!
在数据库中直接执行`SQL`会导致数据不一致。只有当你确信自己在做什么么时，你才应该使用它。

使用记录集

我们现在将探索`ORM`如何工作，并了解如何使用它执行最常见的操作。我们将使用`shell`命令提供的提示，来交互式地探索记录集的工作方式。

查询模型

对于`self`，我们只能访问方法的记录集。但`self.env`环境引用，允许我们访问任意模型。例如，`self.env['res.partner']`返回对合作伙伴模型的引用（实际上是一个空的记录集）。然后，我们可以使用`search()`或`browse()`来生成其中的记录集。

`search()`方法需要传入一个域表达式，之后返回一个匹配这些条件的记录集。传入空域`[]`，则将返回所有记录。有关域表达式的更多细节，请参阅第6章，视图—设计用户界面（*Views – Designing the User Interface*）。如果该模型有特殊字段`active`，默认情况下只会考虑`active=True`的记录。

一些可选的关键字参数，如下所示：

- `order` 是在数据库查询中用作`ORDER BY`子句的字符串。字符串内容通常是一个由逗号分隔的字段名列表。
- `limit` 设置要检索的最大记录数。
- `offset` 偏移记录数，设置从第一条开始，要忽略的检索结果的条数，可以同`limit`一起使用，一次性查询多个记录块。

有时我们只需要知道满足特定条件的记录的数量。为此，我们可以使用`search_count()`，它返回记录数量而不是记录集。它节省了检索记录列表的成本，因此，当我们没有记录集，而且只需要计算记录的数量时，它的效率会更高。

`browse()`方法需要传入一个`ID`或一个`ID`列表，并返回一个对应记录集。当我们已经知道我们想要的记录的`ID`时，这个方法更为方便。

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)
```

单例

只有一个记录的记录集很特殊，称为**singleton**记录集。单例仍然是一个记录集，并且可以在任何需要使用记录集的地方使用。

但是与多元素记录集不同的是，单例记录集可以使用点符号访问它们的字段，如下所示：

```
>>> print self.name
Administrator
```

在下一个示例中，我们可以看到同一个`self`单例记录集，也表现出了普通记录集可迭代的特性。它只有一个记录，所以只有一个名字被打印出来：

```
>>> for rec in self:
      print rec.name
Administrator
```

在包含多个记录的记录集上，尝试访问字段值会出错。因为我们不确定是否使用了单例记录集，这就可能会产生问题。在设计为单例模式的方法中，我们可以使用`self.ensure_one()`在开始时进行检查。如果`self`不是单元素，它会抛出错误。

注意，空记录也是单例。

写记录

记录集实现了活动记录模式。这意味着我们可以为它们赋值，并且这些更改将在数据库中持久化。这是一种直观且方便的操作数据的方法，如下所示：

```
>>> admin = self.env['res.users'].browse(1)
>>> print admin.name
Administrator
>>> admin.name = 'Superuser'
>>> print admin.name
Superuser
```

记录集也有三个方法来处理它们的数据：create()、write()和unlink()。

create()方法需要传入一个将字段映射到值的字典，并返回创建好记录。默认值按预期自动应用，如下所示：

```
>>> Partner = self.env['res.partner']
>>> new = Partner.create({'name': 'ACME', 'is_company': True})
>>> print new
res.partner(72,)
```

unlink()方法可删除记录集中的记录，如下所示：

```
>>> rec = Partner.search([('name', '=', 'ACME')])
>>> rec.unlink()
True
```

write()方法同样需要传入一个将字段映射到值的字典。这些更新在记录集的所有元素上生效，同时方法没有返回值，如下所示：

```
>>> Partner.write({'comment': 'Hello!'})
```

使用活动记录模式有一些不足；它一次只能更新一个字段。另一方面，write()方法可以使用单个数据库指令同时更新多个记录的多个字段。在对性能有要求的情况下，应该记住这些差异。

值得一提的是copy()可以复制一条现有的记录;使用时可以传入一个可选参数：包含值的字典，用以写入新记录。例如，通过复制演示用户，来创建一个新的用户：

```
>>> demo = self.env.ref('base.user_demo')
>>> new = demo.copy({'name': 'Daniel', 'login': 'dr', 'email':''})
```

请记住，带有copy = False属性的字段将不会被复制。

处理时间和日期

出于历史原因，ORM记录集处理的是date和datetime的字符串值，而不是实际的Python Date和Datetime对象。在数据库中，它们以日期字段形式存储，但datetimes以 UTC时间格式存储。

- odoo.tools.DEFAULT_SERVER_DATE_FORMAT
- odoo.tools.DEFAULT_SERVER_DATETIME_FORMAT

上面两个分别映射到%Y-%m-%d和%Y-%m-%d %H:%M:%S。

fields.Date和fields.Datetime提供了几个方法以协助日期处理。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)
```

日期和时间是由服务器以一种简单的UTC格式处理和存储的，不能自动调整时区，会导致可能与用户的时区不同。因此，我们可以利用一些其他的函数来帮助我们处理这个问题：

- `fields.Date.today()` 以服务器所需要的格式返回当前日期(`date`)的字符串，并使用UTC作为参考。这足以计算默认值
- `fields.Datetime.now()` 以服务器所需要的格式返回当前日期时间(`datetime`)的字符串，并使用UTC作为参考。这足以计算默认值
- `fields.Date.context_today(record, timestamp=None)` 将在会话上下文中返回当前日期的字符串。时区值取自记录的上下文，可选参数是一个新的`datetime`值。
- `fields.Datetime.context_timestamp(record, timestamp)`将一个简单的日期时间(没有时区)转换为一个时区感知的日期时间。时区是从记录的上下文中提取的，这也应对了此函数的名称。

为了格式之间的转换， `fields.Date`和`fields.Datetime`对象都提供了这些函数:

- `from_string(value)` 将字符串转换为日期或日期时间对象。
- `to_string(value)`将日期或日期时间对象转换为服务器格式的字符串

记录集操作

记录集也支持额外的操作。我们可以检查记录是否包含在记录集中。如果x是单例记录集，`my_recordset`是一个包含许多记录的记录集，我们可以使用：

- `x in my_recordset`
- `x not in my_recordset`

下面的操作也是可用的：

- `recordset.ids` 返回记录集元素的ID列表。
  `recordset.ensure_one()`检查是否为单个记录(singleton);如果不是，则会引发`ValueError`异常。
- `recordset.filtered(func)` 返回一个过滤后的记录集。
- `recordset.mapped(func)` 返回映射值列表。
- `recordset.sorted(func)`返回一个排序后的记录集。

下面是这些函数的一些用法示例：

```
>>> 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)
```

操控记录集

我们希望能够添加、删除或替换这些相关字段中的元素，这就引出了一个问题:如何更改记录集?

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，修改记录集就意味着要根据现有的记录集组成生成新的记录集。

一种方法是使用集合操作:

- `rs1 | rs2`是集合的**union**操作，结果记录集中包含来自两个记录集的所有元素。
- `rs1 + rs2`是集合的**addition**操作，将两个记录集连接到一个。它可能会返回一个有重复记录的集合。
- `rs1 & rs2`是集合的**intersection**操作，结果是一个由两个记录集中的都存在的元素所组成的集合。
- `rs1 - rs2` 是集合的**difference**操作，并导致了`rs1`的记录集在`rs2`中不存在的元素

截取符号在这里也可以使用，如这些示例所示:

- `rs[0]` 和`rs[-1]` 分别用来检索第一个元素和最后一个元素。
- `rs[1:]`返回本记录集的不包含第一个元素的副本。这会检索出与`rs – rs[0]`相同

的记录，但同时保存了它们的顺序。

在`Odoo 10`中，操控记录集不会改变其中元素顺序。这与之前的`Odoo`版本不同，在这些版本中，操控记录集并不一定能保证维持元素顺序。只有添加`(addition)`和截取会维护记录顺序。

我们可以在删除或添加元素的时候，使用这些操作更改记录集。下面是一些例子：

- `self.task_ids |= task1` 如果task1记录之前不在结果集中，增加这条记录。
- `self.task_ids -= task1` 如果task1存在于结果集中，则删除这条指定的记录。
- `self.task_ids = self.task_ids[:-1]`删除结果集中的最后一条记录。

关系字段会包含记录集的值。`Many-to-one`字段可以包含单例记录集，而`one-to-many`，`many-to-many`字段中包含的记录集可以存储任意数量的记录。我们使用常规赋值语句对它们设置值，或者给`create()`和`write()`方法传入值字典进行设置。在上一个例子中，我们使用了一个特殊的语法来修改`one-to-many`，`many-to-many`字段。在XML记录中使用相同的方法为关系型字段提供值，并且在第4章，*模块数据（Module Data）*中的*设置关系字段的值（Setting values for relation fields）*一节有说明。

下面给出了关于`write()`语句如何达到于前面的三个赋值结果的示例：

- `self.write([(4, task1.id, None)])` 增加task1记录。
- `self.write([(3, task1.id, None)])` 从结果集中删除task1。
- `self.write([(3, self.task_ids[-1].id, False)])` 删除结果集的最后一条记录。

使用关系型字段

正如我们前面所看到的，模型可以有关系字段：**many-to-one(多对一)**、**one-to-many(一对多)**、**many- to-many(多对多)**。这些字段类型使用记录集作为值。

在`many-to-one`的情况下，该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。例如，以下指令是正确且安全的：

```
>>> 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'
```

很方便的是,空记录集也会像单例记录集一样,访问它的字段不会返回错误,而是返回False。因此，我们可以使用点符号来遍历记录，而不用担心空值的错误，如下所示：

```
>>> self.company_id.country_id
res.country()
>>> self.company_id.country_id.name
False
```

关系型字段值的操作

在使用活动记录模式时，可以将关系字段指定为记录集。

对于many-to-one字段，分配的值必须是单个记录（单例记录集）。

对于to-many字段，它们的值也是一个记录集。可以用一个新的记录集来替换原有的记录集（即使不存在原有记录集也可以替换）。这里允许任何大小的记录集。

使用create()或write()方法，在使用字典分配值时，不能直接分配记录集给关系型字段。应该使用相应的ID或ID列表。

例如，我们应该使用self.write({'user_id': self.env.**user.id**})，而不是self.write({'user_id': self.env.**user**})。

小结

在前几章中，我们了解了如何构建模型和设计视图。在这里，我们进一步了解了如何实现业务逻辑并使用记录集来处理模型数据。

我们还看到了业务逻辑如何与用户界面交互，并学会创建与用户通信的向导——一个可以启动高级流程的平台。

在下一章中，我们将学习如何为`addon`模块添加自动化测试，以及一些调试技术。

8

编写测试与调试你的代码

开发人员工作的一个重要部分是测试和调试代码。自动化测试是构建和维护健壮软件的非常有效的工具。在本章中，我们将学习如何将自动测试添加到addon模块中，以使它们更加健壮。还提供了服务器端调试技术，允许开发人员检查和了解他的代码中正在处理的事情。

单元测试

自动化测试通常被认为是软件开发中的最好的习惯。它不仅帮助我们确保代码得到正确的实现。更重要的是，它为将来的代码增强或重写提供了一个安全网。

在使用动态编程语言（如Python）的情况下，由于没有编译步骤，语法错误就可能被忽略。使用单元测试尽可能多的代码，变得更加重要。

编写测试的指导方针强调两个目标。测试的第一个目标应该是提供一个良好的测试覆盖率，设计能调用所有代码行的测试用例。这通常会让第二个目标——也就是显示代码的功能正确性——进展的更顺利。因为在完成第一个目标之后，我们就会很容易来为非常见的使用场景构建额外的测试用例。

增加单元测试

Python测试通过使用一个tests/子目录添加到addon模块中。测试运行器将自动发现该名称的子目录中的测试。

我们的todo_wizard addon的测试将在一个tests/test_wizard.py文件中。我们需要添加tests/_init_.py文件：

```
from . import test_wizard
```

这将是tests/test_wizard.py的基本框架：

```
# -*- 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提供了一些用于测试的类。TransactionCase测试对每个测试使用不同的事务，结束时自动回滚。我们还可以使用SingleTransactionCase，它在一个事务中运行所有的测试，只在最后一次测试结束时才回滚。当您希望每个测试的最终状态是之后测试的初始状态时，这一点非常有用。

setUp()方法是我们准备将要使用的数据和变量的地方。我们通常将它们存储为类属性，以便可以在测试方法中使用它们。

测试应该被实现为类方法，如test_populate_tasks()。测试用例方法名称必须以test_前缀开头。它们会被自动被发现，而这个前缀标识了实现测试用例的方法。

方法将按测试函数名称的顺序运行。在使用TransactionCase类时，将在每个结束时执行回滚。方法的docstring在运行测试时显示，其应该提供一个方法作用的简短描述。

这些测试类是unittest库的测试用例的封装器。这是Python标准库的一部分，您可以参考它的文档了解更多https://docs.python.org/2/library/unittest.html。

更确切地说，Odoo使用的是unittest的扩展库，unittest2。

编写测试用例

现在，让我们扩展在最初的框架中看到的test_populate_tasks()方法。我们可以编写的最简单的测试，从测试对象运行一些代码，验证查询结果，然后使用assert同预期的结果进行比较。

test_populate_tasks()方法将测试待办任务的do_populate_tasks()方法。由于我们的设置，确保了我们有两个open Todo，在运行它之后，我们希望向导task_ids将引用这两个记录。

```
# 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')
```

方法的第一行的docstring对描述测试非常有用，并且会在运行方法时打印到控制台上。

使用self.assertEqual语句来进行测试成功或失败的检查。最后一个参数是可选的，但是推荐使用，因为当测试失败时，它可以提供更多的消息。

assertEquals是最常用的方法之一，但它只是可用的assert方法之一。我们应该对每个

案例使用assert函数，因为它们将有助于理解失败测试的原因。例如，与其比较task_ids的长度，还不如用两个预期的任务准备一个记录集，然后使用：

```
self.assertItemsEquals( self.wizard.task_i
    ds, expected_tasks, 'Incorrect set of
    populated tasks')
```

在测试失败时，assert方法的返回的信息是最全的，同时能将预期结果与实际结果进行比较。

unittest文档提供了一个很好的参考

https://docs.python.org/2/library/unittest.h tml#test-cases.

通过往类中添加另一个方法，来实现加入新的测试用例。接下来，我们将测试do_mass_update()向导方法。当我们点击向导的**OK**按钮时，这就是将要执行的工作：

```
def test_mass_change(self):
    "Mass change deadline date"
    self.wizard.do_populate_tasks()
    self.wizard.new_deadline = self.todol.date_deadline
    self.wizard.do_mass_update()
    self.assertEqual( self.todol.
        date_deadline,
        self.todo2.date_deadline)
```

我们从再次运行do_populate_tasks()开始。请记住，在TransactionCase测试中，每次测试结束时都会执行回滚。因此，在之前的测试中所做的操作将被还原，我们需要再次填充向导的Todo任务列表。接下来，我们模拟用户填写新的截止期字段并执行批量更新。最后，我们检查是否两个任务都以相同的日期结束。

设置测试

我们应该从准备在测试中使用的数据开始。

使用特定用户执行测试操作是很方便的，这也是为了测试是否正确配置了访问控制。这是使用`sudo()`模型方法实现的。记录集携带着用户相关信息，所以在使用`sudo()`创建之后，在同一记录集中的后续操作将使用相同的上下文来执行。

这是`setUp`方法的代码，加上一些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
        tO = date.today()
        Todo = self.env['todo.task'].sudo(demo_user)
        self.todol = Todo.create((
            'name': 'Todol',
            'date_deadline': fields.Date.to_string(tO)})
        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({})
```

为了测试我们的向导，我们想要只存在两个`open Todo`。因此，我们从关闭任何现存的`Todo`开始，这样它们就不会妨碍我们的测试，并使用演示用户`(Demo user)`为测试创建两个新的`Todo`。最后，我们使用演示用户创建了一个新的向导实例，并将其分配给`self.wizard`，因此可以对实例使用测试方法。

测试异常

有时，我们需要测试来检查方法是否生成了异常。常见的情况是测试验证方法是否正确执行。

在我们的示例中，`test_count()`方法使用警告异常作为向用户提供信息的方式。为了检查是否有异常，我们将相应的代码放入`self.assertRaises()`块中。

We need to import the `Warning` exception at the top of the file:

我们需要在文件的顶部导入`Warning`异常：

```
from odoo.exceptions import Warning
```

并与另一个测试用例一起添加到测试类中：

```
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()方法没有引发异常，则检查失败。如果它确实引发了异常，则检查成功，而引发的异常则存储在e变量中。

我们用它来进一步检查。异常消息包含统计的任务数量，我们期望是两个。在最后的语句中，我们使用assertIn检查异常文本是否包含' 2 '字符串。

运行测试

测试写好了，是时候运行它们了。为此，我们只需要将--test-enable选项添加到Odoo服务器的启动命令，并安装或升级addon模块(-i 或 -u)。

启动命令是这样的：

```
$ ./odoo-bin -d todo --test-enable -i todo_wizard --stop-after-init
--addons-path="..."
```

只有启动时安装或升级的模块才会被测试。如果需要安装一些依赖模块，那么它们的测试也会运行。如果您想避免这种情况，您可以用通常的方式先安装需要被测试模块，然后在执行模块的升级(-u)时运行测试。

关于YAML测试

Odoo还支持第二种类型的测试，测试时使用YAML数据文件。最初所有的测试都使用YAML，直到最近才引入基于unittest的测试。虽然两者都被支持，但许多核心插件仍然包括YAML测试，当前的官方文档没有提到YAML测试。过去的文档可以在
https://doc.odoo.com/v6.0/contribute/15_guidelines/coding_guideline s_testing/.

具有Python背景的开发人员可能会更熟悉unittest，因为它是一个标准的Python特性，而YAML测试则是使用odoo特定的约定来设计的。目前明显的趋势是更倾向于unittest，

而YAML的支持在将来的版本中可能会被放弃。

由于这些原因，我们不会对YAML测试做深入的解释。但对它们的工作方式有一些基本的了解可能还是有用的。

YAML测试是类似于CSV和XML的数据文件。实际上，YAML格式是一种更紧凑的数据格式，可以用于替代XML。与Python测试必须在test/子目录中不同，而YAML测试文件可以在addon模块内的任何地方。但它们通常会在tests/或test/子目录中。Python测试可以被自动发现，而YAML测试必须在____manifest__.py文件中声明。这是通过test键完成的，类似于我们已经知道的data键。
虽然在Odoo 10不再使用YAML测试，但在point_of_sale addon模块中有一个例子02_order_to_invoice.yml:

```
-
  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_posl"), "active_ids":
[ref("pos_order_posl")]}' }:
    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_posl')} )
```

从a !开始的行是YAML标签，等价于我们在XML文件中找到的标签元素。在前面的代码中，我们可以看到一个a !record标签，相当于XML 的<record>，和a !python。这允许我们在模型上运行python代码，在这个例子中是pos.make.payment。

如您所见，YAML测试使用需要学习odoo的特殊语法。相比之下，Python测试使用现有的unittest框架，只添加odoo的特殊包装类，比如TransactionCase。

开发工具

开发人员应该学会一些能工作中帮助到他们的技巧。在第1章中，开始使用Odoo开发中（*Getting Started with Odoo Development*），我们已经介绍了**Developer Mode**的用户界面。我们还有能够提供部分开发人员友好特性的服务器选项。接下来我们将更详细地介绍它。之后，我们将讨论另一个有关开发人员的相关主题：如何调试服务器端代码。

服务器开发选项

Odoo服务器提供了 ——dev 选项，以启用一些开发人员的特性加速开发周期，例如：

- 当在addon模块中发现异常时，进入调试器
- 自动重新加载Python代码，一旦保存了Python文件，避免手动重启服务器
- 直接从XML文件读取视图定义，避免手动模块升级

尽管所有选项全部都启用在大多数情况下都是合适的，--dev选项接受一个逗号分隔的列表用于选择需要的选项。我们还可以指定我们喜欢使用的调试器。默认情况下，使用Python调试器pdb。有些人可能更喜欢安装和使用其他调试器。这里也支持ipdb和pudb。

在Odoo 10之前，我们有--debug选项，允许在addon模块异常上打开调试器。

在使用Python代码时，需要在每次修改代码时重新启动服务器，以便重新加载它。--dev 命令选项可直接启动重载：当服务器检测到Python文件被更改时，它会自动重复服务器装载序列，使代码更改立即生效。

要使用它，只需在服务器命令上添加选项`--dev=all`：

```
$ ./odoo-bin -d todo --dev=all
```

为此，需要使用`watchdog Python`包，并且应该使用如下安装方式：

```
$ pip install watchdog
```

注意，这只适用于`Python`代码更改和`XML`文件中的视图架构。对于其他的更改，例如模型数据结构，需要进行模块升级，重新加载是不够的。

调试

我们都知道开发人员工作的很大一部分，就是调试代码。为此，我们通常使用可以设置断点并逐步运行我们的程序的代码编辑器。

如果您使用`Microsoft Windows`作为开发工作站，那么创建一个能够从源代码运行`Odoo`代码的环境是一项非常重要的任务。还有一个事实是，`Odoo`是一个等待客户端调用的服务器，并且只对客户端的调用产生回应，这与客户端类型程序的调试差别很大。

Python调试器

虽然对新手来说可能有点吓人，但是最实用的调试`Odoo`的方法是使用`Python`集成调试器`pdb`。我们还将介绍拥有更丰富的用户界面的扩展，类似于复杂的IDE通常提供的功能。

要使用调试器，最好的方法是在我们要检查的代码上插入一个断点，代码通常位于一个模型方法中。这是通过在需要的地方插入以下行来完成的：

```
import pdb; pdb.set_trace()
```

现在重新启动服务器，以便加载修改过的代码。一旦程序执行到达该行，一个`(pdb) Python`

提示符将显示在服务器正在运行的终端窗口中，等待我们的输入。

`--dev`选项不需要使用手动设置`Python`调试器断点。

这个命令行以`Python shell`形式运行，您可以在当前的执行上下文中运行任何表达式或命令。这意味着当前变量可以被检查甚至修改。这些是最重要的快捷命令：

- `h` (help) 显示可用的`pdb`命令的摘要
- `p` (print) 计算并输出表达式
- `pp` (pretty print) 用于打印数据结构（如字典或列表）
- `l` (list) 列出要执行的指令周围的代码
- `n` (next) 转到下一条指令
- `s` (step) 进入当前指令的
- `c` (continue) 继续正常执行命令
- `u` (up) 在执行堆栈中向上移动
- `d` (down) 在执行堆栈中向下移动

Odoo服务器也支持dev=all选项。如果被激活，当一个异常被抛出时，服务器将在相应的行中进入一个事后剖析（*post mortem*）模式。这是一个pdb命令行，如前面所述的，允许我们在发现错误时检查程序状态。

一个示例调试会话

让我们看看一个简单的调试会话是怎样的。我们可以在do_populate_tasks向导方法的第一行中添加调试器断点：

```
def do_populate_tasks(self):
    import pdb; pdb.set_trace()
    self.ensure_one()
    # ...
```

现在重新启动服务器，打开一个**To-do Tasks Wizard**表单，并单击**Get All**按钮。这将触发do_populate_tasks向导在服务器上的方法，而web客户端将保持在**Loading...**状态，等待服务器响应。查看服务器正在运行的终端窗口，您将看到与此类似的内容：

```
> /home/daniel/odoo-dev/custom-
addons/todo_wizard/models/todo_wizard_model.py(54)do_populate_tasks()
-> self.ensure_one()
(Pdb)
```

这是pdb调试器命令提示行，这两个提示行的第一行提供了关于您在Python代码执行中的位置的信息。第一行通报文件、行号和函数名，第二行是要运行的下一行代码。

在调试会话期间，服务器日志消息也会显示。这些不会妨碍我们的调试，但它们会干扰我们。我们可以通过减少日志消息的冗余度来避免这种情况。大多数时候这些日志消息将来自于werkzeug模块。我们可以用这个选项来让他们沉默--log-handler=werkzeug:CRITICAL。如果这还不够，我们可以使用--log-level=warn来降低日志级别。

如果我们现在输入h，我们将看到可用的命令的快速引用。键入l显示当前代码行和周围代码行。

输入n将运行当前代码行，并移动到下一行。如果我们只按Enter，前面的命令将会被重复。那么做三次，我们就应该听在方法的返回语句。

我们可以检查任何变量的内容，比如这个方法中使用的open_tasks。输入p open_tasks或print open_tasks将显示该变量。任何Python表达式都是允许的，甚至是变量赋值。例如，为了显示更友好的任务名列表，我们可以使用：

```
(pdb) p open_tasks.mapped('name')
```

运行返回行，再次使用n，我们将显示函数的返回值。是这样的：

```
--Return--
> /home/daniel/odoo-dev/custom-
addons/todo_wizard/models/todo_wizard_model.py(59)do_populate_tasks()->('re
s_id': 14, 'res_model': 'todo.wizard', 'target': 'new', 'type':
'ir.actions.act_window', ...}
-> return self._reopen_form()
```

调试会话将继续在被调用的代码行上，但是我们可以输入c完成它并继续正常的执行。

选择Python调试器

虽然pdb有许多的优点，但它太简单了，目前存在一些更好用的选择。

The Iron Python debugger--ipdb是一个流行的选择，它使用与pdb相同的命令，但

是添加了一些改进，如tab补全和语法高亮显示，以达到更舒适的使用。可通过下面的方法安装：

```
$ sudo pip install ipdb
```

并在一行中添加断点：

```
import ipdb; ipdb.set_trace()
```

另一个替代调试器是pudb。它同样支持pdb中的命令，而且在纯文本终端上工作，但使用的图形显示类似于一个IDE调试器。比较有用的是，例如当前上下文中的变量和它们的值，在它们自己的窗口中是很容易获得的：

```
PuDB 2016.2 - ?:help  n:next  s:step into  b:breakpoint  !:python command line
 37                                                     Variables:
 38     @api.multi                                      pudb: <module 'pudb' from '
 39     def _reopen_form(self):                           /home/daniel/.local/lib/p
 40         self.ensure_one()                             ython2.7/site-packages/pu
 41         action = {                                    db/__init__.pyc'>
 42             'type': 'ir.actions.act_window',        self: todo.wizard(14,)
 43             'res_model': self._name,
 44             'res_id': self.id,
 45             'view_type': 'form',
 46             'view_mode': 'form',                    Stack:
 47             'target': 'new',                        >> do_populate_tasks [todo.
 48         }                                              call_kw_multi api.py:672
 49         return action                                  call_kw api.py:681
 50                                                        _call_kw [DataSet] main.
 51     @api.multi                                         call_button [DataSet] ma
 52     def do_populate_tasks(self):                       response_wrap http.py:50
*53         import pudb; pudb.set_trace()                  __call__ [EndPoint] http
> 54         self.ensure_one()                             checked_call http.py:324
 55         Task = self.env['todo.task']                   wrapper model.py:119
 56         open_tasks = Task.search([('is_done', '='   Breakpoints:
 57         self.task_ids = open_tasks
 58         # reopen wizard form on same wizard recor
 59         return self._reopen_form()
Command line: [Ctrl-X]

>>>                                    < Clear  >
```

可以通过系统包管理器或通过pip安装它，如下所示：

```
$ sudo apt-get install python-pudb  # using OS packages
$ sudo pip install pudb  # using pip, possibly in a virtualenv
```

添加一个pudb断点的方式，就和您预计的一样：

```
import pudb; pudb.set_trace()
```

印刷信息和日志记录

有时我们只需要检查一些变量的值，或者检查是否执行了一些代码块。Python的print语句可以完美地完成工作，而无需停止执行流。当我们在终端窗口中运行服务器时，输出的文本将显示在标准输出中。如果将print语句写入文件，是不会将输出存储到服务器日志中。

另一个要记住的选择是在我们的代码的敏感点设置调试级日志消息，我们可能需要这些消息来调查部署实例中的问题。只需要将服务器日志级别提升到debug，然后检查日志文件。

检查运行的流程

还有一些技巧可以让我们检查运行中的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然后输入：

```
$ ps ax | grep odoo-bin
```

输出中的第一列是该进程的PID。请记下这个进程的PID，因为我们将需要它。

现在我们要给进程发送一个信号。用需要使用的命令就是终止进程(kill)。默认情况下，它发送一个信号来终止一个进程，但是它也可以发送其他更友好的信号。

在知道了Odoo服务器进程的PID后，我们可以打印当前正在使用的代码的记录：

```
$ kill -3 <PID>
```

如果我们查看正在写入服务器输出的终端窗口或日志文件，我们将看到在运行的代码行上运

行的几个线程的信息和正在执行的代码行的详细堆栈跟踪信息。

我们还可以删除缓存/内存统计数据，使用：

```
$ kill -USR1 <PID>
```

小结

自动化测试对于一般的业务应用程序来说都是一种有价值的实践，并且可以确保动态编程语言（如Python）中的代码健壮性。

我们了解了如何为addon模块添加和运行测试的基本原则。我们还讨论了一些帮助我们调试代码的技术。

在下一章中，我们将深入到视图层，并将讨论看板视图。

9
QWeb和看板视图

QWeb是Odoo使用的模板引擎。它是基于xml的，用于生成HTML片段和页面。QWeb最初是在7.0版本中引入的，以启用更丰富的看板视图，并且自8.0版本以来，也用于报告设计和CMS网站页面。

在这里，您将了解QWeb语法以及如何使用它创建自己的看板视图和自定义报告。让我们开始学习看板的更多知识。

关于看板

Kanban是一个日语单词，用来表示工作队列管理方法。它从丰田生产系统和精益制造中获得灵感。随着敏捷开发的广泛应用，它已经在软件业中流行起来。

kanban board是一个将工作队列可视化的工具。看板由许多列组成，一列则代表工作过程中的一个**stages**。工作项，则由放置在适当的列上的**cards**来表示。新的工作项目从最左边的列开始，并在板子上移动，直到他们到达最右边的列，成为已完成的工作。

看板的简单性和视觉效果使它们能够很好地支持简单的业务流程。看板的一个基本示例可以包含三列，如下图所示：**To Do**、**Doing**和**Done**。

当然，它可以扩展出任何流程，只要我们有需求：

Photo credits„ “A Scrum board suggesting to use kanban” by JeR.lasovski. Courtesy of Wikipedia.

看板视图

对于许多业务场景来说，相比于典型的，笨重的工作流引擎，看板更有效地管理相应的流程。除以及经典列表和表单视图外，Odoo也支持看板视图。这使得实现这种类型的视图变得很容易。现在让我们学习如何使用它们。

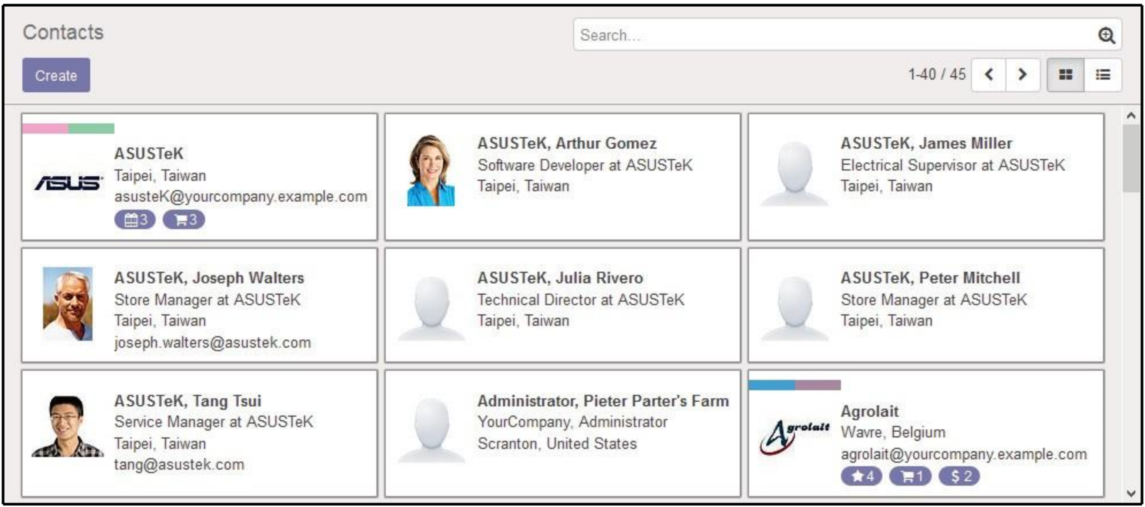
在表单视图中，我们主要使用特定的XML元素，如<field>和<group>，以及很少的HTML元素，如<h1>或<div>。对于看板视图，情况正好相反；它是基于html的模板，只支持两个特定的odoo的元素，<field>和<button>。

HTML是使用QWeb模板动态生成的。QWeb引擎可以处理特殊的XML标记和属性，并生成在web客户机中呈现的最终HTML。虽然这让我们可以很大程度上控制渲染内容，但也使得视图设计更加复杂。

看板视图设计非常灵活，因此我们将尽力为您提供能够快速构建看板视图的简单方法。一个好的方法是，通过找到一个类似于您需要的现有看板视图并查看了解它，来学习如何构建一个您自己的看板。

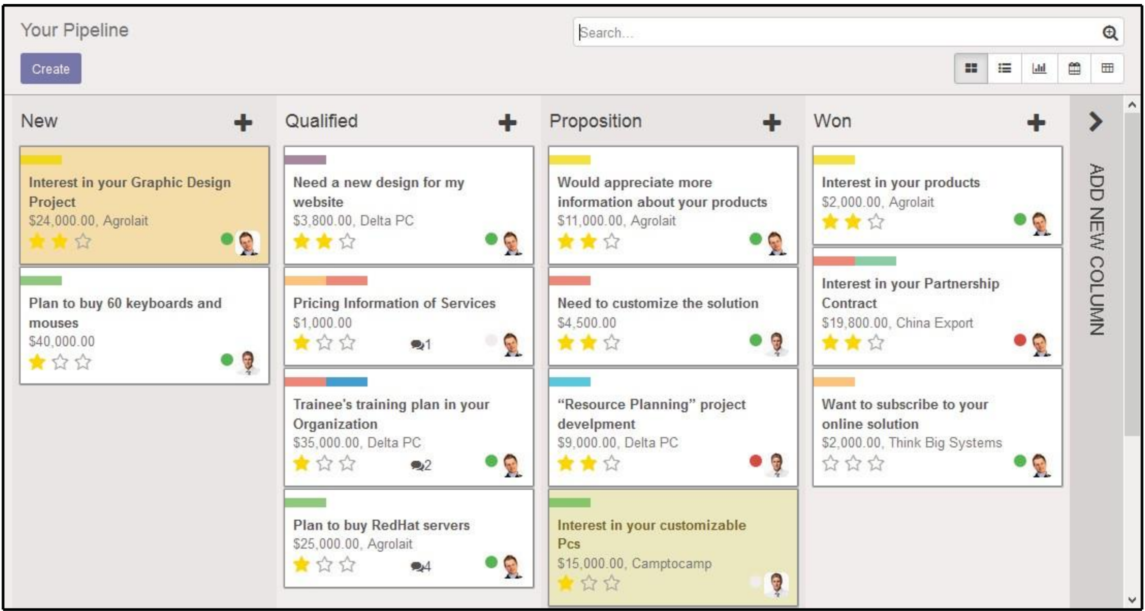
我们可以看到使用看板视图的两种不同的方法。一个是卡片列表。它用于联系人、产品、员工目录或应用程序。

下面是**Contacts**看板视图的样子：



但这并不是一个真正的看板。看板应该由列组成，当然，看板视图也支持这种布局。我们可以在**Sales | My Pipeline**或**Project Tasks**找到例子。

Sales | My Pipeline界面是这样的：



在上面的视图中卡片被组织在看板列中，这是和联系人看板之间最显著的区别。这是由**Group By**特性实现的，类似于列表视图。通常，分组效果使用**stage**字段完成。看板视图一个非常有用的特性是，它支持卡片在列之间的拖拽，并自动为分组字段赋相应的值。

在这两个例子的卡片中，我们可以发现一些差异。事实上，卡片的设计是相当灵活的，而且不止存在一种方法来设计看板卡片。但这两个例子可以为您的设计提供一个起点。

Contact卡片在左边基本上有一个图像，在主区域有一个加粗醒目的标题，后面是一个值列表。**My Pipeline**卡结构略微复杂一些。主卡区域也有一个标题，后面是相关信息列表和页脚区域，在这个例子中，左边是一个优先级（priority）小部件，右边是责任用户。虽然这种卡没有显示图片，但是在右上角也有一个选项菜单，会在鼠标悬停的时候显示。例如，这个菜单允许改变卡片的背景颜色。

我们将使用这个更精细的结构，作为我们的`Todo`看板上的卡片的模型。

设计看板视图

我们将使用一个新的addon模块，将看板视图添加到Todo任务中。将它直接添加到todo_ui模块会更简单。但是，为了更清楚地解释，我们将使用一个新模块，避免在已经创建的文件中出现太多、可能令人困惑的更改。

我们将把这个新的addon模块命名为todo_kanban，并创建通常的初始文件。编辑描述文件todo_kanban/_manifest_.py如下：

```
{'name': 'To-Do Kanban',
 'description': 'Kanban board for to-do tasks.',
 'author': 'Daniel Reis',
 'depends': ['todo_ui'],
 'data': ['views/todo_view.xml'] }
```

还要添加一个空的todo_kanban/_init_.py文件，根据Odoo addon模块的要求，使目录Python可导入。

接下来，创建一个XML文件，在这个文件中，我们闪亮的新看板视图将会执行，并将看板设置为“待办任务”窗口动作的默认视图。编辑todo_kanban/views/todo_view.xml，包含以下代码：

```
<?xml version="1.0"?>
<odoo>
  <!-- Add Kanban view mode to the menu Action: -->
  <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>
```

现在我们有了模块的基本框架。

在开始使用看板视图之前，我们需要在待办任务模型中添加几个字段。

优先级，看板状态和颜色

除了阶段(stage)之外，在看板上还有一些有用的和经常使用的字段。

- priority允许用户组织他们的工作项，标识哪些项目应该优先处理。
- kanban_state 标识一个任务已准备好移动到下一个阶段，或者由于某种原因处于阻塞状态。在模型定义层，两者都是选择型字段。在视图层，它们有特定的小部件，可以用于表单和看板视图。
- color 用于存储看板卡应该显示的颜色，并且可以通过看板视图的颜色选择器菜单进行设置。

要将这些字段添加到模型中，我们要添加一个models/todo_task_model.py文件。

但是首先，我们需要让它可被导入。编辑todo_kanban/_init__.py文件导入models子目录：

```
from . import models
```

然后创建models/__init__.py文件：

```
from . import todo_task
```

现在，让我们编辑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', default='normal')
```

现在我们可以开始编写看板视图了。

看板卡片元素

看板视图的架构，包含<kanban>顶级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>
```

请留心在<kanban>元素中使用的`default_group_by="stage_id"`属性。我们使用它，可以让看板卡片，默认按照卡片的阶段分组在列中。对于简单的卡片看板，例如在**Contacts**，我们不需要这个操作，只使用一个简单的<kanban>开头的标签即可。

顶级元素<kanban>支持一些有趣的属性：

- `default_group_by` 设置默认分组时的使用字段。
- `default_order` 设置看板项目的默认排列顺序。
- `quick_create="false"` 禁用在每个列的顶部仅填写一个标题来创建新

项目的快速创建选项（视图中的大“+”号）。false值供JavaScript语法使用，必须是小写的。

- class将CSS类添加到（渲染出的）看板视图的根元素。相关类o_kanban_small_column，使列间距比默认设置更小。可以通过模块定制的CSS文件，提供额外的类。

在<kanban>元素之后，我们将添加模板中使用的字段列表。确切地说，仅在QWeb表达式中使用的字段需要在这里声明，以确保从服务器获取其数据。

接下来，我们使用了<templates>元素，其中包含一个或多个QWeb模板来生成使用的HTML片段。我们必须有一个名为kanban-box的模板用来渲染看板卡片。同时还可以添加其他模板，通常被用来定义要在主模板中重用的HTML片段。

这些模板使用标准HTML和QWeb模板语言。QWeb提供了特殊的指令，通过处理这些指令来动态生成最终要呈现的HTML。

Odoo使用Twitter Bootstrap 3 web样式库，因此，这些样式类通常可以在渲染HTML的地方使用。您可以在https://getbootstrap.com了解更多关于Bootstrap的知识。

现在，我们将更深入地了解在看板视图中使用的QWeb模板。

看板卡片布局

看板卡的主要内容区域是在kanban-box模板内定义的。这个内容区域也可以有一个页脚子容器。

对于单个页脚，我们将在kanban-box底部使用一个oe_kanban_footer CSS类的<div>元

素。这个类会自动对<div>的内部元素灵活地分配空间，使其内部的左对齐和右对齐变得多余。

在卡片的右上角，可能出现能够打开一个动作菜单的按钮。作为另一种选择，Bootstrap提供了可在card中的任何位置（包括oe_kanban_footer 页脚等处）添加的pull-left和pull-right类来实现左对齐和右对齐效果。

这是我们的看板卡上QWeb模板的第一次迭代：

```
<!-- 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>
```

以上展示了看板卡的总体结构。您可能注意到，在顶部的<div>元素中使用color字段来动态设置卡片的颜色。我们将在下一节中详细解释t-attf QWeb指令。

现在我们来编写主要内容区域，并选择在那里放置什么：

```
<!-- 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>
```

这个模板大部分是常规的HTML，但是我们也看到了用于渲染字段值的<field>元素；以及在常规表单视图按钮中的type属性， 在这里供<a>锚标签使用。

在左边的页脚，我们将插入优先级(priority)小部件：

```
<div>
  <!-- Left hand footer... -->
  <field name="priority" widget="priority"/>
</div>
```

在这里，我们可以看到priority字段被添加进来，就像我们在表单视图中所做的那样。

在右边的页脚，我们将放置看板状态小部件和任务所属者的头像：

```
<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>
```

可以使用<field>元素添加看板状态，这与常规的表单视图一样。用户头像则使用HTML 的<img>标签插入。图像内容是使用QWeb的t-att- 指令动态生成的，我们稍后会解释。

有时我们想要在卡片上显示一个小头像，比如在Contacts示例中。作为参考，可以通过添加以下内容作为第一个内容元素来完成：

```
<img t-att-src="kanban_image( 'res.partner', 'image_medium',
  record.id.value)" class="o_kanban_image"/>
```

添加看板卡选项菜单

看板卡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（在菜单上）添加自定义按钮以调用任何动作也是可行的。我们也有一个小部件来设置卡片的颜色。
下面是在oe_kanban_content元素顶部添加选项菜单的基线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" aria-labelledby="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>
```

请注意，因为其中的一个表达式中使用了`t-if="!record.is_done.value"`，除非我们在视图中有字段`<field name="is_done" />`，否则表达式将不起作用。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，就像我们在定义`<kanban>`视图时所做的那样。

下拉菜单，基本上是HTML中`<a>`元素的列表。一些选项，例如**Edit**和**Delete**，只有在满足某些条件时才可用。这是用`t-if` QWeb指令完成的。在本章后面，我们将更详细地解释这个和其他QWeb指令。

`widget`全局变量表示负责渲染当前看板卡的`KanbanRecord()` JavaScript对象。两个特别有用的属性是`widget.editable`和`widget.deletable`，可以用来检查动作是否可用。我们还可以看到如何根据记录字段值显示或隐藏选项。如果`is_done`字段未设置，则**Set as Done**选项才会显示。

最后一个选项添加特殊小部件——颜色选择器，使用`color`数据字段来选择和更改卡片的背景颜色。

看板视图中的动作

在QWeb模板中，`<a>`标签可以有一个`type`属性。它设置了链接将执行的操作类型，以便链接可以像普通表单中的按钮一样操作。因此，除了`<button>`元素之外，`<a>`标签也可以用来执

行Odoo操作。

在表单视图中，动作类型可以是action或object，并且它应该伴随一个name属性，确定要执行的具体操作。此外，以下操作类型也可用：

- open打开关联的表单视图
- edit打开关联的表单视图，并直接进入编辑模式
- delete删除这条记录并在看板视图上移除这个项目

Qweb模板语言

QWeb解析器在模板中查找特殊的指令，并用动态生成的HTML替换它们。这些指令是XML元素属性，可以在任何有效的标签或元素中使用，比如<div>，<span> 或<field>。

有时我们想使用QWeb指令，但不想把它放在模板中的任何XML元素中。对于这些情况，我们有一个特殊元素<t>，它可以有QWeb指令，例如t-if或t-foreach，但它是静默的，并且不会在生成的最终XML/HTML上有任何输出。

QWeb指令将经常通过计算表达式，根据当前的记录值生成不同的结果。有两种不同的QWeb实现:客户端JavaScript和服务器端Python。

报告和网站页面使用服务器端Python实现。另一方面，看板视图使用客户端JavaScript实现。这意味着在看板视图中使用的QWeb表达式应该使用JavaScript语法编写，而不是Python。

在显示看板视图时，内部步骤大致如下：

1. 获取模板的XML用于渲染。
2. 调用服务器 read()方法获取模板中字段的数据。
3. 找到kanban-box模板并使用QWeb解析它来输出最终的HTML片段。

4. 在浏览器的显示中注入HTML (DOM)。

这并不是严格意义上的说明。它只是一种思维导图，可以帮助理解看板视图是如何工作的。

接下来，我们将学习QWeb的计算表达式，并探索可用的QWeb指令，来增强我们的任务任务看板卡片。

QWeb JavaScript求值上下文

许多QWeb指令使用可以用表达式的计算值来产生一些结果。当从客户端使用时，例如看板视图，这些表达式都是用JavaScript编写的。它们在包含可用变量的上下文中进行求值操作。

record对象，可以表示被渲染的记录，其中从服务器返回的字段值。可以使用raw_value或value属性访问字段值：

- raw_value是服务器方法read()返回的值，因此它更适合在条件表达式中使用。
- value是根据用户设置格式化的，并用于在用户界面中显示。通常与date/datetime和float/monetary字段相关。

QWeb求值上下文中也有JavaScript web客户端实例的引用。为了使用它们，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讨论这个问题。为了便于参考，在QWeb表达式求值时可以使用以下标识符：

- widget是对当前KanbanRecord()小部件对象的引用，它负责将当前记录渲染为看板卡片。它公开了一些有用的辅助函数以供开发者调用。

- `record`是`widget.records`的快捷方式。使用点符号，并提供对可用字段的访问。
- `read_only_mode`指示当前视图是否处于读取模式(而不是编辑模式)。 它是`widget.view.options.read_only_mode`的快捷方式。
- `instance`是对完整web客户端实例的引用。

值得注意的是，某些字符不能在表达式中使用。小于号(<)就是这样一个例子。这是因为XML标准，这些字符具有特殊的含义，不应该在XML内容上使用。一个否定的>=是一个有效的替代方法，但是通常的做法是使用下列可用于不等式的替代符号：

- `lt` 表示“小于”
- `lte` 表示“小于等于”
- `gt` 表示“大于”
- `gte` 表示“大于等于”

使用t-attf进行属性字符串替换

我们的看板卡使用`t-attf QWeb`指令为顶部的`<div>`元素动态设置一个类，使卡片根据`color`字段值着色。为此，使用了`t-attf- QWeb`指令。

`t-attf-` 指令通过字符串替换，动态地生成标签属性。这允许动态生成较长字符串中的部分内容，比如URL地址或CSS类名。

该指令寻找将被求值的表达式块，并替换结果。这些表达式块都是通过{{and }}或#{and }限定的。块的内容可以是任何有效的JavaScript表达式，并且可以使用任何可用于QWeb表达式的变量，比如`record`和`widget`。

在我们的例子中，我们还使用了`kanban_color() JavaScript`函数，特别提供了将颜色索引号映射到CSS类颜色名称的逻辑。

作为一个更详细的例子，我们可以使用这个指令动态地改变**Deadline Date**的颜色，这样过期的日期就会显示为红色。

为此，用如下代码替换我们看板卡片中的`<field name="date_deadline"/>`：

```
<li t-attf-
  class="oe_kanban_text_(( record.da
  te_deadline.raw_value and
  !(record.date_deadline.raw_value > (new Date()))
  ? 'red' : 'black' }}">
  <field name="date_deadline"/>
</li>
```

结果将会是`class="oe_kanban_text_red"`或`class="oe_kanban_text_black"`，这取决于截止日期。请注意，虽然在看板视图中有`oe_kanban_text_red CSS`类，但是`oe_kanban_text_black CSS`类并不存在，例子中使用它只是为了更好的说明`t-attf`的作用。

小于号，`<`，在表达式中是不允许的，我们选择通过使用否定大于来解决这个问题。另一种可能是使用`<`(小于)转义符号。

使用t-att配置动态属性

`t-att- QWeb`指令通过计算表达式的结果，动态生成属性值。我们的看板卡片使用它来动态地在`<img>`标签上设置一些属性。

`title`元素的动态渲染是通过：

```
t-att-
```

这个字段.value返回字段将会显示在屏幕上的值，对于many-to-one字段，这通常是相关记录的name值。对于用户，这是用户名。因此，当鼠标指针停留在图像上时，您将看到相应的用户名。

src标签也会动态生成，以提供对应用户的图像。图像数据由JavaScript帮助函数提供，kanban_image()：

```
t-att-src="kanban_image('res.users', 'image_small',
  record.user_id.raw_value)"
```

函数参数是：读取图像的模型、读取的字段名和记录的ID。在这里，我们使用.raw_value，获取用户的数据库ID而不是它的表示文本。

除此之外，可以使用t-att-NAME和t-attf-NAME来呈现任何属性，因为所生成的属性的名称取自使用的NAME后缀。

使用t-foreach循环

通过循环遍历，可以重复一个HTML块。我们可以使用它，将任务的全部关注者的头像添加到任务的看板卡片上。

让我们从渲染任务中的合作伙伴ID开始，如下：

```
<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表达式来计算一个集合的迭代。在大多数情况下，这将只是一个*to-many*关系字段的名称。它与t-as指令一起使用，以设置用于在迭代中引用每个项目时使用的名称。

接下来使用的t-esc指令对所提供的表达式进行计算，在本例中只使用rec变量名，并将其视作安全转义后HTML进行渲染。

在前面的示例中，我们循环了存储在message_parter_ids字段中的任务关注者。由于看板卡上的空间有限，我们可以使用slice() JavaScript函数来限制显示的数量，如下所示：

```
t-foreach="record.message_partner_ids.raw_value.slice(0, 3)"
```

rec变量保存了每个迭代出的值，在本例中为客户ID。了解了这一点，我们可以重写关注者的循环如下：

```
<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，这可以添加到在右页脚处，负责用户头像的旁边。

还有一些辅助变量。它们的名称，以t-as中定义的变量名作为前缀。在我们的示例中，我们使用了rec，因此可用的辅助变量如下:

- rec_index是元素的索引，从0开始
- rec_size是集合中元素的数量
- rec_first在迭代的第一个元素上,结果为true
- rec_last在迭代的最后一个元素上,结果为true
- rec_even 元素的索引为偶数时，结果为true
- rec_odd 元素的索引为奇数时，结果为true
- rec_parity当前元素索引的奇偶性，结果不是odd就是even
- rec_all 表示正在遍历的对象
- rec_value当遍历字典(key: value}时保存键值 (rec 保存键名)

例如，我们可以使用下面的方法来消除ID列表上的末尾逗号：

```
<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>
```

使用t-if实现条件渲染

我们的看板视图在卡片选项菜单中使用了t-if指令，并根据某些条件来提供选项。当在客户端渲染看板视图时，t-if指令需要一个JavaScript计算表达式。只有当条件计算为true时，标签及其内容才会呈现。

另一个例子则是，如果任务效果预计有值，则在卡片中显示它。这需要在date_deadline字段后面添加以下内容:

```
<t t-if="record.effort_estimate.raw_value gt 0">
  <li>Estimate <field name="effort_estimate"/></li>
</t>
```

我们使用了<t t-if="...">元素，如果条件为false，元素就不会产生输出。只有是true时，其中的<li>元素才会被渲染输出。注意，条件表达式使用的是gt符号而不是>，来表示大于运算符。

使用t-esc和t-raw渲染值

我们使用<field>元素来渲染字段内容。但是，如果没有<field>标签，字段值也可以直接显示。

t-esc指令计算一个表达式，并将其呈现为html转义值，如下所示：

```
<t t-esc="record.message_parter_ids.raw_value" />
```

在某些情况下，如果能确保源数据是安全的，就可以使用t-raw来呈现字段的原始值，而不需要进行任何转义，如下面的示例所示：

```
<t t-raw="record.message_parter_ids.raw_value" />
```

出于安全考虑，避免使用t-raw。它的使用，应该严格地限制在不使用任何用户数据的情况下，输出特定的HTML数据，或者所有用户数据，都是显式转义HTML特殊字符后的值。

使用t-set设定变量值

对于更复杂的逻辑，我们可以将表达式的结果存储到一个变量中，之后在模板中使用它。这将使用t-set指令来完成变量的命名，然后t-value指令，计算表达式的值并赋值给变量。

例如，下面的代码以红色表示错过了最后期限，就像前面的部分一样，但是使用了一个red_or_black变量，来确定要使用的CSS类，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>
```

变量也可以被赋值为HTML内容，例如下面的例子：

```
<t t-set="calendar_sign">




  <span class="oe_e">&#128197; </span>
</t>
<t t-raw="calendar_sign" />
```

oe_e CSS类使用Entypo图标。日历符号的HTML表示，则存储在一个变量中，该变量可以在模板需要时使用。**Font Awesome**的图标集也是可以使用的。

使用t-call插入其他模板

QWeb模板可以是可重用的HTML片段，所以可以被插入到其他模板中。这样，我们就可以通过设计基础代码块，来构建更复杂的用户界面视图，而不是一次又一次地重复相同的HTML块。

可重用模板，需要在<templates>标签中定义。用一个带有t-name的顶部元素标识，而不是kanban-box。然后t-call指令就可以将其他模板插入进来。对于在同一个的看板视图中声明的模板，或者在相同的addon模块中的其他地方的模板，甚至不同的addon中的模板，这么做都是可行的。

关注者头像列表，可以隔离为一段可重用代码片段。让我们使用子模板重做它。我们应该首先在XML文件中添加另一个模板，在<templates>元素中，<t t-name="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主模板调用它非常简单。我们应该使用以下内容，而不是<div>元素包含每个for each指令：

```
<t t-call="follower_avatars" />
```

要调用在其他addon模块中定义的模板，我们需要使用module.name完整标识符，就像我们处理其他视图一样。例如，可以使用完整标识符todo_kanban.follower_avatars来引用上面的代码片段。

被调用的模板与调用方在相同的上下文中运行，因此在调用模板时，调用方中可用的任何变

量名也可用。

一个更优雅的替代方法是将参数传递给调用的模板。这是通过在t-call标记内设置变量来完成的。这些仅在子模板上下文中可用，并且不会存在于调用者的上下文中。

我们可以使用它来获得关注者头像，最大数量由调用者设置，而不是在子模板中硬编码。首先，我们需要用变量arg_max替换固定值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>
```

之后，当调用子模版时来设定变量值：

```
<t t-call="follower_avatars">
  <t t-set="arg_max" t-value="3" />
</t>
```

t-call元素内的所有内容，子模版可以通过一个魔法变量0来访问。我们通过<t t-raw="0"/>定义一个HTML代码片段供子模板使用，而不是使用参数变量。

更多使用t-attf的方法

我们已经学习了最重要的QWeb指令，但是还有一些指令也需要了解。我们会做一个简短的解释。

我们已经见过t-att-NAME和t-attf-NAME风格的动态标签属性。另外，可以使用固定的t-att指令。它接受一个键值字典映射或pair（两个元素列表）。

使用如下的映射：

```
<p t-att="('class': 'oe_bold', 'name': 'testl'}" />
```

结果如下：

```
<p class="oe_bold" name="testl" />
```

使用如下的pair：

```
<p t-att="['class', 'oe_bold']" />
```

结果如下：

```
<p class="oe_bold" />
```

看板视图间的继承

在看板视图和报告中使用的模板，可以使用应用于其他视图的常规扩展技术，例如XPath表达式。参见第3章，继承–扩展现有应用程序（*Inheritance – Extending Existing Applications*），了解更多细节。

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，可以多次声明同一个字段，例如，在模板之前声明一次，之后在模板内部再声明一次。在这种情况下，选择器将匹配第一个字段元素，而不会像预期的那样在模板中添加我们的修改。

为了解决这个问题，我们需要使用XPath表达式来确保匹配的是模板内的字段。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>
```

在上面的示例中，XPath在<t tname="kanban-box">元素中查找<field name="display_name">元素。这排除了在<templates>部分之外相同名字的字段元素。

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研究其正确的语法。您的Linux系统可能已经安装了xmllint命令行工具，这个工具有一个——xpath选项来执行对XML文件的查询。

另一个能够提供更好输出的选项，是Debian/Ubuntu包中libxml-xpath–perl工具的xpath命令：

```
$ sudo apt-get install libxml-xpath-perl
$ xpath -e "//record[@id='res_partner_kanban_view']" -e
```

```
"//field[@name='display_name']]" /path/to/*.xml
```

定制CSS和JavaScript素材

正如我们所见，看板视图大多是HTML，并大量使用CSS类。我们一直在介绍标准产品提供的一些常用的CSS类。但为了获得最佳效果，模块还可以添加自己的CSS。

这里我们不会详细讨论如何编写CSS代码，但它与解释模块如何添加自己的CSS（和JavaScript） web素材有关。Odoo用于后端的素材在assets_backend模板中声明。要添加我们的模块素材，我们应该扩展该模板。这个XML文件通常放在一个名为views/ 模块子目录中。
以下是向todo_kanban模块中添加CSS和JavaScript文件的一个示例XML文件，放置位置可以是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>
```

像往常一样，它应该在__manifest__.py描述文件中引用。注意，这些素材位于一个/static/src子目录中。虽然这个子目录不是必需的，但它是一个普遍的惯例。

小结

您了解了看板和如何构建看板视图来实现它们。我们还介绍了`QWeb`模板，以及它如何用于设计看板卡片。`QWeb`也是为网站`CMS`提供动力的渲染引擎，所以它在`Odoo`工具集的重要性越来越大。在下一章中，我们将继续使用`QWeb`，但是在服务器端，我们将创建自定义报表。

10

创建QWeb报表

报表对于商业应用来说是非常宝贵的功能。自从8.0版本开始内置的QWeb报表引擎，是默认的报表引擎。报表设计使用QWeb模板来生成HTML文档，然后转换成PDF格式。

Odoo的内置报表引擎曾经历了重大变化。在7.0版本之前，报表基于ReportLab库并使用了特定的RML标记语法。在7.0版本中，Webkit报表引擎被包含在核心模块中，允许使用常规的HTML来设计报表。最后，在8.0版本中，这个概念得到了进一步的发展，QWeb模板成为了内置报表引擎背后的主要概念。

这意味着我们可以方便地将我们所学到的QWeb，应用于创建业务报表。在本章中，我们将向我们的**To Do**应用程序添加一个报表，并将审查使用QWeb报表的最重要的技术，包括报表计算，如合计、翻译和打印格式。

但是在开始之前，我们必须确保已经安装了用于将HTML转换为PDF文档的工具的推荐版本。

安装wkhtmltopdf

要正确地生成报表，需要安装wkhtmltopdf库的推荐版本。它名字是下列英文的缩写，**Webkit HTML to PDF**。Odoo使用它，将渲染后HTML页面转换成PDF文档。

旧版本的wkhtmltopdf库有一些问题，比如不打印页眉和页脚，所以我们一定要注意安装正确的版本。对于odoo 的9.0版本，在本书写成之时，推荐版本是0.12.1。不幸的是，提供给您的主机系统(如Debian/Ubuntu或其他的Linux发行版)的打包版本是不够的。因此，我们应该下载并安装针对我们的操作系统和CPU架构所推荐的包。下载链接 http://wkhtmltopdf.org 或http://download.gna.org/wkhtmltopdf.。

我们应该首先确保我们的系统中没有安装错误的版本：

```
$ wkhtmltopdf --version
```

如果上面的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，我们需要下载合适的包并安装它。在 http://download.gna.org/wkhtmltopdf/0.12/0.12.1中检查文件名是否正确。对于Ubuntu14.04 LTS (Trusty) 64位，下载命令如下：

```
$ wget
http://download.gna.org/wkhtmltopdf/0.12/0.12.1/wkhtmltox-0.12.1_linux-trus
ty-amd64.deb -O /tmp/wkhtml.deb
```

下一步我们应该安装它。安装一个本地deb文件并不会自动安装依赖项，因此需要第二个步骤来完成这个任务并完成安装：

```
$ sudo dpkg -i wkhtml.deb
$ sudo apt-get -f install
```

现在我们可以检查wkhtmltopdf库是否正确安装，并确认它的版本号是我们想要的：

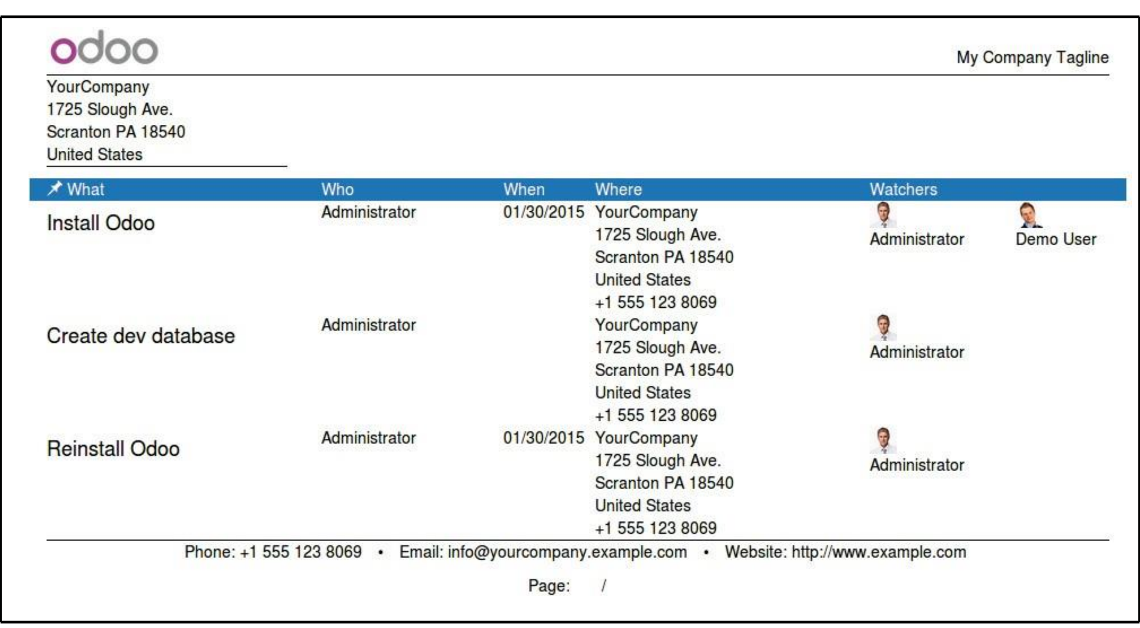
```
$ wkhtmltopdf --version
wkhtmltopdf 0.12.1 (with patched qt)
```

在此之后，Odoo服务器启动序列就不再会显示**You need Wkhtmltopdf to print a pdf version of the report's** 的信息。

创建商业报表

通常我们会在我们的To Do模块中实现报表。但是为了学习目的，我们将为我们的报表创建一个新的`addon`模块。

我们的报表效果如下：


My Company Tagline

YourCompany
1725 Slough Ave.
Scranton PA 18540
United States

What	Who	When	Where	Watchers	
Install Odoo	Administrator	01/30/2015	YourCompany 1725 Slough Ave. Scranton PA 18540 United States +1 555 123 8069	Administrator	Demo User
Create dev database	Administrator		YourCompany 1725 Slough Ave. Scranton PA 18540 United States +1 555 123 8069	Administrator	
Reinstall Odoo	Administrator	01/30/2015	YourCompany 1725 Slough Ave. Scranton PA 18540 United States +1 555 123 8069	Administrator	

Phone: +1 555 123 8069 • Email: info@yourcompany.example.com • Website: http://www.example.com

Page: /

我们将命名这个新的`addon`模块为`todo_report`。首先要做的是创建一个空的`__init__.py`文件和`__manifest__.py`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文件可以通过声明以下内容开始：

```
<?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>标签是将数据写入到ir.actions.report.xml模型的快捷方式。这个模型代表一种特定类型的客户端动作。可以在**Settings** | **Technical** | **Reports**菜单选项中浏览这个模型的数据。

在报告的设计过程中，最好先设置报表类型report_type="qweb-html"，并在完成报表设计后将其修改回qweb-pdf文件。这样可以更快地生成报表，也更容易检查来自QWeb模板的HTML结果。

在安装了这个之后，待办任务表单视图将在顶部在**More**按钮的左边，显示一个**Print**按钮，其中包含执行报表打印的选项。

因为我们还没有定义报表，所以现在无法使用。这将是一个QWeb报表，因此它需要一个QWeb模板。name属性可以标识将使用的模板。与其他标识符引用不同，name属性中的模块前缀是必需的。我们必须充分参考<module_name>.<identifier_name>。

QWeb报表模板

报表通常遵循基本框架，如下所示。这可以添加到reports/todo_report.xml文件，在<report>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>
```

这里最重要的元素是使用标准报表结构的`t-call`指令。`report.html_container`模板将运行支持`HTML`文档的基本设置。`report.external_layout`模板，将使用公司的相应设置，处理报表页眉和页脚。作为另一种选择，我们可以使用`report.internal_layout`模板，它只使用一个基本页眉。

现在，我们有了模块和报表视图的基本框架。请注意，由于报表是`QWeb`模板，所以就像其他视图一样可以使用继承。在报表中使用的`QWeb`模板，可以使用**XPATH**表达式进行常规的继承视图扩展。

报表中的数据

与看板视图不同，报表中的`QWeb`模板是在服务器端渲染的，并使用`Python QWeb`实现。我们可以将此视为同一规范的两种实现，故我们需要了解其中的差异。

首先，使用`Python`语法对`QWeb`表达式进行计算，而不是使用`JavaScript`。对于最简单的表达式，可能很少或没有区别；但是若是更复杂的操作，表达式可能会有所不同。

表达式的计算方法也不同。对于报表，我们有以下变量：

- `docs`是一个可重复的集合，其中包含要打印的记录。
- `doc_id`是要打印的记录的`ID`列表。
- `doc_model`标识记录的模型，例如`todo.task`。

- time是对Python时间库的引用。
- user是运行报告的用户的记录。
- res_company是当前用户公司的记录。

报表内容是用HTML编写的，字段值可以使用t-field属性引用。可以用t-field-options属性来进行补充，效果是以特定的小部件来渲染字段内容。

现在我们可以开始为我们的报表设计页面内容了：

```
<!-- 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 Bootstrap HTML网格系统。简单地说，Bootstrap有一个包含12个列的网格布局。可以使用<div class="row">添加新的行。在一行中，我们有单元格，每一个都跨越一定数量的列，总计应当占据12列。每个单元格都可以用<div class="col-xs-N">来定义，其中N是它跨越的列数。

可以在http://getbootstrap.com上找到对Bootstrap的完整引用，描述上面介绍过的以及更多的样式元素。

在这里，我们添加标题行和标题，然后我们有一个t-foreach循环，遍历每个记录，并为每个记录渲染一行。

由于渲染是在服务器端完成的，所以记录是对象，我们可以使用点表示法从相关数据记录访问字段。这使得通过关系字段来访问它们的数据变得很容易。请注意，这在客户端渲染的Qweb、视图（如web客户端看板视图）中是不可能的。

以下XML可以渲染记录行的内容：

```
<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>
```

如我们所见，字段使用时可附带额外选项。这些非常类似于表单视图中使用的options属性。在第6章，视图-设计用户界面（*Views – Designing the User Interface*），使用附加widget来设置渲染字段的小部件。

一个例子是上面使用的货币小部件，紧挨着截止日期。

一个更复杂的示例是用于格式化地址的contact小部件。我们使用了公司地址res_company.partner_id，并且因为它有一些默认数据，我们可以立即看到渲染出的地址。但是，使用当前用户的地址，o.user_id.partner_id更有意义一些。默认情况下，contact小部件显示带有一些图标的地址，例如一个电话图标。我们使用的no_marker="true"选项禁用了它们。

渲染图片

我们报表的最后一部分将以带头像的关注者列表结束。我们将使用Bootstrap组件media-list，并循环关注者列表来渲染其中的每一个人：

```
<!-- 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>
```

二进制字段的内容以base64形式提供。<img>元素的src属性可以直接使用这类数据。因此，我们可以使用QWeb指令 t-att-src来动态生成每个图像。

汇总总数和累计总数

报表中经常需要提供总数。可以使用Python表达式来计算这些总数。

在<t t-foreach>标签结束之后，我们来添加用于计算总数的最后一行：

```
<!-- 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元素的数量。总数可以用sum()在一个值列表上计算。在前面的示例中，我们使用一个Python列表推导式从docs记录集生成一个值列表。您可以将列表推导式看作一个嵌入式的for循环。

有时我们想在报告中执行一些计算。例如，一个累计总数（循环到当前记录时的总和）。这可以用t-set来定义一个累加变量，然后在每一行上更新它。

为了说明这一点，我们可以试着计算关注者的累计数量。我们应该在循环docs记录集的t-foreach之前，初始化这个变量：

```
<t t-set="follower_count" t-value="0" />
```

然后，在循环中，将变量每次都将加上当前记录的关注者数量。我们选择在渲染出关注者列表后执行这个指令，并在每一行打印出当前的总数：

```
<!-- Running total-->
  <t t-set="follower_count"
    t-value="follower_count + len(o.message_follower_ids)" />
  Accumulated # <t t-esc="follower_count" />
```

定义纸张格式

现在，我们的报表在HTML格式中看起来不错，但是它在PDF页面上并没有很好地打印出来。我们可以使用横向页面获得更好的结果。所以我们需要添加这种纸张格式。

在XML文件的顶部，添加以下记录：

```
<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>
```

它是一份欧洲A4格式的拷贝，源代码在addons/report/data, report_paperformat.xml文件中定义。但我们将纸张方向由纵向改到横向。从web客户端可以看到定义的文件格式，**Settings | Technical | Reports | Paper Format.**。

现在我们可以在报表中使用它了。默认的文件格式是在公司设置中定义的，但是我们也可以指

定特定报表使用的纸张格式。这是在报表操作中使用paperfomat属性完成的。

让我们编辑用于打开报告的动作，并添加此属性：

```
<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"
/>
```

在9.0版本中添加了 <report>标签的paperformat属性。对于8.0，我们需要使用<record>元素来添加具有paperformat值的报表动作。

在报表中使用翻译

要为报表启用翻译，需要使用一个模板从模板调用它。这里使用的是带有t-lang属性的<t t-call>元素。

t-lang属性应该赋值为一个语言代码，比如es或en_US。它需要一个含有语言字段的记录。我们经常使用的是接收这张报表的合作伙伴的语言字段。客户的语言设置存储在partner_id.lang字段。在我们的例子中，我们没有合作伙伴字段，但是我们可以使用负责用户的语言，而相应的语言首选项是user_id.lang。

该函数需要传入一个模板名称，并将渲染和翻译它。这意味着我们需要在一个单独的模板中定义报告的页面内容，如下所示：

```
<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>
```

基于自定义SQL的报表

我们所建的报表是建立在常规记录的基础上的。但是在某些情况下，我们需要转换或聚合数据。在渲染报表时，处理动态数据并不是一件容易的事情。

其中一种方法是编写一个SQL查询来构建我们需要的数据集，然后通过一个特殊的模型存储这些结果，并根据获得记录集进行报表工作。

为此，我们将创建一个`reports/todo_task_report.py`文件，使用代码如下：

```
# -*- 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')
```

要加载这个文件，我们需要添加一个`from . import reports`行到 `__init__.py`文件的顶部，同时添加`from . import todo_task_report`到`reports/_init_.py`文件。

`sql`属性用于覆盖数据库表的自动创建特性，同时提供一个SQL语句。我们希望它创建一个数据库视图来提供报告所需的数据。我们的SQL查询实际上非常简单，但关键是此时我们可以使用任何有效的SQL查询来生成我们的视图。

我们还映射了我们需要的ORM字段类型的字段，以便它们可以在这个模型上生成的记录集上使用。

接下来，我们可以根据这个模型添加一个新的报表，`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，我们可以使用一个不同的解决方案:向导。为此，我们应该创建一个具有相关字段的暂态模型，其中头部包含了由用户引入的报告参数，并且这些字段会将查询出的数据提供给报告使用。这些行是由一个模型方法生成的，它可以包含我们可能需要的任何逻辑。强烈建议从现有的类似报告中获取灵感。

小结

在前一章中，我们学习了`QWeb`，以及如何使用它来设计看板视图。在本章中，我们了解了`QWeb`报告引擎，以及使用`QWeb`模板语言构建报告时最重要的技术。

在下一章中，我们将继续使用`QWeb`，并构建网站页面。我们还将学习编写`web`控制器，为我们的`web`页面提供更丰富的特性。

11

创建网站前端特性

Odoo最初是一个后台系统，但很快就有了前端界面的需求。早期的门户功能，基于与后端相同的接口，并不是很灵活，也不是移动设备友好型。

为了解决这个问题，version 8中引入了新的网站特性，为产品添加了**Content Management System (CMS)**。这将使我们能够构建漂亮而有效的前端，而不需要集成第三方CMS。

在这里，我们将学习如何利用Odoo提供的网站功能，开发我们自己的前端addon模块。

路线图

我们将创建一个网站页面，列出我们的待办任务，并允许我们导航到每个现有任务的详细页面。我们还希望能够通过web表单添加新的待办任务。

由此，我们将能够涵盖网站开发的基本技术：创建动态页面，将参数传递到另一个页面，创建表单并处理它们的验证和计算逻辑。

但是首先，我们将通过一个非常简单的**Hello World** web页面来介绍基本的网站概念。

我们的第一个web页面

我们将为我们的网站功能创建一个addon模块。我们可以叫它todo_website。为了介绍Odoo web开发的基础知识，我们将实现一个简单的**Hello World** web页面。这个挺有想象力的，对吧？

像往常一样，我们将首先创建它的清单文件。创建todo_website/_manifest_.py文件：

```
(
  'name': 'To-Do Website', 'description':
  'To-Do Tasks Website', 'author':
  'Daniel Reis',
  'depends': ['todo_kanban']}
```

这个模块在todo_kanban addon模块的基础上构建，这样就可以拥有我们整本书中添加到待办任务模型的所有特性。

注意，现在我们并不依赖于website addon模块。虽然website提供了一个有用的框架来构建功能齐全的网站，但是基本的web功能已经是构建在核心框架中的。现在让我们来了解他们。

Hello World!

为了提供我们的第一个web页面，我们将添加一个控制器对象。我们可以从将其文件导入模块开始：

首先添加一个有下列语句的todo_website/ init .py文件：

```
from . import controllers
```

然后添加一个有下列语句的todo_website/controllers/_init_.py文件：

OSCG 開源智造

```
from . import main
```

现在添加控制器的实际文件，todo_website/controllers/main.py，有以下代码：

```
# -*- coding: utf-8 -*-
from odoo import http

class Todo(http.Controller):

    @http.route('/hello', auth='public') def
    hello_world(self):
        return('<h1>Hello World!</h1>')
```

odoo.http模块提供了Odoo网络相关的特性。我们负责页面渲染的控制器，应该是odoo.http.Controller的子类的对象。这个类的实际名称并不重要；这里我们选择使用Todo。

在控制器类中，我们有匹配路由，进行一些处理，然后返回结果的方法。结果则要显示给用户的页面。

odoo.http.route 修饰器用于将方法绑定到URL路由。我们的示例使用了/hello路由。访问http://localhost:8069/hello，你将会看到一个**Hello World**消息。在本例中，该方法执行的处理非常简单：它返回一个带有**Hello World**消息的HTML格式文本字符串。

您可能注意到我们在路由中添加了auth='public'参数。当页面需要对未经过身份验证的用户可用时，需要此参数。如果我们删除它，只有经过身份验证的用户才能看到页面。如果不存在活动的会话(session)，则将显示登录屏幕。

使用QWeb模板的Hello World

使用Python字符串来构建HTML会很快变得乏味。QWeb模板在这方面做得更好。因此，让我们使用模板改进**Hello World** web页面。

QWeb模板是通过XML数据文件添加的。从技术上讲，类似表单或树视图，QWeb模板也是一种

视图。实际上，它们都存储在同一个模型中，`ir.ui.view`。

像往常一样，要加载的数据文件必须在清单文件中声明，所以要编辑`todo_website/__manifest__.py`文件添加键：

```
'data': ['views/todo_templates.xml'],
```

然后添加实际的数据文件`views/todo_web.xml`，包含以下内容：

```
<odoo>
  <template id="hello" name="Hello Template">
    <h1>Hello World !</h1>
  </template>
</odoo>
```

`<template>`元素实际上是声明`ir.ui.view`模型的`<record>`的快捷方式，实现方法是用`type="qweb"`行，和在内部编写`<t>`模板。

现在我们需要我们的控制器方法使用这个模板：

```
from odoo.http import request
# ...
@http.route('/hello', auth='public')
def hello(self, **kwargs):
    return request.render('todo_website.hello')
```

`request`通过它的`render()`函数，提供了模板渲染的功能。

注意，我们在方法参数中添加了`**kwargs`。如果`HTTP`请求提供的其他参数（如查询字符串或`POST`参数）可以被`kwargs`字典捕获。这使我们的方法更加健壮，因为提供意外的参数不会导致错误。

扩展web特性

我们期望在Odoo的所有特性都具有可扩展能力，web特性也不例外。实际上，我们可以扩展现有的控制器和模板。作为一个例子，我们将扩展**Hello World** web页面，这样它就可以使用一个带有致以问候的名字的参数来生成问候语：使用URL / helloworld?name=John会返回**Hello John!**问候。

扩展通常是由一个不同的addon模块完成的，但是它在同一个addon中也同样有效。为了保持简洁和简单，我们就不创建一个新的addon模块了。

让我们添加一个新的todo_website/controllers/extend.py,代码如下：

```
# -*- 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什么来扩展控制器。

首先，我们使用Python的import来获引用要扩展的控制器类。与模型相比，模型们有一个中央注册表，由env对象提供，在那里可以获得对任何模型类的引用，而不需要知道模块和实现它们的文件。对于控制器，我们则没有这个特性，需要知道实现了我们想要扩展的控制器的模块和文件。

接下来，我们需要（重新）定义从被扩展的控制器的方法。它需要用至少一个简单的

@http.route()来装饰，以使其保持活动状态。可选地，我们可以为route()提供参数，并替换和重新定义它的路由。

被扩展的hello()方法现在有一个name参数。参数的值可以从路由URL的片段、查询字符串的参数或POST参数中获取。在这种情况下，路由没有可提取的变量（我们稍后会展示这一点），由于我们处理GET请求，而不是POST，所以name参数的值将从URL查询字符串中提取。测试URL可以是 http://localhost:8069/hello?name=John。

Inside the hello() method we run the inherited method to get its response, and then get to

在hello()方法中，我们运行先被继承的方法并得到它的响应对象，然后根据我们的需要修改响应对象。控制器方法的常见模式是让它们以一个渲染模板的语句结束。在我们的例子中：

```
return request.render('todo_website.hello')
```

这会生成一个http.Response对象，但是实际的渲染被推迟到调度结束时才进行。

这意味着，继承方法可以更改用于渲染QWeb模板和上下文。可以通过修改response.template来更改模板，但我们不需要这么做。我们更希望修改response.qcontext将name键添加到渲染上下文的。

不要忘记将新的Python文件添加到todo_website/controllers/_init__.py:

```
from . import main
from . import extend
```

现在我们需要修改QWeb模板，这样它就可以使用这些额外的信息。添加文件t 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>
```

Web页面模板是XML文档，就像其他的Odoo视图类型一样我们可以使用xpath来定位元素，然后更改它们，跟我们处理其他视图类型继承是一样。继承的模板在<template>元素的inherited_id属性中标识。

我们不应该忘记在我们的addon 的manifest文件todo_website/_manifest__.py里声明这个额外的数据文件：

```
'data':
  [ 'views/todo_web.xml',
  'views/todo_extend.xml'],
```

在这之后，访问http://localhost:8069/hello?name=John 应该会展示一条 **Hello John!**消息。

我们还可以通过URL片段提供参数。例如，我们可以使用这种替代方案，用 http://localhost:8069/hello/John 这个URL得到完全相同的结果：

```
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
```

正如您所看到的，路由可以包含与要提取的参数对应的**placeholders**，然后传递给方法。占位符还可以指定一个转换器来实现特定的类型映射。例如，<int:user_id>会将user_id参数提取为整数值。

转换器是由Odoo使用的werkzeug库提供的一个特性，大多数可用的转换器都可以在werkzeug库的文档中找到，网址是http://werkzeug.poc oo.org/docs/routing/

Odoo添加了一个特定的、特别有用的转换器：提取一个模型记录。例如 @http.route('/hello/<model("res.users"):user>)将用户参数提取为res.users模型的记录对象。

HelloCMS!

现在我们尝试把它变得更有趣，并创建我们自己的简单的CMS。为此，我们可以用路由提取在URL中的一个模板名称（一个页面），然后将其渲染。然后我们可以动态地创建web页面，并由CMS提供服务。

事实证明这是很容易做到的：

```
@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，在你的网页浏览器，你将看到我们的**Hello World**网页！

实际上，内置的网站提供了CMS特性，包括在/page端点路由上更健壮的实现。

在werkzeug术语中，端点是路由的别名，并由其静态部分（没有占位符）表示。对于我们简单的CMS示例，端点是/hellocms。

大多数时候，我们希望我们的页面集成到Odoo的网站模块。在本章剩下的部分，我们将着手于修改website addon。

搭建网站

前面的例子所给出的页面没有集成到Odoo网站：我们没有页脚、菜单等等。Odoo website addon模块提供了所有这些功能，这样我们就不用自己操心这些问题了。

要使用它，我们应该首先在我们的工作实例中安装website addon模块，然后将它添加到我们的模块依赖中。manifest .py的depends键应该是这样的：

```
'depends': ['todo_kanban', 'website'],
```

为了使用website模块，我们还需要修改控制器和模板。

控制器的路由需要一个额外的website=True参数：

```
@http.route('/hello', auth='public', website=True)
def hello(self, **kwargs):
    return request.render('todo_website.hello')
```

模板需要插入到网站的总体布局中：

```
<template id="hello" name="Hello World">
  <t t-call="website.layout">
```

```
    <h1>Hello World!</h1>
  </t>
</template>
```

有了这个，我们以前使用的**Hello World!**例子现在应该可以显示在一个Odoo网站页面中了。

增加CSS和JavaScript素材

我们的网站页面可能需要一些额外的CSS或JavaScript素材。web页面的这个方面是由website模块管理的，所以我们需要某种方法来命令它也使用我们的文件。

我们将添加一些CSS来为完成的任务添加简单的删除线效果。为此，创建todo_website/static/src/css/index.css文件，内容如下：

```
.todo-app-done (
    text-decoration: line-through;
}
```

接下来，我们需要将它包含在网站页面中。这是通过在website.assets_frontend模板上添加它们来实现的，这个模板负责加载特定于网站使用的素材。编辑todo_website/views/todo_templates.xml数据文件，以扩展该模板：

```
<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>
```

我们将很快使用这个新的todo-app-done样式类。当然，也可以使用类似的方法添加JavaScript素材。

待办任务列表的控制器

现在已经完成了基础工作，让我们接着来完成待办任务（Todo Task）列表。我们将用一个/todo URL展示一个带有待办任务列表的web页面。

为此，我们需要一个控制器方法，准备当前的数据，以及一个QWeb模板来将该列表呈现给用户。

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，并使其可用于渲染模板。在这种情况下，控制器需要一个经过身份验证的会话（session），因为路由具有auth='user'属性。即便这是默认的参数值，也很好地说明了用户会话是必需的。

这样，待办任务的search()语句将与当前会话用户一起运行。

公共用户可以访问的数据非常有限，当使用这种类型的路由时，我们通常需要使用sudo()来提高访问权限，使页面数据可用；否则用户无法访问数据。

这也可能是一个安全风险，所以要注意验证两点：输入参数以及所做的动作。还应将sudo()记录集使用限制在的尽量小范围的操作中。

request.render()方法需要：1）被渲染的QWeb模板的标识符，2）一个用于模板中计算求值的上下文字典。

待办任务列表的模板

QWeb模板应该由一个数据文件添加，我们可以将其添加到现有的todo_website/views/todo_templates.xml数据文件中：

```
<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>
```

前面的代码使用t-foreach指令来渲染任务列表。在输入复选框中使用的t-att指令允许我们根据is_done值选择添加或不添加checked属性。

我们有一个输入复选框，并希望在任务完成时检查它。在HTML中，复选框的检查取决于其是否有checked属性。为此，我们使用t-att-NAME指令根据表达式动态渲染checked属性。在

这种情况下，表达式计算为`None`， `QWeb`将省略该属性，这对于本例来说是很方便的。

在渲染任务名称时，使用`t-attf`指令动态创建打开每个特定任务的详细表单的`URL`。我们使用特殊函数`slug()`为每个记录生成一个可读的`URL`。由于我们还未创建相应的控制器，所以这个链接现在无法工作。

在每个任务中，我们还使用`t-att`指令来为完成的任务设置`todo-app-done`样式。

最后，我们有一个**Add**按钮来打开一个表单页面，以创建一个新的待办任务。接下来我们将使用它来介绍`web`表单处理。

To-do Task详情页

待办任务列表中的每个项目都是一个超链接，目标是一个详情页面。我们应该为这些链接实现一个控制器，并提供一个QWeb模板进行页面展示。在这一点上，这应该是一个简单的练习。

在todo_website/controllers/main.py文件中添加方法：

```
#class Main(http.Controller):

    @http.route('/todo/<model("todo.task"):task>', website=True)
    def index(self, task, **kwargs):
        return
            http.request.render( 't
            odo_website.detail',
            ('task': task})
```

注意，该路由使用一个带有model (“todo.task”)转换器的占位符，它可以映射到任务变量。它从URL中捕获任务的标识符，也即一个简单的ID号或一个slug表示。之后将其转换为相应的浏览记录对象。

而对于QWeb模板，将以下代码添加到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">
    <hl 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>
```

这里值得注意的是<t t-field>元素的用法。它用于处理字段值的适当表示，就像在后端一样。例如，它正确地显示了日期值和many-to-one值。

网站表单

表单是网站上常见的功能。我们已经有了实现表单所需的所有工具:QWeb模板可以为表单提供HTML，相应的提交操作可以是一个由可以运行所有验证逻辑的控制器处理的URL，最后将数据存储在适当的模型中。

但对于非常规表单，这可能是一项艰巨的任务。执行所有的验证并向用户提供关于错误的反馈并不简单。

由于这是一个常见的需求，所以可以使用website_form addon来帮助我们。让我们看看如何使用它。

回头看待办任务列表中的**Add**按钮，我们可以看到它打开了/todo/add URL。这将呈现一个用于提交新的待办任务的表单，可用的字段将是任务名称、负责任务的人员（用户）和文件附件。

我们应该从添加website_form依赖到我们的addon模块开始。我们可以替换website，保留它是多余的。在todo_website/_manifest_.py编辑depends键为:

```
'depends': ['todo_kanban', 'website_form'],
```

现在我们使用website_form来添加我们页面。

表单页面

我们可以从实现支持表单渲染的控制方法开始。编辑`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，渲染了`todo_website.add`模板，并向它提供一个用户列表用来构建一个选择框。

现在处理对应的`QWeb`模板。我们可以把它加入到`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>
```

正如预期的那样，我们可以找到odoo特有的`<t t-call="website.layout">`元素，负责在网站布局中插入模板，以及用于设置一个网站布局需要的额外标题的`<t t-set="additional_title">`。

对于内容，我们在这个模板中看到的大部分内容都可以在一个典型的Bootstrap CSS表单中找到。但是我们也有一些特定于网站表单的属性和CSS类。我们在代码中以粗体标记它们，这样您就更容易识别它们。

JavaScript代码需要CSS类来正确执行其表单处理逻辑。然后我们有一些关于<form>元素的特定属性：

- action 是标准的表单属性，但必须具有"/website_form/"值。后面的斜杠是必需的。
- data-model_name 标识要写入的模型，并将传递给/website_form控制器。
- data-success_page 是在表单成功提交之后重定向的URL。在本例中，我们将被定向回/todo列表。

我们不需要提供我们自己的控制器方法来处理表单提交。/website_form路由将为我们处理这个操作。它从表单中获取所需的所有信息，包括刚刚描述的特定属性，然后对输入数据执行基本验证，并在目标模型上创建一个新记录。

对于高级用例，我们可以强制使用自定义控制器方法。为此，我们应该向<form>元素添加一个data-force_action属性，并使用目标控制器的名字。例如，data-force_action="todo-custom"将会在表单提交时调用/website_form/todo-custom URL。然后我们应该提供一个附着在该路由上的控制器方法。然而，这样做已经超出了本书涵盖的范围。

我们现在需要完成表单。添加字段以从用户那里获取输入。在<form>元素添加：

```
<!-- Description text field, required -->
<div class="form-group form-field">
  <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>
```

这里我们添加两个字段，一个存储描述信息的常规文本字段和一个用于上传附件文件字段。除了`o_website_from_input`类之外，所有的标记都可以在常规Bootstrap表单中找到。`o_website_from_input`需要网站表单逻辑来准备要提交的数据。

用户选择列表并没有太大的不同，只是需要使用一个`t-foreach QWeb`指令来渲染可选择用户的列表。这样得到操作之所以可行，是因为控制器检索了记录集，并使它可以在名为`users`的模板中使用:

```
<!-- 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>
```

但是，在我们进行一些访问安全设置之前，我们的表单仍然无法工作。

访问安全和菜单项目

由于这种通用的表单处理是相当开放的，并且依赖于客户机发送的不可信数据，出于安全原因，它需要服务器端去设置客户端所允许的操作。模型字段，特别是基于表单数据的那些，必须是在白名单中的才可以写入数据。

为了向这个白名单添加字段，可以在XML数据文件中使用一个辅助函数。我们应该创建todo_website/data/config_data.xml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>
```

为了创建一个能够被表单使用的模型，我们必须做两件事:在模型上启用website_form_access属性，以及声明可以使用的字段白名单。这是在前面的数据文件中执行的两个操作。

不要忘记，对于我们的addon模块可以装载这个数据文件，需要将它添加到清单文件的data键中。

如果我们的待办任务页面也可以出现在网站菜单中，那就更好了。让我们使用相同的数据

文件添加这个功能。添加另一个`<data>`元素如下：

```
<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>
```

如您所见，要添加一个网站菜单项，我们只需要在创建一个`website.menu`模型的记录，此记录需要设置名称、`URL`和父菜单项的标识符。此菜单的父菜单为顶级菜单项`website.main_menu`。

增加自定义逻辑

网站表单允许我们将自己的验证和计算逻辑，插入到表单处理逻辑中。这是通过在目标模型上实现一个`website_form_input_filter()`方法来实现的。向此方法传入一个`values`字典，验证并对其进行更改，然后返回被修改后的`values`字典。

我们将使用它来实现两个特性:从任务标题中删除任何前置和尾随空格，并强制任务标题长度至少有三个字符。

向`todo_website/models/todo_task.py`文件中添加以下代码：

```
# -*- 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：`request`对象和`values`字典。验证表单提交失败时，应该引发`ValidationError`异常。

大多数情况下，表单的扩展应该不需要我们自定义表单提交处理程序。

像往常一样，我们必须`Python`导入这个新文件，方法是在`todo_website/_init_.py` 文件中添加`from .import models`。另外，新建一个`todo_website/models/_init_.py`文件并添加`from . import todo_task`行。

小结

你现在应该对网站的基本功能有一个很好的了解。我们已经了解了如何使用`web`控制器和`QWeb`模板来呈现动态`web`页面。然后我们学习了如何使用`website addon`并在其基础上创建我们自己的页面。最后，我们介绍了`website_form addon`，它帮助我们创建了一个`web`表单。通过这些我们掌握了创建网站功能所需的核心技能。

接下来，我们将学习如何让外部应用程序与我们的`Odoo`应用程序交互。

12

外部API-集成其他系统

Odoo服务器还提供一个外部API，该API由其web客户端使用，也可用于其他客户端应用程序。

在本章中，我们将学习如何从我们的客户端程序中使用Odoo外部API。任何支持XML-RPC或JSON-RPC协议的编程语言，我们都可以使用。例如，官方文档为四种流行的编程语言提供了代码示例：Python、PHP、Ruby和Java。

为了避免引入读者可能不熟悉的额外语言，这里我们将重点介绍基于python的客户端，尽管处理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，然后使用它构建一个使用Python的简单的待办任务桌面应用程序。

最后，我们将介绍ERPPeek客户端。它是一个Odoo客户端库，它可以作为Odoo RPC调用的

简易抽象层，也是Odoo的命令行客户端，允许远程管理Odoo实例。

设置Python客户端

可以使用两种不同的协议来访问Odoo API: XML-RPC和JSON-RPC。任何能够实现其中一个协议的客户端的外部程序都可以与Odoo服务器进行交互。为了避免引入额外的编程语言，我们将继续使用Python来探索外部API。

到目前为止，我们只在服务器上运行Python代码。这一次，我们将在客户端使用Python，所以您可能需要在您的工作站上做一些额外的设置。

要实现本章中的示例，您需要能够在工作计算机上运行Python文件。Odoo服务器需要Python 2，但是我们的RPC客户机可能会使用任何语言，因此Python 3也是可以使用的。但是，由于一些读者可能在他们的工作计算机上运行着服务器（你好，Ubuntu用户!），所以我们继续使用Python 2，这样对所有读者都更简单些。

如果你正在使用Ubuntu或Mac， Python可能已经安装好了。打开终端控制台，输入python，您应该得到如下内容：

```
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.
>>>
```

Windows用户可以为Python找到一个安装程序，并且还能快速上手。官方安装包可以在https://www.python.org/downloads/找到。

如果您是Windows用户，并且在您的机器中安装了Odoo，您可能会奇怪为什么您没有Python解释器，需要额外的安装。简短的回答是，Odoo安装有一个嵌入式Python解释器，但不方便在外部使用。

使用XML-RPC调用Odoo API

访问服务器的最简单方法是使用XML-RPC。我们可以在Python的标准库中使用xmlrpclib库。请记住，我们编程客户端目的是为了连接到服务器，因此我们需要一个正在运行的Odoo服务器实例来连接。在我们的示例中，我们假设一个Odoo服务器实例运行在同一个机器上(localhost)，但是如果服务器在另一台机器上运行，那么您可以使用任何可访问的IP地址或服务器名。

打开XML-RPC连接

让我们和Odoo外部API进行第一次接触。启动一个Python控制台并输入以下内容：

```
>>> 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}
```

Here, we import the xmlrpclib library and then set up a variable with the information for the server address and listening port. Feel free to adapt these to your specific set up.
在这里，我们导入xmlrpclib库，然后为服务器地址和监听端口设置一个变量。请随意调整这些设置。

接下来，我们在/xmlrpc/2/common端点上设置了访问服务器公共服务（不需要登录）的权限。可用的方法之一是version()，它可以检查服务器版本。我们使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，与您认为的恰恰相反，这个方法并没有创建一个会话。此方法只确认用户名和密码被服务器接受，并返回应在请求中使用的用户ID，而不是用户名，如下所示：

```
>>> db = 'todo'
>>> user, pwd = 'admin', 'admin'
>>> uid = common.authenticate(db, user, pwd, {})
>>> print uid
```

如果登录凭证不正确，将返回False值，而不是用户ID。

从服务器读取数据

使用XML-RPC时，不会产生会话，并且每个请求都要发送身份验证凭据。这增加了协议的开销，但是使用起来更简单。

接下来，我们设置访问需要登录的服务器方法。这些在/xmlrpc/2/object端点上公开，如下所示：

```
>>> api = xmlrpclib.ServerProxy('%s/xmlrpc/2/object' % srv)
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'search_count'[[]]) 40
```

在这里，我们将第一次访问服务器API，执行对**Partner**记录数量的计数。相关方法将被execute_kw()方法调用，该方法需要以下参数：

- 连接的数据库的名称。
- 当前连接用户ID。
- 用户密码。
- 目标模型标识符。
- 调用的方法。
- 位置参数列表。
- 一个带有关键字参数的可选字典。

前面的例子调用了res.partner模型的search_count方法，其中有一个位置参数，[]，没有关键字参数。位置参数是一个搜索域；由于我们提供了一个空的列表，所以它会计算所有的合作伙伴。

相对频繁的动作是search和read。当从RPC调用时，search方法返回一个匹配域的ID列表。RPC不能获得browse方法，所以应该在其位置使用read方法通过记录ID列表检索数据，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'}]
```

请注意，对于read方法，我们使用的是ID列表的一个位置参数，[18]，以及一个关键字参数fields。我们还可以注意到，many-to-one关系字段是作为pair检索的，结果同时包含相关记录的ID和显示名。在处理代码中的数据时，要记住这一点。

search和read的组合执行非常频繁，因此提供了一个search_read方法来在单个步骤中执

行这两个操作。与前两个步骤相同的结果可以得到如下：

```
>>> api.execute_kw(db, uid, pwd, 'res.partner', 'search_read',
[[('country_id', '=', 'be'), ('parent_id', '!=', False)]],  {'fields':
['id', 'name', 'parent_id']})
```

search_read方法的行为类似于read，但它希望将域作为第一个位置参数，而不是ID列表。值得一提的是，read和search_read的field参数不是强制的。如果没有提供，将检索所有字段。这可能会触发功能字段的（消耗极多性能的）计算和检索大量不需要的数据。因此通常建议提供一个字段的显式列表。

调用其他方法

所有其他的模型方法都是通过RPC公开的。预先设置的方法（有下划线），则被认为是私有的。这意味着我们可以使用create、write和unlink修改服务器上的数据如下：

```
>>> 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
```

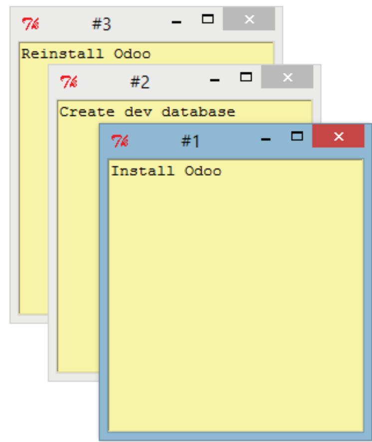
XML-RPC协议的一个限制是它不支持None值。其含义是，不返回任何结果的方法不能通过XML-RPC使用，因为它们隐式返回None。这就是为什么方法应该总是至少以一句return True结束。

值得强调的是，大多数编程语言都可以使用Odoo外部API。在官方文档中，我们可以找到关于Ruby、PHP和Java实例。地址为

https://www.odoo.com/documentation/10.0/api_integration.html.

编写一个笔记桌面应用程序

让我们用RPC API做一些有趣的事情。Odoo提供了一个简单的笔记应用程序。如果用户想要直接从他们的电脑桌面上管理他们的个人笔记呢?让我们编写一个简单的Python应用程序来实现这一点，如下面的截图所示：



为了清晰起见，我们将把它拆分为两个文件:一个处理与服务器后端的交互的note_api.py；还有一个带有图形用户界面的note_gui.py。

通信层与Odoo

我们将创建一个类来设置连接并存储其信息。它应该公开两种方法: get()来检索任务数据和set()来创建或更新任务。

选择一个目录来托管应用程序文件，并创建`note_api.py`文件。我们可以从添加类构造器开始，如下所示:

```
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，我们存储创建对象中需要的所有信息，以便在模型上执行调用：API引用、uid、密码、数据库名称和使用的模型。

接下来，我们将定义一个辅助方法来执行调用。它利用对象存储的数据提供一个较小的函数签名，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级别get()和set()方法。

get()方法将可以接受一个ID列表作为可选参数来检索。如果没有列出，所有记录将返回：

```
def get(self, ids=None):
    domain = [('id',' in', ids)] if ids else []
    fields = ['id', 'name']
    return self.execute('search_read', [domain, fields])
```

set()方法可以传入任务文本，以及一个ID作为可选的参数。如果没有提供ID，将创建一个新记录。它将返回写入或创建的记录的ID，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
```

让我们以一小段测试代码结束该文件，如果我们运行Python文件，将执行以下代码：

```
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Python脚本，我们应该可以看到打印出来的待办任务的内容。这样，我们在Odoo后端已经有了一个简单的包装器，现在让我们来处理桌面用户界面。

创建GUI

我们的目标是学习在外部应用程序和Odoo服务器之间编写接口，我们在前一节中已经基本完成。但如果不进一步开发并让终端用户能够使用它，是我们学习过程中极大的不足。

为了使安装尽可能简单，我们将使用Tkinter来实现图形用户界面。由于它是标准库的一部分，所以不需要任何额外的安装。我们的目标并不是解释Tkinter是如何工作的，所以我们只简要介绍它。

每个**task**应该在桌面上有一个小的黄色窗口。这些窗口将有一个单独的**text**小部件。按下*Ctrl + N*将会打开一个新笔记，按下*Ctrl + S*将会把当前笔记的内容写到Odoo服务器上。

现在，除了note_api.py文件外，再创建一个新的note_gui.py文件。它将首先导入我们将要使用的Tkinter模块和小部件，然后是NoteAPI类，如下所示：

```
from Tkinter import Text, Tk
import tkMessageBox
from note_api import NoteAPI
```

如果前面的代码产生错误，并提示ImportError: No module named _tkinter, please install the python-tk package，这意味着系统需要额外的库。在Ubuntu上，你需要运行以下命令：

```
$ sudo apt-get install python-tk
```

接下来，基于Tkinter one，我们将创建自己的文本(Text)小部件。在创建实例时，它需要传入一个用于save动作的API引用，以及任务文本和ID，如下所示：

```
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UI窗口，文本小部件将自己放入其中。因此创建一个新的NoteText实例时，将自动打开一个桌面窗口。

接下来，我们将实现create和save动作。create动作将打开一个新的空白窗口，但只有在执行save操作时，它才会将信息存储在服务器中。这是对应的代码：

```
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)
```

Save动作既可以在现有任务中执行，也可以在新任务上执行，但是在这里没有必要担心，因为这些情况已经由NoteAPI的set()方法处理了。

最后，我们将添加在程序启动时检索并创建所有笔记窗口的代码，如下所示：

```
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()
```

最后一个命令在创建的最后一个注释窗口中运行`mainloop()`，以开始等待窗口事件。

这是一个非常基本的应用程序，但这里的重点是要以有趣的方式来利用`Odoo RPC API`。

引入ERPpeek客户端

`ERPpeek`是一种多功能的工具。它既可以作为**Command-line Interface (CLI)**，也可以作为**Python library**使用。它提供的`API`比`xmlrpclib`的更方便。可以从`PyPi`索引中获得`ERPpeek`，安装语句如下：

```
$ pip install -U erppeek
```

在`Unix`系统上，如果您准备在系统全局安装它，那么您可能需要将`sudo`添加到该命令之前。

ERPpeek的API

`ERPpeek`库提供了一个对`xmlrpclib`包装过的编程接口，这与我们服务器端代码中的编程接口类似。

下面的介绍是让您了解`ERPpeek`库提供的内容，而不是对其所有特性提供完整的解释。

我们可以使用erppeek替代xmlrpclib重新生成我们的第一个步骤，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'])
```

正如您所看到的，这个API调用参数更少，且类似服务器端的对应代码。
除此之外，ERPpeek还有更多的特性，如它还提供了模型的表达。我们有以下两种方法来获得模型的实例，1）使用model()方法，2）或者使用驼峰命名法的模型名称来进行访问：

```
>>> m = api.model('res.partner')
>>> m = api.ResPartner
```

现在我们可以对该模型执行以下操作：

```
>>> m.count([('name', 'like', 'Packt%')]) 1
>>> m.search([('name', 'like', 'Packt%')])
[44]
```

它还为记录提供了客户端对象形式的表达：

```
>>> recs = m.browse([('name', 'like', 'Packt%')])
>>> recs
<RecordList 'res.partner,[44]'>
>>> recs.name ['Packt
Publishing']
```

正如您所看到的，erppeek库比普通的xmlrpclib要涵盖了更多的特性，同时让重用服务器端的代码编写客户端成为可能。这个过程中代码只需要很少的修改，或根本无需修改。

ERPpeek CLI

erppeek库不仅可以用作Python库，它同时也是一个用于在服务器上执行管理操作的交互式命令行界面(CLI)。当odoo shell命令在主机服务器上提供本地交互会话时，erppeek库允许客户机通过网络进行远程交互会话。

打开命令行，我们可以查看可用的选项，如下所示：

```
$ erppeek --help
```

让我们来看一个示例会话：

```
$ 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()
3
todo >>> rec = model('res.partner').browse(43)
todo >>> rec.name
'Packt Publishing'
```

如您所见，我们连接到了服务器上，而执行上下文提供了对model()方法的引用，并在获取了模型实例后对它们执行动作。
用于连接服务器的erppeek.Client实例也可以通过client变量引用。

值得注意的是，它提供了一个替代web客户端来管理安装的附加组件的功能：

- client.modules()：列出可用的或已安装的模块。
- client.install():执行模块安装

- `client.upgrade()`：执行模块升级
- `client.uninstall()`：卸载模块

因此，`erppeek`还可以为`Odoo`服务器提供良好的远程管理工具。

小结

本章的目标是了解外部API的工作原理以及它的功能。我们最开始使用了一个简单的Python XML-RPC客户机来探索它，但是外部API可以在任何编程语言中使用。事实上，官方文档为Java、PHP和Ruby提供了代码示例。

有许多的库都可以处理XML-RPC或JSON-RPC，Odoo使用了其中一些通用库以及一些odoo特定的库。本书除了erppeek之外，尽量不涉及其他任何库。因为它不仅是一个久经考验的Odoo/OpenERP XML-RPC包装器，而且它在远程服务器管理和检查中也具有极高的应用价值。

直到现在，我们还使用了Odoo服务器实例进行开发和测试。但是，要拥有一个生产级服务器，还需要进行额外的安全性和优化配置。在下一章中，我们将关注这些内容。

13

部署清单-实战

在本章中，您将了解如何准备您的Odoo服务器，以便在生产环境中使用。

我们有许多用来部署和管理一个Odoo生产服务器的策略和工具。接下来我们将指导你使用其中的一种。

这是我们将遵循的服务器设置清单：

- 安装依赖项和一个专用用户来运行服务器
- 用源代码安装Odoo
- 设置Odoo配置文件
- 设置处理多进程的workers
- 设置Odoo系统服务
- 使用SSL支持设置反向代理

可用的预先构建包

Odoo有一个Debian/Ubuntu软件包可供安装。使用它，您将得到一个在系统启动时自动启动的工作服务器进程。这个安装过程很简单,你可以在 https://nightly.odoo.com找到相关信息。您还可以找到针对CentOS的rpm构建以及 .exe安装程序。

虽然这是安装Odoo的一种简单且方便的方式，但大多数集成商更喜欢部署和运行版本控制的源代码。这样可以更好地控制部署的内容，使之生产过程中管理更改和修复bug变得更加容易。

安装依赖

在使用Debian发行版时，默认情况下，您的登录是具有管理员权限的root，您的命令提示符显示#。在使用Ubuntu时，root帐户的登录被禁用，并且在安装过程中配置的初始用户是一个*sudoer*，这意味着它可以使用sudo命令来运行具有根权限的命令。

首先，我们应该更新包索引，然后进行升级，以确保所有已安装的程序都是最新的：

```
$ sudo apt-get update
$ sudo apt-get upgrade -y
```

接下来，我们将安装PostgreSQL数据库，并使我们的用户成为数据库超级用户：

```
$ sudo apt-get install postgresql -y
$ sudo su -c "createuser -s $(whoami)" postgres
```

我们将从源代码运行Odoo，但在此之前，我们需要安装所需的依赖项。以下是所需的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
```

我们不应该忘记安装wkhtmltox，需要用它打印报表：

```
$ wget

$ sudo dpkg -i wkhtmltox-0.12.1.2_linux-jessie-amd64.deb
$ sudo apt-get -fy install
```

安装指令将报告一个丢失的依赖项错误，但是最后一个命令强制安装这些依赖项，并最终正确地完成安装。

现在我们只缺少了Odoo所需要的Python包。这些Python包大多也有Debian/Ubuntu系统的版本。官方的Debian安装包也使用它们，您可以在Odoo源代码中的debian/control文件中找到这些包名。

但是，这些Python依赖项也可以直接从**Python Package Index (PyPI)**中安装。这对那些喜欢在virtualenv中安装Odoo的人来说更友好。所需包的列表在Odoo的requirements.txt中（大部分基于python的项目都会有这个文件）。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
```

现在我们已经安装了所有依赖项，包括数据库服务器、系统包和Python包，接下来就可以安装Odoo了。

准备一个专用的系统用户

使用在系统上没有特殊的特权的专用用户运行Odoo，是一项更加安全的选择。

我们需要为此创建系统和数据库用户。我们可以命名为`odoo-prod`，例如：

```
$ sudo adduser --disabled-password --gecos "Odoo" odoo
$ sudo su -c "createuser odoo" postgres
$ createdb --owner=odoo odoo-prod
```

这里，`odoo`是用户名，`odoo-prod`是支撑`odoo`实例的数据库的名称。

注意，这些都是没有任何管理特权的常规用户。系统会为新用户自动创建一个主目录。在本例中，它是`/home/odoo`，用户可以使用 `~` 快捷符号引用它自己的主目录。我们将使用它来处理用户的`Odoo`特定配置和文件。

我们可以使用以下命令打开一个会话：

```
$ sudo su odoo
```

`exit`命令终止该会话，并回到我们的原始用户。

从源代码安装

或早或晚，您的服务器将需要升级和补丁。到那个时候，一个版本控制的仓库可能使你获益良多。我们使用`git`从仓库中获取代码，就像我们安装开发环境一样。

接下来，我们将进入`odoo`用户并将代码下载到其主目录中：

```
$ sudo su odoo
$ git clone https://github.com/odoo/odoo.git /home/odoo/odoo-10.0 -b 10.0
--depth=1
```

`-b`选项确保我们得到了正确的分支，并且`——depth=1`选项忽略了更改历史，只检索最新的代码修改，使下载内容变得更小，下载速度更快。

`Git`肯定会成为管理您`Odoo`部署版本的宝贵工具。我们只是粗略地了解了如何管理代码版本。如果您还不熟悉`Git`，那么就值得进一步了解它。`http://git-scm.com/doc`将会是一个好的学习起点。

现在，我们应该拥有从源代码运行`Odoo`所需的一切。我们可以检查它是否正确启动，然后退出专用用户的会话：

```
$ /home/odoo/odoo-10.0/odoo-bin --help
$ exit
```

接下来，我们将设置一些系统级的文件和目录供系统服务使用。

建立配置文件

在启动一个Odoo服务器时添加--save选项，可以保存用于~/.odoorc文件的配置。我们可以将该文件作为服务器配置的起点，服务器配置将存储在/etc/odoo中，代码如下所示：

```
$ sudo su -c "~/odoo-10.0/odoo-bin -d odoo-prod --save --stop-after-init"
odoo
```

这将使我们的服务器实例使用配置参数。

之前版本的.openerp_serverrc配置文件仍被Odoo10支持，如果系统找到这种配置文件，那么将会使用它。当一台机器还被用来运行之前版本的Odoo，同时在此机器上而配置Odoo 10时，这可能会造成一些混乱。因为在这种情况下，--save选项可能正在更新.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服务存储日志文件的目录。这应该是在/var/log内的某个地方：

```
$ sudo mkdir /var/log/odoo
$ sudo chown odoo /var/log/odoo
```

现在，我们应该确保配置了一些重要的参数。以下是一些最重要参数的建议值：

```
[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
```

对于它们的解释：

- addons_path是一个逗号分隔的路径列表，位于其中的附加模块将被查询。文件从左到右读取，最左边的目录具有更高的优先级。
- admin_passwd是访问web客户端数据库管理功能的主密码。重要的是设置一个强大的密码，或者最好设置为False来禁用此功能。
- db_user在服务器启动序列中初始化的数据库实例。
- dbfilter是用于访问数据库的过滤器。这是一个Python解释的正则表达式。为了不提示用户去选择数据库，以及使未经身份验证的url能够正常工作，应设置成^dbname$样式，例如，dbfilter=^odoo-prod$。它支持%h和%d占位符，由HTTP请求主机名和子域名替换。
- logfile是服务器日志应该写入的位置。对于系统服务，预期的位置在/var/log中。如果不写，或设置为False，则将日志打印到控制台。
- proxy_mode应该设置为True，因为我们将在反向代理后访问Odoo。
- without_demo应该在生产环境中设置为True，这样新的数据库就不会有演示数据。
- workers 设置为两个或两个以上的值时，系统将支持多进程模式。我们稍后将更详细地讨论这个问题。 .
- xmlrpc_port是服务器侦听的端口号。默认情况下使用端口8069.

下面的参数也很有用:

- data_dir是存储会话数据和附件文件的路径。要记住备份。

- xmlrpc-interface设置将被侦听的地址。默认情况下，它会监听所有的0.0.0.0。但是，当使用反向代理时，为了只响应本地请求可以将其设置为127.0.0.1。

我们可以使用-c或--config选项运行服务器来检查的设置的效果，如下：

```
$ sudo su -c "~/odoo-10.0/odoo-bin -c /etc/odoo/odoo.conf" odoo
```

使用上述设置运行Odoo时不会控制台不会显示任何日志输出。因为输出都被写入到配置文件中定义的日志文件中。要了解服务器的情况，我们需要打开另一个终端窗口，然后运行：

```
$ sudo tail -f /var/log/odoo/odoo-prod.log
```

即便使用配置文件，也仍然可以强制将日志输出打印到控制台，通过添加--logfile=False选项，如下所示：

```
$ sudo su -c "~/odoo-10.0/odoo-bin -c /etc/odoo/odoo.conf --logfile=False"
odoo
```

多进程的workers

预计生产实例将处理大量工作负载。默认情况下，因为Python语言GIL，服务器只运行一个进程，并且只能使用一个CPU内核进行处理。但是可以使用多进程模式，以便处理并发请求。选项workers=N设置要使用的工作进程的数量。作为一个指导方针，您可以尝试将其设置为1+2*P，其中P是处理器的数量。对于每种情况，最好的设置都是需要调优的。因为最优设置取决于服务器负载以及服务器上运行的其他负载密集型服务，比如PostgreSQL。

把workers设置的高一些会比较好。因为大多数浏览器使用并行连接，所以最小值应该是6，而最大值通常被机器上的RAM大小限制。

这里有几个limit-* 配置参数来调整workers。workers在达到这些限制时被循环利用——相应的进程则被停止，然后启动新的进程。这可以保护服务器不受内存泄漏的影响，也可以防止服务器资源过载。

官方文档对优化worker参数已经提供了好的建议，更多供参考更多的细节，请访问 https://www.odoo.com/documenta tion/10.0/setup/deploy.html。

设置为系统服务

接下来，我们希望将Odoo设置为系统服务，并在系统启动时自动启动。

在Ubuntu/Debian中，init系统负责启动服务。从历史上看，Debian(和派生的操作系统)使用了sysvinit，Ubuntu使用了一个名为Upstart的兼容系统。目前情况已经发生了变化，在最新版本中使用的init系统现在被称作systemd。

这意味着有两种不同的方式来安装系统服务，您需要根据操作系统的版本选择正确的方法。

在最新的Ubuntu稳定版本16.04，我们应该使用systemd。但是像14.04这样的旧版本仍

然在许多云服务提供商中使用，所以您也有可能需要使用它。

为了检查您的系统是否使用systemd，请尝试以下命令：

```
$ man init
```

这会打开当前使用的init系统的文档，您可以检查正在使用的是什么系统。

创建一个systemd服务

如果您正在使用的操作系统是最新的，比如Debian 8或Ubuntu 16.04，那么您应该使用systemd作为init系统。
为了向系统添加新服务，我们只需要创建一个描述它的文件。创建一个/lib/systemd/system/odoo.service文件，内容如下：

```
[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
```

要启动此新服务，请使用以下命令：

```
$ sudo systemctl odoo start
```

检查它的状态运行如下：

```
$ sudo systemctl odoo status
```

最后，如果您想要停止它，请使用以下命令：

```
$ sudo systemctl odoo stop
```

创建一个Upstart/sysvinit服务

如果您使用的是旧的操作系统，比如Debian 7、Ubuntu 15.04，甚至是14.04，那么您的系统可能是sysvinit 或Upstart。在这种情况下，两个系统的操作是基本一样的。许多云VPS服务仍然基于Ubuntu 14.04，所以这可能是您在部署Odoo服务器时会遇到的场景。

Odoo源代码包括用于Debian封装发行版的init脚本。我们可以使用它作为我们的服务初始化脚本，有一些细微的修改，如下：

```
$ sudo cp /home/odoo/odoo-10.0/debian/init /etc/init.d/odoo
$ sudo chmod +x /etc/init.d/odoo
```

此时，您可能需要检查init脚本的内容。关键参数被分配到文件顶部的变量。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
```

这些变量应该足够，我们需要记住它们的默认值并在头脑中准备剩下的配置。但当然，你可以改变它们以更好地满足你的需求。

USER变量是服务器运行的系统用户。好在我们已经创建了预期的odoo用户。

DAEMON变量是通往服务器可执行文件的路径。我们启动Odoo的实际可执行文件在其他的位置，但是我们可以创建一个符号链接到它：

```
$ sudo ln -s /home/odoo/odoo-10.0/odoo-bin /usr/bin/odoo
$ sudo chown -h odoo /usr/bin/odoo
```

CONFIG变量是要使用的配置文件。在前一节中，我们在默认位置创建了一个配置文件：/etc/odoo/odoo.conf。

最后，LOGFILE变量是存储日志文件的目录。预期目录是我们在定义配置文件时创建的/var/log/odoo。

现在我们应该能够开始和停止我们的Odoo服务，命令如下：

```
$ sudo /etc/init.d/odoo start
Starting odoo: ok
```

停止服务的方式类似，如下所示：

```
$ sudo /etc/init.d/odoo stop
Stopping odoo: ok
```

在Ubuntu中，也可以使用service命令：

```
$ sudo service odoo start
$ sudo service odoo status
$ sudo service odoo config
```

我们现在只需要让这个服务在系统启动时自动启动：

```
$ sudo update-rc.d odoo defaults
```

在此之后，当我们重新启动服务器时，Odoo服务应该自动启动，并且不会出现错误。现在是检查一切是否正常工作的好时机。

从命令行检查Odoo服务

此时，我们可以确认我们的Odoo实例是否已经启动并响应请求。

如果Odoo运行正常，我们现在应该能够从它得到响应，并且在日志文件中没有出现错误。如果Odoo使用以下命令响应HTTP请求，我们可以在服务器内部进行检查：

```
$ curl http://localhost:8069
<html><head><script>window.location = '/web' +
location.hash;</script></head></html>
```

为了查看日志文件中的内容，我们可以使用以下内容：

```
$ sudo less /var/log/odoo/odoo-server.log
```

如果您刚刚开始使用Linux，那么告诉您一条小技巧，就是可以使用tail -f来跟踪日志文件中正在发生的事情：

```
$ sudo tail -f /var/log/odoo/odoo-server.log
```

使用反向代理

虽然Odoo本身可以提供web页面服务，但强烈建议在它之前添加一个反向代理。反向代理充当客户端发送请求和响应它们的Odoo服务器之间的通信的中介。使用反向代理有几个好处。

在安全方面，它可以做到以下几点：

- 处理（和强制使用）HTTPS协议来加密通信
- 隐藏内部网络特征
- 作为一个应用程序防火墙，限制用于处理的URL

在性能方面，它可以提供显著的改进：

- 缓存静态内容，从而减少Odoo服务器的负载
- 压缩内容以加速加载时间
- 充当负载均衡器，在多个服务器之间分配负载

Apache是反向代理的流行选项。Nginx是新出现的一种选择，技术参数很不错。在这里，我们将选择使用Nginx作为反向代理，并展示如何使用它来完成这里提到的安全性和性能方面的功能。

为反向代理设置Nginx

首先，我们应该安装Nginx。我们希望它侦听默认的HTTP端口，因此我们应该确保它们没有被其他服务占用。执行此命令会导致错误，如下：

```
$ curl http://localhost
curl: (7) Failed to connect to localhost port 80: Connection refused
```

如果没有，您应该禁用或删除使用此端口的服务，以允许Nginx使用这些端口。例如，要停止现有的Apache服务器，您应该使用以下命令：

```
$ sudo service apache2 stop
```

或者更好的是，您应该考虑从系统中删除Apache，或者重新配置它以侦听另一个端口，这样Nginx就可以自由地使用HTTP和HTTPS端口(80和443)。

现在我们可以安装Nginx，它是如下的方式完成的：

```
$ sudo apt-get install nginx
```

要确认其工作正常，我们在使用浏览器访问服务器地址或在我们的服务器中使用curl http://localhost，结果会显示一个**Welcome to nginx**页面。

Nginx配置文件遵循与Apache相同的规则:它们存储在/etc/nginx/available-sites/并通过添加一个符号链接到/etc/nginx/enabled-sites/激活。我们还应该禁用Nginx安装提供的默认配置，如下所示:

```
$ 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nano或vi，我们应该通过如下方式编辑Nginx配置文件：

```
$ sudo nano /etc/nginx/sites-available/odoo
```

首先，我们添加了上游后端服务器，之后Nginx将把通信重定向到我们的案例中的Odoo服务器，该服务器正在监听端口8069，如下所示：

```
upstream backend-odoo
  ( server
  127.0.0.1:8069;
}
server
  ( location /
  (
    proxy_pass http://backend-odoo;
  }
}
```

要测试编辑的配置是否正确，请使用以下命令：

```
$ sudo nginx -t
```

如果发现错误，请确认配置文件是否正确输入。另外，一个常见的问题是默认的HTTP会被另一个服务（比如Apache或默认的Nginx网站）接收。再次我们回顾之前给出的关于这个问题的指令，并确保指令都被正确执行，然后重新启动Nginx。在此之后，我们可以让Nginx重新加载新配置，如下所示：

```
$ sudo /etc/init.d/nginx reload
```

我们现在可以确认Nginx正在将流量重定向到后端Odoo服务器：

```
$ curl http://localhost
<html><head><script>window.location = '/web' +
location.hash;</script></head></html>
```

使用HTTPS

接下来，我们应该安装一个证书来使用SSL。要创建自签名证书，请遵循以下步骤：

```
$ 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
```

这将在/etc/nginx/目录中创建一个ssl/目录，并创建一个无密码自签名SSL证书。在运行openssl命令时，会询问一些附加信息，并生成一个证书和密钥文件。最后，这些文件的所有权归于运行web服务器的用户www-data。

使用自签名证书可能会带来一些安全风险，比如中间人攻击，甚至某些浏览器不允许使用此类证书。为了得到健壮的解决方案，您应该使用由认可的证书颁发机构签署的证书。如果你在经营一个商业或电子商务网站，这一点尤其重要。

现在我们有了SSL证书，我们准备配置Nginx来使用它。

为了执行HTTPS，我们将把所有的HTTP通信都重定向到它。替换前面定义的服务器指令：

```
server
  ( listen
  80;
  add_header Strict-Transport-Security max-age=2592000;
  rewrite ^/.*$ https://$host$request_uri? permanent;
}
```

如果我们现在重新加载Nginx配置并使用web浏览器访问服务器，我们将看到http://地址将被转换为https://地址。

但是在我们正确配置HTTPS服务之前，它不会返回任何内容。需要添加以下服务器配置：

```
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，并使用/etc/nginx/ssl/证书文件加密通信。我们还向请求头添加了一些信息，以便让Odoo后端服务知道它被代理了。

出于安全原因，Odoo必须确保proxy_mode参数设置为True。原因是，当Nginx作为代理时，所有的请求都将来自服务器本身而不是远程IP地址。在代理中设置X-Forwarded-For请求头，并启用--proxy-mode可以解决这个问题。请注意，启用--proxy-mode而在代理设置上面的请求头时，将允许任何人使用其远程地址进行欺骗。
最后，location指令定义所有请求都传递到上游backend-odoo。

重新加载配置，我们的Odoo服务应该已经是通过HTTPS工作了，检查命令如下所示：

```
$ 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>
```

最后一个输出确认Odoo web客户端是通过HTTPS服务的。

对于使用Odoo POSBox的特殊情况，我们需要为/pos/ URL添加一个异常，以便能够以HTTP模式访问它。POSBox位于本地网络中，但没有启用SSL。如果POS接口被加载到HTTPS中，它将无法与POSBox通信。.

Nginx的优化

现在，是时候对Nginx设置进行一些微调了。建议启用它们的响应缓冲和数据压缩，以提高网站的速度。我们还为日志设置了一个特定的位置。

在端口443上的侦听器中应该添加以下配置，例如，在代理定义之后：

```
# 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;
```

我们还可以激活静态内容缓存，以更快地响应前面代码示例中提到的请求类型，并避免它们在Odoo服务器上的负载。在location /部分后，添加以下第二位置部分：

```
location ~* /web/static/ ( #
  cache static data
  proxy_cache_valid 200 60m;
  proxy_buffering on; expires
  864000;
```

```
        proxy_pass http://backend-odoo;
    }
```

这样，静态数据就会被缓存60分钟。在此期间内对这些请求的进一步请求将由Nginx从缓存中直接响应。

长轮询

长轮询用于支持即时消息应用程序，在使用多进程wokers时，它在一个单独的端口上处理，默认情况下是8072。

对于我们的反向代理，这意味着应该将长轮询请求传递到这个端口。为了支持这一点，我们需要向Nginx配置添加一个新的上游，如下代码所示：

```
upstream backend-odoo-im ( server 127.0.0.1:8072; }
```

接下来，我们应该向服务器添加另一个位置来处理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应该将这些请求传递给适当的Odoo服务器端口。

服务器和模块更新

Odoo服务器准备好并开始运行后，过一段时间您可能需要在Odoo上安装更新。这包括两个步骤：首先，获得新版本的源代码（服务器或模块），然后安装它们。

如果您遵循了从源代码部分安装的方法，我们可以获取和测试staging仓库中的新版本。强烈建议您复制生产数据库，并用之测试升级。如果odoo-prod是您的生产数据库，可以使用以下命令来完成：

```
$ 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，在生产服务上执行升级应该是安全的。记住要记住当前版本的Git引用，以便能够通过再次检出这个版本来回滚。强烈建议在执行升级之前，保存数据库的备份。

可以使用以下createdb命令以更快的方式生成数据库副本：

```
$ createdb --template odoo-prod odoo-stage
```

注意事项：要运行此指令，odoo -prod数据库就不能存在任何开放连接，因此需要停止Odoo服务器来执行复制。

在此之后，我们可以使用Git将新版本拖到生产库，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发布策略，目前时不再发布任何小版本。GitHub分支代表着最新的稳定版本。
夜间构建被认为是最新的官方稳定版本。

在更新频率上，更新太频繁是没有意义的，但也不要等待一年再更新。每隔几个月进行一次更新就可以了。通常，服务器重启将足以支持代码更新，并不需要模块升级。

当然，如果您需要一个特定的bug修复，可以尽早进行更新。还记得要关注在公共频道– GitHub Issues或社区邮件列表上公布的安全漏洞。作为服务的一部分，企业合同客户可以更早地收到此类问题的邮件通知。

小结

在本章中，您了解了在基于Debian的生产服务器中设置和运行Odoo的额外步骤。访问了配置文件中最重要的设置，并学习了如何利用多进程模式。

为了提高安全性和可伸缩性，您还学习了如何在Odoo服务器进程前使用Nginx作为反向代理。

这些应该涵盖了运行Odoo服务器，并为用户提供稳定和安全的服务所需的基本知识。

好了本书终于完结，本书仅仅只是入门的odoo开发指南手册，更高级的开发技巧和完整开发实战案例请参阅OSCG开发团队所编写的《Odoo开发手册》