Drupal 8 从入门到精通

www.drupalc.com

Drupal 中文网　编著

目　录

Drupal 8 从入门到精通 1
Drupal 中文网　编著 1
Drupal 8 从入门到精通 1
第 **1** 章 **Drupal 8** 概述 2
1.1 Drupal 基本概念 2
1.2 Drupal 适用领域 4
1.3 Drupal 系统基本术语简介 5
1.4 用户、角色、权限 8
1.5 内容类型 8
1.7 Drupal 是否安全 10
1.8 Drupal 8 新特性 11
第 **2** 章 服务器环境搭建 14
2.1 安装 **Ubuntu 16.04** 14
2.2 安装 **Web** 服务器 **Apache** 15
2.3 安装 **MySQL** 17
2.4 安装 **PHP5** 17
2.5 安装 **phpMyAdmin** 管理 **MySQL** 数据库 19
第 **3** 章 **Drupal 8** 安装和基本配置 21
3.1 Drupal 8 系统需求 22
3.2 常见 **web** 服务器软件 22
3.3 PHP 配置 23
3.4 下载并解压 **Drupal 8** 24
3.5 创建 **Settings.php** 25
3.6 运行 **Drupal 8** 安装脚本 26
3.7 配置 **cron** 任务 31
3.8 配置简洁 **URL** 31
第 4 章 站点配置 33
4.1 设置站点信息 33
4.2 站点缓存和带宽优化 34
4.3 导入、导出站点配置 35
4.4 文本格式与编辑器 36
4.4.1 系统内置的文本格式 37
4.4.2 添加文本格式 37
4.4.3 将 **CSS** 常用类配置到 **CKEditor** 工具栏 38
4.4.4 创建自定义 **CKEditor** 38
4.5 区域与日期设置 40
4.5.1 设置国家和时区 40
4.5.2 日期时间格式设置 41
4.6 搜索设置 42
　　索引进度**(INDEXING PROGRESS)** 44

www.drupalc.com

索引负荷管制(INDEXING THROTTLE) 45
默认索引设置(DEFAULT INDEXING SETTINGS) 45
搜索日志设置(LOGGING) 45
搜索页面设置(SEARCH PAGES) 45
4.7 日志与错误设置 45
4.8 站点维护模式 46
4.9 站点报告 47
第 5 章 用户管理 48
5.1 账户设置 48
5.2 添加-编辑-删除用户 50
5.3 用户角色管理 52
5.4 设置角色权限 52
5.5 封锁用户 IP 53
5.6 用户注册、登录、找回密码 54
5.7 隐藏用户登陆 54
第 6 章 Drupal 8 内容管理 55
6.1 内容管理界面 55
6.2 添加内容 56
6.3 编辑内容 57
6.4 删除内容 58
6.5 内容类型 58
6.5.1 默认的内容类型 58
6.5.2 创建新内容类型 58
6.5.3 管理表单显示与管理显示 59
6.6 评论管理 60
第 7 章 区块管理 61
7.1 Bartik 区域演示 61
7.2 常见区块简介 62
7.3 放置与配置区块 63
7.4 添加谁在线区块 63
7.5 自定义区块管理 64
第 8 章 Drupal 8 语言管理 65
8.1 配置语言 65
8.2 用户界面翻译 67
8.3 内容语言翻译 69
8.4 配置翻译 69
第 9 章 Drupal 8 菜单管理 70
第 10 章 Drupal 8 分类管理 73
第 11 章 视图管理 74
11.1 视图概念 74
11.2 管理视图页面 74
11.3 创建简单的区块视图 75
11.4 创建简单页面视图 76
11.5 向视图添加显示 77

www.drupalc.com

11.6 向视图添加字段......77
11.7 自定义视图字段的输出样式......78
11.8 重写视图字段的输出......78
11.9 在视图中获得字段的自定义名称......79
11.10 向视图添加上下文过滤器......79
11.11 配置视图过滤规则......80
11.12 向视图添加关系......80
11.13 向站点访客显示过滤器......81
11.14 在视图中获得一个过滤器的自定义名称......82
11.15 管理视图的显示设置......82
11.16 配置视图调试模式和缓存......83
第 **12** 章 **Drupal 8** 模块开发......84
12.1 模块开发基本知识......84
12.2 准备一个模块框架......85
12.3 给你的模块命名并创建目录......85
12.4 创建模块信息文件**.info.yml**......86
12.5 添加一个 **composer.json** 文件......88
12.6 Hello World 自定义页面模块......90
12.6.1 添加一个基本控制器......90
12.6.2 添加一个路由文件......91
12.6.3 添加一个菜单链接......92
12.7 向自定义模块添加自定义区块......92
12.7.1 创建自定义区块......93
12.7.2 添加区块配置表单......94
12.7.3 处理区块配置表单......95
12.7.4 在区块显示中使用配置......95
12.7.5 添加一个默认配置......96
12.8 在 **Drupal 8** 模块中包含缺省的配置......97
12.9 创建自定义内容和配置实体......98
12.10 在 **Drupal 8** 中定义和使用配置......108
12.11 创建自定义字段......110
12.11.1 创建自定义字段类型......111
12.11.2 创建自定义字段格式化器......114
12.11.3 创建自定义字段部件......115
12.12 创建自定义页面......119
第 **13** 章 使用配置系统 **API**......121
13.1 配置系统概述......121
13.2 简单的配置 **API**......122
13.3 使用配置表单......126
13.4 Drupal 8 配置的存储......128
13.5 配置覆写系统......129
13.6 配置 **schema/metadata**......133
13.7 配置实体依赖......142
13.8 在 **Drupal 8** 中创建配置实体类型......143

www.drupalc.com

第 14 章 菜单系统 151
14.1 Drupal 7 菜单 API 与 Drupal 8 菜单 API 的异同 151
14.2 提供模块定义的菜单链接 151
14.3 提供模块定义的局部任务 152
14.4 提供模块定义的局部动作 155
14.5 提供模块定义的上下文链接 156
第 15 章 路由系统 160
15.1 Drupal 8 路由系统概述 160
15.2 Drupal 8 路由与控制器引例 161
15.3 Drupal 8 路由系统的基本功能 163
15.4 路由数据结构 164
15.5 路由访问检测 168
15.6 修改或增加已存在的路由 171
15.7 路由参数 172
15.8 路由参数转换 173
15.9 在路由中使用实体参数 176
15.10 存取路由参数中的原始值 179
15.11 实现自定义参数转换 179
15.12 验证路由参数 180
15.13 提供动态路由 180
15.14 路由对象 Route CurrentRouteMatch RouteMatch Url 184
第 16 章 插件开发 186
16.1 插件开发概述 186
16.2 为什么要使用插件 187
16.3 基于注释的插件 188
16.4 视图中基于注释的插件 192
16.5 创建插件管理器 196
16.6 D8 插件发现机制 198
16.7 插件定义 200
16.8 插件上下文环境 201
16.9 插件派生 203
16.10 创建能在主题中定义的插件 205
第 17 章 实体系统 207
17.1 Drupal 8 实体概述 207
17.2 实体类型 207
17.3 实体常用操作 208
17.4 Bundles 212
17.5 配置实体 213
17.6 内容实体 214
17.7 创建自定义内容实体 214
17.8 创建自定义内容类型 218
17.9 在内容类型中附加自定义字段 222
17.10 用 UI 导出自定义字段的代码 227
17.11 实体 API 实现类型化数据 API 228

www.drupalc.com

17.12 定义和使用内容实体字段 231
17.13 实体翻译 API 238
17.14 显示模式、查看模式和表单模式 241
17.15 字段类型、字段控件和字段格式 243
17.16 升级 Code Snippets 模块到 Drupal 8:创建一个自定义字段 251
17.17 使用 computed 字段属性实现动态字段值 257
17.18 处理器 Handlers 265
17.19 使实体可修订 266
17.20 实体注释结构 268
17.21 实体验证 270
第 18 章 表单系统 273
18.1 创建一个表单 273
18.2 创建表单的相关知识 276
18.3 使用 html5 元素 277
18.4 Drupal 表单元素的工作原理 279
18.5 验证表单数据 281
18.6 添加多个表单验证器 283
18.7 处理提交的表单数据 284
18.8 添加多个提交处理器 286
18.9 修改其它表单 287
18.10 修改表单的原理及表单验证 290
第 19 章 渲染系统 292
19.1 渲染数组与缓存 292
19.2 Drupal 怎样使用缓存优化渲染 294
19.3 可冒泡的元数据 295
19.4 自动占位符 296
19.6 Drupal 8 渲染管线 297
19.5 渲染数组 299
第 20 章 缓存系统 302
20.1 Drupal 缓存系统架构 302
20.2 后端缓存 304
20.3 缓存 bin 与缓存配置 306
20.4 访问检测与缓存 308
20.5 缓存 API 309
20.6 缓存上下文 309
20.7 缓存 max-age 314
20.8 缓存 tags 314
20.9 缓存依赖接口 317
20.10 外部缓存 Varnish 319
第 21 章 服务与依赖注入 322
21.1 Drupal 8 的服务与依赖注入机制 322
21.2 修改已有服务 提供动态服务 325
21.3 表单的依赖注入 326
21.4 服务标签 Service Tags 328

www.drupalc.com

21.5 服务的文件结构......330
第 **22** 章 数据库 **API**......332
22.1 数据库 **API** 概述......332
22.2 数据库基本概念......332
22.3 数据库配置......334
22.4 静态查询......337
22.5 使用自定义类获取查询结果......340
22.6 动态查询......341
22.6.1 动态查询介绍......341
22.6.2 条件子句......344
22.6.3 统计行数(COUNT QUERIES)和 DISTINCT......348
22.6.4 表达式 **Expressions**......348
22.6.5 扩展 **Select** 查询......349
22.6.6 查询字段......351
22.6.7 分组查询 **Grouping**......353
22.6.8 数据库连接操作......354
22.6.9 查询结果排序......355
22.6.10 查询修改 **tagging**......356
22.6.11 范围查询......359
22.6.12 表格排序......360
22.7 结果集......360
22.8 插入查询 **INSERT**......362
22.9 更新查询......367
22.10 删除查询......368

www.drupalc.com

Drupal 8 从入门到精通

Drupal 8 从入门到精通是一部全面介绍 Drupal 8 的书籍。Drupal 系统虽然在国外很流行，其功能也非常强大，又有强大的社区支持，人气超高，但可惜的是中国用户非常少，究其原因，可能是其定制性太强，过于灵活，难以掌握，加之中文资料很少。Drupal 7 是 Drupal 系列的一个里程碑，Drupal 8 又是一个革新性版本，无论是界面还是内核都有较大的变化。作者写作本书的目的在于引导国人能认识并理解 Drupal，并通过团队合作开发出具有中国特色的 Drupal 发行版。全书分为三个部份讲述，第一部份为基础部份，主要讲述 Drupal 的安装配置及用 Drupal 建站的一些基础知识；第二部份为为高级部份，主要讲述 Drupal 原理、内核、模块和主题等开发知识；第三部份为实例部份，主要分析当前流行的网站系统特点及 Drupal 解决方案。由于作者精力和水平有限，书中难免有许多错误之处，希望广大读者批评指正。

www.drupalc.com

第 1 章 Drupal 8 概述

Drupal 是一个强大的内容管理系统，也是一个轻量级的应用程序开发框架。本章将介绍 Drupal 系统的基本概念，诸如 Drupal 的适用领域、URL 处理、存储机制、用户管理、安全等等。

1.1 Drupal 基本概念

灵活性与易用性

灵活性与易用性一直是内容管理系统的设计目标。如果一个系统过于简单，它能做的事就不多，相反一个系统过于复杂则难以撑握、难以控制。

Drupal 系统努力调和这两方面的矛盾。它为用户提供了易于使用、易于管理的界面接口，使用户能轻松地建立、管理站点，又能轻易地添加丰富的内容，从这个意义上讲，它称得上一个内容管理系统(CMS)。同时 Drupal 提供了强大的 API 用作二次开发，在 Drupal 官网上有成千上万的模块可供选用，这使 Drupal 易于扩展，拥有极强的灵活性，从这方面讲，它又是一个轻量级的应用开发框架

Drupal 是一个高度模块化的内容管理系统，它的核心集成了常用的模块以完成站点的基本功能。它为用户提供了基本的模块与基本的主题，以方便非技术用户建立站点。它拥有非常活跃的社区，在社区中有数以万计的贡献模块可供建站使用。使用 Drupal 建站就像搭积木一样简单，选用可适的模块，站点创建者可以创建新闻站、在线商店、社交网络、博客站、威客站、分类网站、综合性门户网站等等。只有你想不到的，没有 Drupal 做不到的。

与其它的 CMS 区别

为了更好地说明 Drupal 与其它的 CMS 的区别，考虑这样一个例子。假如你想在站点上发布新文章，并且将最新的五篇文章按发布日期显示在首页上，你还想添加博客功能，并把最新的五篇博客文章显示在首页上。

过去的 CMS，首先你需要安装处理文章的插件，使你发布的文章能显示在首页上并显示摘要。其次你需要安装博客插件以实现博客功能，每一种插件都只是针对某种内容，它们之间相对独立。

如果你想将两种功能融为一体，又该怎么做呢？这可能需要你开发新插件来实现。对于 Drupal 来讲就不一样了，Drupal 具有极强的灵活性，你能很快找出一套解决方案，因为 Drupal 的模块是以标准的方式完成任务，它能很好地将各种自定义功能组合在一起。

www.drupalc.com

当然强大的灵活性带来的是学习与撑握的困难，Drupal 的核心和数以万计的模块是需要花费大量时间学习的，一旦你撑握了它们，你就会立即受益，到那时你发现你再也不想使用其它 CMS。

Drupal 是如何工作的?

在大多数人看来，web 站点不过是许多静态页面的集合。伴随着博客网站、新闻网站、电子商务网站的出现，站点变得越来越复杂，当管理站点时，管理员希望能以树形结构来管理内容。

然而，Drupal 把内容类型当作节点，静态页、博客文章、新闻条目等都以节点的形式存储，也就是说 Drupal 中的内容存储方式是统一的、相同的。菜单、视图、区块可以用来设计站点的导航结构，它实现了存储与显示的分离，XHTML 存储结构化的信息，CSS 负责它们如何显示。在 Drupal 中，节点存储结构化的数据，包括文章、新闻条目等。菜单、分类标签、视图用于组织内容，最终由主题负责将页面内容呈现给访客。

节点—Drupal 灵活性的秘密

我们每天都在谈论结点，因为它是 Drupal 系统的核心，值得我们深入研究。最基本的结点是一组相关信息的集合。当你创建一个博客类型时，你不仅可以定义它的主体，还可以定义标题、内容、作者、日期、分类等信息。这些信息可以通过主题显示给用户。

内容的每一项信息都是以一个节点进行存储，因此它们都以相同的方式被 Drupal 的核心或贡献模块处理。网站创建者可以确定内容显示方式。网站创建者可以创建多套自定义的主题来显示内容，这就像我们平时所说的换肤功能。

评论是 Drupal 系统的又一大亮点，评论不是文章特有的，博文也可以有评论，产品也可以有评论。也就是说 Drupal 的评论系统并不说针对某种特定类型。评论功能的开启与关闭需在内容类型中配置。

www.drupalc.com

团结协作

建立一个大量内容的 web 站点，单靠一个人或少数人是难以实现的，Drupal 增强了用户内容创建，强化了内容之间的链接。Drupal 拥有强大的用户管理系统以及权限系统，站点创建者可以把内容创建工作交给用户，通过设置用户权限规定用户能做什么不能做什么，比如要写一本书，就可以发动一群爱好者共同来写作，设置好各个作者的权限，然后作者们就可以共同创作这本书。

快速开始

Drupal 的灵活性令人难以置信，但它安装极为简单。将安装文件上传至服务器，进行少量的设置，不到 1 小时你的 Drupal 站点就建好了。

接下来你可以为站点指定主题，为站点创建内容。你可以对用户注册与登陆进行设置，打开或关闭用户注册，设置用户验证 Email 等。你可以到站点扩展页面开启或关闭模块以扩展站点功能。打开 book 模块使你创建的内容以书籍大纲的形式显示；开启论坛模块允许用户讨论；创建分类用来组织内容等等。

Drupal 拥有强大的主题系统，允许你为网站创建皮肤，你只需创建模板，写 CSS 或 JS 代码就可以创建出令人惊叹的网站外观。Drupal 产生的 HTML 标签是简洁的，符合标准的，没有冗余，没有乱七八糟的东西。

Drupal 的层次结构

如果想深入理解 Drupal，你需要理解 Drupal 系统中的信息是如何处理的。Drupal 将系统分成五个层:

节点数据层

由于 Drupal 中的数据是以节点来存储的，所以这一层是整个数据的源头，站点的一切输出都来自于这里，任何输入的数据也存储在这里。

模块层

模块是为 Drupal 系统提供功能的程序集，Drupal 内核自带一些基本的模块以实现站点的基本功能，Drupal 的社区中有数以万计的模块可供选用，如 e-commerce 模块可以实现在线商店功能。

区块与菜单

区块与菜单用来组织内容的显示，如热点内容区块可以将点击次数最多的内容显示在页面上，区块功能非常强大，可以配置区块的显示位置，或为某一类用户显示等等。菜单用于网站的导航。

用户权限层

这一层控制用户在站点上能做什么能看到什么。权限通过角色定义和分配。

主题层

主题层控制网站的最终外观，它包含模板引擎、资源库(CSS、JS、图像等)。

1.2 Drupal适用领域

www.drupalc.com

Drupal 是一个强大的、灵活的内容管理系统，它几乎可以建立任何类型的 web 站点，虽然它足够强大，但是我们应该充分发挥它的优势，Drupal 在以下情况更能体现它的优势。

功能定制:如果你觉得其它的 CMS 不能满足你的要求，并且限制太多。那 Drupal 应该是你最佳的选择，它强大的自定义功能以及成千上万的模块总能满足你。

灵活性:Drupal 允许你在任意方向发展，如你的站点开始只是一个博客网站，随着时间的推移，你可能想把它做成资讯站或电子商城，Drupal 完全可以满足你。

站点复杂:复杂的表单、工作流程、多语言站点、站群等。Drupal 能轻易实现这些。Drupal 社区有数以万计的模块可供选用，你只需选择合适的模块将它们组装在一起就可以实现。

自定义内容类型:Drupal 提供自定义内容类型功能，比如你可以创建一个产品内容类型。

另一方面，针于特定的领域，Drupal 可能不是最佳选择，如下面的情况:

个人博客:有许多实现个人博客功能的程序，如 WordPress、Blogger 等，虽然 Drupal 也能实现个人博客功能，但它不如以上程序专业。

Wiki 网站:你应该首先考虑专业的 Wiki 系统如 MediaWiki。你也可以将 Drupal 配置成 Wiki 网站，但你还需要做许多事。

论坛:有很多论坛程序如 Discuz、phpBB、Vanilla 等等，其中 Discuz 在国内使用非常广泛，其功能也非常强大，足可以满足你的需要。如果你不想使用以上程序，而是想自己实现论功能，你也可以尝试使用 Drupal，Drupal 自带论坛功能，但你可能觉得有些简陋，你可以用贡献模块如 Advanced Forum 创建更好的论坛。如果你是一个模块开发者，你也可以自己开发论坛模块。

需要注意的是，当你选用一款 web 软件时，你总是应该把软件的安全和维护放在首位，如果软件没有人维护，这表示此软件永远停步不前，总有一天会被淘汰，也可能存在潜在的安全问题。

1.3 Drupal系统基本术语简介

这一节我将介绍 Drupal 的基本术语，以便于读者能更好的理解 Drupal。如果你读完成觉得不够详细，请阅读 drupal 的官方文档。

节点

www.drupalc.com

一个节点是你网站上的一段内容。节点的类型定义了节点包含的字段。根据不同的节点类型附加不同的字段，这就是内容类型。例如'base page'类型包含标题、主体字段。还有许多内容类型如文章、书页、讨论主题、博客页等。节点这个概念并不清晰，它将逐渐被实体概念所取代。

实体类型

实体类型是包含一组字段的抽象概念。实体用于存储和显示数据，节点内容、评论、分类、用户资料等都是实体，模块可以自定义实体类型。

评论

评论是一种内容类型，它由 Drupal 核心 Comment 模块开启。每一条评论是用户写的一段内容，这段内容被附加到评论的节点。例如论坛中的评论被附加到特定的话题。

分类

Drupal 系统使用 taxonomy 来对内容进行分类。它是由 Drupal 的核心模块 Taxonomy 提供。你可以定义词汇并在词汇下添加术语。每一个词汇能附加到一个或多个内容类型，节点内容使用分类、标签等进行分组。

用户

这里的用户是指访问你站点的真正访问者，它是一个实体类型。一般地用户具有用户名、密码、角色、e-mail 地址等属性。贡献模块也可以自定义其它的用户属性。例如你可以为用户的 Twitter 地址增加一个"Link"。

模块

Drupal 模块是扩展 Drupal 系统功能的软件集合。模块可以分为以下三类:

核心模块:核心模块是包含在 Drupal 的发布版本中，不需要下载其它的组件就可以开启或关闭它们，如 Blog、Book、Poll、Taxonomy 等。

贡献模块:贡献模块是由 Drupal 的模块开发者开发的模块，它们被提交到 Drupal 官网统一管理，并可以通过 Drupal 站点模块管理界面进行在线安装，如 Pathauto、Metatag、Rule 等等。

自定义模块:自定义模块是指你自己开发的模块。这需要你深入理解 Drupal、熟悉 PHP 编程、撑握 Drupal API 等等。模块开发请阅读 章。

区域与区块

www.drupal.com

Drupal 站点的页面由区域构成。核心主题 Bartik 包含 header,footer,sidebars,featured top,featured bottom,main content 等等区域。你可以通过主题定义区域。

区块用来在网站页面的区域中显示内容，它可以是 HTML 代码，也可以是文章列表，或者你定义的其它信息。

菜单

Drupal 使用菜单来导航网站，方便用户访问内容。如页面头部显示网站主导航菜单，站点创建者可以对主导航菜单进行配置。又比如用户菜单用来查看用户资料或退出网站。你可以自己创建菜单并显示它们，也可以创建文章时为它们指定一个菜单等。

主题

Drupal 的主题层与数据层、模块层是相分离的，主题层决定站点的外观，控制图像样式，菜单显示，网站页面布局，网站配色等。主题包含模板文件与资源文件，模板文件将 HTML 标签、CSS 类、内容等组织在一起，CSS 文件控制页面内容如何显示。

视图

视图是一个创建动态页面的强大工具，Drupal 8 已经将视图模块集成到内核中。用户可以使用视图工具创建页面、区块、RSS 等。比如我们想将最新内容显示在网站主页的某一位置上，就可以利用视图创建这一区块，然后在相应区域中开启就行了。

数据库

Drupal 将数据存入数据库中。在数据库中，每一种信息都有相应的数据表，例如节点信息有节点表，每一种字段存储数据在它们自己的表中，这些表由 Drupal 自动创建。评论、用户、角色、权限等也有相应的数据表。一般来说 Drupal 使用 MySQL 数据库，但它也支持其它数据库如 PostgreSQL、SQLite 等。

URL 路径

Drupal 站点的 URL 路径是指站点的基本 URL 后的部份，如 http://example.com/node/1，Drupal 路径是 node/1。

当你访问 Drupal 站点时，Drupal 根据路径对寻找需要返回给访客的信息，Drupal 会检测路由信息或菜单项以找到定义这个路径的模块，然后交由模块响应。正如上例，定义路径的是 node 模块，Drupal 会让节点模块决定如何处理这一路径。

Bootstrap

www.drupalc.com

Bootstrap 是 Drupal 系统的 CPU(中央处理器)，在其它的软件中可能叫事件循环。Drupal 的核心就是这样，它一直等待 URL 需求，然后处理 URL 需求。

权限

Drupal 拥有强大的权限系统，通过配置用户权限规定用户能做什么、不能做什么。权限系统是配置用户角色进行分配，用户角色就相当于用户组，具有相同权限的用户就是一个用户组。模块可以定义权限，以规定哪些用户可以使用这一项功能。

1.4 用户、角色、权限

Drupal 把每一个访问你站点的人都视为一个用户，如果一个访客在你站点上注册了，则他是一个注册用户，被 Drupal 赋于一个唯一的用户 ID，没有注册的叫匿名用户。

用户类型

每一个用户都有一个 ID，ID 为 1 的用户是站点管理员，这个用户只能在安装 Drupal 时创建。这个用户非常特殊，因为它具有站点的一切权限。

在你站点上注册的用户被 Drupal 赋于一个用户 ID。在注册时需提供用户名、密码、E-mail 等信息，以及模块定义的扩展信息。

匿名用户是指访问你站点的访客，他们没有在你站点上注册，这些用户共享一个用户 ID(0)。

权限

站点上的用户通过用户角色分配权限。用色通过站点用户管理进对创建，一般地站点上会创建'站点编辑'角色。你可以为每一种角色分配权限，让 Drupal 知道每种角色在站点上能做什么事，你还需要给每个用户授予角色，以使他们能获得角色的权限。

Drupal 内建了三种角色，分别是匿名用户、注册用户、管理员，并为它们分配了权限。Drupal 的权限系统是非常灵活的，你可以根据站点的需要创建多种角色并为它们分配合适的权限。

www.drupalc.com

1.5 内容类型

单个 web 站点拥有多种内容类型，如新闻条目、文章、投票、博文、产品等等。在 Drupal 中，内容以节点存储，内容类型定义了节点类型的设置，如节点自动发布、开启评论等。

如果你以默认方式安装 Drupal，Drupal 将为你创建"文章"和"静态页"两种内容类型，你可以开启 Drupal 的核心模块或贡献模块，你会发现这些模块定义了其它的内容类型，你还可以自定义内容类型。

Drupal 8 的内容类型

文章(Article)

在 Drupal8 默认安装情况下，它支持文章类型，文章类型一般是指更新频繁需要分类组织的信息比如新闻条目。一般地，文章按照发布时间排列，发布最新的文章显示在最上面，但这可以通过 View 模块修改。

静态页(Basic page)

Drupal 8 默认安装就开启了静态页内容类型。一般地静态页用于将固定内容链接到菜单上，如网站的'关于我们'一般是固定的，我们可以创建一个静态页让'关于我们'指向这一页面。

博客文章(Blog Entry)

博客类型已被 Drupal 8 删除。博客文章常被称为日志，博客模块允许在站点上注册用户，以创建他们自己的日志内容。

书籍页面(Book page)

Drupal 核心 Book 模块提供了书籍页面类型，它允许以书籍大纲的形式组织内容，并允许多用户共同创作内容，任何被授权的用户都可以向某一本书添加子页面。在 Drupal 8 中任何类型都能添加到书籍中。

论坛话题(Forum topic)

论坛话题定义讨论的标题，任何人都可以发布评论以对讨论进行回复，论坛话题通过分类进行组织。分类也就是我们所说的版块。

投票(Poll)

投票是对一组问题的选择式响应，一旦你创建了投票问题，用户就可以针对问题进行选择，投票模块会对投票结果进行处理，如统计每一个问题被选的计数等。

自定义内容类型

www.drupalc.com

你可以在 Menu->Structure->Content Types ->Add content type(admin/structure/types/add)这里创建自定义的内容类型。使用自定义的内容类型组织内容。例如，你可以创建'信息'和'焦点图'类型，而不使用'文章'类型。创建内容类型，你需要为其附加字段，如标题、内容、图像等，你可以管理这些字段的显示方式，或不显示它们。

1.6 Drupal 路径

在 Drupal 中，路径是指站点 URL 后面的查询字符串，举一个例子，一个页面的完整 URL 是 http://example.com/?q=node/7，则路径是 node/7。如果你站点开启了简洁 URL，上例完整的 URL 是 http://example.com/node/7，路径仍然是 node/7，Drupal 8 必须开启简洁 URL。URL 别名能够完全替换访客看到的 URL，我们这里讨论的 URL 并不是指 URL 别名，而是 Drupal 的内部路径，无论是否指定 URL 别名，其内部路径总是不变的。

Drupal 内部路径是非常重要的，因为 Drupal 的管理是依靠内部路径来工作的。举一个例子，当你为一个菜单添加一个链接时，你需要为它指定一个内部路径以告诉 Drupal 这个菜单指向的页面。

下面是一些路径的例子:

node/82
taxonomy/term/3
admin/content/add
user/login
user/2

在 Drupal 中你可以为一个 URL 指定别名，Drupal 8 已集成了 path 模块，其功能是提供 URL 别名管理，你可以对 URL 别名进行编辑、删除等。也可以批量更新 URL 别名，这需要使用贡献模块 pathauto，这个模块允许你设置别名 pattern，它在内容类型表单创建了自动别名复选框，勾选则按 pattern 自动产生别名。

当使用了别名后，可能导致一个页面有多个 URL，搜索引擎可能认为这是重复内容，从而对你的网站进行降权，甚至封杀，因此你需要规划好 URL，不能随意更改，一旦出现重复内容你需要使用重定向模块来处理。

1.7 Drupal 是否安全

www.drupalc.com

Drupal 拥有优秀的安全机制，它对安全问题进行跟踪记录，并组织人员调查、验证、处理安全问题。

Drupal 的安全团队经常与 Drupal 社区一起工作以共同解决 Drupal 安全问题。Drupal 安全团队的成员会对 Drupal 核心或贡献项目代码进行分析，以及时发现潜在的安全问题。

使用 Drupal 的任何人都可以订阅 Drupal 安全邮件，以便于及时查看安全问题及其更新。

开源软件的安全性

开源软件比私用软件更加安全，开源软件的安全问题更容易发现并修复。这是一篇来自 IBM 的文章'开源软件的安全性'总结了开源软件安全相关的问题。另外美国白宫的官方网站使用的是 Drupal 系统。

Drupal 如何处理常见的安全漏洞

Drupal 的 API 和默认配置是安全的。像注入式攻击、跨站点脚本攻击、会话管理、跨站点请求伪造等问题，在 Drupal 中这些问题都有完美的解决方案。详情请阅读 Drupal 的安全报告。

不像其它的商业私有项目，Drupal 作为一个社区驱动的开源项目，没有理由掩盖任何安全漏洞或潜在的安全问题。社区的贡献远比商业影响更重要。安全报告的数量不适于比较，Drupal 有超过 29000 的贡献项目，它们的用户检验着它们的安全问题，一个安全公告可能仅仅是公布一个小问题。详情请阅读安全风险等级。

一个安全公告指出了一个潜在的安全问题，并且这个问题已经被处理。一个安全问题在修复前被人利用，这是极其罕见的。因此你只需保持最新的 Drupal 核心和贡献项目就已经很安全了。

据 Drupal 专业统计，90%以上的安全问题是来自于自定义主题或自定义模块，这些代码并没有提交给 drupal.org 进行管理，没有受到公众的监督。

另外，对服务器的攻击不一定与 Drupal 有关，可能是你服务器的软件不够健状，又或者是服务器的安全设置没做好。

1.8 Drupal 8 新特性

Drupal 8 使用了 Symfony 2 框架，一个轻量级和快速的内核为模块和主题提供了更好的支持。Drupal 8 的内核使用面向对象的方法开发，使其有更好的重用性，更易维护。下面是 Drupal8 的一些新功能:

主题引擎 Twig

Drupal8 使用了新的主题引擎 Twig，它是一个基于 PHP 的、灵活的、快速的、更如安全的模板引擎。使用 Twig 能更容易创建既美观功能又强的 Drupal 站点，它的语法相对比较简单，模板文件中不能使用 PHP 代码，因此它比 PHP template 更加安全。

移动优先的主题

Drupal 8 的内建主题设计为可响应式的，它的管理主题能适应各种屏幕尺寸，单击'Back To Site'按钮可以回到首页。表格能适应任何屏幕，管理工具栏在移动设备中工作得很好。

采用 HTML5

HTML5 将会成为 web 业界的事实标准，Drupal 8 原生系统支持 HTML5，它将会让你访问如 date、e-mail、phone 等字段，甚至为移动设备、掌上设备提供更多的功能和兼容性。

多语言支持

Drupal 8 提供了多语言支持，需要到站点管理界面开启多语言相关的四个模块。管理界面已内建了语言和翻译，用户能轻松添加语言并能对其进行设置，能导入导出翻译文本，能创建基于内容的翻译。能自动检测来自于社区的翻译更新。

配置管理

Drupal 8 在后台集成了配置管理功能，通过这个功能你可以导入导出配置文件，能轻易将本地开发配置传送到服务器上。你也可以使用一个版本控制系统对配置进行跟踪，配置数据是存储在文件中而不是数据库中。

易于写作

Drupal 8 拥有强大的内容创建和编辑功能，它内置了强大 WYSIWYG 编辑器 CKEditor，并且可以到 CKEditor 官网定制编辑器功能。但最受欢迎的还是 Drupal 8 提供的就地编辑功能，在 Drupal 8 中，站点的内容创建者或网站编辑可以在任何页面编辑内容而无需切换到完整的编辑模式。现在更容易创建草稿，并且更加安全。

快速编辑

当我们浏览网站内容时，可能会发现一些错误，需要即时修改。Drupal 8 提供了就地编辑功能能方便地对内容作出修改。如果你登录了 Drupal 站点，你可以直接在前端页面对内容进行修复或补充。

核心集成 Views

www.drupalc.com

Views 是 web 站点的必备部份，没有它就会失去很多美好的东西，站点设计者已经使用这一个贡献模块输出了相册、地图、图表、列表、文章、表格、菜单、区块、报表等许多特性的页面。现在 Views 已经集成到了 Drupal 8 的核心中。首页和一些管理页面都使用了 Views，用户也可以快速地创建页面、区块等，并且毫不费力地修改它们。

更好的可访问性

Drupal 8 对可访问性技术提供了良好的支持，像 WAI-ARIA。在 Drupal 8 中 ARIA 即时通知 API 和 TabManager 显著增强，它为丰富的 internet 应用程序提供控制，如字体大小、颜色调整、jQuery UI 自动完成、模式对话框等，这些功能在 Drupal 8 中能轻而易举地完成。

内建 Web 服务功能

现在，Drupal 8 自身可用作数据源，它以 JSON 或 XML 格式输出内容。你甚至可以从前端向 Drupal 8 发送数据。Drupal 8 实现了超文本应用语言(Hypertext Application Language)即 HAL，使在 Drupal 8 中进行 web 服务开发不再困难。

字段丰富

Drupal 8 核心集成了大量的字段类型，这使得内容管理能力更强大。像实体引用(entiry reference)、链接(link)、日期(date)、电子邮件(e-mail)、电话(telephone)等新类型能帮助内容创建。新字段能附加到任意内容类型，也可以附加到自定义内容类型。

向导

现在描素性文本置于帮助链接下，用户能单出它获得向导；弹出窗口用以解释其工作原理，这对 Drupal 8 新手来说是非常有益的，这一友好功能使得 Drupal 这一 CMS 系统更易理解。

加载提速

Drupal 8 缓存所的有实体，仅当使用 JS 时加载 JS。当查看页面时，缓存的页面不需重新加载。从缓存中加载内容是非常快的，一旦缓存被开启，它将完全自动化。

工业标准

Drupal 8 遵循 PHP7 的最新标准，如 PSR-4、命合空间(namespace)、多重继承(trait)等，集成外部库如 Composer、PHPUnit、Guzzle、Zend Feed Component、Assetic 等。同时，Drupal 8 使用了 Symfony 2 框架，内核使用面向对象代码。

www.drupalc.com

第 2 章 服务器环境搭建

为了更好的学习与研究 Drupal 8，我们先来搭建 Drupal 8 的服务器环境，由于 Ububtu 操作系统有非常友好的图形操作界面，又是基于 Linux 的操作系统，因此选择它作为服务器的操作系统。Web 服务器选择 Apache，数据库选择 MySQL，语言不用说自然是 PHP，搭建服务器软件环境是 LAMP，我这里使用 VM10 虚拟机进行安装，Ubuntu 选择 ubuntu-16.04-desktop-i386。

2.1 安装Ubuntu 16.04

假定你已经安装好了 VM10，如果不会安装请自行百度。下面是安装过程:

1.打开 VM10，出现 VM10 的管理界面，如图 2-1 所示:

2.点击创建新的虚拟机。弹出创建虚拟机向导。

3.勾选典型，然后点击下一步，出现安装客户机操作系统页，勾选安装程序光盘映像文件，点击浏览按钮选择操作系统映像 ubuntu-16.04-desktop-i386.iso，然后点击下一步。

4.设置 Ubuntu 用户名与密码，输入完信息后，点击下一步。

5.命名虚拟机，输入虚拟机的名字如 ubuntu16.04，选择虚拟文件存储位置。然后点击下一步。

6.指定磁盘容量，对于 Ubuntu 系统 20G 就行了，选择将虚拟磁盘文件存储为单个文件，然后点击下一步。

7.显示虚拟机信息，可以自定义硬件配置，这里不作修改，点击完成按钮，开启虚拟机开始安装 Ubuntu 系统。安装过程完全自动化，现在你可以到一边去喝喝茶什么的，一会儿功夫就安装好了。

2.2 安装Web服务器Apache

Apache 是一个十分优秀的 Web 服务器软件，其性能十分优异，并拥有广大的用户，现在我将在 Ubuntu 16.04 下安装 Apache 的最新版本 apache 2.4.23。使用虚拟机启动 Ubuntu 系统并以管理员登录，打开命令行终端，输入以下命令

Sudo apt-get install apache2

可完成 apache2 的安装，但安装的并不是最新版，安装最新版本可以进行手动安装，为安装 apache2 需下载相关的软件包:

apr (apr-1.5.2.tar.gz)　　　http://apr.apache.org
apr-util (apr-util-1.5.1.tar.gz)　http://apr.apache.org
pcre (pcre-8.39.tar.gz)　　　http://sourceforge.net/projects/pcre/files/pcre/
apache (httpd-2.4.23.tar.gz)　http://httpd.apache.org

1. 安装 apr（解决 APR not found）

apr 全称为 Apache Portable Runtime，中文翻译为 Apache 可移植运行库，是 Apache HTTP 服务器的支持库，提供了一组映射到下层操作系统的 API。如果操作系统不支持某个特定的功能，APR 将提供一个模拟的实现。这样程序员使用 APR 编写真正可在不同平台上移植的程序。

安装命令：

```
tar -zxvf apr-1.5.2.tar.gz //解压压缩文件
cd apr-1.5.2   //进入这个目录
./configure    //配置信息
make         //编译
sudo make install   //这里要使用 sudo 获得超级用户权限，不然会安装失败
```

2. 安装 apr-util（解决 APR util not found）

apr-util 是 apr 的一些补充的工具包，其作用应该类似上面谈到的 apr。

安装命令

www.drupalc.com

tar -zxvf apr-util-1.5.1.tar.gz //解压压缩文件
cd apr-util-1.5.1　//进入这个目录
./configure --prefix=/usr/local/apr-util --with-apr=/usr/local/apr　//配置信息,并告诉系统配置的目录将放在哪儿，以及 apr 在哪（apr 在默认情况下放在/usr/local/apr，所以在第一步没有 prefix）
make　　　//编译
sudo make install　//以管理员运行

3.安装安装 pcre（解决 prce not found）

pcre 的全称为 Perl Compatible Regular Expressions，是一个 c 语言库的正则表达式，被包含在很多自由软件项目中，其中就包括 apache，所以这个也是在安装 apache 时必备的。下面这个网址可以对 pcre 了解更多的内容。
http://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions

安装命令：

tar -zxvf pcre-8.32.tar.gz //解压压缩文件
cd pcre-8.32 //进入这个目录
./configure　--prefix=/usr/local/pcre //配置信息
make　　　//编译
sudo make install　//以管理员运行

4.安装 apache

安装命令：

tar -zxvf httpd-2.4.4.tar.gz //解压压缩文件
cd httpd-2.4.4　//进入这个目录
./configure --prefix=/usr/local/apache2 --with-apr=/usr/local/apr
--with-apr-util=/usr/local/apr-util --with-pcre=/usr/local/pcre　--enable-so //配置信息
make　　　//编译
sudo make install　//以管理员运行
cd /usr/local/apache2/conf　//进行 apache 配置目录
sudo chmod 777 httpd.conf　//修改 apache 配置文件权限
gedit httpd.conf　//编辑 apache 配置文件

查找 ServerName 将其改为 ServerName loaclhost:80

cd　/usr/local/apahce2/bin
./apachectl start

在 firefox 中输入 http://localhost/，显示 It works!　表明 apache 已正常安装并提供服务。

www.drupalc.com

2.3 安装MySQL

MySQL 的安装方式多种多样，可以使用 Linux 下通常二进制文件安装，也可以使用 apt-get 工具安装，还可以使用源代码进行编绎安装。现在我们使用 apt-get 进行安装。在安装前先对系统软软包进行更新，按 Ctrl+Alt+T 调出命令行终端，然后输入命令:

sudo apt-get install update

使用 sudo 指令是为了获得管理员超级权限

软件包更新完毕后，就可以输入以命令安装 MySQL 了。

sudo apt-get install mysql-server

如果出现:

E: Could not get lock /var/lib/dpkg/lock - open (11: Resource temporarily unavailable)
E: Unable to lock the administration directory (/var/lib/dpkg/), is another process using it

表示另一个程序正在使用，目前资源锁不可用，解决办法是输入以下命行:

sudo rm /var/cache/apt/archives/lock
sudo rm /var/lib/dpkg/lock

然后再输入上面安装 MySQL 的命令进行安装。系统会读取包列表，建立依赖树，读取状态信息，并显示需要安装哪些软件，然后询问是否继续，输入 y 继续安装。系统会远程下载所需的软件包，这个过程可能会有些慢长。安装过程会要求输入 MySQL 的 root 用户密码，输入密码后，过一会儿就安装完了。

2.4 安装PHP5

安装 PHP 有好几种方式，这里我们采用编译安装，到 PHP 官网下载 PHP 的最新版本，我这里下载的是 PHP-5.6.24。

1.解压 PHP 源码包:

www.drupalc.com

tar zxvf   php-5.6.24.tar.gz
cd php-5.6.24

2.在安装 PHP 之前，需要对 PHP 进行编译配置，如开启特定扩展等。使用./configure –help 命令可以查看所有支持的配置参数，为了能使 apache 能加载 php，需要指出 apxs 的路径。

如下是配置参数命令:

./configure --with-apxs2=/usr/local/apache2/bin/apxs --with-libxml-dir
--with-openssl --with-zlib-dir –with-bz2 --enable-bcmath --enable-calendar
--with-curl --enable-exif --enable-ftp --with-gd --with-mhash --enable-mbstring
--with-mcrypt   --with-pdo-mysql --enable-opcache   --with-pdo-pgsql
--enable-sockets –enable-zip
--with-tidy   --enable-wddx   -with-xmlrpc   --with-pear

以上是配置参数，如果运行结果出现 Thanks php 等字样表明配置成功。配置过程会检测开启的扩展是否已安装到系统或是否存在相应的库文件，如果出现 not found 字样，说明缺少相应支持库，安装即可，这里收集了一份清单。

sudo apt-get install libxml2-dev　　　　//安装 libxml2 支持库
sudo apt-get install libcurl4-gnutls-dev　　//安装 curl 扩展支持库
sudo apt-get install libjpeg-dev　　　　//安装 jpeg 支持库，安装 GD 扩展必备
sudo apt-get install libpng-dev　　　　//安装 png 支持库，安装 GD 扩展必备
sudo apt-get install libmcrypt-dev　　　　//安装 mcrypt 加密扩展支持库
sudo apt-get install libmysql++-dev　　　　//安装 mysql 支持库
sudo apt-get install libtidy-dev　　　　//安装 tidy 支持库

配置好后可以进行编译安装了

make clean　　//确保编译连接正确
make　　//编译
make install　　//编译并安装

安装过程会显示安装的目录及文件及其它一些信息。

3.配置 php.ini

cp php.ini-devlopment   /usr/local/lib/php.ini

将 php 开发配置文件复制为加载的 php 配置文件。

www.drupalc.com

可以编辑 php.ini 来设置 PHP 运行时的选项。如果想把配置文件存放到其它位置，需要在配置编译时加上—with-config-file-path=/path 选项。PHP 源码包中已自带几个 php.ini 文件，其后缀名已表明了其用途，如 php.ini-developmonet 用于开发，php.ini-production 用于生产环境。如果选择 php.ini-production 文件作为配置文件，你最好在本地机器上进行充分地测试。

4.编辑 apache2 的配置文件 httpd.conf 使其加载 PHP 模块。这需要使用到 Apache2 的 LoadModule 指令，此指令用于动态加载模块。以上的 make install 命令已经作了一些事，但请还是自行检查一下。

```
LoadModule php5_module modules/libphp5.so
```

5.通知 Apache 将哪些扩展名解析为 PHP，例如将扩展名.php 解析为 PHP。出于安全考虑，例如避免将 exploit.php.jpg 文件解析为 PHP 执行，我们不再使用 Apache 的 AddType 指令来设置。而是使用文件匹配，如下面例子:

```
<FilesMatch \.php$>
   SetHandler application/x-httpd-php
</FilesMatch>
```

或者你想将.php .php2 .php3 .php4 .php5 .php6 以及.phtml 文件都当作 php 执行，我们需要像下面这样设置。

```
<FilesMatch “\.ph(p[2-6]?|tml)$”>
  SetHandler application/x-httpd-php
</FilesMatch>
```

查找 DirectoryIndex，添加 index.php 作为默认首页。

```
DirectoryIndex index.php index.html
```

6.重启 Apache 服务

```
/usr/local/apache2/bin/apachectl start
```

或

```
service httpd restart
```

以上便把 PHP 作为 Apache 的 SAPI 模块了，Apache 和 PHP 都有许多编译配置选项，可以在相在相应源码目录中使用./configure –help 查看。

2.5 安装phpMyAdmin管理MySQL数据库

www.drupalc.com

PhpMyAdmin 是一个基于 Web 管理 MySQL 数据库的管理工具，其自身使用 PHP 写成，执行效率很高，安全性好，简便易用。现在我们来安装 phpMyAdmin。

1.到 phpMyAdmin 官网(http://www.phpmyadmin.net)下载 phpMyAdmin 的最新版本，目前最新版为 4.6.4 版，包括英文版和所有语言版，我们需要下载所有语言本，下载 phpMyAdmin-4.6.4-all-languages.tar.gz 到本地机器。

2.将下载压缩包复制到 web 根目录下，使用 tar 命令进行解压。

tar –xzvf phpMyAdmin-4.6.4-all-languages.tar.gz

3.重命名 phpMyAdmin

mv phpMyAdmin-4.6.4-all-languages   phpMyAdmin   //将其进行重命名
rm –f   phpMyAdmin-4.6.4-all-languages.tar.gz   //删除压缩包

4.配置安装，可以使用两种方式对 phpMyAdmin 进行配置。以往需要手工编辑 config.inc.php 配置文件。现在可以使用安装脚本创建配置文件。创建一个 config.inc.php 仍然是一种快速度配置方式并可以设置一些高级特性。

手工创建配置文件

使用文本编辑器创建 config.inc.php，如使用 vi config.inc.php。当然你可以将 phpMyAdmin 目录下的 config.sample.inc.php 复制为 config.inc.php   cp config.sample.inc.php   config.inc.php   这样你获得一个最小化的配置文件)。phpMyAdmin 首先加载 libraries/config.default.php，然后加载 phpMyAdmin 目录下的 config.inc.php 进行覆盖变量的默认值。如果觉得默认值已足够，就不需要使用 config.inc.php。下面是官方文档中的一个例子。

```
<?php
$cfg['blowfish_secret'] = 'ba17c1ec07d65003';  // use here a value of your
choice

$i=0;
$i++;
$cfg['Servers'][$i]['auth_type']      = 'cookie';

?>
```

如果你不想每次都输入 MySQL 数据库用户名和密码，可以将其加入。

```
<?php

$i=0;
$i++;
```

www.drupalc.com

```
$cfg['Servers'][$i]['user']            = 'root';
$cfg['Servers'][$i]['password']        = 'cbb74bc'; // use here your password
$cfg['Servers'][$i]['auth_type']       = 'config';
```

使用安装脚本配置

如果不想自己创建配置文件，可以使用 phpMyAdmin 的安装脚本生成配置文件，为此需要创建 config 目录并设置读写权限。

```
cd phpMyAdmin
mkdir config    //创建配置文件临时目录
chmod o+rw config    //给配置目当添加读写权限
```

如果已有一个配置文件，可优先复制它

```
cp config.inc.php config/       //将当前配置文件复制到配置目录下以例编辑
chmod o+w config/config.inc.pohp    //给配置文件设置写权限
```

在浏览器器地址栏输入带有/setup 后缀地地址，如 http://ipaddr/phpMyAdmin/setup，其中/ipaddr 是服务器地址。如果已有配置文件，则可以在安装面板中点击"Load"按钮进行加载。注意所有的修改必须要点击"Save"按钮进行保存以生效。安装脚本会在 config 目录下生成 config.inc.php 文件或修改已有的 config.inc.php 文件，但如果这个目录权限不对的话，可能会得到一个"Cannot load or Save configuration"的错误。

一旦生成了这个配置文件，需要把它复制到 phpMyAdmin 目录下，并取消它的写权限，这是出于安全考虑。

```
mv config/config.inc.php    //将文件移动到 phpMyAdmin
chmod o-rw config.inc.php    //取消其它用户的读写权限
rm –rf config    //删除配置目录
```

至此，phpMyAdmin 安装完成。

第 3 章 Drupal 8 安装和基本配置

www.drupalc.com

上一章我们已经讲述了服务器的搭建，本章我们可以开始安装 Drupal 8 了，在安装之前，先要了解一下 Drupal 8 的系统需求。

3.1 Drupal 8 系统需求

在安装 Drupal 8 之前，我们先来看看 Drupal 8 的基本需求，大多数 web 服务商都能满足 Drupal 8 的需求。

空间需求

Drupal 8 的压缩包大概 20MB，解压后大概 100M，考虑到建站还会用到大量贡献模块，以及数据库占用的空间，网站内容占用的空间等，建议使用 500M 以上的空间。

Web 服务器

大多数支持 PHP 的 web 服务器都可以，如 Apache、Nginx、IIS 等。

数据库

Drupal 8 支持 MySQL、MariaDB、Percona Server、PostgreSQL、SQLite 等数据库软件。MySQL 需 5.5.3 或更高版本，MariaDB 5.5.20 或更高版本，Percona Server 5.5.8 或更高，PostgreSQL 9.1.2 或更高版本，SQLite 3.7.11 或更高。MySQL、MariaDB、Percona Server 需带有 PDO 并使用 InnoDB 存储引擎。PostgreSQL 需带有 PDO。

其它的如 Microsoft SQL Server 或 Oracle 等数据库需要安装相应的驱动，这些驱动可以在 Drupal 官网的上找到。

PHP

Drupal 8 要求 PHP 5.5.9 或更高版本。

3.2 常见web服务器软件

Apache

Apache 是一个使用很广泛的 web 服务器，Drupal 8 在 Apache 2.x 上工作得很好。有许多 Drupal 站点都部署在 Apache 服务器上，因此在 Apache 服务器上性能测试将超过其它的 web 服务器软件。

www.drupalc.com

Apache 服务器允许 Drupal 启用简洁 URL，这需要 Apache 配置 mod_rewrite 模块，Drupal 8 默认开启简洁 URL，并且不能关闭，因此要运行 Drupal 8 必须配置好 mod_rewrite。

Apache 虚拟主机必须配置指令 AllowOverride All 以允许使用 Drupal 的.htaccess 文件覆盖服务器配置。

Nginx

Nginx 是一个高并发、高性能的 web 服务器软件，它使用较少的内存。Drupal 可以工作在 Nginx 1.8.x 及以上。Nginx 是 Apache 的一个很好替代品，因此 Drupal 已在 Nginx 上作了充分的测试。Nginx 支持简洁 URL，需要安装 ngx_http_rewrite_module。

Hiawatha

Hiawatha 是一个非常安全的服务器。它的目标是易用性和轻量级。据相关研究表明，Hiawatha 有更好的抗攻击能力。Hiawatha 完全兼容 Drupal，它支持 URL 重写。

Microsoft IIS

Microsoft IIS 是微软公司提供的 web 服务器软件，Drupal 核心能工作在 IIS5、IIS6、IIS7 上。为了使用 URL 重写，可能需要安装第三方模块。IIS7 可以使用微软的 URL 重写模块。

www.drupalc.com

3.3 PHP配置

本节详细讲述 Drupal 8 核心系统对 PHP 的需求，其它的贡献模块可能有一些特殊需求，具体请参考模块文档。强烈建议开启 Drupal 8 系统缓存功能，以提升 Drupal 8 站点性能。

文件和目录权限

Drupal 和 PHP 应该能够读和写/sites/default/files 目录。这个目录用于存放缓存文件(压缩后的 CSS 和 JS 文件)和通过 Drupal 界面上传的文件。精确的权限依赖 PHP 的安装配置。一般来说，所有者能读、写、执行，不推荐设为(777)这可能存在安全风险。

PHP 设置

为了使 Drupal 更好地工作，需对 PHP.ini 作一些设置:

session.cache_limiter = nocache
session.auto_start = 0
expose_php = off
allow_url_fopen = off
magic_quotes_gpc = off
register_globals = off
display_errors = off

内存需求

Drupal 8 核心需求内存 64MB。Drupal 8 大量采用了"延迟加载"数据和代码，每页所占用的内存差异较大，管理页面之间的调用较多，差异可能更大，你应该为每种类型的页面和视图查看内存使用情况。

如果 PHP 配置不满足 Drupal 8 的需求，Drupal 8 将会在页面上显示警告。就算满足了 Drupal 8 的安装条件，因为 drupal 8 站点可能还用了较多的贡献模块，可能会使用更多的内存。建议 PHP 内存设为 128MB 到 256MB。

扩展

Drupal 8 需要 PHP 启用必要的模块扩展，如 PDO、XML、GD2、Open SSL、JSON、cURL、APC、ssh、Twig C 等。可通过 Drupal 8 状态报告查看 PHP 环境。

www.drupalc.com

3.4 下载并解压Drupal 8

在安装 Drupal 8 之前需到 Drupal 官网下载 Drupal 8 最新稳定版，你可以使用 Drush、Drupal Console、命令行、FTP、Git、composer 等方式下载。

Drush

Drush 是一个管理和维护 Drupal 站点的工具。它提供了很方便的下载 Drupal 的方式：pm-download，别名为 dl。

Drush dl drupal

这个命令将会下载最新的推荐版本到当前目录。至于如何下载 Drupal 指定版本请查看 drush help dl。

Drupal Console

Drupal Console 是 Drupal 的新的命令行界面(CLI)。它提供了极为方便的方式下载 Drupal:site:new:

Drupal site:new mydrupalsite 8.1.5

Drupal site:new mydrupalsite

这个命令将下载 Drupal 指定版本到当前目录。

命令行方式

使用 SSH 登陆到服务器并进入到你 Drupal 站点的目录，在*nix 系统中，目录大概是/var/www/html，使用 wget 或 curl 命令进行下载:

Wget http://ftp.drupal.org/files/projects/drupal-8.15.tar.gz

或者

Curl –o http://ftp.drupal.org/files/projects/drupal-8.15.tar.gz

解压 drupal 8

tar –xvzf   drupal-8.15.tar.gz

rm drupal-8.15.tar.gz

接下来将解压后的内容移动到站点根目录

mv drupal-8.15

在 windows 机器上操作十分简单，到 Drupal.org 官网下载 Drupal 最新版，解压，剪切到 web 根目录下。

www.drupalc.com

3.5 创建Settings.php

为了使 Drupal 8 能正常运行，你需要配置数据库，数据库信息存放在 settings.php 文件中。在你下载并解压 Drupal 8 后，它并不会产生 settings.php 文件，而是生成一个 default.settings.php，当你安装 Drupal 8 时，它会复制 default.settings.php 并重命名为 settings.php。但有极少数情况需要你手工创建 settings.php。

Drupal 8 的 settings.php 文件存放在 sites/default 目录下，在该目录下还有一个名为 default.services.yml 的文件，它和 default.settings.php 相似，default.servicesm.yml 也必须重命名为 services.yml 以便工作。这个文件的作用是用来覆写 Drupal 核心中的 services.yml 文件的，一般来说，你不需要覆写它，但在开发环境可能需要重新设置一些变量，因此需要创建它。

自动创建 settings.php

Drupal 8 的安装脚本会在运行时自动创建和处理 settings.php 文件，当处理完后会对其重新设置权限以增加站点的安全性。有些主机的服务器配置可能导致创建 settings.php 文件失败，在这种情况下，你需要手工创建该文件并对其进行一些设置，当你创建它后，drupal 安装脚本会自动地处理。

手工创建 settings.php

Drupal 8 自动一个 default.setting.php 文件对系统进行了简单的配置，在运行安装脚本以前，你需要复制 defalt.setting.php 并重命名为 settings.php 文件，并设置可读写权限，在安装完后，你应该设置其权限为只读。

创建 files 目录以及权限设置

安装脚本会为你创建 sites/default/files 目录，如果创建失败，你可以手工创建

mdir sites/default/files

修改该目录的权限为 755

cmod 755 sites/default/files

权限设为 777 是有一定的安全风险。

安装脚本会自动设置权限，sites/default 目录权限设为 555，settings.php 文件权限设为 444，如果失败，请手工设置它们的权限:

cmod　555 sites/default
cmod　444 sites/default/settings.php

www.drupalc.com

3.6 运行Drupal 8 安装脚本

现在我们可以运行安装脚本来安装 Drupal 8，在浏览器地址栏输入 Drupal 8 安装脚本网址，假如你的 Drupal 文件在本地机器 web 根目录 drupal 下，则这个网址为 http://localhost/drupal，如果你在 web 服务器上安装，则这个网址是 http://域名/。输入网址回车后，你会看到如下选择语言画面。

1.选择语言

一般来说，选择英语(English)进行安装，如果选择其它语言，语言包会自动从 Drupal 翻译服务器上下载，可能有点慢。如下图：

选择好语言后，点击 Save and continue 进入下一步。

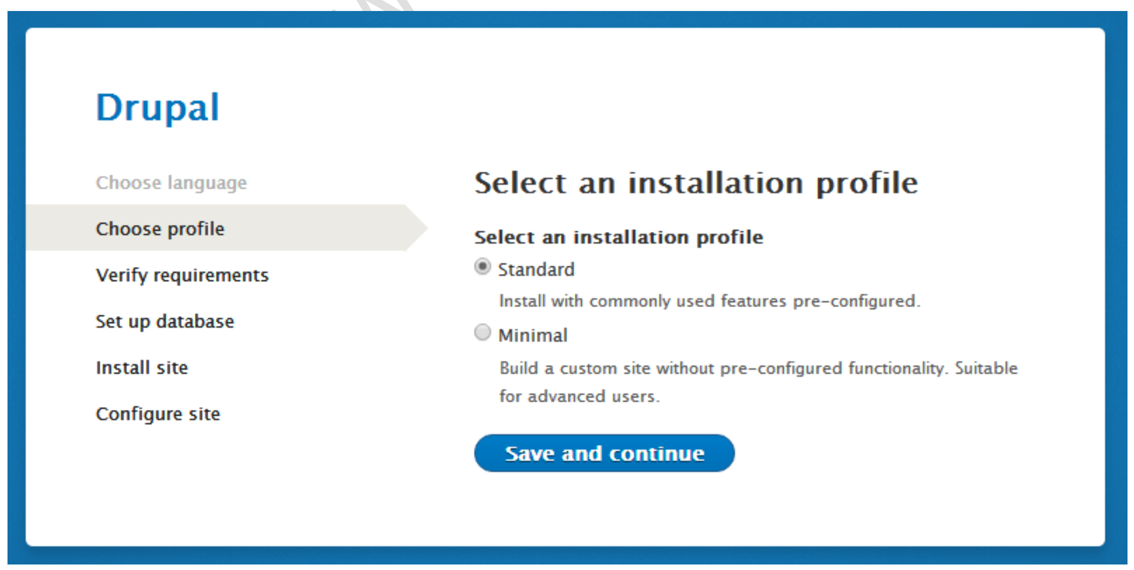
2.选择安装方式

一般来说选择"standard"安装方式即可，标准安装方式已经创建了一些内容类型如'article'、'page'等，标准安装方式为你启用了一些有用的模块。

最小化安装"minimal"推荐有经验的用户选择，可以创建自定义的内容类型，以及相关的发布设置，最小化安装方式仅仅开启了 Block、Database logging、Update status 等少数模块。

如果你下载了一个分发，分发将会显示在这里，你可以选择分发进行安装。

www.drupalc.com



选择好后点击"Save and continue"

3.检测环境

这一步检测安装环境，检测出的任何错误信息都会显示在这一页，按照提示信息修改错误，改好继续安装。

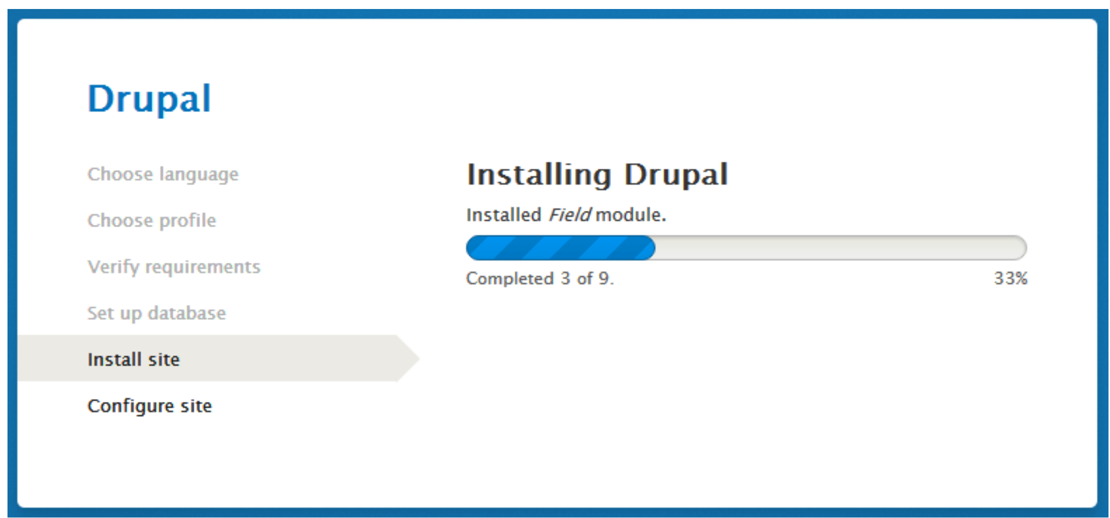
4.设置数据库

这一步选择数据库，一般来说选择 MySQL 数据库。然后输入数据库名、数据库用户名、数据库密码，这些你可以从主机商那里取得。你可以在高级设置中改变数据库主机地址，端口号，一般来说不需要修改数据库主机信息。如果你在单个数据库中创建了多个 Drupal 实例，你需要为每一个实例指定一个表前缀。

设置好后，点 Save and continue 继续安装。

5.安装站点

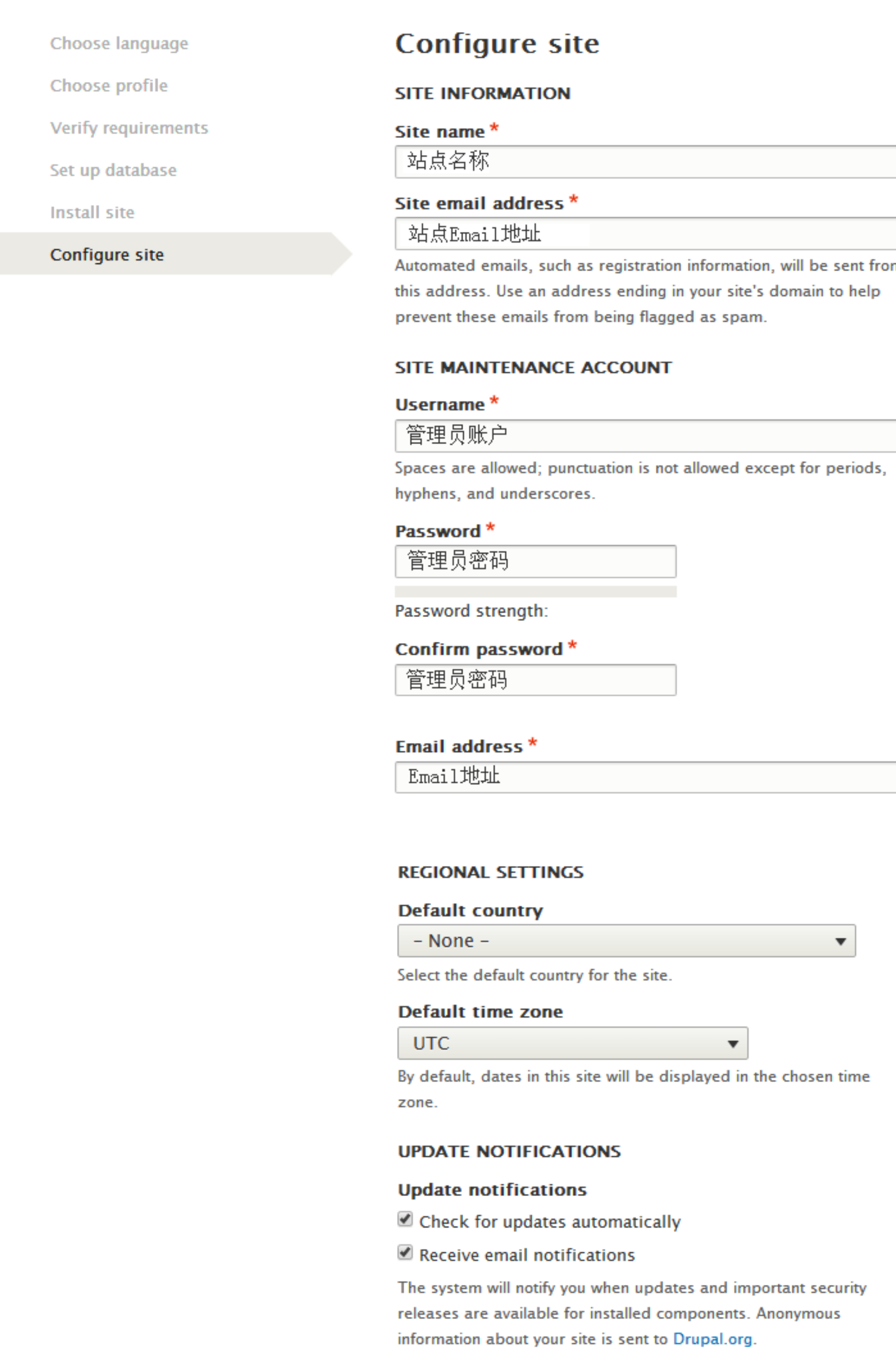
这一步安装站点，安装过程会显示一个进程条以指示安装进程，如果没有出现错误，则自动进入下一步。



6.配置站点

这一步将对站点进行基本配置，包括设置站点名称、站点 E-mail 地址、管理员账户、国家、时区、自动检测更新等。其中管理员账户特别重要，这是一个十分特殊的用户，它拥有 Drupal 站点一切权限，因此在设置时尽量复杂一些以提高站点安全性。

www.drupalc.com



设置好后点 Save and continue 完成完装。

3.7 配置cron任务

在安装完 Drupal 8 后，就可以配置 cron 任务，配置 cron 任务是十分重要的，它自动完成搜索结果索引，检测 Drupal 更新，删除临时文件等。

con 是什么?

Cron 即计划任务是间隔一定时间执行约定命令的守护进程，这些命令称为"cron 任务"。Cron 可以在 Unix、Linux、Mac 机器上执行。Windows 服务器采用 Scheduled Task 完成计划任务。计划任务靠时间触发。

在 Drupal 中真正执行计划任务的是访问 cron.php 文件，这个文件的地址可以在站点管理>报告>状态报告页面查看。

启用 cron

在 Drupal 8 中你可以开启 cron，到管理>配置>系统>cron(admin/config/system/cron)页面设置 cron。默认执行 cron 的时间间隔是 3 小时，意思是说 3 小时后当有人访问你站点时，将触发 cron 任务。在低流量的站点上开启 cron 是不错的，如果访问量比较大，出于性能考虑，你最好禁用自动 cron 任务。

关闭 cron

出于性能考虑，或者你想以其它方式运行 cron 任务(在站点状态报告页面有一个手工执行 cron 的链接)，你可以关闭自动 cron 任务。到管理>配置>系统>cron(admin/config/system/cron)页面，单击 Run cron every 下接框，将其设为 Never(从不)。另外你可以设置'cron_safe_threshold'变量的值为 0 来关闭。如下:

drush –y vset cron_safe_threshold 0

另一种关闭 cron 的方式是修改 settings.php 文件:

$conf['cron_safe_threshold'] = 0;

注意这将设置 admin/config/system/cron 为 Never，并且管理员无法从管理界面更改这一设置。

3.8 配置简洁URL

简洁 URL 是服务器软件提供的 URL 重写规则，它的功能是把用户浏览器中的 URL 转化为服务器能处理的 URL，在大多数靠数据库驱动的网站，其内部处理的 URL 带有查询字符串，这对用户和搜索引擎来说都是不友好的。为了使用更友好的 URL 地址，大多 CMS 系统都提供了 URL 重写功能，当然这还需服务器软件支持。

大多数主机商都采用 Apache 软件作为服务器软件，Apache 软件有一个模块 mod_rewrite，它的功能是支持 URL 重写。Drupal 8 系统默认是开启简洁 URL 的，并且不能被关闭，这意为着你的服务器若不支持 URL 重写，则无法运行 Drupal 8。Drupal 8 已将 URL 重写规则写入了.htaccess 文件中，该文件的作用是覆盖 Apache 服务器的设置。具体的重写规则请查看该文件或到 drupal 官网查询。

www.drupalc.com

第 4 章 站点配置

上一章，我介绍了 Drupal 8 的安装过程以及注意事项。这一章我将介绍 Drupal 8 站点的基本配置，主要包括站点信息、带宽优化、导入、导出配置、编辑器、时区等内容。

4.1 设置站点信息

在安装 Drupal 8 的最后一步，是对站点进行配置，包括站点的基本信息，这个站点信息可以通过管理界面进行修改和设置。

具体步骤如下:

1.以管理员身份登录你的网站。

2.点击管理工具栏的配置或在浏览器地址栏输入 http://域名/admin/config，如果你开启了语言功能如简体中文，这个 url 可能可能是 http://域名/zh-hans/admin/config，当然简体中文(zh-hans)也可以修改，请参考 Drupal 8 语言功能。

3.点击"站点信息"链接，进入站点信息设置页面。

4.在这一页你可以设置站点名称、站点口号、站点电子邮件、默认首页 URL、错误页面等。设置好后点击保存。

站点名称、站点口号会显示在你网站 Logo 旁边，一般来讲我们并不想让它们显示出来，这可以通过主题设置进行关闭，或者在区块中关闭。站点电子邮件用于向注册的用户发送电子邮件或密码修改等。

默认首页为/node，这个一般不需要修改。留空为用户登陆页面，可设为已存在的路径，如站点有"关于我们"/about-us，可以将其设为默认首页，如果设置了一个错误路径，系统会提示路径无效或无法访问。

设置默认的 403（拒绝访问）页面，如果留空不设置，Drupal 8 会采用默认的 403（拒绝访问）页面。你可能会觉得并不友好，你可以创建一个基本页面，然后对其进行主题化，以达到你想要的效果。

设置默认的 404 错误（页面未找到）页面，留空不设置，Drupal 8 会采用默认的 404 错误（页面未找到）页面。你可能会觉得并不友好，你可以创建一个基本页面，然后对其进行主题化，以达到你想要的效果。

www.drupalr.com

4.2 站点缓存和带宽优化

在 Drupal 8 系统配置页面，有一项是对系统性能进行配置，其 URL 为 http://域名/admin/config/development/performance，在浏览器地址栏输入 URL 回车后，会显示 Drupal 8 的系统性能配置页面，在这一页你可对缓存和带宽优化进行配置。

开启 Drupal 8 的缓存有利于提高站点性能，提高站点的响应速度，点击页面缓存生存期下方的列表框如图 4-3，选择一个间隔时间就开启了页面缓存，页面缓存将数据存放于数据库的缓存表中，以便请求时从缓存中直接提取数据，从而加快页面响应，缓存也可以存放在文件中，在 Drupal 社区中能找到相关的贡献模块。

Performance ☆

Home » Administration » Configuration » Development

▼ CLEAR CACHE

Clear all caches

▼ CACHING

Note: Drupal provides an internal page cache module that is recommended for small

Page cache maximum age

<no caching> ▼

The maximum time a page can be cached by browsers and proxies. This is used as the value for ma

▼ BANDWIDTH OPTIMIZATION

External resources can be optimized automatically, which can reduce both the size

☑ Aggregate CSS files

☑ Aggregate JavaScript files

Save configuration

出于某些原因，你可能想关闭缓存，如你正在进行主题开发。点击缓存生存期下方的列表框，选择 从不(Never) 就关闭了页面缓存。如果你在进行主题开发，即使关闭了页面缓存，你所做的修改可能还是无法立即显现，如果发生这种情况，请点击清除所有缓存按钮，以清空系统所有的缓存，如果还不行，则将 Drupal 8 配置为开发模式，详情请阅读 Drupal 主题开发。

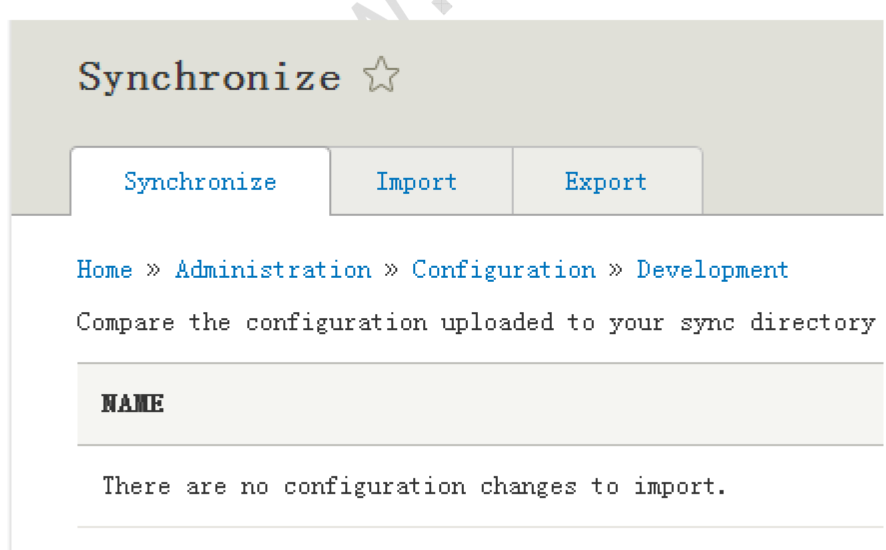
Drupal 8 支持强大的主题功能，其 CSS 以组件方式进行组织，因此一个主题中包括许多 CSS 文件，在没有开启 CSS 合并的情况下查看页面源代码，你会发现页面包含了许多 CSS 文件如下图所示，文件太多不利于缓存和优化，Drupal 8 提供了带宽优化功能对 CSS、JS 文件进行压缩，开启方法很简单，勾选合并 CSS 文件和合并 JavaScript 文件复选框就行了，合并后再查看网页源代码你会发现仅仅只有 3 到 4 个 CSS 文件了。JS 文件也是一样，合并之前在页脚你会发现很多 JS 文件，合并后就少许多了。Drupal 8 对 JS 的加载是使用就加载，不是每一页都进行加载，这也是出于性能考虑，与主题相关的更多内容请阅读主题开发的相关章节。

注意在开启了 CDN 的服务器上开启合并 CSS 文件可能会导致一些问题，如果出现这种情况，请关闭 CDN 功能。

4.3 导入、导出站点配置

Drupal 8 为了使站点配置管理更加灵活，提供了 Configuration Manager 模块，这个模块的功能是允许管理员修改配置，包括导入导出配置，这有利于同步站点开发和线上站点。

现在我们一起来看看 Drupal8 的配置管理界面，点击管理->配置->配置管理或输入 admin/config/development/configuration(域名部份省略)，来到系统同步页面，如图 4-2 所示。在这个页面会显示你对 Drupal8 配置所作的任何修改，由于我们没有作出任何修改，这里没有列出。



让我们先来看看 Drupal 8 的配置是什么样子的，点击单个导入导出选项卡，在随后的页面中点击导出选项卡，来到导出单个配置页面如下图，随便选择一个配置进行导出。例如类型为菜单，配置名称为管理，这时我们会看到下面的文本区

域中显示了配置代码，这个文本区域下面有这个配置的文件名，这段代码是用 YMAL 语言写的。

在 Drupal 8 中大多数配置信息都使用 YMAL 语言描述，如果你对 YMAL 语言不熟，请阅读本站中的 YMAL 中文教程。将这段代码进行修改，然后再导入，看看会发生什么。

点击导入选项卡，来到导入单个配置页面，上面的例子是导出菜单，因此我们需要选择菜单作为导入类型，将修改后的代码粘贴到下方的文本区域中，点击导入，系统会提示你这个指令不能撤销，继续点确定，看到配置已保存，表示已导入成功。到 管理->->结构->菜单 页面进行查看发现菜单标题已更改。

Drupal 8 还提供了完全导入/导出配置功能，在这一页你需要提供需导入的配置文件，也可以将系统的所有配置导出。

4.4 文本格式与编辑器

Drupal 8 提供了强大的内容创建与编辑功能，并支持多用户创建内容，同时加强了安全风险防控。Drupal 8 已内置了功能强大又十分好用的 CKEditor 编辑器，有此编辑器就实现了内容的所见即所得功能，方便用户创建与编辑内容。为了加强对安全的防控，Drupal 8 提供了文本过滤模块，这个模块会对用户的输入进行过滤以剔除有害的文本，它也允许站点管理员对每种角色的用户能使用的 HTML 标签、能使用的编辑器功能进行设置。

点击 Manage->configuration->Text formats and editors 或输入 admin/config/content/formats 导航到内容写作页面，如下图 4-3 所示。这一页列出了系统内置的文本格式和管理员创建的文本格式。每种文本格式可以配置编辑器和有权使用的角色，它们显示的顺序可以通过拖移来改变，也可以设置各自的权重，权重较大的排在下面，如果一个角色具有多种文本格式，这些格式会按各自的权重排列，在改变文本格式顺序后别忘了点击下面的"保存变更"按钮。

Text formats and editors ☆

Home » Administration » Configuration » Content authoring

Text formats define how text is filtered for output and how HTML tags and other text is displayed, replaced,

Text formats are presented on content editing pages in the order defined on this page. The first format avai

+Add text format

NAME	TEXT EDITOR	ROLES
Basic HTML	CKEditor	Authenticated user, Administ
Restricted HTML	—	Anonymous user, Administrato
Full HTML	CKEditor	Administrator
Plain text	—	*This format is shown when*

Save changes

www.drupalc.c

4.4.1 系统内置的文本格式

Drupal 8 安装时选择标准安装方式，Drupal 8 会为站点创建四种基本的文本格式，如上节图 4-3 所示。

- Basic HTML:基本的 HTML，默认使用 CKEditor，注册的用户、管理员可以使用此种格式，这种格式的 CKEditor 将提供一些基本的功能。
- Restricted HTML:受限的 HTML，没有配置编辑器，管理员、匿名用户可使用，这种 HTML 不可使用编辑器，所输入的 HTML 的标签也是受到限制的，输入未经允许的 HTML 标签将会被过滤。
- Full HTML:完全的 HTML，配置 CKEditor，管理员可用，这种格式可以使用所有的 HTML 标签，但有些标签请谨慎使用，可能会造成安全风险。这种格式一定不能配置给普通用户使用。
- Plain text:纯文本，不使用任何编辑，任何人可以使用，不能使用任何 HTML 标签，这是最安全的方式。

前三种基本的文本格式是可编辑和禁用的，纯文本格式不能禁用，但这个操作不可逆且所有使用这种文本格式的内容不再显示，请谨慎操作。

4.4.2 添加文本格式

一般的网站都有内容编辑用户，现在我们为这类用户创建一种文本格式"内容编辑"。

点击列表上面的添加文本格式按钮来到添加文本格式页面，在这一页对本文格式进行设置。在名称文本框中输入"内容编辑"，角色栏中勾选"内容编辑"角色，如果没有这种角色，可以到角色管理页面进行创建，这里假定已创建好。

因内容编辑用户要使用到编辑器创建内容，因此选择一个编辑器，因为系统中就只有一个 CKEditor，选择它。当然可以安装多个编辑器，详情请阅读相关章节。选择编辑器后，会在下面自动出现工具栏配置，你可以使用拖移的方式在启用的工具栏上添加功能。可以在工具栏上创建新的组，当启用某些插件后，在下方还会出现插件配置菜单。

接下来是设置内容过滤功能，出于安全的考虑，你需要添加过滤器，勾选过滤器以开启它，有的过滤器会出现设置表单，进行相应设置即可。过滤器也可以通过模块来添加，具体请阅读有关章节。过滤器下方会出现已开启的过滤器，过滤器的执行顺序可以通过拖移或修改权重来修改。

配置好后，点击保存配置。这时我们发现内容编辑格式已经出现在列表中，这表示具用该角色的用户能使用这种文本格式了。

4.4.3 将CSS常用类配置到CKEditor工具栏

选择你要编辑的文本格式并点击编辑，来到文本格式的配置页面，这和添加文本格式是同一个页面，只是这个页面上的表单已进行了设置。将 CKEditor 可用的按钮 样式 拖动到工具栏上，此时下方会出现 CKEditor 插件设置"未配置样式"表单，在右边的文本区域中输入 CSS 类，其格式为 CSS 类|显示名，每行输入一个。

现在我们需要在前端主题和管理主题中创建刚才输入的 CSS 类，定位到你前端主题和管理主题目录，打开基本的 CSS 文件，将 CSS 类代码写入其中。这时创建文章，你会发现你可以在编辑器中使用刚才定义的类，且它能在编辑器中正常显示，发布文章后，也能和在编辑器中显示一样。

4.4.4 创建自定义CKEditor

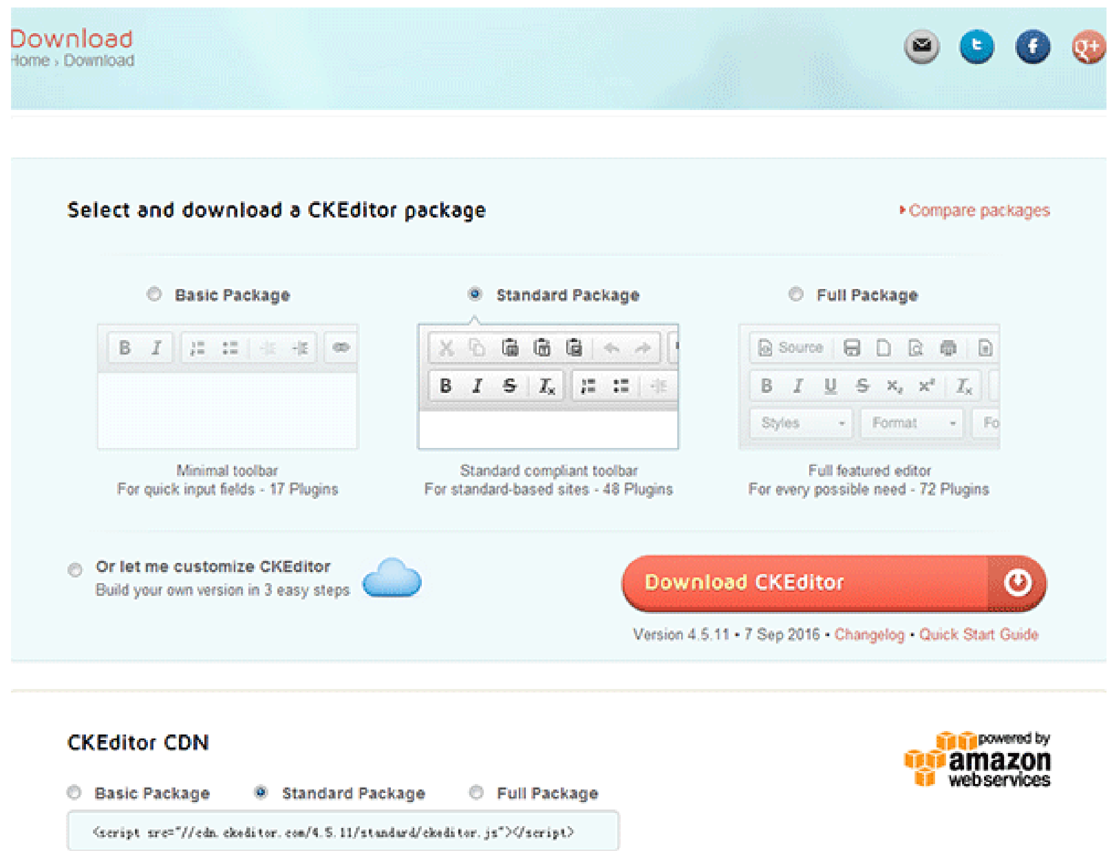
CKEditor 已经是最好用的网络在线编辑器，其强大的功能，丰富的插件支持，赢得了用户的喜爱。Drupal 8 默认已经安装了 CKEditor 编辑器，其可配置的插件功能虽不多，但基本的文本编辑已经够用。如果觉得现有 CKEditor 编辑功能不

www.drupalc.com

够，可以到 CKEditor 官网下载标准版、完全版或自定义版，下面我们将介绍自定义下载。

定位到 D8 中 CKEditor 目录/core/assets/vendor/ckeditor，目录下有一个名为 build-config.js 的文件，这个文件的作用是用来创建 CKEditor 自定义功能的配置文件。这个文件是由 CKEditor 创建器自动添加的，主要目的是实现再一次创建 CKEditor 编辑器时重用。这个文件只是在创建 CKEditor 时使用，其本身在 CKEditor 中并无作用，因此删除它并不会影响其功能。

定位到 CKEditor 官网(http://ckeditor.com)，点击右上角的 Download 按钮，进入下载 CKEditor 编辑器页面，如图所示。



有三个版本可供选择：

- Basic Package:基本版，最小化工具栏—17 个插件。
- Standard Package:标准版，标准工具栏—48 个插件。
- Full Package:完全版，包含所有特性—72 个插件。

仍选一个版本，然后点击 Download CKEditor 按钮下载。下载下来是一个压缩文件，解压后替换掉现有 CKEditor 即可完成 CKEditor 功能扩展。

CKEditor 除了提供这三个版本之外，还支持自定义创建 CKEditor 编辑器。在下载页面上点选"Or let me customize CKEditor"，然后点击 Customize &Download

CKEditor 进入自定义 CKEditor 页面。在这一页你将完成 CKEditor 的创建设置，分为 3 步:

1.选择基本包，Basic、Standard 或 Full。
2.选择需要的插件和皮肤。
3.选择编辑器支持的语言。

最后点选优化(Optimized)选项并点下载。

在这一页你还可以上传创建好的配置文件即先前提到的 build-config.js 文件进行自动创建配置，我们来修改 CKEditor 的默认皮肤，找到 build-config.js 文件，用 UE 打开，然后查找 skin 将后面的字符串由 moono 改为 office2013，然后保存，点击自定义下载 CKEditor 页面的右上角 Upload build-config.js 按钮，随后会弹出一个文件选择对话框，选择你刚才编辑的 build-config.js 文件，上传完成后，这一页的设置会根据你的 build-config.js 文件进行设置,你还可以作必要的修改，最后点击下载按钮进行下载。并将下载下来的压缩包解压，并将其替换掉现有的 CKEditor 目录，然后你进行 Drupal 8，你会发现编辑器的皮肤已变成 office2013 风格了。

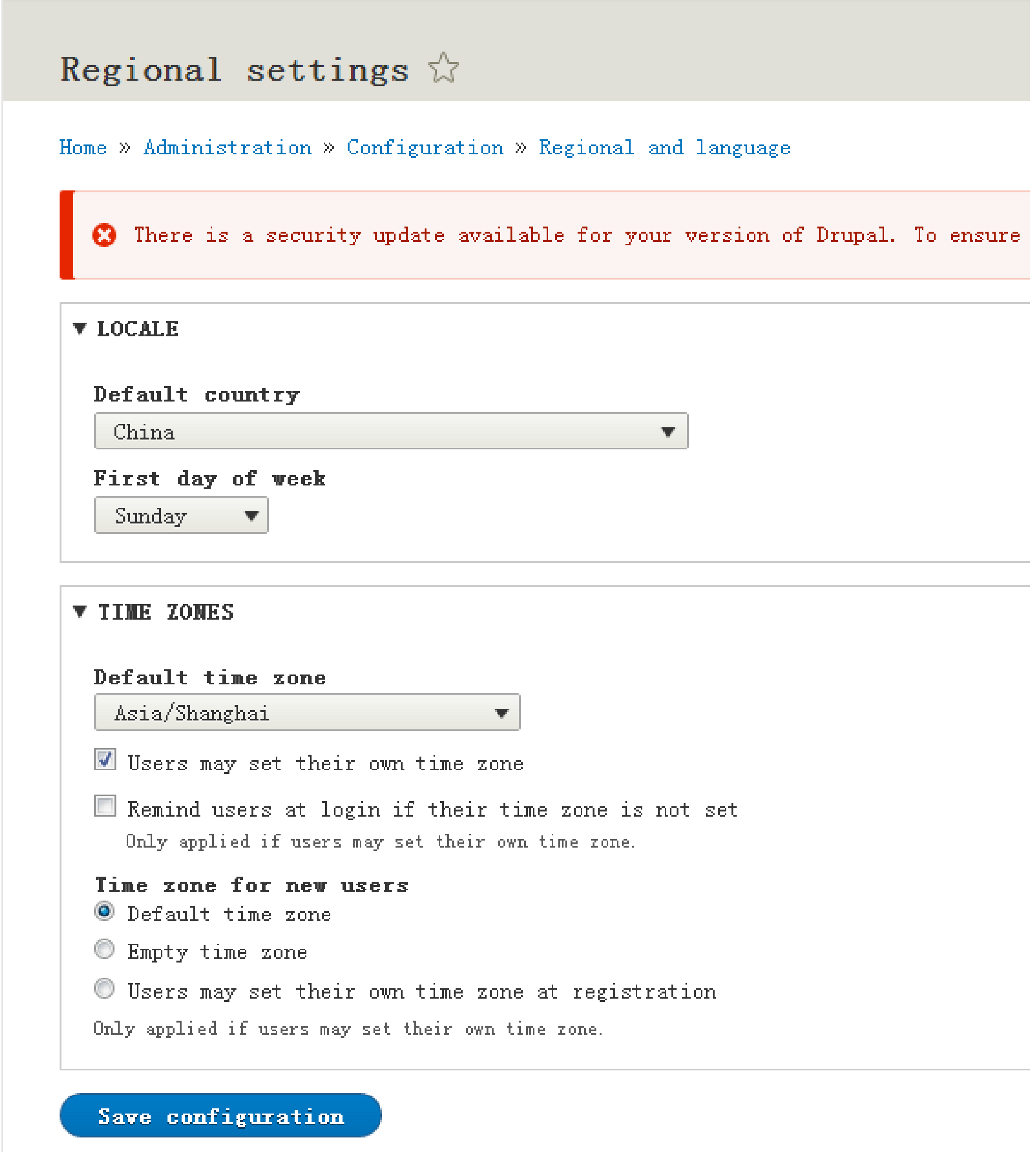
4.5 区域与日期设置

Drupal 8 在安装的最后一步，要求站点管理员设置国家和时区，这个设置就成为站点的默认国家和时区。在安装完 Drupal 8 后，可以对默认的国家的时区及日期格式进行修改。

4.5.1 设置国家和时区

点击->Manage->Configuration->Regional settings 或输入 admin/config/regional/settings(域名已省略)，在随后的页面中设置国家和时区，如下图所示。

www.drupalc.com



在 LOCALE->Default country 下拉列表框中，你会看到 Drupal 8 支持很多国家，默认国家可以设置为无，这里设置为中国(China)。接下来设置每周的第一天，按中国人的习惯设置为星期一。

时区设置，从 TIME ZONES->Default time zone 下拉列表框中选择一个时区，中国的话可以选择 Asia/Shanghai 或 Asia/Chongqing，这两个地方分别分中国上海和中国重庆。在默认时区下方还可以进一步进行设置。一般来说不需要用户设置他们自己的时区，新用户的时区设为默认时区即可。设置完后点 保存配置。

4.5.2 日期时间格式设置

点击 Manage->Configuration->Date and time formats，出现日期和时间格式管理界面，在这一页列出了 drupal 8 已创建的日期时间格式，这些格式与中国的时间日期格式有点不同，你可以对已存在的格式进行编辑和删除，也可以添加新的日期时间格式。

添加新的日期格式，在日期时间格式列表页面，点击添加格式按钮(Add format)或输入 admin/config/regional/dete-time/formats/add(以后这种不完整的 URL 都是省略了域名)后，出现添加日期时间格式页面，如下图:

Add date format ☆

Home » Administration » Configuration » Regional and language » Date and time fo

There is a security update available for your version of Drupal. To ensure

Name

Name of the date format

Format string *

A user-defined date format. See the PHP manual for available options.

Language

English

Add format

名称(Name):日期时间格式名称，如中国标准时间格式

格式字符串(Format string):PHP 支持的日期时间格式字符串，如果不知道，可查阅 PHP 手册，这里设为 Y-m-d H:i:s，设置后，文本框的右边会出现显示示例。设好后点添加格式(Add format)完成操作。

编辑日期格式所使用的表单和添加日期格式所使用的表单相似，只是表单已设置值，只需作修改即可。

4.6 搜索设置

Drupal 8 内置了关键字搜索功能，它由 search 模块实现，它提供了一个搜索区块，这个区块默认在每一页开启，这个特性可以通过区块管理进行修改。Drupal 的搜索模块支持插件功能，其它模块可以提供搜索插件以扩展 Drupal 核心搜索功能。在 Drupal 8 核心中提供两种类型的搜索插件，它们是由基于节点的内容搜索和用户搜索。贡献模块还可以提供其它的搜索功能。

Drupal 8 搜索引擎维护站点内容关键字索引，为了创建和维护关键字索引，需要配置一个 cron 维护任务。索引行为可以进行设置，点击 Manage->Configuration->Search pages 会出现搜索设置页面，主要对索引进度、索引负荷管制、默认索引设置、搜索日志、搜索页面共五个方面进行设置，如下图：

www.drupalc.com

Search pages ☆

Home » Administration » Configuration » Search and metadata

There is a security update available for your version of Drupal. To ensure

▼ INDEXING PROGRESS

Only items in the index will appear in search results. To build and maintain t

100% **of the site has been indexed. There are 0 items left to index.**

Re-index site

▼ INDEXING THROTTLE

Number of items to index per cron run

The maximum number of items indexed in each run of the cron maintenance task. If necessary,

▼ DEFAULT INDEXING SETTINGS

Search pages that use an index may use the default index provided by the Searc
rebuilt to reflect the new settings. Searching will continue to work, based o

The default settings should be appropriate for the majority of sites.

Minimum word length to index

The minimum character length for a word to be added to the index. Searches must include a k

☑ Simple CJK handling

Whether to apply a simple Chinese/Japanese/Korean tokenizer based on overlapping sequenc

▼ LOGGING

☐ Log searches

If checked, all searches will be logged. Uncheck to skip logging. Logging may affect pe

索引进度(INDEXING PROGRESS)

索引进度显示站点的内容索引情况，如有未索引内容，可以点击重建站点索引(Re-index site)内容进行索引或等待下一次自动维护任务，重建站点索引可能会花费较多时间。

索引负荷管制(INDEXING THROTTLE)

索引负荷管制是对索引自动维护任务进行设置，可设置每次自动索引的数量，数量设得太大会使用较大的内存花费较多的时间，可能会导致内存出错或超时，如果遇到此类情况请设置较小值。一些搜索页面类型可自行设置。

默认索引设置(DEFAULT INDEXING SETTINGS)

默认索引设置主要对索引关键字设置，搜索模块提供的搜索页面可以使用默认的索引，或者使用不同的索引机制。下面是对默认设置进行修改，这会导致搜索引擎重建索引，但这会等到下次索引自动维护任务运行，在这之前，搜索仍然使用以前的设置进行工作。索引关键词最小字数默认为 3，这个数字越小，搜索结果越精确，但会使用更大的数据库空间。每一个搜索查询可能包含一个或多个关键字。简单 CJK(中日韩字符)处理，如果勾选则启用基于重叠排列的简单 CJK(中日韩)语言 php 文件分析器。如果使用外部处理器，则关闭它，该设置不影响其它语言。

搜索日志设置(LOGGING)

一般不勾先记录搜索，如果勾选，则记录所有搜索。这会影响站点性能。

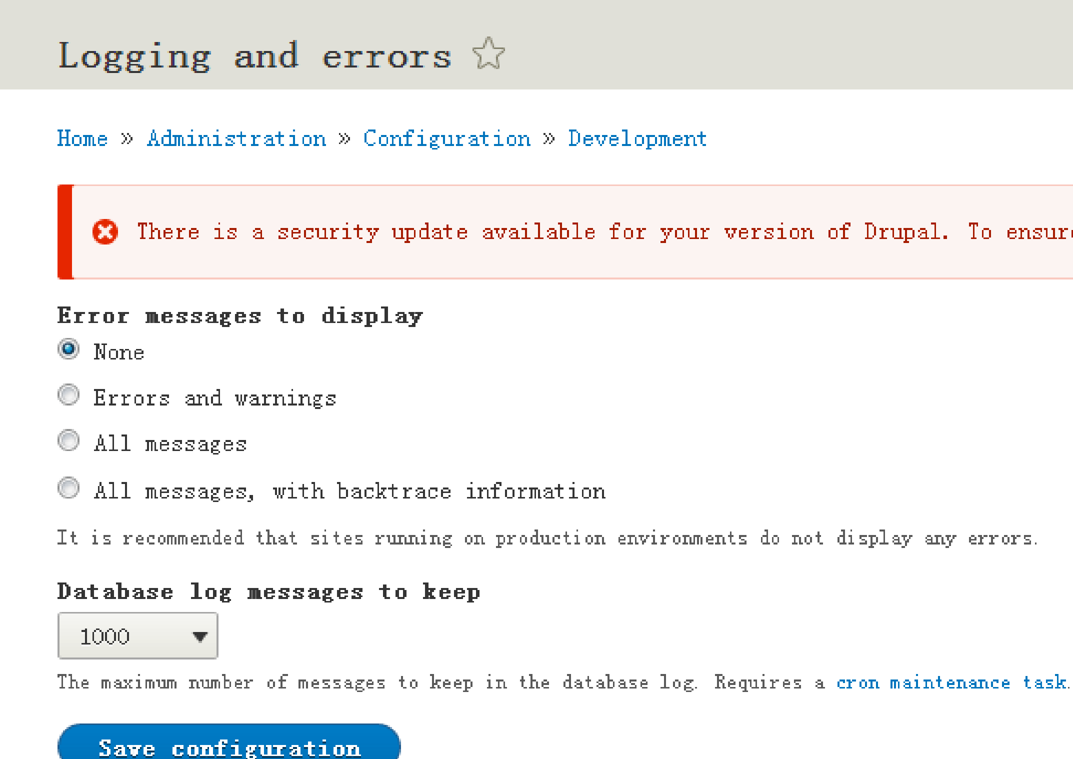
搜索页面设置(SEARCH PAGES)

这里列出了搜索页面类型，每一种页面类型都是模块为搜索模块提供的一个插件。可以对搜索页面进行添加、编辑、删除等操作。

www.drupalc.com

4.7 日志与错误设置

Drupal 8 提供了日志记录功能，在站点管理中可对其进行参数设置，到 Manage->Configuration->Logging and errors(admin/config/development/logging)对日志和错误进行配置，如下图:



错误显示方式：线上站点设置为无，以提高安全性。开发站点可以设为所有消息、带回溯信息。下方设置数据库日志最大数默认为 1000，这个需要一个自动维护任务，到站点报告查看日志，可手动清除日志。

如果开启了 syslog 模块，下方将出现相关设置，该模块的功能是将日志记录到操作系统日志文件中。

4.8 站点维护模式

有时站点需要处于维护模式，如 Drupal 8 核心升级，站点出现错误，有大量工作需要维护，数据库维护优化，手动运行 cron 等等。

点击 Manage->Configuration->Maintenance mode 出维护模式页面，勾选 将站点置于维护模式复选框，开启站点维护模式，开启站点维护后，具有"Access site in maintenance mode"的权限的用户仍然可以访问站点，注册用户可以输入用户登录 URL 登录系统。其它的访客只能看到维护提示消息，维护提示消息在下面的文本域中设置。

4.9 站点报告

Drupal 8 提供了非常详细的站点报告，方便站点创建者对站点状态进行监控。主要包括站点更新、日志信息、翻译更新、字段列表、拒绝访问错误、页面没找到、热门搜索、状态报告、视图插件共九项功能。

可用更新:指的是在站点上安装的主题或模块的更新信息，在这里列出可更新的主题或模块，你可以手动检查更新，也可以对可用更新进行设置，如设置检查的频率、设置更新通知邮件等等。

最新日志信息:这一页可让你查看站点日志信息，这些信息存放在数据库中，以分页的形式列出。由于日志非常多看起来并不方便，Drupal 的日志系统提供日志过滤功能，你可以对日志类型和严重性进行筛选以方便查看。系统中日志过多会占用数据库空间，Drupal 的自动 cron 会自动清空日志信息，你也可以在这一页通过点击"清空日志信息"按钮清除日志信息。

可用的翻译更新:显示已安装模块和主题的翻译更新报告，点击后会打开翻译报告页面，在这一页显示所有的翻译状况，也可以进行手动更新。

字段列表:显示所有实体类型使用的字段。这一页列出文章、区块、评论、手册页面、基本页面等实体类型所使用的字段。如果你创建了视图并使用了字段，它会显示在"用于视图中"选项卡列表中。

最常见"拒绝访问"错误:列出引发 403 错误消息的路径。

最常见的"没有找到页面"错误:列出引发 404 错误消息的路径。

热门搜索:列出站点搜索过的关键字，显示它们的计数。

状态报告:显示 Drupal 系统的运行环境以及出现的问题。主要包括 cron 维护任务、Drupal 核心更心状态、PHP 版本、PHP 扩展支持情况、web 服务器、信任主机配置、数据库环境等。如果节点访问权限出现问题，你可以在这一页点击"重建权限"进行重建。

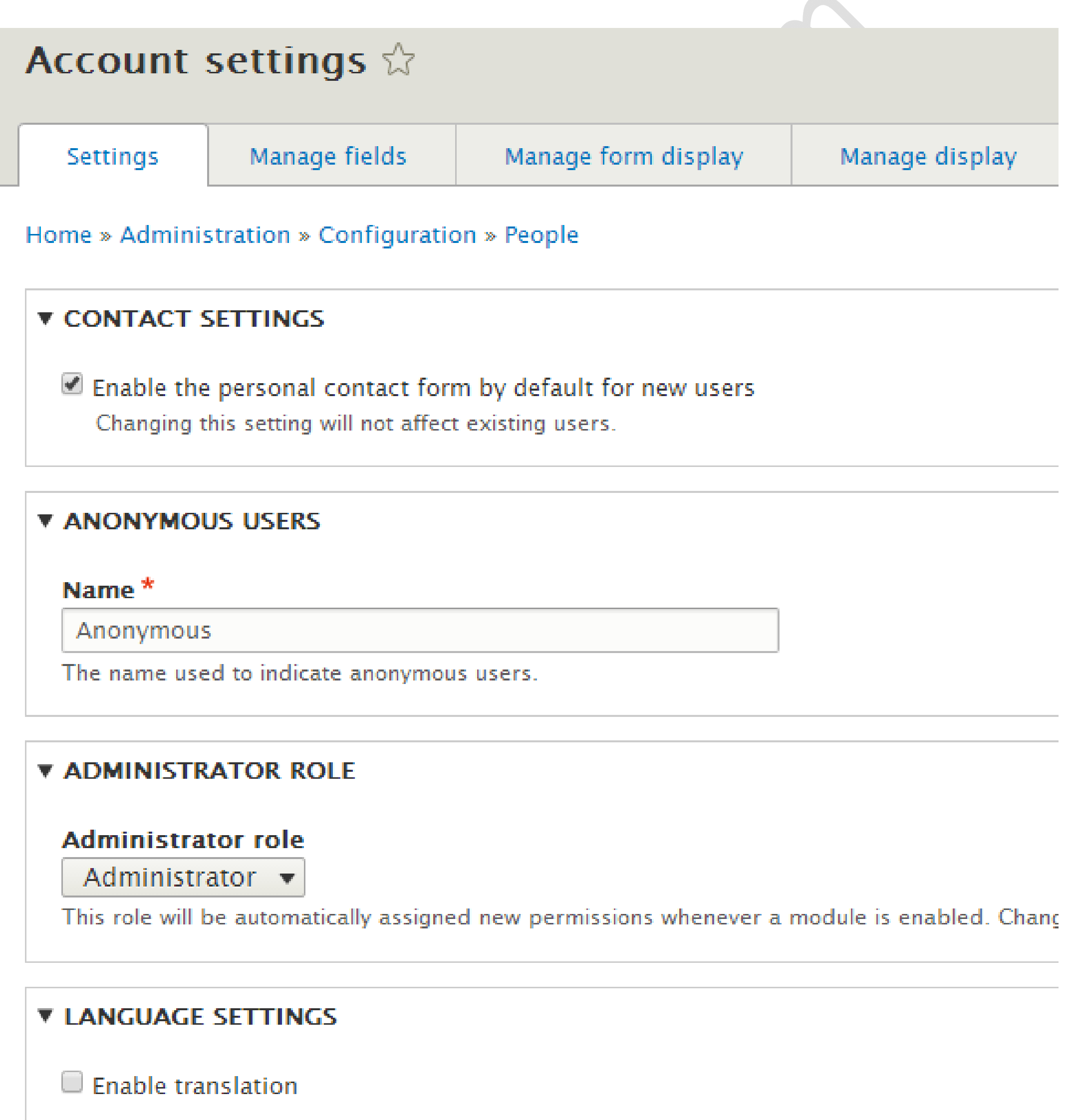
www.drupalc.com

第 5 章 用户管理

任何一个 CMS 系统都支持用户系统，用户可以是站点创建者、站点编辑、注册用户、访客等。Drupal 8 内建了强大的用户系统，它允许站点创建者对用户进行动态管理，为用户指定角色，赋予权限等等。

5.1 账户设置

账户设置主要是对 Drupal 用户的基本功能进行设置，主要包括设置、管理字段、管理表单显示、管理显示等选项页面如图 5-1 所示。



打开账户设置页面

点击 管理->配置->账户设置或输入 admin/config/people/accounts 进入账户设置页面，在这一页对账户进行设置。

联系设置

新用户默认启用个人联系表:勾选此项，注册的新用户可以使用个人联系表单，此项不会影响到已存在的用户。

匿名用户

匿名用户名称:默认为 Anonymous，可以修改。

管理员角色

管理员角色:设置此项，用户会获得管理员权限。

注册设置

设置谁可以注册账户，有三个选项分别是"仅限管理员"、"访客"、"访客，但须管理员批准"，默认为第三个选项，在没有启用验证码之前，最好选第一个选项，不然可能会有机器人进行大量注册，导致站点产生大量垃圾用户。一般来说需要勾选"访客创建账号需要电子邮件确认"，这样使用户的注册过程得到验证，不会随意注册。"启用密码强度指示器"勾选此项在用户设置密码时，会显示密码的强弱情况，这可以引导用户设置一个更安全的密码。

注销设置

有三个选项"禁用账户，并保留其它内容"，"禁用此账户并撤下其甩有内容"，"删除这个帐号，把此帐号所有的内容转到匿名用户下"，默认选择第一项，具有"选择取消账户方法"权限的用户或者管理用户权限可以覆盖这个默认方法。

通知所用的电子邮件地址

这项用于设置向用户发送邮件使用的电子邮件地址，在注册设置中选择"访客，但须要管理员批准"，系统会向这个地址发送用户审核提醒邮件。留空使用系统默认邮件地址。

电子邮件模板设置

这里设置向用户发送电子邮件的内容，主要包括欢迎(管理员创建新用户)、欢迎(等待批准)、管理员(用户等待确认提醒)、欢迎(不需审核)、账户激活、账户被阻止、账户取消配置、账户已取消、密码修复。电子邮件默认内容是英文写的，你可以开启翻译模块对此进行翻译，或直接在右边的文本域中输入内容，在电子邮件中可用使用占位符同。主要有:

[site:name]站点名

www.drupalc.com

[site:url]站点网址

[user:name] 用户名

[user:mail]用户邮件

[site:login-url]登录 URL

[site:url-brief]站点短 URL

[user:edit-url]编辑用户 URL

[user:one-time-login-url]用户第一次登录 URL

[user:cancel-url]用户注销 URL

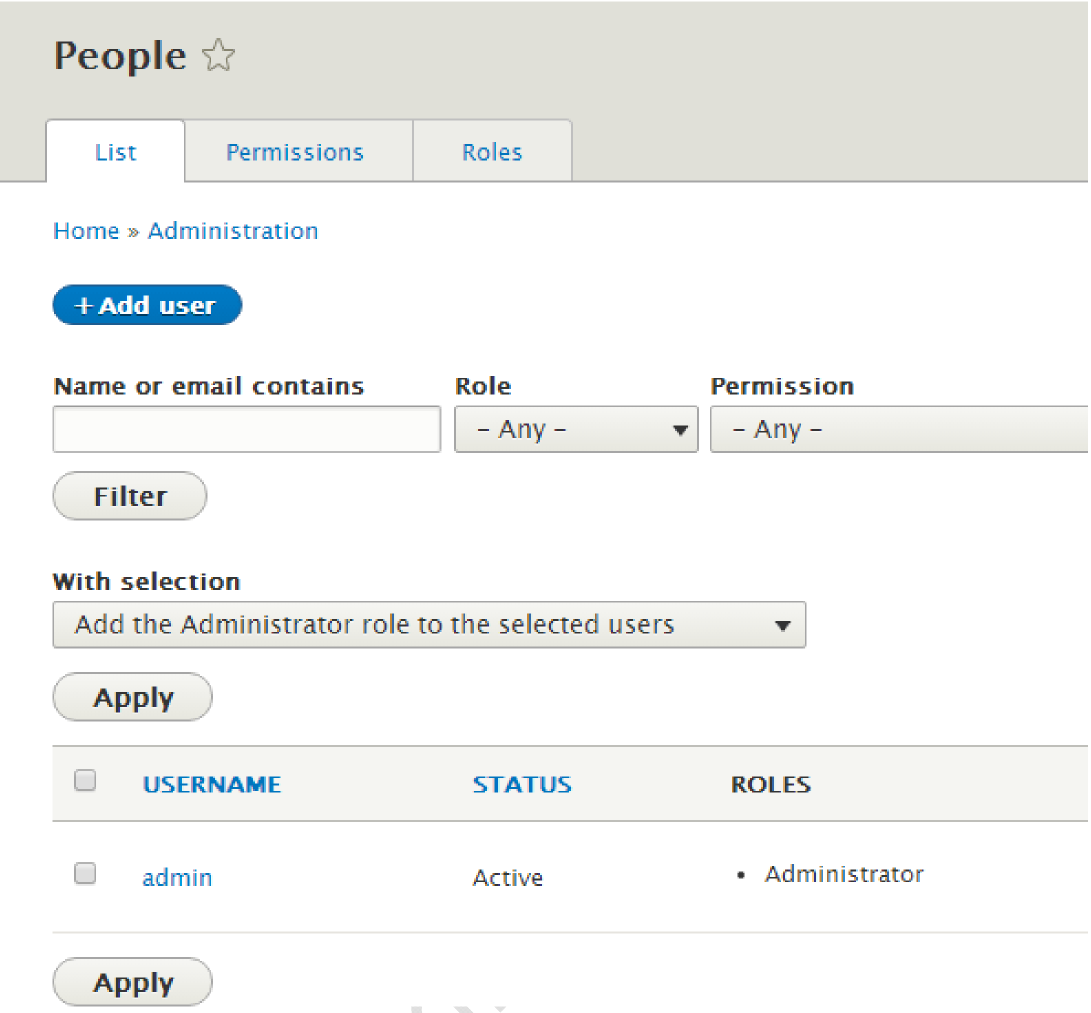
5.2 添加-编辑-删除用户

用户列表页面的 URL 为 admin/people，可通过点击 Manage->People 进入，在这一页对用户进行管理。这一页使用分页的方式列出站点已注册的所有用户，可以对它们进行批量管理。

添加新用户

点击用户列表上方的"添加用户"按钮，系统返回添加用户页面，在这一个页面显示一个用户注册表单，管理员按要求填写表单内容以完成添加用户。主要对电子邮件、用户名、密码、用户状态、用户角色、头像、联系表、时区等进行设置。如图 5-2 所示:

www.drupalc.com



编辑用户

用户列表的最后一列为操作列，这一列对每一个用户显示一个编辑按钮，管理员可以通过点击这个按扭来编辑用户信息，编辑用户的表单与添加用户的表单相似，这里不再赘述，编辑好后点保存完成用户信息编辑。

搜索用户

Drupal 8 提供强大的用户过滤功能，支持模糊查询，管理员可输用用户名或 Email 关键字进行查找，用户名、E-mail 可以只输入其中一部份，Drupal 会列出所有匹配的用户。还可以通过设置角色、权限、状态等进一步过滤。例如，我们想列出具有"创建新内容"的活动用户，从权限下拉列表中选择"创建新内容"，状态下拉列表中选择"有效"，然后点击"过滤"按钮，如果有匹配的用户则会在下方列出，没有则显示没有可用的用户。

批量操作

批量操作主要包括给选中的用户添加管理员角色、从管理员角色中删除选中的用户、封锁选中的用户、取消对选中用户的封锁、取消选中的用户，这些操作都需要选择一个或多个用户，这就实现了对用户的批量操作。

5.3 用户角色管理

Drupal 8 使用用户角色来对用户进行分组，以便于用户组织与管理。用户角色即用户组，它代表具有相同用户权限的一类用户，站点管理员可以在用户角色管理页面对网站角色进行管理，包括定义角色名、排序等。排序最好按照权限的多少来排，最少的排最前面，最多的排最后。其 URL 为 admin/people/roles。

角色列表

角色页面显示站点上已存在的角色，Drupal 8 内建三种角色分别是匿名用户、已登陆用户、管理员。它们显示的顺序按照权重排列，你可以通过拖移重排它们的顺序，重新排序后，记住要点"保存次序"按钮，角色列表的操作栏显示一个下拉按钮实现对角色的操作，包括编辑、编辑权限、删除等，点击编辑权限按钮可以快速来到角色权限设置页面。

添加角色

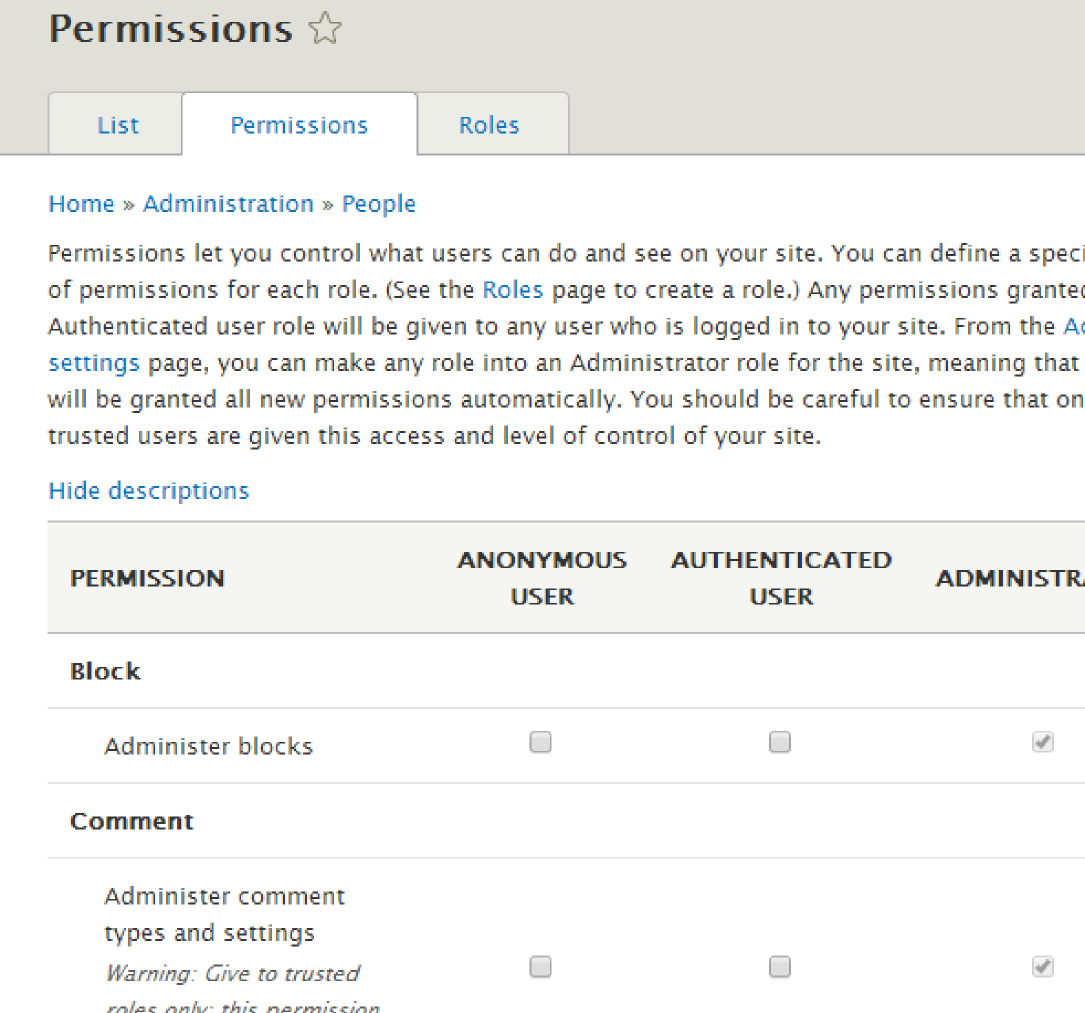
点击角色列表上方的"添加角色"按钮，在随后出现的页面中输入角色名称，然后点击保存完成角色的创建，编辑角色与此类似。

www.drupalc.com

5.4 设置角色权限

权限使站点创建者有能力控制站点用户能做什么以及能看什么。你可以为每一种角色定义权限，登录站点的用户将自动继承注册用户角色的权限，这个权限可以重新设置。你可以将任意一种角色设为管理角色，具有这种角色的用户将自动获得所有的新权限，这需谨慎使用。

权限设置页面的 URL 为 admin/people/permissions，其进入方式为，点 管理->人员->权限 或输入它的 URL，如下图所示：



这一页列出了站点的所有权限，并以模块分组的形式展示，列表的第一列为权限，后面的列为站点的角色名，给一种角色设置一个权限，只需点击这种角色下方相应权限的复选框即可，取消权限只需取消相应权限的勾选，由于权限非常多，上图只是一部份。

5.5 封锁用户IP

Drupal 8 核心提供 Ban 模块，该模块的功能是允许站点管理员封锁用户的 IP 地址，默认情况下并没有启用这一模块，如需封锁用户的 IP 地址，需到 管理->扩展 页面开启此模块。

开启 Ban 模块后，此模块会在 管理->配置 页面生成一个“IP 地址屏蔽”链接，点击该链接，出现 IP 地址屏蔽页面，在该页面查看和删除已封锁的 IP 地址、添加新的封锁地址。当一个被封的 IP 地址访问站点时，只能看到一条解释性的简短消息，输入的 IP 地址必须格式正确。要解除一个 IP 地址的封锁，只需点击该 IP 地址后的删除按钮即可。

5.6 用户注册、登录、找回密码

Drupal 8 系统为用户提供了基本的注册、登录、找回密码功能。点击右上角的登录链接，会出现用户登录页面，该页面带有三个选项卡，分别是登录、注册、重置您的密码。让我们先注册一个用户，单击"创建新帐号"选项卡(在用户设置中开启了注册功能才会出现该选项卡)，出现创建新帐号表单，填写电子邮件地址、用户名、选择时区后点击创建新帐号。如果 email 无效可能会出现不能发送 email 的错误提示，这个可以不用管。下方提示，感谢你申请新帐户，你的账户需要管理员审核，同时站点将向你发送一封欢迎 email。

新申请的帐号审核通过后，就可以登录系统了，点击登录选项卡，输入用户名和密码，点击登录即可登录系统。如果账户设置中注册无需审核则访客注册后可直接登录。如果你忘记了密码，你可以通过重置您的密码选项卡来找回，在该选项卡，输入你的用户名或 email，系统将向你发送一封重置密码的 email，你可以通过点击其中的重置密码链接来重新设置密码。

Drupal8 的用户系统看起来有些简陋，但这并不是问题，你可以到 drupal 官网下载用户扩展模块来扩展用户功能，或者自己开发个性化的用户系统模块。

5.7 隐藏用户登陆

出于安全或其它一些目的，你可能想隐藏用户登录。可以到区块管理中并闭用户登录菜单区块，具体如下:

点 管理->结构->区块布局 或输入 admin/structure/block，出现区块布局页面，这个页面包含两个选项卡，区块布局和自定义区块库。在区块布局页面，需要分主题设置，默认显示的是当前主题的区块布局。默认用户登录区块是放在 Secondary menu 区域中，也就是屏幕右上角位置。定位到 User account menu 菜单，在后面的下拉列表中选择"无"以关闭该区块。关闭用户登录后，用户仍可以直接在浏览器输入登录的 URL 进行登录。如果你开启了 User Login 区块，请使用同样的方法关闭。

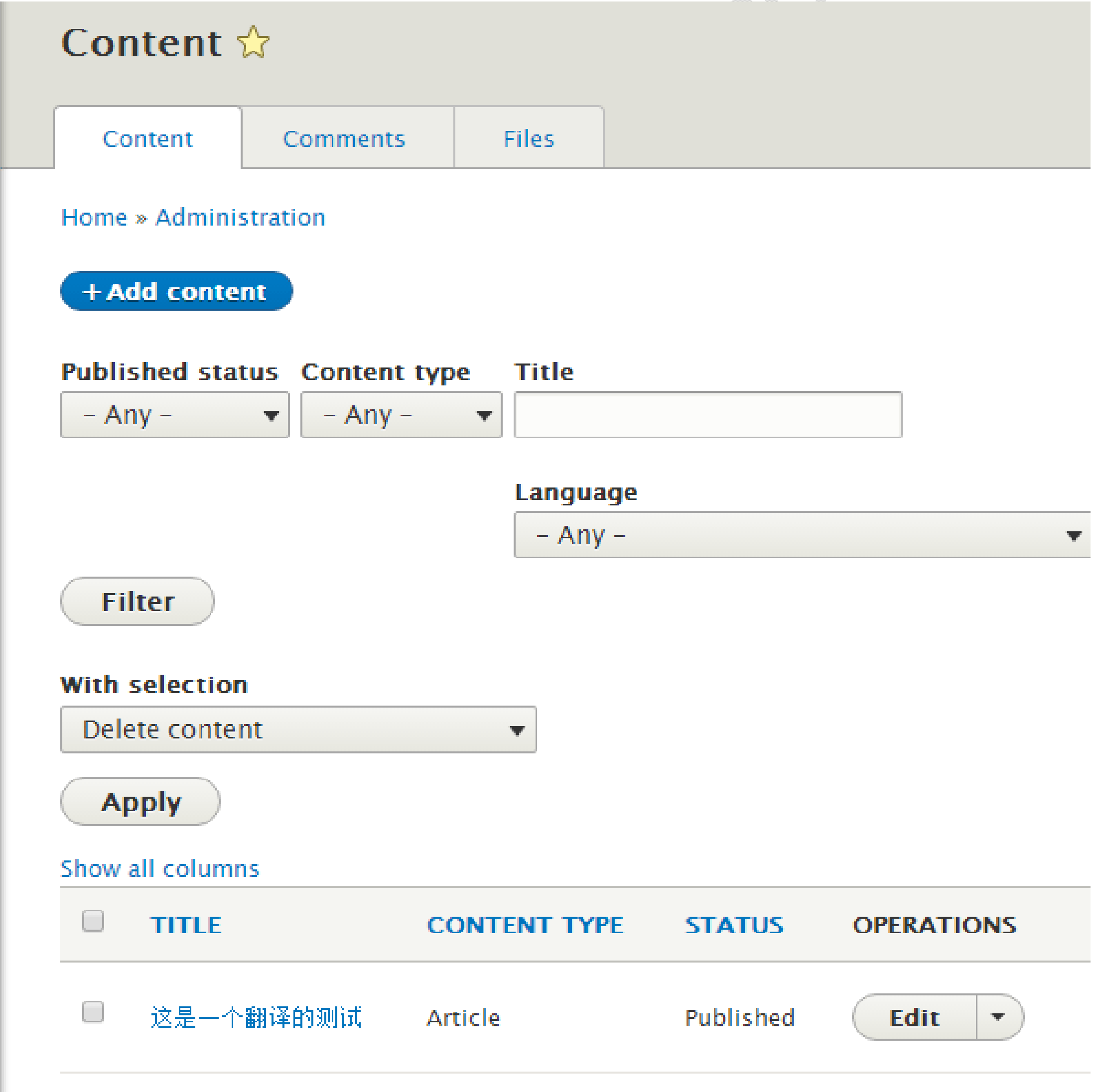
www.drupalc.com

第 6 章 Drupal 8 内容管理

Drupal 8 拥有强大的内容管理系统，内容在 Drupal 8 中被称为节点，根据内容的类型不同，有不同的节点类型，如文章、图片、产品等。节点类型也被称为内容类型，一个内容类型是由一个或多个字段组成，具体数据由字段存储在数据库中。管理员能轻易地创建内容，并对内容进行批量管理，如内容过滤、编辑、删除、置顶以及其它模块定义的功能，Drupal 8 还提供了就地编辑内容功能，使内容管理更方便。

6.1 内容管理界面

点 管理->内容或输入 admin/content 就会出现内容管理界面，如图所示：



这一页一共有三个选项卡，分别是内容、评论、文件。内容这一页显示了一个内容列表，大致包括标题、内容类型、作者、状态、已更新、操作等列。在列表上方可以对内容进行过滤，过滤包含发布状态、内容类型、标题等，其中标题可输入内容完整标题，也可只输入标题关键字，其它两个为下拉列表。每一行内容的操作列显示一个下拉按钮，包含编辑和删除两种功能。

6.2 添加内容

因为内容类型的原因，添加内容时需要选择一种内容类型。大多内容类型的添加很相似，这里以添加文章为例进行讲述。

点击内容列表上方的"添加内容"按扭，在随后出现的页面中点击"文章"类型或者直接输入 node/add/article，跳转到添加文章的页面。如图所示:

添加文章页面显示一个表单，此表单包括了添加文章的所有设置。在标题下的文本框中给文章取一个名称。正文是一个带有摘要的文本域，点击"编辑摘要"可出现输入摘要的文本域，在正文中输入文章内容，在文本域下方列出了可以使用的文本格式以及对文本格式的简要介绍，其中完全 HTML 和基本 HTML 可以使用 CKEditor 编辑，实现内容的所见即所得功能。

接下来是输入标签，多个标签使用逗号隔开。标签实际是一种分类，一种组织内容的手段，它使用了自动完成机制，输入了不正确的标签是无法保存的，Drupal 会给出提示，并用红色标出边框以示输入错误。

接下来是上传图像。如果你使用CKEditor上传了图像，图像不会在这里列出，你可以在这里上传图像，上传的图像会自动附加到文章中，你可以到文件列表中查看文件使用情况，注意如果你上传了翻译文件，可能导致这一页显示一个错误，清空所有缓存表就好了，详细请阅读http://www.drupacl.com/article/58。

如果是在管理模式下创建文章，右方会显示一个拆叠的菜单组，分别是菜单设置、评论设置、URL 路径设置、编著信息、推荐选项。非管理模式下，菜单组会显示在下方。一般不需要设置菜单。可以在评论设置中开启或关闭评论，开启评论后，具有"发表评论"权限的用户可以发表评论。URL 路径设置允许你自定义 URL 别名，这个 URL 将显示给用户和搜索引擎，这个很重要，在定义时一定要想好，不能随便改，否则有可能遭到搜索引擎的惩罚，你可以安装 pathauto 模块自动生成 URL 别名，具体请阅读 pathauto 相关内容。编著信息主要是对作者进行设置，默认为当前登录用户，如果你是管理员你可以设置为其它用户，还可以设置发布时间，一般不需要设置。最后一个为推荐选项，一般需要勾选"推荐到首页"选项，还有一项功能是将内容置顶。

全都设置好后，可以点击"预览"按钮查看文章预览，不满意可以返回继续修改，修改好后点击"保存和发布"按扭完成文章的创建，如果点击"保存为未发布"则只是保存了内容，网站访客不能查看该文章，管理员和作者不受影响。

这个过程只是默认系统发布文章的过程，因为贡献模块或自定义模块都可以修改这个表单，导致发布文章的一些选项有所改变。

6.3 编辑内容

定位到你需要编辑的内容，如你想修改刚才创建的文章，你可以查看该文章，在文章标题附近你会看到编辑文章的链接，点击后页面直接跳转到编辑文章页面。如果你知道它的 URL，你可以直接输入 URL 转到内容编辑页面。你也可以在后台内容管理页面对内容进行编辑，内容管理页面会以分页的方式列出站点所有内容，你可能会感到眼花缭乱，这时你需要对内容进行过滤或搜索，找到你想编辑的内容后，直接点击操作列中的编辑按钮即可。

编辑内容的表单与创建内容的表单是一样的。Drupal 从数据库中取出内容将其置于表单中，方便你对内容进行修改，修改完之后点击保存并发布就好了。

6.4 删除内容

在 Drupal8 中，删除内容也是很方便的，在查看内容附近就会有删除内容的链接，你只需点击这个链接，Drupal8 会提示你是否要删除内容，此动作无法撤销，点击"是"则真的删除，点击取消则会返回。在内容管理界面可以对内容进行批量删除，也可以单个删除，单个删除只需点击相应内容操作列的下接按钮'删除'就行了。批量删除则需要勾选待删除的内容，然后从"对所选项目"下拉列表中选择"删除内容"，最后点击"应用"按钮完成内容的删除。

www.drupalc.com

6.5 内容类型

在 Drupal 中，文章、日记、产品、广告、试题等都是内容，但这些内容之间也有区别，如产品与试题就不一样，试题包括题目、答案等，但产品则包含标题、图片、描述、价格等。为了体现这些不同，Drupal 使用内容类型这一概念来区分不同种类的内容，内容类型是由一系列与信息相关的字段组成的数据类型。内容类型定义了每种内容的属性，以及它们由哪些字段组成，它们的表单有哪些，它们的显示方式等。贡献模块或自定义模块能定认内容类型，也可以通过内容类型管理界面定义内容类型。

6.5.1 默认的内容类型

使用标准方式安装 Drupal 8，Drupal 8 将为你配置好了两种内容类型，分别是文章类型和基本页面类型，这两种类型在大多数网站中都是很常用的。文章类型主要是指站点经常更新的内容如新闻、日记等。基本页面类型则用于创建不常更新的内容如关于我们、联系我们等。

文章类型主要包含标题、正文、评论、图片、标签等字段。基本页面类型主要包含标题、正文等字段。这两类型都可以进行编辑或删除。

6.5.2 创建新内容类型

除了 Drupal 8 自带的两种内容类型外，你可以自定义内容类型。点击 管理->结构->内容类型 转到内容类型列表，你可以看到列表中显示了系统自带的两种内容类型，文章类型和基本页面类型，在每种类型的操作列显示了一个下拉按钮，主要包括管理字段、管理表单显示、管理显示、编辑、删除等功能。

我们现在添加一个产品内容类型。点击添加内容类型按钮，跳转到添加内容类型页面。

首先给内容类型取一个名称，在名称下的文本框中输入"产品"，此名称必须唯一，它会显示在添加内容页面的列表中，系统自动在文本框后面显示了机读名称，点后面的编辑可进行修改。接下来输入此类型的描述，如"用于管理产品类型，然后是对提交表单、发布选项、显示设置、菜单设置等进行设置，不要勾选发布选项中的推荐到首页，把显示设置->显示作者和日期信息前的勾去掉，不要任何菜单，因为产品类型不需要它们。完全设置好后点击保存和管理字段按钮进入下一步。

我们已经添加了产品这一类型，它已经具有了标题、正文、作者等一些默认的字段，但产品还有其它一些属性如图片、生产厂家、价格等。现在我们为产品添加生产厂家和价格字段，点击本页的"添加字段"按钮，转到添加字段页面，在标签下输入生产厂家，帮助文本可有可无，其它默认就行，然后点击保存设置。还可以对它的存储方式进行设置，这里将其数量限制为 1。再添加一个价格字段，字段类型选择小数，设置小数位数为 2，在标签中输入价格，还可以设置最大值、最小值，可以设置后缀为货币符号。设置好后点保存。如果觉得不满意，可以通过点击"编辑"进行修改。

www.drupalc.com

6.5.3 管理表单显示与管理显示

对于创建好的产品内容类型，可以对它的表单以及内容的显示方式进行管理。点击"产品"类型一行操作列的管理表单显示按钮(admin/structure/types/manage/products/form-display)，转到管理表单显示页面，在这一页，你可以设置产品类型表单的每个字段是否显示以及显示方式。

产品标题使用控件文本框，其长度为 60 个字符，可以隐藏该字段，根据需要设置其长度。产品描述，设置为文本域，默认行数为 9 行，这个也可以根据需要修改。生产厂商使用文本框，长度为 60 个字符，可修改。图像使用图像控件，预览图样式设置为缩略图(100*100)，这个可以根据需要修改，但只能选择已有的图像样式，可以到配置图像样式中创建新的图像样式。评论默认为开启，选择隐藏关闭。价格使用文本框，长度设为 10。设置好后点击保存即可。

管理显示，输入 URL admin/structure/types/manage/products/display，转到管理显示页面，在这里可以对不同显示模式进行设置，如产品摘要、产品详情等不同显示模式。

产品详情显示设置

图像的标签设为隐藏，格式设为大图样式，图像链接设为无。描述的标签设为隐藏，格式默认即可。评论的标签设置成位于上方，格式设为评论列表，如果评论比较多，可设置分页。产品价格的标签设为隐藏。生产厂家的标签设为位为上方。在下方可以自定义显示设置。设置好后点击保存。

同样地，我们可以设置其它显示模式，比如，产品摘要模式，我们只需保留缩略图的简短描述与价格等。这完全取决于你想如何展示你的产品。

6.6 评论管理

评论是对别人创建的内容作出自己的看法，在 Drupal 中评论也是一种内容类型。但它相对来说比较特殊，可以为其它的内容类型开启评论，默认情况下，可以对文章进行评论，但这是可以设置的，你可以编辑文章内容类型，关闭它的评论功能。如上文创建的产品内容类型，也可以为其开放评论功能，在已开放评论的内容类型的每一个节点上还可以开放或关闭主论类型(请参考内容创建章节)。

点击 管理->结构->评论类型 转到评论类型列表页面，这一页列出了系统已存在的评论类型，Drupal 已自带了一个"缺省评论"，其描述为允许对内容进行评论。因它是内容类型，所以也可以对它进行编辑、管理字段、管理表单显示、管理显示等操作。我们也可以添加新评论类型。

点击"编辑"按钮，我们发现只能修改它的标签与描述。从管理字段选项可以看出，评论使用了长文本格式，只能编辑这个字段。管理表单显示表出了作者、主题、评论等的设置。

在内容页的下方会显示一个评论表单，填写标题、评论内容，提交即可添加一条评论，添加的评论会显示在下方按评论的先后顺序显示。

管理员能轻易地管理评论内容，可以在浏览评论内容时对其进行编辑或删除，也可以对评论进行审核。在 管理->内容->评论 页面对评论进行批量管理，包括审核评论、删除评论、编辑评论、撤下评论等。管理员应该合适地设置评论权限，如哪些人员可以发表评论，哪些人员发表的评论需要审核等等。这些需要到权限管理中进行设置。

www.drupalc.com

第 7 章 区块管理

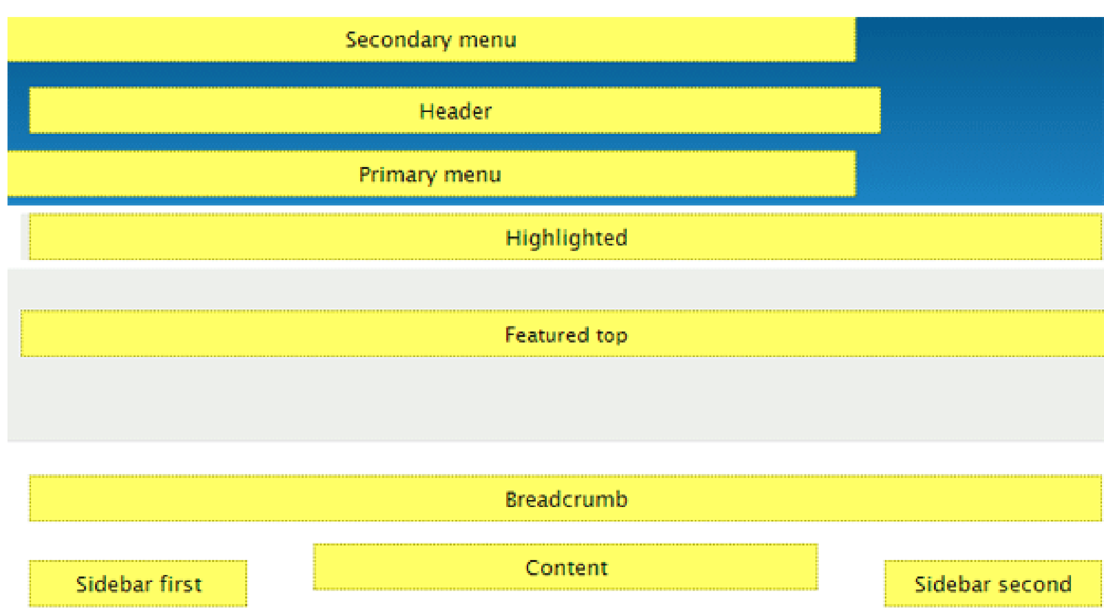
Drupal 8 使用区块来控制相关内容或单独内容的显示位置，这种机制使内容显示非常灵活。例如，一个 web 站点有站点标志、站点菜单导航等，它们的位置是不固定的，但我们只需建好相应的区块，无论它在哪个位置显示，其区块内部总是一样的。在网站上有很多区块，如栏目区块、菜单导航区块、热点内容区块、广告区块等等。Drupal 8 的区块功能是非常强大的，它已经预先创建了一些常用区块，只需将这些区块配置在主题中的合适位置(区域)即可工作。

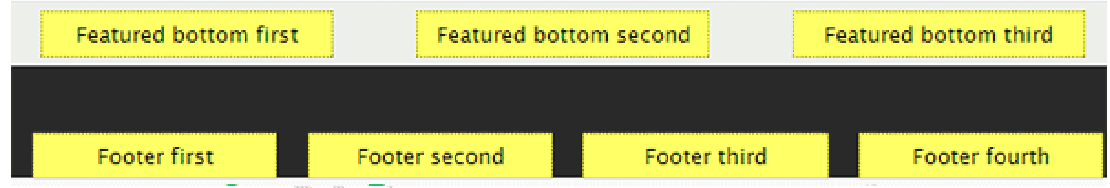
7.1 Bartik区域演示

区域是页面上的一块矩形位置，与区块是包含关系，主题中可以定义区域，一个区域中可以放置一个或多个区块。所以区块需要针对每个主题进行配置，Drupal 8 默认前端主题 Bartik 定义了页眉、主菜单、高亮、特性、面包屑、左边栏、内容、右边栏、底部等区域，先让我们来看看各个区域的位置。

点击 管理->结构->区块布局 或输入 admin/structure/block，转到区块布局页面，默认显示 Bartik 的区块配置情况，在区块配置列表上方有一个"演示块区域"链接，点击主个链接，转到 Bartik 区域布置图，如图 7-1 所示。

www.drupalc.com





这在个演示中，你可以看到每个区域所在的位置，这样便于你对区块进行配置。

7.2 常见区块简介

Drupal 8 已经为我们创建好了一些区块，有的区块是在视图中创建的，有的则是模块创建的。点击任意区域的"放置区块"按钮，系统会弹出放置区块对话框。这个对话框列出了系统所有可用的区块，下面对常用模块进行简要介绍。

标签:显示站点分类标签。

导航路径:一个面包屑的导航菜单。

最新内容:显示网站最新内容，可以设置条目数，此区块由视图(Views)创建，一般放在边栏区域。

最新评论:显示网站最新的评论内容，可设置显示条数，些区块由视图(Views)创建，一般放在边栏。

新进用户:网站最新注册的用户，可设置显示数量，此区块由视图(Views)创建，一般放在页面底部。

主导航:网站导航菜单，这个可以在菜单管理界面进行定义，一般放在导般区域。

工具:指用户能在网站上进行的操作，如添加内容等。

用户登录:用户登录区块，放置用户登录的表单。

还有其它区块，由于不常用，这里不作介绍，有兴趣的读者可自行查看。

7.3 放置与配置区块

在特定主题的区块管理页面，可以在指定区域放置区块，也可以修改区块放置的区域，当选择区域为无时，表示禁用区块。可以对已存在的区块进行配置，不同的区块有不同的配置选项。可以删除已存在的区块。可以对区块进行排序。

一般来说我们不想将站点名称和站点口号显示在页面上，点击页眉区域中的站点品牌的配置按钮，转到区块配置页面，区块的标题不用管。关掉站点品牌的元素，将站点名称、站点口号前的勾去掉。可见性也不用管，点击保存区块。到首页查看，你会发现，站点名称和站点口号都不见了。

站点首页不需要面包屑导航，我们可以对面包屑导航区块进行排除设置。点击面包屑区域下的 Breadcrumbs(不同系统可以有不同名字)的配置按钮，转到面包屑区块配置页面。标题不用管，点击可见性设置中的页面选项卡，在右边的文本域中输入<front>(代表首页 URL)，再点选下方的"为下列页面隐藏"选项，最后保存区块。再回到首页查看，已经没有面包屑导航了，其它页面不受影响。

7.4 添加谁在线区块

对于一些论坛、社区交友类网站，我们想显示当前在线用户。Drupal 8 默认已为我们提供了这一功能，我们只需添加在线用户区块即可。

www.drupalc.com

打开区块布局页面，点击'footer fifth'区域中的放置区块按钮(一般在线用户显示在页面底部)，随后弹出一个系统可用区块的对话框，在该对话框中选择'在线用户'区块，系统会弹出区块配置对话框，不要显示标题，项目数设为 50，可见性设置不用管，机读名称采用默认即可，完全设置好后，点保存区块。现在在线用户区块已经配置好了，你可以返回首页查看。

系统不为我们提供了很多区块可以使用，配置方法差不多，这里不再赘述。

7.5 自定义区块管理

在网站开发中，Drupal 8 提供的区块可能不能满足具体需求，这时我们可以自定义区块，既可以使用区块布局中的自定义区块管理功能添加自定义区块，也可以通过模块开发添加自定义区块。

点击 结构>区块布局>自定义区块库 或输入 admin/structure/block/block-content 转到自定义区块管理页面。在这一页列出了所有自定义的区块，自定义区块其类型为自定义区块类型，这种类型可以自行创建或配置字段以及显示设置等。创建区块后就可以在区域中开启以便页面对区块进行定位。

我们来添加一个统计区块，点击 添加自定义区块 按钮，在随后弹出的页面中选择 基本区块，系统会转到自定义区块设置表单。在区块描述中输入"统计区块"，在正文中输入统计代码，统计代码根据你使用的统计系统而不同，比如你使用 cnzz 统计，则登录 cnzz 官网进行相应设置后，将统计代码复制并粘贴到这里，注意这里的文本格式应该选择支持所有 HTML 的文本格式，设置好后点击保存。

统计区块已创建好，它会显示在自定义区块列表中，现在可以到区块布局中 footer fitth 区块中开启它。

第 8 章 Drupal 8 语言管理

Drupal 8 拥用强大的语言管理功能，能使你轻易开发多语言网站。Drupal 8 核心提供了四个与语言相关的模块。它们是:

Configuration Translation 提供供配置的翻译界面。

Content Translation 允许用户翻译内容实体。

Interface Translation 翻译内建的用户界面。

Language 允许用户配置语言和将它们应用到内容。

标准安装方式，这四个模块是没有开启的，因此要使用语言功能，需要到管理>扩展页面开启语言模块。这四个模块中 Language 是最基本的模块，其它三个模块都要用到，大家可以展开模块描述以查看它们之前的依赖关系。

本章将详细介绍 Drupal 8 的语言功能，主要讲述这四个语言模块的配置使用。

8.1 配置语言

非英语站点和多语言站点需开启语言(Language)模块，该模块提供了基本的语言功能和语言检测。首先开启语言模块，点击 管理>扩展 转到模块列表页面，找到语言(Language)模块并勾选，同时勾选其它三个模块(Configuration Translation，Content Translation，Interface Translation)，然后点击安装(install)，页面上方会提示这些模块已开启。

开启了语言模块后，就可以添加一种语言了。点击 管理>配置>语言 或输入 admin/config/regional/language，转到语言配置页面，这个页面列出了系统已存在的语言，可以对这些语言进行重排序，它们的顺序将会影响到语言区块切换，编辑内容时的语言选择等顺序，但不会影响语言检测与选择的顺序。在这里设置站点默认的语言，但不推荐在正工作的站点上设置。

现在我们为站点添加简体中文语言。点击添加语言(Add language)按钮，在语言名称(Language name)下选择简体中文(Chinese,Simplified)，然后点击添加语言(Add language)按钮，完成简全中文语言的添加。这时页面重定向到语言列表，可以看到简体中文语言已被创建可以使用的提示文本，语言列表中已列出了简体中文语言。将简体中文语言设为站点默认语言，点击简体中文后面的默认(DEFAULT)选项，然后点保存配置(Save configuration)即可。

Drupal 8 已为我们定义大部份语言，如果需添加的语言不再语言名称下拉列表框中，可以选择列表框最下面的自定义语言(Custom language...)，选择后，在下

www.drupalc.com

方会自动出现自定义语言的配置，包括语言代码(Language code)、语言名(Language name)、阅读方向(Direction)。其中语言代码应符合 W3C 对语言代码的定义，阅读方向默认为从左至右(Left to right)。配置好后点击添加自定义语言(Add custom language)按钮完成自定义语言的添加。

编辑与删除站点语言

在语言列表的最后一列，有一个下拉按扭，主要是对站点语言进行编辑和删除，点击简体中文后面的编辑(Edit)按钮，可以编辑语言的名称，阅读方向等，但不能改变语言代码。非站点默认语言可以点击删除(delete)进行删除，系统会提示该动作不可恢复，需要确认才能删除。这个一定要小心，一旦删除与语言相关的内容将不会显示。

语言检测与选择

对于非英语网站和多语言站点，Drupal 8 是如何知道当前使用哪种语言显示页面的呢，Drupal 8 提供了多种方法对语言进行检测，它们按照优先级顺序进行排序。

点击 管理>配置>语言>检测与选择(Manage>Configuration>Language>Detection and selection) 或输入 admin/config/regional/language/detection，转到语言检测与选择页面。这个页面列出了界面文本语言检测方式，如果安装了内容翻译模块，还会列出内容文本语言检测方式。主要有以下几种检测方式。

Account administration pages:账户管理页面

URL:通过 URL 前缀或域名检测语言

Session:Session 参数检测语言

User:跟踪用户的语言设置

Browser:根据浏览器语言设置选择语言

Selected language:基于已选择的语言

各种检测方式需要开启才能生效，开启相应的检测方式只需勾选相应行的 ENABLED 复选框就行，如果开启了多个则按照优先级进行检测，优先级高的生效。其中 Selected language 总是开启的但其优先级最低，非语言站点只需开启这一项就行，因为非英语站点还是只使用一种语言，只需选择好默认语言就行。

基于 URL 前缀或域名的检测方式配置

大多数多语言站点都是基于 URL 前缀或域名区分的，现在我们进行相应配置。首先勾选 URL 检测方式以开启它，再点击 然后点击 Configure(配置)按钮，页面

www.drupalc.com

转至 URL language detection configuration，选择 Path prefix(路径前缀)选项，然后分别对已存在的语言进行 URL 前缀配置。如英语语言前缀 URL http://www.example.com/en，汉语语言前缀 http://www.example.com/zh-hans，站点的默认语言可以留空不设置。如果选择 Domain 域名选项，则需要对各种语言所使用的域名进行配置，如英语语言 en.domain.com，汉语语言 zh-hans.domain.com 等，配置好后点击 Save configuration(保存配置)。

基于浏览器的语言选择

浏览器使用不同的语言代码关联到同种语言，这里可以设置不同的语言代码与站点语言的映射关系，该页面列出了一些常用的语言代码，你可以添加新的语言映射关系，只需输入浏览器的语言代码，然后选择站点语言即可完成语言映射关系。但需注意浏览器语言代码应符合 W3C 的语言代码定义。

8.2 用户界面翻译

用户界面翻译模块的功能是提供一个对用户界面的翻译功能，主要包括在线翻译界面文本，导入、导出界面翻译文本以及翻译设置等功能。

导入一个翻译文件

用户界面翻译提供了导入翻译文件功能，现在我们来导入简体中文语言包。先到 Drupal 官网下载简体中文语言包，文件名如 drupal-8.1.7.zh-hans.po。点击 Manage>Configuration>User interface translation>Import 或输入 admin/config/regional/translate/import，页面转至 Interface translation import(界面翻译导入)页面。当添加语言或模块、主题开启时，系统将自动下载翻译文件并导入。这一页允许你手工导入翻译文件，翻译文件是一个 Gettext Portable Object file(可移植对象)，其扩展名为.po 文件。手工导入可以用于自定义模块、自定义主题的翻译，可以到 Drupal 翻译服务器下载翻译文件或导出翻译文件，并使用 Gettext 翻译编辑器进行自定义翻译，并可以将翻译结果导入系统。如果翻译文件过大，可能需要好几分钟。

点击选择文件按钮，在弹出的文件选择对话框中选择 drupal-8.1.7.zh-hans.po 文件，选择简体中文语言(Chinese,Simplified)，有三个复选项，分别为将导入的字符串视为自定义翻译，覆盖非自定义翻译，覆盖已有的自定义翻译。这些选项首次导入不用管，点击 import(导入)按钮，随后会出现一个导入进程条，指示导入的进度，导入完成后会自动转到翻译界面，并提示简体中文语言已导入等信息。

导入完成后到 管理>配置>语言 页面中将简本中文设为站点默认语言，保存后网站界面变为简体中文。

www.drupalc.com

在线翻译

上一节我们导入了简本中文语言包，这个是 Drupal 的官方翻译文件，但是它并不完善，有的地方可能不符合中文习惯，这时我们可以在翻译页面进行在线翻译。

点击 管理>配置>用户界面翻译或输入 admin/config/regional/translate 即可转到翻译页面。这一页能让翻译者搜索指定的已翻译或未翻译文本串，以用于创建或编辑翻译，但由于翻译任务涉及大量文本串，建议将需翻译的文本串导出以方便离线翻译。

在字符串包括文本框中输入需要翻译的字符串，留空为显示所有字符串。搜索区分大小写，可以对翻译语言进行限制如选择简本中文，可搜索已翻译和未翻译的字符串或仅显示其中一种字符串，最后点击过滤按钮。搜索结果将会显示在下面的字符串翻译列表中，在对应的字符串后的文本框中输入翻译后的字符串，点击下方的保存翻译即可。

导出翻译到文件

因为界面翻译的工作量巨大，最好将需翻译的字符串导出为文件，然后使用专业工具进行翻译，这样事半功倍。

点击 管理>配置>用户界面翻译>导出 转到导出翻译页面。这一页导出你站点使用的翻译字符串。导出的文件可以是 Gettext Portable Object (.po)格式，包含原始字符串及翻译（可以和其它人分享）；或者导出为 Gettext Portable Object (.pot)格式，只包含原始字符串，以便于使用 Gettext 翻译编辑器创建翻译。

选出好语言和导出选项，导出选项有包括非定制翻译，包括定制翻译，包括未翻译文字，最后点击导出，可将翻译文件下载到本地。

用户界面翻译设置

用户界面翻译设置主要对检查更新、翻译来源、导入行为进行设置。点击 管理>配置>用户界面翻译>设置或输入 admin/config/regional/translate/settings 转到用户界面翻译设置页面，如图所示。

检查更新用于设置对已安装模块或主题界面翻译的检查频率，分为从不（手动）、每周、每月三项，可以点击'立即检查更新'进行手动检查。也可以在管理>报告>可用更新页面进行查看、更新和设置。

翻译来源设置界面翻译的源文件位置，可设为 Drupal 翻译服务器和本地文件或仅本地文件。翻译文件存储于本地的 sites/default/files/translations。这个路径可以在文件系统设置中进行更改。

导入行为指导入翻译文件时对已有的翻译的处理方式，可选值有三项，分别为不要覆盖现有翻译，只覆盖已导入翻译，保留定制翻译。覆盖现有翻译。

www.drupalc.com

8.3 内容语言翻译

内容语言翻译这个模块的功能是创建站点内容的翻译文件，比如，你建了一个英语和汉语的双语言站点，你发布一篇使用英语写的文章，但有好多中国人看不懂，你就想创建相应的汉语文章，这个就使用内容语言翻译功能实现。

点击管理>配置>内容语言和翻译或输入 admin/config/regional/content-language，转到内容语言和翻译的设置界面，如图所示。在这个页面你可以设置哪些内容可以翻译，并且可以精确到字段。包括内容类型、分类词汇、用户资料等实体类型。默认隐藏语言选项、语言为站点默认语言。

勾选一个实体类型，在下方会出现翻译设置表单，如选择内容选项，然后选择文章选项，会弹出文章类型所引用的字段，已选择的字段可以进行翻译。

8.4 配置翻译

点击管理>配置>配置翻译转到配置翻译页面，这一页列出了站点上所有的可被翻译的配置项，如站点名，角色名等等。任意点击列表中某项的列表按钮，即可列出可被翻译的字符串，然后可以进行翻译，如图所示。

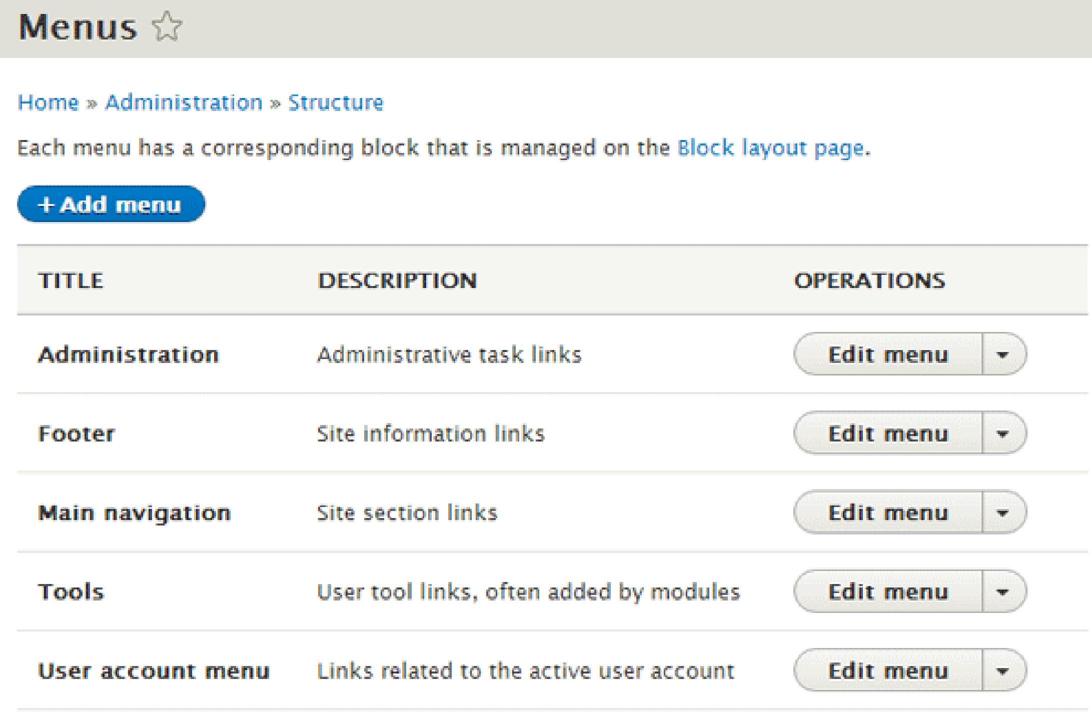
另外在相应的界面菜单中会出现翻译按钮，以便实现就地翻译。例如，点击管理>结构>内容类型，转到内容类型管理页面，你会看到，系经统列出了 Article 和 Basic page 两种内容类型，每种内容类型后面的下拉按钮中有一项是翻译，点击它就可以对内容类型翻译了。

www.drupalc.com

第 9 章 Drupal 8 菜单管理

Drupal 8 的菜单系统与之前的版本有很大的不同，以前的版本菜单系统的功能是非常强大的，因为其与 URL 解析进行映射，一个菜单对应一个函数功能。而 Drupal 8 采用 Symfony 2 进行 URL 解析，菜单系统也因此得到简化。Drupal 8 的菜单管理主要是由 Menu UI 模块提供，该模块对系统菜单进行管理，如添加菜单、删除菜单、编辑菜单等。

点管理>结构>菜单即可转到菜单管理界面，如图所示。在这一页，我们可以看到，系统内置了五大类的菜单，分别是 Administration 管理菜单、Footer 底部菜单、Main navigation 主导航菜单、Tools 工具菜单、User account menu 用户账户菜单。每个菜单都有对应的区块，它们可以在区块布局页面进行管理。



添加菜单

点击页面上的"添加菜单"按扭，然后输入菜单标题和管理用摘要，再点击保存即可完成菜单的添加，如图 9-2 所示。

Add menu ☆

Home » Administration » Structure » Menus

You can enable the newly-created block for this menu on the Block layout page.

Title*

Administrative summary

Menu language

English

Save

删除菜单

除开系统内置的的菜单，其它由用户添加的菜单可进行删除，只需点击相应菜单项后面的下接菜单并选择删除即可。

添加菜单链接

我们常用的菜单主要是主导航菜单，现在我们来添加一些链接。点击主导航菜单中的添加链接按扭即可转到添加链接页面，如图所示。输入菜单链接标题、链接路径、菜单描述，设置好父链接和权重，然后点击保存即可完成添加菜单链接。添加完后系统自动自定向到菜单列表界面。注意:菜单链接路径可以是一个内部路径或外部路径，<front>代表首页。记得勾选启用标志，否则菜单将不可用。

www.drupalc.com

Add menu link ☆

Home » Administration » Structure » Menus » Administration

菜单链接标题 *

此链接在菜单中显示的文本。

Link *

Start typing the title of a piece of content to select it. You can also enter an internal path such as */node/add* or an external URL such as *http://example.com*. Enter *<front>* to link to the front page.

☑ Enabled

A flag for whether the link should be enabled in menus or hidden.

描述

当鼠标移动至菜单链接上时显示。

☐ Show as expanded

If selected and this menu link has children, the menu will always appear expanded.

Parent link

The maximum depth for a link and all its children is fixed. Some menu links may not be available as parents if selecting them would exceed this limit

第 10 章 Drupal 8 分类管理

为了更好地组织内容，方便分类查询，Drupal 8 提供了 Taxonomy 模块对内容进行分类管理。通过合理地配置权限，该模块可以将内容与标签联系在一起。Taxonomy 采用词汇与术语来管理标签。分类常常与菜单一起使用，例如网站栏目或频道即是一个分类，它们常常作为导航菜单出现在主导航中。

分类管理界面

点击管理>结构>分类转到分类管理界面，该页列出了系统已有的词汇表。系统内置了 Tags 词汇表即标签。你可以添加词汇表、编辑术语表、列出术语、管理字段、管理显示等操作，其中管理字段与管理显示和以前讲的差不多。

添加词汇表

点击本页'添加词汇表'按钮转到添加词汇表页面，输入词汇表名称如'下载'，输入词汇表描述，然后点击保存按钮完成词汇表的添加。

添加术语

找到刚才创建的'下载'分类，点击列出术语操作，系统会列出'下载'分类下的术语，由于这是新创建的分类词汇，下面还没有术语，现在我们来添加一个。点击添加术语按钮，在随后出现的表单页面中输入名称、描述、URL 别名，设置好后，点击保存即可。

对已存在的术语可以进行编辑、删除等操作。

为内容类型创建分类

为了使内容能使用结构化的分灰术语，我们需要将分类词汇赋予内容类型。现在我们来给文章类型添加分类。点管理>结构>内容类型>文章>管理字段导航到管理文章字段页面，这一页列出了文章类型已使用的字段，现在我们给它添加一个分类字段。点击'添加字段'按钮，选择分类术语，输入分类术语的标签，如'分类'，点击保存到下一步，设置限制数量为不限，点击保存字段设置进入下一步。在随后的页面中进行详细配置，输入帮助文本，设置非必填字段，可设置默认值。最后点击保存配置按钮。

现在分类字段已经出现在文章类型的字段列表中了，可以进一步对表单显示进行配置，如选择使用列表还是使用自动完成。完全配置好后，在添加文章界面可以输入文章的分类了。

www.drupalc.com

第 11 章 视图管理

Drupal 8 已经内置了强大的视图管理工具，该模块在 Drupal 7 中一直是建站必备的贡献模块。对于一个由数据库驱动的网站来说，我们的任务无非就是将创建的内容存入数据库中，然后根据实际需要从数据库中查询内容显示在网站页面上。然而这并不容易，需要你理解复杂的 SQL 语句，并且很容易出现错误，导致网站出错。

视图模块的出现使得这一切变得轻松和容易了，它允许你通过图形界面管理数据库内容查询，你不需要懂得复杂的 SQL 语句，只需点点鼠标就可以将需要的内容查询出来，并设置为页面又或是区块放置在合适的地方。Drupal8 已经内置了一些常用的视图，如在线用户视图、最新内容等等。你可以轻松创建、编辑这些视图。

11.1 视图概念

视图这个词语源于数据库述语。数据库视图是一个复杂的存储查询，用于使用表或数据库。你可以从数据库视图中获取数据。

Drupal 视图模块允许你以相似的方式来管理数据库查询，当你创建一个 Drupal 视图后，模块会自动写出查询以管理数据库。

正如 Drupal 一样，视图模块提供了强大的功能，以致于你只需点几下鼠标，就可以在首页上放置站点最新内容区块。可以将区块放置于标签页中，第一个标签页上显示热点内容，第二个标签页中显示最新评论，第三个标签页列出最新用户等。

视图模块提供了创建动态站点内容的能力，丰富了你的站点内容，它使得你管理与维护站点更加轻松，它的功能足可以写成一本书。

基于以上原因，视图模块是必需掌握的管理工具，本章将讲述视图模块的用法，以及如何使用它更好地管理和维护你的站点。

11.2 管理视图页面

点管理>结构>视图导航到 admin/structure/views 视图管理页面，在这一页列出了系统已存在的所有视图，每行显示一个视图，主要显示的信息包括视图名称、显示方式、机器名、视图描述、标签、路径、操作等。

www.drupalc.com

在视图的操作列，以下拉菜单组织视图操作，主要有以下操作:

Enable:启用视图以便于向用户显示。

Disable:禁用视图，视图被禁用后将不会显示。

Edit:编辑视图，通这编辑视图修改视图的设置。

Dupalicate:复制视图，此操作将提示你复制一个视图，并可以对此视图的所有设置作出修改。

Delete:删除视图，此操作不能恢复。

11.3 创建简单的区块视图

视图模块允许你创建数据列表区块，数据列表区块可以在站点区域中开启，以便将其显示给站点用户，如图所示。

具体操作如下:

1.导航到管理视图页面 admin/structure/views。

2.点击 Add New View，随后会显示一个创建视图向导，以帮助你正确地设置视图，如图所示。

3.在视图名称字段输入视图名称。

4.点击描述，并输入视图描述。

5.在视图设置>显示下面，选择 Taxonomy terms 分类术语项，如图所示。

下面是各选项的解释:

Comments:评论类型，可以指定排序

Log entries:登录类型，可以指定排序

Files:文件，可以指定排序

Content:内容类型，可以指定排序

Content Revisions:内容修订，可以指定排序

www.drupalc.com

Taxonomy Terms:分类术语，可以指定排序

User:用户，可以指定排序。

6.在区块设置中，启用 Create a block 创建一个新区块选项。

7.在区块显示设置>显示格式中，选择一个格式选项，格式选项依赖于当前视图的 Show 设置，这个格式将决定数据显示方式。例如，你可以选择非格式化列表，或 Grid 列表等，如下图。

8.在 Items per block 字段，输入区块中要显示的行数。

9.如果你想将内容分页显示给用户，请启用 Pager(分页)选项。如果不启用分页选项，用户仅仅可以查看指定显示的项。

10.单击 Save and Edit。转到整个视图编辑界面。如果不需要改变，请单击 Save 按钮。

11.4 创建简单页面视图

上一节我们介绍了使用视图模块创建区块视图，本节将介绍使用视图模块创建页面视图。具体步骤如下:

1.导航到视图管理页面 admin/structure/views。

2.单击 Add New View(添加新视图)。

3.在视图名称字段输入视图名称。

4.单击描述输入视图描述。

5.在 Show 字段，选择以下选项之一:

Comments，Files，Content，Content Revisions，Taxonomy Terms，Users。

6.勾选 Create a Page 选项以启用创建页面并对页面视图进行设置。

7.在页面标题字段输入页面标题，在路径字段输入页面路径。

8.在页面显示格式设置中选择一种显示格式，一共有四种显示格式分别为 Grid，HTML List，Table，Unformatted List。

9.在 Items to display 字段，输入需要显示的项目数量。

www.drupalc.com

10.如果想分页显示，则勾选 Use a pager 选项。

11.如果你想为视图创建一个菜单链接，则勾选 Create a menu link 选项。从下接列表中选择一种菜单类型，并输入链接文字。

12.如果你想让视图提供 RSS feed，请勾选 Include an RSS feed 选项。并输入 Feed 路径以及 Feed 行样式。

13.全部配置好后点 Save and Edit。

11.5 向视图添加显示

你能够向一个视图添加一个或多个显示类型。每一个显示类型是一个不同的视图。举一个例子，你可以制作一个视图用于在页面中显示最新内容，并且创建一个新的显示类型用于在区块或 feed 中显示内容。一个显示可以修改或覆盖视图的所有参数。

1.导航到视图编辑页面(如 admin/structure/views/views/MYVIEWNAME/edit)。

2.点击 Add 按钮并从弹出的列表中选择一项。

3.作出必要的修改后点击 Save。

www.drupalc.com

11.6 向视图添加字段

1.导航到视图编辑页面(如 admin/structure/views/views/MYVIEWNAME/edit)。

2.在 Fields 部份，点击 Add 按钮，将弹出添加字段对话框。

3.在 Type 下拉列表中选择一项。

4.从字段列表中选择一个或多个字段。提示:你可以在搜索框中输入关键词以进行过滤。

5.点击 Add and configure fields 按钮，将弹出配置字段对话框。

6.可选步骤:启用 Create a Lable 选项并输入标签文本。

7.启用或禁用 Exclude From Display 选项。这允许隐藏一些作为配置目的的字段。

8.点击 Apply。如果你选择了一个或多个字段，这个对话框还会显示，你必需重复 6 到 8 步。

11.7 自定义视图字段的输出样式

你能够自定义视图的 HTML 和 CSS。你可能启用或禁用缺省的 CSS 类。

1.导航到视图编辑页面(如 admin/structure/views/views/MYVIEWNAME/edit)。

2.在字段设置部份，点击一个字段。系统弹出字段配置对话框。

3.点击 STYLE SETTINGS 展开字段样式设置。

4.启用以下选项中的一个或多个:

Customize Field HTML

Customize Label HTML

Customize Field and Label Wrapper HTML

5.针对已选择的每一项，指定包围内容的 HTML 元素。你可以点击 Create a CSS Class 来指定一个 CSS 类。

6.启用或禁用 Add Default Classes 选项。当启用这个选项，该字段将使用缺省的 CSS 类。

7.点击 Apply。

www.drupalc.com

11.8 重写视图字段的输出

你可以配置一个视图中的字段显示信息，使之不同于字段本身的真实数据。通过使用 token，你可以强化数据库中字段内容。

1.导航到视图编辑页面(如 admin/structure/views/views/MYVIEWNAME/edit)。

2.在字段配置部份，选择一个字段并单击，系统弹出字段配置对话框。

3.点击 Rewrite Results 展开重写输出结果配置。

4.可以启用以下选项:

Override the output of this field with custom text:使用自定义文本覆盖字段输出。

Output this field as a custom link:输出字段作为一个链接，你必须输入一个路径，你可以为链接显示配置一些选项。

Trim this field to a maximum number of characters:截取字段数据，你需要输入最大字符数。当字段值超过这个长度时会截取。

Strip HTML tags:删除 HTML 标签，启用这个选项，所有的 HTML 标签将会被删除和。你可以指定保留的标签。

Remove whitespace:删除空白字符，启用这项，开头和结尾的所有空白字符将会被删除。

Convert newlines to HTML
tags:启用这项，所有的换行符(\n)将被转化为 HTML 标签。

5.点击 Apply。

11.9 在视图中获得字段的自定义名称

有时一个视图能以不同的方式使用相同的字段，为了避免混淆，你应该为该字段使用不同的名称。具体步骤如下:

1.导航到视图编辑页面(如 admin/structure/views/views/MYVIEWNAME/edit)。

2.在字段设置部份，选择一个字段并单击，系统弹出字段配置对话框。

3.单击 Administrative Title 字段展开管理标题设置，输入字段名称，这个名称会显示在视图编辑页面中。

11.10 向视图添加上下文过滤器

你可以给视图配置一个内容过滤器，以便用户可以通过输入内容关键字动态地获取内容。例如，在页面上放置一个包含内容过小滤器的区块或者是根据作者获取的文章列表。

为了创建一个基于内容而不是基于 URL 信息的上下文过滤器，你需要创建一个关系。具体如下:

1.导航到视图编辑页面(如 admin/structure/views/views/MYVIEWNAME/edit)。

2.点击 Advanced 以展开高级视图设置。

3.点击 CONTEXTUAL FILTERS 旁的 Add 按钮。

4.系统弹出添加上下文过滤器对话框，该对话框列出了可以使用的过滤器，你可以在 Search 文本框中输入过滤器关键字进行搜索，或者从 Type 下拉列表中选择一项进行筛选。从列表中选择一个或几个过滤器，然后点击 Add and configure contextual filters 按钮，系统弹出已选过滤器的配置对话框。

5.设置当过滤器的值不可用时的行为，这里根据情况自行设定。

6.如果需为不同的情况设置不同的值，可以在 EXCEPTIONS 部份进行设置。

7.设置当过滤器的值在 URL 中时的缺省值，这里根据情况自行设定。

8.配置好后点击"Apply and Continue"按钮，并为每个过滤器作出设置。

11.11 配置视图过滤规则

视图的一个主要功能就是从数据库中取出数据，但这需要告诉视图从数据库中取出哪些数据，也就是给视图设置过滤规则。具体如下:

1.导航到视图编辑页面(如 admin/structure/views/views/MYVIEWNAME/edit)。

2.点击过滤规则旁的 Add 按钮。

3.系统弹出添加过滤器的对话框，从过滤器列表中选择一个或多个。然后击点"Add and configure filter criteria"按钮，进入过滤器规则配置对话框。

4.过滤器规则的配置对话框根据你选择的过滤器不同而不同，需要根据实际情况进行配置。配置好后点击"Apply"按钮。

11.12 向视图添加关系

当你第一次创建视图时，你会选择一个如 Comments,Content,Taxonomy terms 等的基本表。这个选择后不能进行修改。你只能从已选的基本表中选择字段，对于内容视图，你可以选择作者的用户 ID，但是无法获取用户名字段。为了获得用户名字段，你需要创建一个关系，将用户表与内容表进行关联。

www.drupalc.com

1.导航到视图编辑页面(如 admin/structure/views/views/MYVIEWNAME/edit)，在这个例子中我们编辑一个内容列表。

2.点击 Advanced 展开视图高级设置。

3.点击 Relationships 旁的 Add 按钮，弹出添加关系对话框，该对话框列出了可以关联的表。

4.在这个例子中我们选择 author 表，在搜索框中输入"author"，列表会更新为与 author 相关的表，这里选择 Content author 选项，并点击 Add and configure relationships 按钮进一步配置。

接下来设置管理标题，这个标题用于显示在视图编辑页面，你可以开启 Require this relationship 选项，开启后将会隐藏不包括这个关系的内容。点击 Apply。

6.点击字段设置旁的 Add 按钮，从已选的关系中搜索如'user'。现在你可以使用更多的字段了，因为有用户表与主表关联。选择 User:Name，点击 Apply。

8.接下来配置 User:Name 字段。在格式中选择 User name，勾选 Link to the user。其它的根据需要进行配置，最后点 Apply。

9.回到视图编辑页面可以查看视图预览。

11.13 向站点访客显示过滤器

可以在视图上配置一个显示给站点访客的过滤器，这允许他们同视图内容进行交互。

1.导航到视图编辑页面

2.在过滤规则部份，选择一个过滤器并点击，系统弹出配置过滤规则对话框。这个对话框是对你选择的过滤规则进行设置。

3.启用 Expose this filter to visitors,to allow them to change it 后，会在下方出现其具体配置。可以选择 Single filter、Grouped filters，依据你选择单一过滤器还是过滤器组，又会出现具体的设置。

4.在 Label 文本框中输入标签，在 Description 文本框输入描述。配置好后，点 Apply。

5.在过滤规则设置部份，你可以对过滤规则进行添加、删除、重新排序。

注意:有些过滤器没有更多的选项，因此不提供群组过滤功能。

www.drupalc.com

11.14 在视图中获得一个过滤器的自定义名称

有时一个视图能以不同的方式使用同种过滤器，为在视图编辑页面出现混淆，你能为每个过滤器指定不同的名称。

1.导航到视图编辑页面(如 admin/structure/views/content)。

2.在过滤规则设置旁，选择一个过滤器并单击。系统弹出过滤规则配置对话框。

3.滚动屏幕，找到 Administrative Title 字段并单击，以展开管理标题设置，然后输入一个名称，这个名称将会出现在视图编辑页面。

11.15 管理视图的显示设置

1.点 Manage>structure>views>settings 导航到视图设置页面 admin/structure/views/settings。

2.启用或禁用以下设置:

Always show the master(default)display:启用这个选项，视图编辑页面将总是显示一个主视图(缺省视图)。如果这个选项没有启用，当创建第一个视图后，主视图将隐藏。

Always show advanced display settings:启用这个选项，视图编辑页面将总是显示高级配置选项，诸如关系和上下文过滤器等。如果选项已禁用，视图高级选项仍然可用，但需单击以展开。

Show the embed display in the ui:在 UI 中镶入显示，镶入显示可以通过 views_embed_view()在代码中使用。

3.Live Preview Settings:启用或禁用以下选项:

Automatically update preview changes:启用这个选项，自动预览功能将会开启。

Show information and statistics about the view during live preview:在预览期间显示视图信息与统计，你可能通过设置此项来决定是否显示视图的信息。

Show the SQL query:显示创建视图显示的 SQL 查询代码。

www.drupalc.com

Show other queries run during render live preview:在视图被渲染时，Drupal 能运行很多查询。启用这项将会在渲染视图期间显示每个查询。

4.配置好后点 Save configuration。

11.16 配置视图调试模式和缓存

除了视图的基本设置之外，Drupal 8 还提供了视图的高级设置，主要对视图调试模式和缓存进行配置。

1.点 Manage>Structure>Views>Settings>Advanced 导航到视图高级设置页面 admin/structure/views/settings/advanced。

2.一般情况下，视图将缓存表、模块、视图等数据。你可以禁用缓存，但请注意这将会带来严重的性能问题，因此在线上站点不推荐禁用。勾选 Disable views data caching 禁用视图缓存功能，视图总是会跳过缓存并每次都重建数据，这会影响站点访问速度。点击 Clear Views' cache 按钮清空视图缓存数据。

3.当我们创建视图时，可能会出各种问题，为了更好地找出问题所在，我们需要开启视图调试来帮助我们。在 DEBUGGING 部份可以启用 Add Views signature to all SQL queries(向所有 SQL 查询添加视图签名)。开启这项后，在数据库服务日志中，将在 SELECT 语句的后面添加"view-name:display-name"字符串。

www.drupalc.com

第 12 章 Drupal 8 模块开发

使用模块可以扩展 Drupal 站点的功能，除了 Drupal 8 核心提供的模块外，还有许多贡献模块可供选择使用。你也可以自定义模块，这涉及到 Drupal 8 的模块开发知识和 PHP 及其必须的编程语言知识。由于 Drupal 8 采用了新的内核，其模块开发方法与以前的版本不同。本章教你一步步编写自定义模块。

12.1 模块开发基本知识

Drupal 8 使用了很多 PHP 的高级特性，它集成了许多第三方库。有经验的 Drupal 7 开发者可能已看出了明显的改变，但许多基本结构仍然很相似。本节介绍的一些基本知识对于 D8 模块开发来说并不是必须的，如果你已比较熟悉，可以跳过这部份内容。

面向对象编程(OOP)

面向对象编程已经成为目前公认的最佳的编程方法。它将一切看作对象，对象具有属性和方法，属性表明对象状态。方法实现对象的操作。面向对象实现了软件的重用性、灵活性、扩展性目标。面向对象还包括诸如抽象、继承、多态等特性。面向对象的内容非常丰富，足可以写成一本厚厚的书，这里不进行细讲，有兴趣的可以阅读面向对象的相关资料。

Drupal 8 支持面向对象特性，其核心大多使用类、接口写成。并遵循一些公共的设计标准，如 The Factory Pattern，Software design pattern，Foundations of Programming:Design Pattern 等等，具体请阅读相关资料。

PHP 命名空间

如果你对 PHP 命名空间概念不熟悉，请阅读 PHP 手册或阅读 PHP 书籍。Drupal 8 使用了 PSR-4 标准开发。PSR-4 标准是基于 PHP 命名空间的自动加载包的方式。它定了如何自动加载基于命名空间和类名的类。Drupal 模块在 Drupal 根命名空间中有自己的命名空间。在大多数情况下，Drupal 代码的命名空间是按照模块命名的如 block.module 命名空间为 Drupal\block。

PSR-4 规定文件仅仅包含一个类、接口或 trait。这些文件的名称必须与所包含的类、接口或 trait 的名称相同。以便于类加载器进行自动加载。

依赖注入

www.drupalc.com

依赖注入已经成为了一种 OOP 设计模式，因为 Drupal 8 需要大量地调用核心 API 或核心服务，所以应该把它作为一种概念来理解。请阅读 dependency injection on PHP the right way，以便更好地理解这种方式。Drupal 8 的服务调用请阅读 Services and dependency injection in Drupal 8。

Symfony 2

Drupal 8 引入上 Symfony 2 开发框架，目的是为了减少代码重复。Drupal 8 主要用它处理路由(routing)、sessions 和服务。阅读 Symfony 2 手册以学习更多的 Symfony 框架知识。学习它根本不需要任何 Drupal 知识，学习它将使你成为更好的 Drupal 开发者和 PHP 开发者。

注释

Drupal 8 采用了文档块的注释方式，一些插件可以找到并为代码提供一些附加内容。注释能被 Doctrine 注释解析器解析，并能转化为 Drupal 能使用的信息，以便于更好地理解你代码的功能。请到 Drupal 8 官网阅读更多的相关信息。

插件

插件提供一些小功能，使用插件更易于在各种功能之间切换。具有相似功能的插件组成一个插件类型。例如'字段部件'是一种插件类型，它包括诸如文本框、文本区域、日期等不同的插件。插件开发请阅读本书第 16 章。

www.drupalc.com

12.2 准备一个模块框架

本节将会指导你创建简单的模块，包括自定义页面、区块、实体、字段等。首先需要创建一个模块目录并创建一个模块信息文件 module.info.yml。可以通过两种方式激活模块，一种方式是使用 Drupal 8 的管理扩展页面启用模块，另一种是直接使用 drush。

12.3 给你的模块命名并创建目录

在开始前，你应该设置 PHP 的错误显示，以便于即时将 PHP 错误显示在屏幕上。

模块命名

首先你应该给你的模块命名，包括模块的描述名字与机器名。机器名将会用于模块的相关文件中，注意不要与 Drupal 核心模块或其它模块重名。

这里列出了模块命名的基本原则:

必段以字母开头。

只能包含小写字母与下划线。

必段唯一，不能与模块、主题、或配置文件重命。

不能使用保留字:src、lib、vendor、assets、css、files、images、js、misc、templates、includes、fixtures、Drupal。

在本例中，我们使用"hello_world"作为模块的机器名。

注意:在模块的机器名中请不要使用大写字母，因为 Drupal 将无法识别你的 HOOK 实现。

为模块创建目录

我们的模块的机器名为"hello_world"，为此我们需要为模块创建 hello_world 目录，在 drupal 的安装目录/modules/custom/hello_world，或者/sites/all/modules/custom/hello_world。如果没有 custom 目录，请创建之。注意模块的目录名与机器名可以不同，比如我们使用 HelloWorld 作为模块的目录名，但请记住，模块的代码文件中必须使用模块机器名。

以前的版本中自定义模块目录是放在/sites/all/modules 下的，核心模块放在/modules。在 Drupal 8 中，所有的核心模块、库等都放在/core 目录中。贡献模块与自定义模块放在/modules 目录中，但你也可以放在/sites/all/modules 目录中，其结果是一样的。

我们这个模块现在还没有任何功能，我们需要先建立一个.info.yml 文件，以使 Drupal 8 能发现它。在后面的章节中将讲述激活它的方法。

编码标准

我们强烈建议你在开发自定义模块时遵循 Drupal 的编码标准。这样使代码更易阅读和理解，并且它是 Drupal 核心的最佳建议。

12.4 创建模块信息文件.info.yml

信息文件.info.yml 是 Drupal 8 模块、主题、配置文件必须的，它提供了基本信息描述。主要包括模块、主题扩展信息，为用户管理界面提供信息，版本兼容性，以及其它的一些信息。

Hello World 模块的信息文件

在 HelloWorld 模块根目录下创建一个 hello_world.info.yml 的文本文件，并输入以下内容:

```
name: Hello World Module
description:  Creates a page showing "Hello World"。
package: Custom
type: module
core: 8.x
```

前三行用于描述模块名、模块描述、所属包(组)，它们将会显示在 Drupal 8 的模块管理页面。name 键指定显示在模块管理页面的名称，description 键批定显示在模块管理页面的描述，package 键允许对模块进行分组。例如 Core 包是由 Drupal 8 核心提供的模块。我们这里使用 package:Custom 表明此模块属于自定义模块包。

Drupal 8 新增了 type 键，其作用是指定扩展的类型，可能值为 module、theme 或 profile。

对于托管于 Drupal.org 上的模块，其 version 键将于包管理脚本自动填充，你不需要手工指定。

core 键是必须的，它指出了模块兼容的 Drupal 核心版本号。

下面提供了一个更加详细的例子，它包括了更多的属性:

```
name: Hello World Module
description: Creates a page showing "Hello World"
package: Custom
type: module
core: 8.x
dependencies:
- datetime: datetime
- link: link
- drupal: views
test_dependencies:
  - drupal: image

configure: hello_world.settings
hidden: true

# Note: do not add the 'version' property yourself!
# It will be added automatically by the packager on drupal.org
```

www.drupalc.com

version:1.0

dependencies:列出你的模块依赖的模块。其格式为{project}:{module}，{project}是出现在 Drupal.org 项目 url 中的项目名称，例如 drupal.org/project/views，{module}是模块的机器名。

test_dependencies:在被添加为你的模块的依赖关系的过程中的其它模块的列表。这允许对依赖关系进行测试。一般地，你应该尽可能地提交 test_dependecies，testbot 会对整个开发过程的依赖关系进行测试。依赖关系是正确的、完整的，你应该将其移动 dependencies 中。在上面的例子中，在测试期间 testbot 会包含 image 模块。但站点安装"Hello World Module"模块时不会要求安装 image 模块。测试依赖的命名规则应该与依赖的命名规则相同。

configure:如果你的模块提供了一个配置表单，你就可能为这个表单指定一个路由。当用户展开模块细节时，它将会在 Extend 页(/admin/modules)显示一个链接。

hidden:true 在站点 Extend 页面(/admin/modules)的模块列表中隐藏你的模块。如果模块仅仅包含一个测试或只是作为一个开发例子实现模块主要的 API，你发现使用这个键是非常有用的。你可以在 settings.php 文件中添加 $settings['extension_discovery_scan_tests'] = TRUE 以显示模块。

调试.info.yml 文件

模块没有在 admin/modules 列出。
确保信息文件按{machine_name}.info.yml 命名并且置于模块根目录下。
确保文件中有下面这一行
type: module
确保模块名以字母或下划线开始。
模块在/admin/modules 但 checkbox 已禁用
确保模块的核心兼容 8.x
core: 8.x
确保所有的模块可用。你能展开模块详细信息以查看模块依赖关系。
模块描述为空
记住 description 键是必须的。
description:Example Module description。

12.5 添加一个composer.json文件

可以向模块添加一个 composer.json 文件以定义项目，在 composer.json 文件中可以定义外部依赖关系。

www.drupalc.com

定义模块包

PHP 社区使用 Composer 管理包；Drupal 也是这样做的。举一个例子，Drupal 项目在"drupal/core"包中有一个依赖，"drupal/core"包定义了一个'drupal-core'类型，Composer 就知道它要做的事。Composer/installers 库定义了一些 Drupal 类型。它们是:

drupal-module
drupal-theme
drupal-library
drupal-profile
drupal-drush

你的模块应该包含一个 composer.json 文件，用它来最小化地定义模块信息。如下面的例子:

```
{
"name": "drupal/example",
"description": "This is an example composer.json for example module",
"type": "drupal-module",
"license": "GPL-2.0+"
}
```

下面是一个更复杂的例子:

```
{
  "name": "drupal/mobile_detect",
   "description": "Mobile_Detect is a lightweight PHP class for detecting mobile
devices.",
   "type": "drupal-module",
   "homepage": "https://drupal.org/project/mobile_detect",
   "authors": [
     {
       "name": "Matthew Donadio (mpdonadio)",
       "homepage": "https://www.drupal.org/u/mpdonadio",
       "role": "Maintainer"
     },
     {
       "name": "Darryl Norris (darol100)",
       "email": "admin@darrylnorris.com",
       "homepage": "https://www.drupal.org/u/darol100",
       "role": "Co-maintainer"
     }
   ],
   "support": {
     "issues": "https://drupal.org/project/issues/mobile_detect",
```

www.drupalc.com

```
    "irc": "irc://irc.freenode.org/drupal-contribute",
    "source": "https://cgit.drupalcode.org/mobile_detect"
  },
  "license": "GPL-2.0+",
  "minimum-stability": "dev",
  "require": {
    "mobiledetect/mobiledetectlib": "~2.8"
  }
}
```

为了给你的包命名，你应该遵循 Composer 包命名规则。

在 composer.json 中定义一个依赖

你可以在 composer.json 文件中为模块定义外部依赖。Drupal 核心不会自动地发现并管理这些依赖。为了利用 composer.json 文件中定义的依赖，你需要使用以下维护方式:
使用 composer 安装 Drupal 核心以及模块
在 Drupal 安装的根目录下手工修改 composer.json 文件

www.drupalc.com

12.6 Hello World自定义页面模块

本节将指导你创建一个简单模块。大多数编程语言的第一个程序是向屏幕输出"hello world"字符串。为了学习 Drupal 的模块编写，这里也以一个 hello world 作为例子。

在编写这个模块之前，先回忆一下上节的知识，我们应先准备一个模块框架。在创建好基本的目录和模块信息文件后才能进入下一步。

12.6.1 添加一个基本控制器

为了响应 hello world 模块页面的请求，你需要添加一个控制器，用来告诉当请求到来时，模块要做什么事。

在你模块目录中，你应该创建符合 PSR-4 标准的目录结构/src/Controller，并在该目录下创建控制器文件 HelloController.php。我们这个模块只是想输出 hello world 这样的字符串，需要在/src/Controller/HelloController.php 文件中输入以下代码:

```
<?php
/**
 * @file
 * Contains \Drupal\hello_world\Controller\HelloController.
 */

namespace Drupal\hello_world\Controller;

use Drupal\Core\Controller\ControllerBase;
class HelloController extends ControllerBase {
  public function content() {
    return array(
        '#type' => 'markup',
        '#markup' => $this->t('Hello, World!'),
    );
  }
}
```

这段代码本身并不会做任何事，它需要一个路由文件来唤醒它。接下来我们给模块添加一个路由文件以激活这个控制器。

www.drupalc.com

12.6.2 添加一个路由文件

回到你的模块根目录，这里已经有一个模块信息文件，现在添加一个名为 hello_world.routing.yml 文件，并输入路由定义如下:

```
hello_world.content:
   path: '/hello'
   defaults:
     _controller: '\Drupal\hello_world\Controller\HelloController::content'
     _title: 'Hello World'
   requirements:
     _permission: 'access content'
```

请注意文件格式，首行 hello_world.content 之前应该留有空格。虽然在这里不一定非要使用模块的机器名，但是为了路由文件与菜单文件保持一致，这里最好使用机器名。hello_world.content 将会在下一节添加菜单链接中用到，它将会链接到路由表中这一项。

如果你已经激活了这个模块，请在 admin/config/development/performance 清空缓存，或使用 Drush 清空缓存(drush cache-rebuild or drush cr)。如果没有激活，请激活它。

现在导航到站点首页，在 url 地址栏输入/hello，你将会看到"Hello,World!"这样的信息。请注意你的 module.routing.yml 文件应该设置适当的缩进，否则将会遇到一问题。

12.6.3 添加一个菜单链接

现在我们已经为模块创建了一个占位符，让我们为它创建菜单链接。本节将指导你给 hello_world 模块创建菜单链接，并在配置页面开发部份显示(admin/config)。

到模块的根目录下，创建一个命为 hello_world.links.menu.yml 的文件，并输入以下内容:

```
 hello_world.admin:
   titke: 'Hello module settings'
   description: 'example of how to make an admin settings page link'
   parent: system.admin_config_development
   route_name: hello_world.content
   weight: 100
```

注意首行空格和适当缩进，第五行的 route_name 应与上一节的路由信息一致。标题和描述将会显示在配置页面的开发部份。parent 键描述的是菜单的父链接菜单。换句话说，这个菜单链接将会在 admin>config>development 下创建。

这将会在管理配置页面添加一个链接，该链接引用 hello_world.content 的路由信息，需要清空缓存已使配置使效。清空缓存后，在管理配置页面开发部份将会看到"Hello module settings"菜单链接，单击这个链接，系统调用 hello_world 模块。

www.drupalc.com

12.7 向自定义模块添加自定义区块

与 Drupal 7 不同的是，在 Drupal 8 中创建区块的一个或多个实例并将它们放置在站点区域中是一件非常容易的事情。

本节讲述如何通过编写代码的方法向区块布局界面添加区块。怎样向区块添加配置表单，以及如何处理表单。最后你会学习如何向表单设置默认值以及区块显示配置。

在编写这个模块之前，请创建好相关的目录及模块信息文件。如果不清楚请查看 12.2 准备一个模块框架

12.7.1 创建自定义区块

区块在 Drupal 8 是区块插件的一个实例。Drupal 8 的区块管理器会扫描模块中包含@Block 注释的任意类。下面是一个使用@Block 注释定义自定义区块的例子:

区块在 Drupal 8 是区块插件的一个实例。Drupal 8 的区块管理器会扫描模块中包含@Block 注释的任意类。下面是一个使用@Block 注释定义自定义区块的例子:

```
<?php

namespace Drupal\hello_world\Plugin\Block;
use Drupal\Core\Block\BlockBase;

/**
 * Provides a 'Hello' Block
 *
 * @Block(
 *   id = "hello_block",
 *   admin_label = @Translation("Hello block"),
 * )
 */

class HelloBlock extends BlockBase {

  /**
   * {@inheritdoc}
   */
  public function build() {
    return array(
      '#markup' => $this->t('Hello, World!'),
    );
  }
}
```

www.drupalc.com

你可以将以上代码复制并粘贴到你模块目录下的 src/Plugin/Block/HelloBlock.php 文件中，并清空缓存。点 Manage>Structure>Block Layout 导航空到区块布局页面 admin/structure/block，并单击一个可用区域的'Place Block'放置区块按钮，系统弹出对话框列出所有可用的区块。搜索并找到'Hello block'。将其配置到站点上。

请注意:类名必须与文件名相同，如果它们不同，区块会被列出，但是不能被添加。

12.7.2 添加区块配置表单

让我们为区块添加一个配置表单，以便于站点创建者实例化区块时对区块进行配置。在 Drupal 8 中配置经常使用，在管理配置页面可以对配置进行导入导出。作为模块创建者应该为区块提供一个默认配置，以便于站点创建者实例化区块时自动填充区块配置。

针对上一节的 HelloBlock 区块类，需要在类所在的文件中添加以下名字空间的引用：

```
use Drupal\Core\Block\BlockPluginInterface;
use Drupal\Core\Form\FormStateInterface;
```

更新类申请以包含新的"implements BlockPluginInterface"语句:

```
class HelloBlock extends BlockBase implements BlockPluginInterface{
```

然后给类添加以下方法。下面的代码仅仅添加一个表单，表单处理和保存结果将会在下一节讲述。

```
/**
   * {@inheritdoc}
   */
  public function blockForm($form, FormStateInterface $form_state) {
    $form = parent::blockForm($form, $form_state);

    $config = $this->getConfiguration();

    $form['hello_block_name'] = array (
      '#type' => 'textfield',
      '#title' => $this->t('Who'),
      '#description' => $this->t('Who do you want to say hello to?'),
      '#default_value' => isset($config['name']) ? $config['name'] : '',
    );

    return $form;
  }
```

www.drupalc.com

在这个例子中，表单首先通过$form = parent::blockForm($form,$form_state)这名句引用它的父类的表单定义；接下来，我们向表单添加了一个新的字段。这个处理过程使用了多态，它是面向对象编程(OOP)中很重要的一种技术。

请到区块布局页面(Manage>Structure>Block Layout)单击该区块旁的配置按钮来查看这个表单。

12.7.3 处理区块配置表单

上一节我们已经创建了区块的配置表单，我们需要处理表单，为此需要在 HelloBlock 类中添加以下方法，打开 src/Plugin/Block/HelloBlack.php 文件，输入以下代码:

```
  /**
   * {@inheritdoc}
   */
  public function blockSubmit($form, FormStateInterface $form_state) {
    $this->setConfigurationValue('name',
$form_state->getValue('hello_block_name'));
  }
```

如果恭迎使用字段集包围表单元素，你应该向 getValue()函数传递一个数组而不是传递字段名。这里 myfieldset 是包围 hello_block_name 字段的字段集。则代码如下:

```
  $this->setConfigurationValue('name',
$form_state->getValue(array('myfieldset', 'hello_block_name')));
```

添加这段代码意味着区块表单将会被处理，输入到表单的值会被存储到区块的实例配置中，它是独立于区块的其它实例的。这个区块还没有使用区块配置的更改。下一节将介绍应用这个设置改变。

12.7.4 在区块显示中使用配置

为了应用区块配置，我们需要修改 HelloBlock 类中的 build()方法，在 build()方法中首先加载配置，然后返回标签。

```
public function build() {
    $config = $this->getConfiguration();
```

www.drupalc.com

```
    if (!empty($config['name'])) {
      $name = $config['name'];
    }
    else {
      $name = $this->t('to no one');
    }
    return array(
      '#markup' => $this->t('Hello @name!', array (
          '@name' => $name,
        )
      ),
    );
  }
```

12.7.5 添加一个默认配置

向模块添加一个简单的配置 yml 文件，Drupal 将会自动地加载配置文件的内容，我们能文访问它以提供一个默认配置。在模块根目录下，创建一个命为'config'的目录，在'config'目录下创建'install'目录。最后在 config/install 目录中创建一个名为'hello_world.settings.yml'的文件。并输入:

```
 hello:
   name: 'Hank Willimas'
```

请记住 yaml 文件是空格敏感的，注意空格和缩进。为了 Drupal 对象加载这个值，我们需要给 HelloBlock 类添加以下方法:

```
/**
   * {@inheritdoc}
   */
  public function defaultConfiguration() {
    $default_config = \Drupal::config('hello_world.settings');
    return array(
      'name' => $default_config->get('hello.name')
    );
  }
```

当模块安装时这个值被使用。当你在区域中又添加这个区块的实例时，你将看到这个默认值。

www.drupalc.com

12.8 在Drupal 8 模块中包含缺省的配置

概述

在 Drupal 7 中，诸如内容类型、字段配置、视图等模块的默认配置，需要在安装和更新时自定义 PHP 代码。在 Drupal 8 中，这些是存储在配置文件(YAML 文件)中的。

例如，管理配置系统需要创建和管理内容类型。通过创建一个正确命名的结构化配置文件，你可以在自定义模块中定义内容类型。

实例

在你的模块目录下创建 config/install 子目录，并在其下创建一个名为 node.type.example_mytype.yml 的文件。对于本例它是 /modules/example/config/install/node.type.example_mytype.yml，模块目录为 /module/example。

请注意命名规则，使用下划线连接你的模块名和内容类型，以防止与其它配置文件冲突。

在这个文件中输入以下代码:

```
type: example_mytype
name: Example
description: 'Use <em>example</em> content to get to Drupal 8 development better.'
help: ''
new_revision: false
display_submitted: true
preview_mode: 1
status: true
langcode: en
```

你可以为其它如放置区块、视图、文本格式、编辑器设置、用户角色等设置默认配置，这和你的模块设置是一样的。你可以为表单字段设置默认值。

如果你在添加内容类型以前已安装了模块，你需要反安装模块并重新安装它，以使模块定义的内容类型生效。

Drupal Console

www.drupalc.com

Drupal Console 实用程序提供命令行完成配置的导入导出，它可以导出与指定内容类型的配置文件包括字段的配置文件，放置配置文件到想要放置的模块配置目录，删除所有出问题的 UUID 行。

12.9 创建自定义内容和配置实体

本节将讲述如何在 Drupal8 中定义实体类型和创建节点实体。这里创建一个实体类型 FooType，它将是内容实体 Foo 的实体类型。Foo 内容实体支持修订和翻译。

首先应在模块中创建 foo.info.yml 文件如下:

```
name: Foo
description: Node-like entity example for Drupal 8
type: module
core: 8.x
version: VERSION
```

然后，我们创建 foo.install 文件，用于定义数据库结构，并通过 hook_install() 自动创建名为 generic 的 FooType 实体类型，它将作为 Foo 内容实体的类型。

```
/**
 * Implements hook_schema().
 */
function foo_schema() {
  $schema = array();

  // In Drupal 8 we'll need 4 tables in order to support revisions and
translations.

  // You can find more information about this here:
https://drupal.org/node/1722906

  // The {entity} main table

  $schema['foo'] = array(
    'description' => 'Foo entity base table.',
    'fields' => array(
      'entity_id' => array(
        'description' => 'The primary entity identifier.',
        'type' => 'serial',
        'not null' => TRUE,
        'unsigned' => TRUE
```

www.drupalc.com

```
      ),
      'uuid' => array(
        'description' => 'The unique entity identifier.',
        'type' => 'varchar',
        'length' => 60,
        'not null' => TRUE
      ),

      // Defaults to NULL in order to avoid a brief period of potential
      // deadlocks on the index.
      'revision_id' => array(
        'description' => 'The active revision identificator.',
        'type' => 'int',
        'size' => 'normal',
        'not null' => FALSE,
        'unsigned' => TRUE,
        'default' => NULL
      ),
      'bundle' => array(
        'description' => 'The {foo_type}.type of this entity.',
        'type' => 'varchar',
        'length' => 48,
        'not null' => TRUE
      )
    ),
    'primary key' => array('entity_id'),
    'unique keys' => array(
      'uuid' => array('uuid'),
      'revision_id' => array('revision_id')
    )
  );

  // The {entity_revision} table
  $schema['foo_revision'] = array(
    'description' => 'Foo entity revision table.',
    'fields' => array(
      'entity_id' => array(
        'description' => 'The primary entity identifier.',
        'type' => 'int',
        'not null' => TRUE,
        'unsigned' => TRUE
      ),
      'revision_id' => array(
        'description' => 'The revision identifier.',
```

www.drupalc.com

```
      'type' => 'serial',
      'not null' => TRUE,
      'unsigned' => TRUE
    ),
    'langcode' => array(
      'description' => 'The language code the entity was created in.',
      'type' => 'varchar',
      'length' => 32,
      'not null' => TRUE
    ),
    'revision_uid' => array(
      'description' => 'The {users}.uid that created this version.',
      'type' => 'int',
      'not null' => TRUE,
      'unsigned' => TRUE
    ),
    'revision_timestamp' => array(
      'description' => 'The Unix timestamp when the version was created.',
      'type' => 'int',
      'not null' => TRUE,
      'unsigned' => TRUE
    )
  ),
  'primary key' => array('revision_id')
);

// The {entity_field_data} table for Foo's own, non-FieldAPI, fields
// holding the current/active revision's data.
$schema['foo_field_data'] = array(
  'description' => 'Foo entity field data.',
  'fields' => array(
    'entity_id' => array(
      'description' => 'The primary entity identifier.',
      'type' => 'int',
      'not null' => TRUE,
      'unsigned' => TRUE
    ),
    'revision_id' => array(
      'description' => 'The revision identifier.',
      'type' => 'int',
      'not null' => TRUE,
      'unsigned' => TRUE,
    ),
    'langcode' => array(
```

www.drupalc.com

```
      'description' => 'The language code the entity was created in.',
      'type' => 'varchar',
      'length' => 32,
      'not null' => TRUE
    ),
    'default_langcode' => array(
      'description' => 'Boolean indicating if the entry holds values for the
original language of the entity.',
      'type' => 'int',
      'size' => 'tiny',
      'not null' => TRUE
    ),
    'bundle' => array(
      'description' => 'The {foo_type}.type of this entity.',
      'type' => 'varchar',
      'length' => 48,
      'not null' => TRUE
    ),
    // Custom entity fields from now on
    'label' => array(
      'description' => 'The entity label.',
      'type' => 'varchar',
      'length' => 255,
      'not null' => TRUE
    ),
    'uid' => array(
      'description' => 'The entity author identificator.',
      'type' => 'int',
      'size' => 'normal',
      'not null' => TRUE,
      'unsigned' => TRUE
    ),
    'published' => array(
      'description' => 'The status of this entity. 0 - disabled, 1 - enabled',
      'type' => 'int',
      'size' => 'tiny',
      'not null' => TRUE
    ),
    'created' => array(
      'description' => 'The UNIX timestamp when the entity was created.',
      'type' => 'int',
      'size' => 'normal',
      'not null' => TRUE,
      'unsigned' => TRUE
```

www.drupalc.com

```
      ),
      'changed' => array(
        'description' => 'The UNIX timestamp when the entity was updated.',
        'type' => 'int',
        'size' => 'normal',
        'not null' => TRUE,
        'unsigned' => TRUE
      )
    ),
    'primary key' => array('entity_id', 'langcode'),
    'indexes' => array(
      'foo_revision' => array('revision_id')
    )
  );

  // The {entity_field_revision} table for Foo's own, non-FieldAPI, fields
  // holding the data of all existing revisions.
  $schema['foo_field_revision'] = array(
    'description' => 'Foo entity field data.',
    'fields' => array(
      'entity_id' => array(
        'description' => 'The primary entity identifier.',
        'type' => 'int',
        'not null' => TRUE,
        'unsigned' => TRUE
      ),
      'revision_id' => array(
        'description' => 'The revision identifier.',
        'type' => 'int',
        'not null' => TRUE,
        'unsigned' => TRUE,
      ),
      'langcode' => array(
        'description' => 'The language code the entity was created in.',
        'type' => 'varchar',
        'length' => 32,
        'not null' => TRUE
      ),
      'default_langcode' => array(
        'description' => 'Boolean indicating if the entry holds values for the
original language of the entity.',
        'type' => 'int',
        'size' => 'tiny',
        'not null' => TRUE
```

www.drupalc.com

```
        ),
        // Custom entity fields from now on
        'label' => array(
          'description' => 'The product label.',
          'type' => 'varchar',
          'length' => 255,
          'not null' => TRUE
        ),
        'uid' => array(
          'description' => 'The entity author identificator.',
          'type' => 'int',
          'size' => 'normal',
          'not null' => TRUE,
          'unsigned' => TRUE
        ),
        'published' => array(
          'description' => 'The status of this entity. 0 - disabled, 1 - enabled',
          'type' => 'int',
          'size' => 'tiny',
          'not null' => TRUE
        ),
        'created' => array(
          'description' => 'The UNIX timestamp when the entity was created.',
          'type' => 'int',
          'size' => 'normal',
          'not null' => TRUE,
          'unsigned' => TRUE
        ),
        'changed' => array(
          'description' => 'The UNIX timestamp when the entity was updated.',
          'type' => 'int',
          'size' => 'normal',
          'not null' => TRUE,
          'unsigned' => TRUE
        )
      ),
      'primary key' => array('revision_id', 'langcode')
  );
  return $schema;
}

/**
 * Implements hook_install().
 */
```

www.drupalc.com

```
function foo_install() {
  // Creates the Generic FooType entity
  entity_create('foo_type', array(
    'label' => 'Generic Foo type',
    'type' => 'generic',
    'description' => 'A very informative text goes here.',
    'settings' => array('published' => 0)
  ))->save();
}
```

定义模块权限

/modules/foo/foo.permissions.yml

```
administer entity:
  title: 'Administer Foo entity'
  description: 'Allows user to administer Foo entities.'
administer types:
  title: 'Administer Foo types'
  description: 'Allows user to administer Foo types.'
```

/modules/foo/foo.module

```
use Drupal\Core\Language\Language;
use Drupal\Core\Cache\CacheBackendInterface;

/**
* Menu argument loader: Loads a Foo type by string.
*
* @param $name
*    The machine name of a Foo type to load.
*
* @return \Drupal\foo\Entity\FooTypeInterface
*    A Foo type object or NULL if $name does not exist.
*/
function foo_type_load($name) {
  return entity_load('foo_type', $name);
}

/**
 * Loads a Foo entity from the database.
 *
 * @param int $id
```

www.drupalc.com

```
 *   The Foo entity ID.
 * @param bool $reset
 *   (optional) Whether to reset the static cache. Defaults to
 *   FALSE.
 *
 * @return \Drupal\foo\Entity\FooInterface|null
 *   A fully-populated Foo entity, or NULL if the entity is not found.
 */
function foo_load($id, $reset = FALSE) {
  return entity_load('foo', $id, $reset);
}

/**
 * Implements hook_entity_bundle_info().
 */
function foo_entity_bundle_info() {
  $bundles = array();
  // Bundles must provide a human readable name so we can create help and
error
  // messages.
  foreach (foo_type_get_names() as $id => $label) {
    $bundles['foo'][$id]['label'] = $label;
  }
  return $bundles;
}

/**
 * Returns a list of available Foo type names.
 *
 * This list can include types that are queued for addition or deletion.
 *
 * @return array
 *   An array of Foo type labels, keyed by the Foo type name.
 */
function foo_type_get_names() {
  $cid = 'foo_type:names:' . language(Language::TYPE_INTERFACE)->id; //
this name is completly custom
  if ($cache = cache()->get($cid)) {
    return $cache->data;
  }
  // Not using foo_type_get_types() or entity_load_multiple() here, to allow
  // this function being used in hook_entity_info() implementations.
  // @todo Consider to convert this into a generic config entity helper.
```

www.drupalc.com

```
  $config_names = config_get_storage_names_with_prefix('foo.type.'); // the
'foo.type' is defined in Foo entiy definition
  $names = array();
  foreach ($config_names as $config_name) {
    $config = \Drupal::config($config_name);
    $names[$config->get('type')] = $config->get('name');
  }
  cache()->set($cid, $names, CacheBackendInterface::CACHE_PERMANENT,
array(
    'foo_type' => array_keys($names),
    'foo_types' => TRUE,
  ));
  return $names;
}

/**
 * Returns a list of all the available foo types.
 *
 * This list can include types that are queued for addition or deletion.
 *
 * @return array
 *   An array of Foo type entities, keyed by ID.
 *
 * @see foo_type_load()
 */
function foo_type_get_types() {
  return entity_load_multiple('foo_type');
}
```

定义模块路由处理信息

```
/modules/foo/foo.routing.yml
foo.overview:
  path: '/admin/structure/foo'
  defaults:
    _entity_list: 'foo'
    _title: 'Foo'
  requirements:
    _permission: 'administer foo'

foo.type_list:
  path: '/admin/structure/foo/types'
  defaults:
    _entity_list: 'foo_type'
```

www.drupalc.com

```
    _title: 'Foo Type Configuration'
  requirements:
    _permission: 'administer foo'

foo.type_add:
  path: '/admin/structure/foo/types/add'
  defaults:
    _entity_form: foo_type.add
    _title: 'Add foo type'
  requirements:
    _permission: 'administer foo'

foo.type_edit:
  path: '/admin/structure/foo/types/{foo_type}'
  defaults:
    _entity_form: foo_type.edit
    _title: 'Edit foo type'
  requirements:
    _permission: 'administer foo'

foo.type_delete:
  path: '/admin/structure/foo/types/{foo_type}/delete'
  defaults:
    _entity_form: foo_type.delete
    _title: 'Delete foo type'
  requirements:
    _permission: 'administer foo'

foo.view:
  path: '/foo/{foo}'
  defaults:
    _content: '\Drupal\foo\Controller\FooController::view'
  _title_callback: '\Drupal\foo\Controller\FooController::getLabel'
  requirements:
    _entity_access: 'foo.view'
    foo: \d+

foo.edit:
  path: '/foo/{foo}/edit'
  defaults:
    _entity_form: foo.default
    _title: 'Edit foo'
  requirements:
    _permission: 'administer foo'
```

www.drupalc.com

```
    foo: \d+

foo.delete:
  path: '/foo/{foo}/delete'
  defaults:
    _entity_form: foo.delete
    _title: 'Delete foo'
  requirements:
    _permission: 'administer foo'
    foo: \d+

foo.type_select:
  path: '/admin/structure/foo/add'
  _content: '\Drupal\foo\Controller\FooController::add'
  _title: 'Add foo'
  requirements:
    _permission: 'administer foo'

foo.add:
  path: '/admin/structure/foo/add/{foo_type}'
  defaults:
    _entity_form: foo_type.default
    _title: 'Add foo'
  requirements:
    _permission: 'administer foo'
```

www.drupalc.com

12.10 在Drupal 8 中定义和使用配置

你能够在你的模块中包含其它模块定义的实体如节点类型(node types)，视图(views)，文本格式(text formats)等。举一个例子，节点模块提供节点类型的配置，因此在你的模块中，你可以定义一个节点类型。你可以为你自己的插件、实体、设置等定义配置，并且你定义的配置能被其它模块使用，正如你使用节点类型配置一样。在 Drupal 8 中定义一段配置十分简单。

配置文件

配置文件位于你模块的 config/install 目录下，例如你的模块目录为 /modules/example，则配置文件的路径为 /modules/example/config/install/example.config.yml，配置文件使用 YAML 文件格式编写。

配置文件的名称遵循'配置名.settings.yml'这样的格式，当然这并不是必须的，但我们强烈建议你按照上面的格式命名，这样可以提高可读性，避免混淆发生。最好不要使用诸如 settings.yml 或 system.settings.yml 这样一般性的名称，一旦系统比较庞大时，到处都是这样的文件，就会产生混淆。配置文件(.yml 文件)名总是可以在系统中通过配置名进行调用，并且可以使用 PHP API 将它们联系到一起。

配置文件的结构

配置文件使用 YAML 文件格式。你可以根据你自己的需要来结构化配置文件，这里并没有按照 YAML 格式进行严格限制。例如，如果你需要一个设置来从页面控制器(page controller)输出内容，你可以在设置中包含一个 message 键，并提供一个字符串值:

```
message: 'Hello'
langcode: 'en'
```

在这里使用 langcode 键的目的是使该信息能被语言系统识别和可翻译，这是一种很好的方式。Langcode 键就是这样一个作用，因此你不能在配置文件中将其用于其它作用。

为了使信息能被翻译，你需要包含更多的文件:

/modules/example/config/schema/example.schema.yml
/modules/example/example.config_translation.yml

第一个文件让你定义配置中哪些可被翻译:

```
example.config:
  type: config_object
  label: 'Example config'
  mapping:
    message:
      type: text
      label: 'Message'
```

第二个文件在配置翻译(/admin/config/regional/config-translation)中添加链接以响应翻译表单：

```
hello.config:
  title: 'Example Translatable config'
  base_route_name: example.admin.config
  names:
    - example.config
```

www.drupalc.com

example.admin.config 是你模块的管理配置表单的路由名称。

这个文件可以在树形结构中包含更加复杂的键值对，你可以打开核心模块目录看一看一些复杂的配置文件例子。

使用配置

Drupal 8 带有一个 PHP API 用于读写配置。最简单的方式是使用 Drupal::config()这一静态方法:

```
$config = \Drupal::config('example.settings');
// Will print 'Hello'.
print $config->get('message');
// Will print 'en'.
print $config->get('langcode');
```

如果你想编辑或更新配置，你可以使用
\Drupal::service('config.factorly')->getEditable()方法

```
$config = \Drupal::service('config.factory')->getEditable('example.settings');
// Set and save new message value.
$config->set('message', 'Hi')->save();
// Now will print 'Hi'.
print $config->get('message');
```

www.drupalc.com

12.11 创建自定义字段

本节将会讲述在模块中如何创建自定义字段，包括创建自定义字段类型，创建自定义字段格式化器，创建自定义字段部件三个方面内容。

这些任务是通过实现字段模块的插件来实现的。先来看看实现以上三个方面内容的目录结构的例子:

```
foo_bar_fields
   foo_bar_fields.info.yml
   src/
     Plugin/
        Field/
           FieldType/
              FooItem.php
           FieldFormatter/
              FooBarFormatter.php
```

```
          FieldWidget/
             FooBarWidget.php
```

下面内容我们将会创建一个自定义字段类型，字段格式化器，字段部件，用以产生随机字符串并显示它。

模块命名为 random 并且包含下面的结构:

```
random
   random.info.yml
   src/
      Plugin/
         Field/
            FieldType/
               RandomItem.php
            FieldFormatter/
               RandomDefaultFormatter.php
            FieldWidget/
               RandomDefaultWidget.php
```

www.drupalc.com

12.11.1 创建自定义字段类型

字段类型用于定义字段的属性和行为，在 Drupal 8 中创建字段类型需要定义一个带有字段类型通知的类。它应位于 MODULE_NAME/src/Plugin/Field/FieldType 下，如/modules/foo/src/Plugin/Field/Field/FieldType/BazItem.php，该类的命名空间为\Drupal\MODULE_NAME\Plugin\Field\FieldType，\Drupal\foo\Plugin\Field\FieldType\BazItem 类通知在文档注释中定义，应包含唯一 id 和一个 label。

例如:

```
/**
* Provides a field type of baz
*
* @FieldType(
*    id = "baz",
*    label = @Translation("Baz field"),
* )
*/
```

该类需要实现 FieldItemInterface 结口。也可以扩展 FieldItemBase 类并实现该结口。

```
class BazItem extends FieldItemBase implements FieldItemInterface {
  /**
   * {@inheritdoc}
   */
  public static function schema(FieldStorageDefinitionInterface
$field_definition) {
    return array(
      'columns' => array(
        'value' => array(
          'type' => 'text',
          'size' => 'tiny',
          'not null' => FALSE,
        ),
      ),
    );
  }

  /**
   * {@inheritdoc}
   */
  public static function propertyDefinitions(FieldStorageDefinitionInterface
$field_definition) {
    $properties['value'] = DataDefinition::create('string');
    return $properties;
  }
}

/**
* @file
* Contains Drupal\field_example\Plugin\Field\FieldType\RgbItem.
*/
namespace Drupal\field_example\Plugin\Field\FieldType;
use Drupal\Core\Field\FieldItemBase;
use Drupal\Core\Field\FieldDefinitionInterface;
use Drupal\Core\Field\FieldStorageDefinitionInterface;
use Drupal\Core\TypedData\DataDefinition;

/**
* Plugin implementation of the 'field_example_rgb' field type.
*
* @FieldType(
*   id = "field_example_rgb",
*   label = @Translation("Example Color RGB"),
*   module = "field_example",
*   description = @Translation("Demonstrates a field composed of an RGB
```

www.drupalc.com

```
color."),
*   default_widget = "field_example_text",
*   default_formatter = "field_example_simple_text"
* )
*/

class RgbItem extends FieldItemBase {
  /**
   * {@inheritdoc}
   */
  public static function schema(FieldStorageDefinitionInterface
$field_definition) {
    return array(
      'columns' => array(
        'value' => array(
          'type' => 'text',
          'size' => 'tiny',
          'not null' => FALSE,
        ),
      ),
    );
  }

  /**
   * {@inheritdoc}
   */
  public function isEmpty() {
    $value = $this->get('value')->getValue();
    return $value === NULL || $value === '';
  }

  /**
   * {@inheritdoc}
   */
  public static function propertyDefinitions(FieldStorageDefinitionInterface
$field_definition) {
    $properties['value'] = DataDefinition::create('string')
      ->setLabel(t('Hex value'));
    return $properties;
  }
}
```

www.drupalc.com

12.11.2 创建自定义字段格式化器

字段格式化器格式化字段数据以向最终用户显示。要实现这一功能需定义字段格式化器类。

下面是例子

```
<?php
/**
* @file
* Contains
\Drupal\Random\Plugin\Field\FieldFormatter\RandomDefaultFormatter.
*/
namespace Drupal\random\Plugin\Field\FieldFormatter;
use Drupal\Core\Field\FormatterBase;
use Drupal\Core\Field\FieldItemListInterface;
/**
* Plugin implementation of the 'Random_default' formatter.
*
* @FieldFormatter(
*   id = "Random_default",
*   label = @Translation("Random text"),
*   field_types = {
*     "Random"
*   }
* )
*/
class RandomDefaultFormatter extends FormatterBase {
  /**
   * {@inheritdoc}
   */
  public function settingsSummary() {
    $summary = array();
    $settings = $this->getSettings();
    $summary[] = t('Displays the random string.');
    return $summary;
  }
  /**
   * {@inheritdoc}
   */
  public function viewElements(FieldItemListInterface $items, $langcode) {
    $element = array();
    foreach ($items as $delta => $item) {
      // Render each element as markup.
```

www.drupalc.com

```
      $element[$delta] = array(
        '#type' => 'markup',
        '#markup' => $item->value,
      );
    }
    return $element;
  }
}
```

上面的代码定义格式字段的类，这个类的基类是 FormatterBase，在这个类中实现了两个方法，一个是 settingsSummary，为字段显示一个摘要。另一个是 viewElements 方法，主要作用是设置渲染数组。

12.11.3 创建自定义字段部件

字段部件用于在表单内部渲染字段。

在 Drupal 8 中创建一个字段部件，你需要创建一个带有 FieldWidget annotation 的类。该类位于 MODULE_NAME/src/Plugin/Field/FieldWidget，例如，/modules/foo/src/Plugin/Field/FieldWidget/BarWidget.php。其命名空间为 \Drupal\MODULE_NAME\Plugin\Field\FieldWidget，例如 \Drupal\foo\Plugin\Field\BarWidget。

类通知在注释中定义应该包含唯一 id，label 和字段类型数组。

例子:

```
/**
 * @FieldWidget(
 *   id = "bar",
 *   label = @Translation("Bar widget"),
 *   field_types = {
 *     "baz",
 *     "string"
 *   }
 * )
 */
```

该类需要实现 WidgeInterface 接口，也可以扩展 WidgeBase 类并实现 WidgeInterface 接口。

www.drupalc.com

```
class BarWidget extends WidgetBase implements WidgetInterface {
  public function formElement(FieldItemListInterface $items, $delta, array
$element, array &$form, FormStateInterface $form_state) {
    $element = [];
    // Build the element render array.
    return $element;
  }
}
```

可以使用下面的代码实现字段部件设置。

```
class BarWidget extends WidgetBase implements WidgetInterface {
  public static function defaultSettings() {
    return array(
      'size' => 60,
      'placeholder' => '',
    ) + parent::defaultSettings();
  }

public function settingsForm(array $form, FormStateInterface $form_state) {
    $element['size'] = array(
      '#type' => 'number',
      '#title' => t('Size of textfield'),
      '#default_value' => $this->getSetting('size'),
      '#required' => TRUE,
      '#min' => 1,
    );
    $element['placeholder'] = array(
      '#type' => 'textfield',
      '#title' => t('Placeholder'),
      '#default_value' => $this->getSetting('placeholder'),
      '#description' => t('Text that will be shown inside the field until a value is
entered. This hint is usually a sample value or a brief description of the
expected format.'),
    );
    return $element;
  }

  public function settingsSummary() {
    $summary = array();
    $summary[] = t('Textfield size: @size', array('@size' =>
$this->getSetting('size')));
    $placeholder = $this->getSetting('placeholder');
    if (!empty($placeholder)) {
```

www.drupalc.com

```
      $summary[] = t('Placeholder: @placeholder', array('@placeholder' =>
$placeholder));
    }

    return $summary;
  }
}
```

上面的代码定义了 BarWidget 类，该类扩展至 WidgetBase 基类，并实现了 WidgetInterface 接口，在该类中实现了部件的缺省设置 defaultSettings()。并定义了设置表单，该表单定义了元素的尺寸和占位符。

使用 getSetting(settings_name)方法获取设置的值。

```
class BarWidget extends WidgetBase implements WidgetInterface {
  public function formElement(FieldItemListInterface $items, $delta, array
$element, array &$form, FormStateInterface $form_state) {
    $element['value'] = $element + array(
      '#type' => 'textfield',
      '#default_value' => isset($items[$delta]->value) ?
$items[$delta]->value : NULL,
      '#size' => $this->getSetting('size'),
      '#placeholder' => $this->getSetting('placeholder'),
      '#maxlength' => $this->getFieldSetting('max_length'),
      '#attributes' => array('class' => array('js-text-full', 'text-full')),
    );

    return $element;
  }
}
```

下面代码定义 TextWidget 类，该类扩展至 WidgetBase 类。该类实现了两个方法，一个是 formElement 定义一个表单元素。另一个是 validate 是一个验证方法，用于验证文本字段的值是否是一个有效的颜色值。

```
/**
* @file
* Contains \Drupal\field_example\Plugin\Field\FieldWidget\TextWidget.
*/

namespace Drupal\field_example\Plugin\Field\FieldWidget;

use Drupal\Core\Field\FieldItemListInterface;
use Drupal\Core\Field\WidgetBase;
use Drupal\Core\Form\FormStateInterface;
```

```
/**
* Plugin implementation of the 'field_example_text' widget.
*
* @FieldWidget(
*   id = "field_example_text",
*   module = "field_example",
*   label = @Translation("RGB value as #ffffff"),
*   field_types = {
*     "field_example_rgb"
*   }
* )
*/

class TextWidget extends WidgetBase {

  /**
   * {@inheritdoc}
   */
  public function formElement(FieldItemListInterface $items, $delta, array
$element, array &$form, FormStateInterface $form_state) {
    $value = isset($items[$delta]->value) ? $items[$delta]->value : '';
    $element += array(
      '#type' => 'textfield',
      '#default_value' => $value,
      '#size' => 7,
      '#maxlength' => 7,
      '#element_validate' => array(
        array($this, 'validate'),
      ),
    );
    return array('value' => $element);
  }

  /**
   * Validate the color text field.
   */
  public function validate($element, FormStateInterface $form_state) {
    $value = $element['#value'];
    if (strlen($value) == 0) {
      $form_state->setValueForElement($element, '');
      return;
    }
    if (!preg_match('/^#([a-f0-9]{6})$/iD', strtolower($value))) {
```

www.drupalc.com

```
      $form_state->setError($element, t("Color must be a 6-digit hexadecimal value, suitable for CSS."));
    }
  }
}
```

12.12 创建自定义页面

在 Drupal 中创建一个简单页面需要两步：

1.定义路径和选项

这一步包含定义页面标题，存取需求以及其它。在 Drupal 7 中，你应该实现 hook_menu()。在 Drupal 8 中，你需要在模块根目录下创建 <module_name>.routing.yml 文件。

2.写代码定义页面内容

在 Drupal 7 中，你需要在 hook_menu()实现页面回调。在 Drupal 8 中，页面回调必须是类的方法或一个已注册的服务。根据不同的条件会有所不同(HTTP vs. HTTPS，content headers,others)，但这超出了本节的范围。

跟随下面这个例子，你应该能够在自定义模块中创建一个简单页面，这不需要学习太多 Drupal 的内部知识。

Example 模块的路由文件

路由文件应该存放于 example/example.routing.yml 中:

```
example.my_page:
  path: '/mypage/page'
  defaults:
    _controller: '\Drupal\example\Controller\ExampleController::myPage'
    _title: 'My first page in D8'
  requirements:
    _permission: 'access content'
```

example.my_page:路由的机器名，一般地它应为 module_name.sub_name.route_name 构成。

path:页面的路径，注意前导符(/)。

defaults:描述页面回调与页面标题。

www.drupalc.com

requirements:页面显示条件。你能定义存取权限，开启模块，以及其它一些条件。

Example 模块的页面控制器实现

该模块的页面控制类定义于 example/src/Controller/ExampleController.php 文件中，具体如下:

```
/**
* @file
* Contains \Drupal\example\Controller\ExampleController.
*/

namespace Drupal\example\Controller;
use Drupal\Core\Controller\ControllerBase;

/** * Provides route responses for the Example module.
*/

class ExampleController extends ControllerBase {

  /**
   * Returns a simple page.
   *
   * @return array
   *   A simple renderable array.
   */
  public function myPage() {
    $element = array(
      '#markup' => 'Hello, world',
    );
    return $element;
  }
}
```

namespace:定义足够的类前缀。编译文件的文档区块与类名，类自动加载就能发现该类。\Drupal\example\Controller\ExampleController，它应该是这个文件 modules/example/src/Controller/ExampleController.php。

use:允许你使用 ControllerBase 替换长长的名字。这使类行更简单易读。

myPage():这个在 YAML 文件中指定的方法必须是 public。它应该返回一个可渲染的数组。

www.drupalc.com

第 13 章 使用配置系统API

配置(configuration)API 用于处理模块的配置数据，像站点名称、口号亦或是更加复杂的数据(视图、内容类型等)都能进行存储、处理。配置数据通常用于在开发站点和线上站点的同步。在站点创建期间产生的配置数据能轻松地导入到服务器站点中。

如果是处理一些不需要在实例之间传递的局部变量，则使用状态(state)API，而不是使用配置。使用状态隐藏系统的值，并且不需要部署在不同的环境。你可以重设系统，删除所有的状态数据，但配置数据仍在。

有两种配置 API，它们是 Config API 和 Configuration Entity API。它们关健的不同是 Config API 是单一实例。单一实例的数据如站点名称。而配置实体API(Configuration Entiy API)是用于处理配置数据集合，如节点类型(node types)、视图(views)、词汇(vocabularies)和字段(fields)。在 Drupal 8 更新主题和模块时更新配置 API 是很重要的，这样会避免错误。

13.1 配置系统概述

在 Drupal 8 中信息可以分为内容(content)、会话(session)、状态(state)、配置(configuration)、缓存(cache)，下面作一下介绍:

内容:向站点用户显示的信息，包括文章、基本页面、图像、文件、广告等。

会话:跟踪用户与站点交互的信息，如当前用户选择的选项，这个信息是短暂的个性化的。

状态:指网站上经常改变的状态信息，如 cron 运行的时间，重建节点权限等。

配置:指站点上非内容又不会经常更改的信息，如站点名，内容类型，视图等。

缓存:是将站点的信息处理后存放于另一个地方，以便于更快的获取数据。缓存不会破坏数据存储，清除缓存，原来的数据也不会丢失。

给你的信息分类

要清楚地界定信息属于以上的哪一类，确实比较困难，下面给出一些建议:

如果信息需要从开发环境布署到生产环境，那它可能使用配置好一些。

考虑站点创建者和站点编辑，如果需要站点编辑修改信息，那它应该是内容。如果仅仅是站点创建者能编辑的信息，可以认为它是配置信息。但这并不是绝对的。

www.drupalc.com

考虑信息数量，如果是大量信息，应该将它视为内容。如果仅仅是少量信息，可考虑使用配置信息。

配置信息趋向于定义事物的类型，如内容类型、分类词汇等。具有某种类型的具体事物应视为内容，如一个内容节点，一个具体的分类等。

简单配置和配置实体

配置信息分两种，一种为简单配置，另一种为配置实体，在使用配置信息前，应判断信息属于哪一种配置信息。

简单配置是最基本也是最简单的配置设置，它可以是一些如布尔值、整数、文本等。例如模块的打开与关闭或站点名称等。简单配置也可以包含你模块需要的任何设置。例如 JS 聚合有开和并的配置。如果它不存在，系统模块无法正确地执行指令。简单配置可以是仅仅由模块提供。例如 system.site 由系统模块提供而非其它模块。

配置实体存储实体的信息集合，实体可以由用户创建和删除，不管它的实例有多少，它都会工作得很好。例如:图像样式、视图、节点类型等。配置实体带有 CRUD hook，这些 hook 可以被其它模块利用，以便于生成它们自已的配置。配置实体可以有动态依赖，正如简配置一样，它们依赖于提供它们的模块。例如，view.view.frontpage 依赖视图模块，因为它要列出一些节点，则它又依赖于节点模块，如果只显示最近更新的文章，则它依赖于配置实体文章类型(node.type.article)。

www.drupal.com

13.2 简单的配置API

这一页介绍使用简单配置 API 读取和设置配置数据。

配置数据

每个模块可以提供默认配置。例如，维护模式设置被定义在 core/modules/system/config/install/system.maintence.yml。这个文件的前面部份是模块的命名空间(这里是 system 模块)，随后是子系统(这里是 maintenance)，这个文件具体内容如下:

message: '@site is currently under maintence. We should be back shortly. Thank you for your patience.'
langcode: en

配置也可以嵌套，例 performance settings(system.performance.yml):

```
cache:
   page:
     enabled: '0'
     max_age: '0'
preprocess:
   css: '0'
   js: '0'
response:
   gzip: '0'
```

注意:根键必须映射

必须对配置的根数据进行映射。例如，如果要为每一个实体类型变量存储数据，则应该给它们添加父键 entity_type 作为根键。如下面例子:

```
entity_type:
  commerce_order:
foo: 'bar'
  node:
foo: 'bar'
  user:
foo: 'bar'
```

如果根键是实体类型，那么就不可以使用 schema 表现配置。另外当配置作为模块或主题的一部份安装时，Drupal 的核心不可能添加额外的信息。

与配置交互

使用 Config 对象与配置文件进行交互，通过调用 config()函数实例化一个 Config 对象，调用 config()函数将会返回一个\Drupal\Core\Config\ImmutableConfig 的实例。

```
//Immutable Config (Read Only)
$config = \Drupal::config('system.performance');
//Mutable Config (Read/Write)
$config=\Drupal::service('config.factory')->getEditable('system.performance')
;
```

一旦你获得了一个 Config 对象，你就可以与配置对象交互。

读取配置

使用 get()方法读取配置。可以使用以下几种方式读取配置，如下面的代码:

```
$config = \Drupal::config('system.maintenance');
$message = $config->get('message');
```

\Drupal::config()也可以链式书写

```
$message = \Drupal::config('system.maintenance')->get('message');
```

读取嵌套配置时，使用'.'字符分隔。

```
$enabled =
\Drupal::config('system.performance')->get('cache.page.enabled');
```

你可以在任何层级读取配置,如果在你读取的所在层有嵌套,将会返回一个数组。

```
$page_cache = \Drupal::config('system.performance')->get('cache.page');
```

上面的代码将返回带有两个键的数组，这两个键分别为'enabled'和'max_age'。

如要在 config 对象上返回所有数据，只需要调用不带参数的 get()方法。

写配置

为了修改配置，你需要调用配置工厂中的 getEditable()方法来获取 \Drupal\Core\Config\Config 的实例。直接调用 \Drupal\Core\Config\ImmutableConfig 实例的 delete()或 save()来修改配置将抛出一个异常。

如下面代码:

```
\Drupal::service('config.factory')->getEditable('system.performance');
```

使用 set()方法修改或添加配置，使用 save()方法保存配置。注意配置必须显示地保存，在配置中简单地设置数据不用保存。

保存配置数据与读取配置数据类似，保存配置数据也需要用到键名。其语法与之前版本的 variable_set()类似。

```
$config =
\Drupal::service('config.factory')->getEditable('system.performance');
//set a scalar value.
$config -> set('cache.page.enabled',1);
//set an array of values.
$page_cache_data = array('enabled' => 1,'max_age' => 5);
$config->set('cache.page',$page_cache_data);
//save your data to the file system.
$config->save();
```

set()函数也可以连写，如果你仅仅需要修改一个值，你可以将其写在同一行。

www.drupalc.com

```
\Drupal::service('config.factory')->getEditable('system.performance')->set('cache.page.enabled',1)->save();
```

如果想在配置对象中替换所有数据，请使用 setData()函数。你可以使用 setData()函数替换数据的子集。如果你只想设置少量数据，则可以通过多次调用 set()方法代替。当使用 setData()时你必须设置一个由 get()返回的关联数组。下面是性能设置(system.performance.yml)的例子:

```
//set all values
\Drupal::service('config.factory')->getEditable('system.performance')->setData(array(
     'cache' => array(
        'page' => array(
          'enabled' => '0',
          'max_age'=> '0',
         ),
      ),
      'preprocess' => array(
         'css' => '0',
         'js' => '0',
      ),
      'response'=> array(
         'gzip' => '0',
      ),
   ))
   ->save();
```

删除配置

清除单个配置可以使用 clear()函数，它们都可以连写。

```
$config =
\Drupal::service('config.factory')->getEditable('system.performance');
$config ->clear('cache.page.max_age')->save();
$page_cache_data = $config->get('cache.page');
```

在这个例子中$page_cache_data 返回一个带有'enabled'键的数组，因为'max_age'已经被删除。

可以使用 delete()函数删除整个配置。

```
\Drupal::service('config.factory')->getEditable('system.performance')->delete();
```

注意随后不要调用 save()函数，不然会创建一个空配置。

www.drupalc.com

应避免在同一函数中多次实例化 config 对象，这会影响性能。下面的代码无需两次实例化 config 对象。

```
\Drupal::service('config.factory')->getEditable('foo.bar')->set('foo','foo')->save();
\Drupal::service('config.factory')->getEditable('foo.bar')->set('bar','bar')->save();
```

最好的方法是只实例化 config 对象一次，并将其存到一个变量中。

```
$config = \Drupal::service('config.factory')->getEditable('foo.bar');
$config
  ->set('foo','foo')
  ->set('bar','bar')
  ->save();
```

13.3 使用配置表单

你可以使用配置表单获取用户输入并修改{module}.settings.yml 文件。这儿有一段代码用于在表单中申请一个$config 对象，你可以在表单设置的 PHP 文件中找到。

Drupal 核心 ConfigFactory 类用于读写配置数据，它可以基于配置文件实例化一个$config 对象。新的配置对象可以被用于实现数据的 CRUD 操作。

下面是表单定义例子，其位于 example/src/Form/exampleSettingsForm.php:

```
/**
* @file
* Contains \Drupal\example\Form\exampleSettingsForm
*/
namespace Drupal\example\Form;
use Drupal\Core\Form\ConfigFormBase;
use Drupal\Core\Form\FormStateInterface;
/**
* Configure example settings for this site.
*/
class exampleSettingsForm extends ConfigFormBase {
  /**
   * {@inheritdoc}
   */
  public function getFormId() {
    return 'example_admin_settings';
  }
```

www.drupalc.com

```
  /**
   * {@inheritdoc}
   */
  protected function getEditableConfigNames() {
    return [
      'example.settings',
    ];
  }
  /**
   * {@inheritdoc}
   */
  public function buildForm(array $form, FormStateInterface $form_state) {
    $config = $this->config('example.settings');
    $form['example_thing'] = array(
      '#type' => 'textfield',
      '#title' => $this->t('thing'),
      '#default_value' => $config->get('things'),
    );
    return parent::buildForm($form, $form_state);
  }
  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state)
{
    $config =
\Drupal::service('config.factory')->getEditable('example.settings');
    $config->set('things', $form_state->getValue('example_thing'))
      ->save();
    parent::submitForm($form, $form_state);
  }
}
```

路由文件(example.routing.yml)

```
example.settings:
  path: '/admin/structure/example/settings'
  defaults:
    _form: '\Drupal\example\Form\exampleSettingsForm'
    _title: 'example'
  requirements:
    _permission: 'administer site configuration'
```

使用 config 对象，可以简化表单数据收集。以上代码将把数据存放于{module}.settings.yml。

www.drupalc.com

任意扩展至 ConfigFormBase 的类必须实现 getEditableConfigNames 方法，并返回一个可编辑的配置字段的数组。

13.4 Drupal 8 配置的存储

一般情况下，在 Drupal 8 中配置信息存储于数据库中。

配置文件格式(YAML)

Drupal 扩展(模块、主题、档案)使用 YAML 文件提供配置。下面是一个配置文件例子:

```
some_string: 'Woo kittens!'
some_int: 42
some_bool: true
```

配置也可以嵌套，如下面的例子:

```
name: thumbnail
label: 'Thumbnail(100×100)'
effects:
  1cfec298-8620-4749-b100-ccb6c4500779:
id: image_scale
data:
  width: 100
  height: 100
  upscale: true
weight: 0
uuid: 1cfec298-8620-4749-b100-ccb6c4500779
```

扩展的默认配置

一个扩展(模块、主题、或档案)可以为它的配置提供默认值，应该将包含配置的 YAML 文件放在 config/install 子目录下。

如果扩展仅仅需要进行简单地配置设置，则可以将这些配置放到一个配置文件中(modulename.settings.yml)。对于更复杂的设置，你应该将其分开放在多个配置文件中。配置实体必须放置在他们自己的 YAML 文件，由模块自己产生。

为了提供动态的默认值，可以实现 hook_install()，而不需要设置相应的配置文件，例如:

www.drupalc.com

```
/**
* Implements hook_install().
*/
function modulename_install() {
  // Set default values for config which require dynamic values.
  \Drupal::configFactory()->getEditable('modulename.settings')
    ->set('default_from_address', \Drupal::config('system.site')->get('mail'))
    ->save();
}
```

扩展的可选配置

扩展的可选配置项存放在 config/optional 子目录中。有些配置项依赖于其它模块的配置项，但并不是模块之间的依赖，当依赖的所有模块安装后，这些配置项才会安装。

举一个例子，模块 A 依赖模块 B，但是模块 A 先安装，模块 B 后安装，然后模块 A 的 config/optional 目录将会被扫描，因为新的依赖已经出现，模块 A 的可选配置项将会安装。如果没有安装模块 B，模块 A 的可选配置也不会安装。

一般地，Drupal 8 将活动的配置存储于数据库中，这样性能和弹性会更好。

更新数据库中的配置信息

在开发期间，你常常需要将 yaml 文件中的配置信息更新到数据库中，你可以使用 drush config-import(cim)命令完成该任务。

到活动配置目录(由 settings.php 定义，如 sites/default/files/config_6dh1U_2YKLGrrh5oLxAgobbledygook/sync)下编辑配置文件，然后运行 drush cim 命令。再使用 drush cr 清除高速缓存使修改生效。当你对设置满意后，将配置文件复制到相应的模块或主题中。

13.5 配置覆写系统

Drupal 8 的配置系统采用统一的方式处理配置信息。一般地，Drupal 将配置信息存放于数据库中，便它可以导出为 YAML 文件，允许对配置进行版本控制。然而有时出于某种目的需要对配置项的值进行覆写。Drupal 7 使用全局变时$conf 来处理 settings.php 文件中的配置。这个系统的最大缺点是覆写是缓慢的。

Drupal 8 引入了新的配置覆写系统，它把覆写作为临时的层，不会在配置表单中使用它们，与其它配置文件分开存储并提供版本控制支持。

www.drupalc.com

全局变量$conf 已更名为$config，配置系统默认激活它。

全局覆盖

Drupal 8 使用全局变时$config 来进行全局覆盖。配置系统通过 Drupal\Core\Config\ConfigFactory::get()来获取配置项，当它从配置中返回一个值后，就可以使用$config 全局变量来修改它的值。

```
$message = \Drupal::config('system.maintenance')->get('message');
```

在使用全局变量$config 覆盖 settings.php 中的值时，需引用配置键:

```
$config['system.maintenance']['message'] = 'Sorry, our site is down now.';
```

对于嵌套的值，使用嵌套的数组键

```
$config['system.performance']['css']['preprocess'] = 0;
```

如果要使用$config 覆盖 settings.php 以外的配置，请使用预定义的全局变量$config。

可以使用以下几种方式来发现可用的配置变量:

使用配置管理模块的 UI 界面来查看
(/admin/config/development/configuration/single/export)

直接查看站点的配置文件(.yml 文件)

通过 drush 查询或列出

```
drush config-list
drush config-get system.performance
```

注意通过$config 覆盖 settings.php 中的配置将不能通过 Drupal 的管理界面查看，也不能使用 drush 列出(除非加上—include-overridden 选项)。管理界面会显示配置的值，你可以将其在不同的环境之间传输。

避免覆盖

有时我们不需要覆写以存取原来的配置值。如果你正写一个配置表单，这是很有用的。在多语言环境中它更有用，因为配置已被重写为翻译。

下面是例子:

```
// Get the site name, with overrides.
$site_name = \Drupal::config('system.site')->get('name');
```

www.drupalc.com

```
// Get the site name without overrides.
$site_name = \Drupal::config('system.site')->getOriginal('name', FALSE);
// Note that mutable config is always override free.
$site_name =
\Drupal::configFactory()->getEditable('system.site')->get('name');
```

你可以通过实现了 StorageInterface::read()的 config.storage 服务直接地访问配置存储。然而这是很少见的。

语言覆写

为了向用户发送电子邮件,这应该基于用户的语言配置,而不是页面的语言配置。将当前语言存入变量中，基于用户的语言设定设置合适的语言，发送电子邮件，最后进行语言回退。

如下面的例子

```
// Load the language_manager service
$language_manager = \Drupal::service('language_manager');
// Get the target language object
$langcode = $account->getPreferredLangcode();
$language = $language_manager->getLanguage($langcode);
// Remember original language before this operation.
$original_language = $language_manager->getConfigOverrideLanguage();
// Set the translation target language on the configuration factory.
$language_manager->setConfigOverrideLanguage($language);
$mail_config  = \Drupal::config('user.mail');
// Now send email based on $mail_config which is in the proper language.
// Set the configuration language back.
$language_manager->setConfigOverrideLanguage($original_language);
```

www.drupalc.com

语言覆盖系统总是使用配置存储来存储覆盖。语言覆盖被存放于文件中。如上例提到的 user.mail 的语言覆写文件是 lang.config.$langcode.user.mail，首先是 language.config 前缀，然后是语言代码$langcode，最后是原来的配置键。

由模块提供覆写

在模块这一级进行覆写是可能的，Drupal 的核心支持全局覆写、语言覆写、以及其它种类的覆写，如基于角色、基于内容、基于域名、基于分组等等。模块可以定义它们自己的覆写规则。当 ConfigFactory 收集模块提供的覆写时，它会调用任意带有 config.factory.override 的服务：

```
config_example.services.yml
services:
  config_example.overrider:
class: \Drupal\config_example\ConfigExampleOverrides
tags:
  - {name: config.factory.override,priority: 5}
```

设置键的优先级以实现优先覆写，具有更高优先级的覆写将会胜过更低优先级的覆写。

```
src/ConfigExampleOverrides.php
/**
* @file
* Contains \Drupal\config_example\ConfigExampleOverrides.
*/
namespace Drupal\config_example;
use Drupal\Core\Cache\CacheableMetadata;
use Drupal\Core\Config\ConfigFactoryOverrideInterface;
use Drupal\Core\Config\StorageInterface;
/**
* Example configuration override.
*/
class ConfigModuleOverrides implements ConfigFactoryOverrideInterface {
  public function loadOverrides($names) {
    $overrides = array();
    if (in_array('system.site', $names)) {
      $overrides['system.site'] = ['name' => 'Overridden site name!'];
    }
    return $overrides;
  }
  /**
   * {@inheritdoc}
   */
  public function getCacheSuffix() {
    return 'ConfigExampleOverrider';
  }
  /**
   * {@inheritdoc}
   */
  public function getCacheableMetadata($name) {
    return new CacheableMetadata();
  }
  /**
   * {@inheritdoc}
```

www.drupalc.com

```
   */
  public function createConfigObject($name, $collection =
StorageInterface::DEFAULT_COLLECTION) {
    return NULL;
  }
}
```

配置覆写在三个不同的层级进行，它们是语言、模块、settings.php，按照 settings.php、模块、语言的优先顺序。在 settings.php 中的覆写的优先级比模块的覆写的优先级更高，模块覆写的优先级比语言覆写的优先级更高。同一层次覆写的优先级可由键的 priority 的值指定。

13.6 配置schema/metadata

Drupal 8 包含了由 Kwalify 创造的 schema/metadata 配置语言，这种语言使用 YAML 文件格式。Kwalify 自身是在 Ruby 中写成，我们对它的格式进行了轻微调整。

本节主要内容如下:

引例
Schema 文件用途
属性
Metadata 文件支持的类型
动态类型引用
Schema 文件代码风格
PHP API
调试 schema

引例

先来看一看站点维护模式的相关配置，这个配置由系统模块提供:

```
<?PHP
$config = \Drupal::config('system.maintenance');
$message = $config->get('message');
$langcode = $config->get('langcode');
```

维护模式的开启与否是存储 state 系统中，而不是在配置中。

维护模式配置对象的默认值存储于 core/modules/system/config/install/system.maintenance.yml 文件中同，如下:

www.drupalc.com

```
message: '@site is currently under maintenance. We should be back shortly. Thank you for patience.'
langcode: en
```

每个模块可以有一个或多个配置对象。这些对象被存储于一个或多个 schema 文件中，在上面的例子中，它是存储在 core/modules/system/config/schema。在 system.schema.yml 文件中响应 schema 的部份如下:

```
system.maintenance:
  type: config_object
  label: 'Maintenance mode'
  mapping:
message:
  type: text
  label: 'Message to display when in maintenance mode'
```

文件中的 system.maintenance 与它的基本文件名 system.maintenance.yml 相关联，并与它的配置对象名(config('system.maintence'))相关联。在该文件中存在嵌套的描述。系统在配置文件中预定义了两种类型的配置对象。它们分别是 config_object 和 config_entity，config_object 用于全局配置，config_entity 用于实体配置。文件 core.data_types.schema.yml 中定义了 config_object 类型：

```
#root of a configuration object.
_core_config_info:
  type: mapping
  mapping:
default_config_hash:
  type: string
  label: 'Default configuration hash'
config_object:
  type: mapping
  mapping:
langcode:
  type: string
  label: 'Language code'
_core:
  type: _core_config_info
```

映射(maping)是一个带有键值对的基本类型。通过使用"config_object"类型，维护模式(maintenance mode)定义重用了 langcode 和_core 键并为 message 添加了更多的键。回到定义中，维护模式(maintenance mode)描述了 content 的内容。真正元素是列在'mapping'键之下。每个元素都有'type'和'label'键用于指出数据的类型和数据的描述。标签(label)用于配置表单中，它的值能被系统管理员编辑修改。

www.drupalc.com

Schema 文件的主要用途

1.在.yml 文个中，Drupal 核心支持的所有根键都将被映射成具体的元素。你只能使用两种映射的子类型 config_object 或 config_entity。映射的单个元素可以基于你的数据定义。核心(_core)键及其下的所有键是为 Drupal 核心预留的。

Schema 文件的用途是用于多语言支持。我们需要标识配置中所有可翻译的字符串，因为你需要在视图、用户角色、菜单等之间使用你自已的设置。我们可以提供模块或主题的翻译。

2.我们总是基于你的数据为配置提供真实的翻译表单。在这种情况下类型很重要，标签很关键。Drupal 核心的配置翻译模块(Configuration translation module)使用 schema 来生成翻译表单和存储翻译。系统内建的两种很重要的可翻译类型是'label'和'text'，其中'label'用于单行输入的文本，'Text'用于多行输入的文本。

3.使用配置 schema 存储配置实体，配置实体的属性需用 schema 来定义，也可以使用它来为配置实体属性设置默认值。

4.使用配置 schema 实现自动类型转换。这是为了确保 PHP 和 web 表单的字符串到其它类型的转换，以保证存储配置时类型正确。

请使用 config_inspector 模块来调试你的 schema，这个模块可以帮助你找出 schema 中的错误。

属性

type:值的类型，可以是基本类型或派生类型。

label:值的用户界面标签，标签不一定要与配置表单的标签一致，但一致会更清晰。

translatable:定义的类型是否可翻译。

nullable:是否可为空；如果不设置默认必须设置它。

class:解析基本类型的类。

type 特定的属性

mapping:映射类型的属性值，用于列出在映射中的基本元素。在映射时需要在 schema 中定义键的类型和值。在映射时仅仅允许字符串键。

sequence:顺序类型的属性值，用于列出在顺序表中的基本元素。键可以是整数或字符串，它们没有关系。

Metadata 文件支持的类型

www.drupalc.com

正如上面提到的，大多数基本类型以及一些复杂的类型都在 core.data_types.schema.yml 文件中定义。

```
# Undefined type used by the system to assign to elements at any level where
# configuration schema is not defined. Using explicitly has the same effect as
# not defining schema, so there is no point in doing that.
undefined:
  label: 'Undefined'
  class: '\Drupal\Core\Config\Schema\Undefined'
# Explicit type to use when no data typing is possible. Instead of using this
# type, we strongly suggest you use configuration structures that can be
# described with other structural elements of schema, and describe your
schema
# with those elements.
ignore:
  label: 'Ignore'
  class: '\Drupal\Core\Config\Schema\Ignore'
# Basic scalar data types from typed data.
boolean:
  label: 'Boolean'
  class: '\Drupal\Core\TypedData\Plugin\DataType\BooleanData'
email:
  label: 'Email'
  class: '\Drupal\Core\TypedData\Plugin\DataType\Email'
integer:
  label: 'Integer'
  class: '\Drupal\Core\TypedData\Plugin\DataType\IntegerData'
float:
  label: 'Float'
  class: '\Drupal\Core\TypedData\Plugin\DataType\FloatData'
string:
  label: 'String'
  class: '\Drupal\Core\TypedData\Plugin\DataType\StringData'
uri:
  label: 'Uri'
  class: '\Drupal\Core\TypedData\Plugin\DataType\Uri'
```

可以看到，大多基本类型被映射到它们相对应的 TypedData API。下面的例子显示定义你自己的类型也是很容易的。定义映射到类型的类，这两个复杂数据类型的定义基于实现的类。

```
# Container data types for lists with known and unknown keys.
mapping:
  label: Mapping
  class: '\Drupal\Core\Config\Schema\Mapping'
```

www.drupalc.com

```
  definition_class: '\Drupal\Core\TypedData\MapDataDefinition'
sequence:
  label: Sequence
  class: '\Drupal\Core\Config\Schema\Sequence'
  definition_class: '\Drupal\Core\TypedData\ListDataDefinition'
```

上面的 mapping 是一个键值对的列表类型(关联数组或 hash)，它的第一个元素可以有不同的类型。而顺序表是一个简单的索引列表，它的元素是同一类型或基于同一种动态类型名称并和键不相关。换句话说，映射(mapping)与顺序表(sequence)的关键不同是顺序表不知道键名和键的数量，而映射所有的键是明确定义的。顺序表可以使用字符串键。

其它的配置 schema 定义由基本的类型派生，例如'label','path','text','date_format',color_hex 被定义为字符串。这些类型可以帮助解析 schema 以确定用于不同目的的文本类型。

```
# Human readable string that must be plain text and editable with a text field.
label:
  type: string
  label: 'Label'
  translatable: true
# Internal Drupal path
path:
  type: string
  label: 'Path'
# Human readable string that can contain multiple lines of text or HTML.
text:
  type: string
  label: 'Text'
  translatable: true
# PHP Date format string that is translatable.
date_format:
  type: string
  label: 'Date format'
  translatable: true
  translation context: 'PHP date format'
# HTML color value.
color_hex:
  type: string
  label: 'Color'
```

注意 label,text,date_format 类型都被标记为可翻译。这意为着核心模块界面翻译(interface translation module)将识别具有这些类型的项并且基于社区或管理员翻译(存于数据库中)或创建的翻译覆盖文件进行翻译。注意可翻译的字符串文本可以从 translation context 键获得，正如上面日期格式中显示的。在这种方式

中，像 Y 这个字符串将获得一个附加的 PHP 日期格式文本，因为翻译器知道它不是'Yes'的缩写，而是表示 PHP 日期格式中的年份。

同样地，你可以定义可重用的复杂类型，它们基于基本类型：

```
# Mail text with subject and body parts.
mail:
  type: mapping
  label: 'Mail'
  mapping:
    subject:
      type: label
      label: 'Subject'
    body:
      type: text
      label: 'Body'
```

上面的代码定义了一个可重用的'mail'类型，它用于 eamil 文本设置，其中 subject 和 body 是一个映射列表。这与为一个配置键定义 schema 是一样的，但你应该给它一个合适的命名，以便于与其它的 schema 定义相区别。基于'mail'的用途进行定义是一个不错的主意。(如在用户模块中的 email 设置 schema user.schema.yml):

```
# Mail text with subject and body parts.
mail:
  type: mapping
  label: 'Mail'
  mapping:
    subject:
      type: label
      label: 'Subject'
    body:
      type: text
      label: 'Body'
```

这两种重要的复杂的类型定义在 core.data_types.schema.yml 文件中。

```
config_object:
  type: mapping
  mapping:
    langcode:
      type: string
      label: 'Language code'
    _core:
      type: _core_config_info
```

www.drupalc.com

```
config_entity:
  type: mapping
  mapping:
    uuid:
      type: string
      label: 'UUID'
    langcode:
      type: string
      label: 'Language code'
    status:
      type: boolean
      label: 'Status'
    dependencies:
      type: config_dependencies
      label: 'Dependencies'
    third_party_settings:
      type: sequence
      label: 'Third party settings'
      sequence:
        type: '[%parent.%parent.%type].third_party.[%key]'
    _core:
      type: _core_config_info
```

动态类型引用

正如上面所说，简单类型引用是必须的，复杂类型引用像'like'也是常见的。有时值的类型不是静态的而是依赖于数据，如图像样式有不同的特效，它们由不同的插件完成。你可以将数据中的键引用为与动态类型相关的类型名的一部份。

类型中的变量值应该在[]中强制关闭，变量值可以与已知组件合并。有三种可能的引用类型:

1.元素键引用:如 book.[%key]，%key 会被元素键代替。

2.子键引用:如类型 views.field.[table]-[field]，类型的计算基于 table 和 field 键的值来决定。

3.父键引用:如'views.display.[%parent.display_plugin]'，将会从它的父亲的 display_plugin 键找出元素的类型。

在图像样式插件和视图插件中有很多这样的例子。比如图像样式的例子 core/modules/image/config/install/image.style.medium.yml 的 YAML 数据结构:

```
name: medium
label: 'Medium (220x220)'
```

www.drupalc.com

```
effects:
  bddf0d06-42f9-4c75-a700-a33cafa25ea0:
    id: image_scale
    data:
      width: 220
      height: 220
      upscale: true
    weight: 0
    uuid: bddf0d06-42f9-4c75-a700-a33cafa25ea0
langcode: en
```

下面的数据结构依赖于效果的类型，效果的类型由它的 id 属性指定。使用的类型依据数据并且不能静态地指定。设置不同的图像样式将使用不同的效果。因此我们需要创建不同的类型引用。下面是在 image.schema.yml 中的响应部份:

```
image.style.*:
  type: config_entity
  label: 'Image style'
  mapping:
    name:
      type: string
    label:
      type: label
      label: 'Label'
    effects:
      type: sequence
      sequence:
        type: mapping
        mapping:
          id:
            type: string
          data:
            type: image.effect.[%parent.id]
          weight:
            type: integer
          uuid:
            type: string
```

上面代码定义了图像样式的元数据，如 name,label,effects 等键的映射。效果(effects)本身是一个顺序表，顺序表的键使用效果的 uuid，但这并不重要，因为顺序表并不并心键，因此我们只需定义元素的类型。效果的通用属性有 id,data 和 weight，然而数据内容依赖它父键的的 id 值。因此应用于数据的 schema image.effect.image_scale 是真正的类型引用。

Schema 文件的代码风格

www.drupalc.com

下面列出的.yml 代码风格可以用在 Drupal 核心的任何地方:

文件的第一行包含注释以说明文件的用途。如果模块中只含有一个schema文件，注释像这样就够了:#Schema for the configuration files of the contact module。

不要使用不清晰的注释。比如，在 comment.settings 的 schema 定义中使用"Comment settings"是多余的。Schema 的每项应含有用于描述的 label。

字符串不要使用双引号，而要使用单引号。

键和类型的定义不要使用引号，在 Drupal 中，键名和类型定义为字符串，并且不包含空格。

在 Drupal 中，YAML 数据文件中的整型值需转换成字符串，因此它需要引在单引号中。

至少应为需要翻译的值添加标签。可以使用配置检查器查看是否可以从 schema 中产生一个表单。

注意代码缩进，这并非代码风格所需，因为合适的 YAML 代码缩进可以让你得到预期的 schema 结构。

PHP API

你可以使用\Drupal::sercice('config.typed')函数返回配置的定义。

```
$definition = \Drupal::service('config.typed')
->getDefinition('system.maintenance');
```

上面的代码将返回下面的数组:

```
array(5) {
  ["label"]=>
  string(16) "Maintenance mode"
  ["class"]=>
  string(34) "\Drupal\Core\Config\Schema\Mapping"
  ["definition_class"]=>
  string(40) "\Drupal\Core\TypedData\MapDataDefinition"
  ["mapping"]=>
  array(2) {
    ["langcode"]=>
    array(2) {
      ["type"]=>
      string(6) "string"
      ["label"]=>
      string(13) "Language code"
```

```
    }
    ["message"]=>
    array(2) {
      ["type"]=>
      string(4) "text"
      ["label"]=>
      string(43) "Message to display when in maintenance mode"
    }
  }
  ["type"]=>
  string(18) "system.maintenance"
}
```

调试 schema

Configuration inspector 模块是用于调试 schema 的模块，它提供了一个用户界面用于比较 schema 数据、表单生成、翻译等。它还可以发现 schema 中的错误。

Drupal 的核心 Configuration translation 模块为网站管理者提供用户界面以翻译 schema。如果你的模块是可翻译的，你也可以使用配置翻译模块来调试翻译文本出现的位置是否正确。

13.7 配置实体依赖

配置实体可以定义依赖。一个依赖可以是模块、主题或实体。

在配置实体安装以前，必须安装配置实体的依赖。如果配置安体所需的依赖不存在或没安装，那么该配置实体将会安装失败。模块应该在模块信息文件中定义依赖。

通常，模块开发者将不需要关于实体依赖的定义。通过扩展核心配置实体基类并且创建相应的插件，依赖将会自动定义和计算。

概述

配置实体通过 config_dependencies 键进行定义。它的值是一个数组，该数组的键可以是以下:

content
config
module
theme

www.drupalc.com

配置实体通过实现
\Drupal\Core\Config\Entity\ConfigEntityInterface::calculateDependencies()确定它们的依赖。

计算配置依赖基于配置实体属性，计算配置实体依赖基于其它的配置实体。在插件和它们的派生中计算依赖

插件派生定义由基本插件派生。如
\Drupal\system\Plugin\Derivative\SystemMenuBlock 派生于
\Drupal\system\Plugin\Block\SystemMenuBlock 插件。系统菜单区块需要在插件区块配置实体和菜单配置实体之间建立依赖关系。

\Drupal\system\Plugin\Block\SystemMenuBlock 实现了
getDervativeDefinitions()方法。因此派生出菜单区块，如 Bartik 底部菜单区块在响应\Drupal\system\Entity\Menu 实体上有一个依赖。

```
public function getDerivativeDefinitions($base_plugin_definition) {
  foreach ($this->menuStorage->loadMultiple() as $menu => $entity) {
    $this->derivatives[$menu] = $base_plugin_definition;
    $this->derivatives[$menu]['admin_label'] = $entity->label();
    $this->derivatives[$menu]['config_dependencies']['config'] =
array($entity->getConfigDependencyName());
  }
  return $this->derivatives;
}
```

在上面的代码中，每一个派生的系统区块都被赋予了区块菜单提供的实体依赖。为了获取实体名称以区别不同的配置依赖，调用了实体的
getConfigDependencyName()方法。实体名称是一个组合字符串，它不应该在申明的地方进行硬编码。

配置依赖的属性可以作为插件的一部份定义，然而配置实体依赖大多是动态值。在插件定义中申明一个静态依赖关系是少见的，应尽量避免。

13.8 在Drupal 8 中创建配置实体类型

本节以实例讲述如何在 Drupal 8 中创建配置实体类型，并带有管理页面。需要定义模块信息文件、路由信息文件、实体类型文件、配置 Schema、实体控制器类等。

安装模块和管理菜单

www.drupalc.com

example/example.info.yml

```
name: Example
description: 'Manages example configuration.'
package: Example
type: module
core: 8.x
```

路由

路由配置文件(routing.yml)定义了管理页面的操作(列表、添加、编辑、删除)的路由。

```
entity.example.collection:
  path: '/admin/config/system/example'
  defaults:
    _entity_list: 'example'
    _title: 'Example Configuration'
  requirements:
    _permission: 'administer site configuration'
entity.example.add_form:
  path: '/admin/config/system/example/add'
  defaults:
    _entity_form: 'example.add'
    _title: 'Add example'
  requirements:
    _permission: 'administer site configuration'
entity.example.edit_form:
  path: '/admin/config/system/example/{example}'
  defaults:
    _entity_form: 'example.edit'
    _title: 'Edit example'
  requirements:
    _permission: 'administer site configuration'
entity.example.delete_form:
  path: '/admin/config/system/example/{example}/delete'
  defaults:
    _entity_form: 'example.delete'
    _title: 'Delete example'
  requirements:
    _permission: 'administer site configuration'
```

example/example.links.action.yml

下面在列表页上添加"Add"链接，主要是使用 appears_on 键。

www.drupalc.com

```
entity.example.add_form:
  route_name: 'entity.example.add_form'
  title: 'Add Example'
  appears_on:
    - entity.example.collection
```

实体类型类

example/src/ExampleInterface.php

假如你的配置实体具有属性，你需要在接口上定义 set/get 方法。

```
namespace Drupal\example;
use Drupal\Core\Config\Entity\ConfigEntityInterface;
/**
* Provides an interface defining a Example entity.
*/
interface ExampleInterface extends ConfigEntityInterface {
  // Add get/set methods for your configuration properties here.
}
```

example/src/Entity/Example.php

这个文件定义配置实体类

```
namespace Drupal\example\Entity;
use Drupal\Core\Config\Entity\ConfigEntityBase;
use Drupal\example\ExampleInterface;
/**
* Defines the Example entity.
*
* @ConfigEntityType(
*   id = "example",
*   label = @Translation("Example"),
*   handlers = {
*     "list_builder" = "Drupal\example\Controller\ExampleListBuilder",
*     "form" = {
*       "add" = "Drupal\example\Form\ExampleForm",
*       "edit" = "Drupal\example\Form\ExampleForm",
*       "delete" = "Drupal\example\Form\ExampleDeleteForm",
*     }
*   },
*   config_prefix = "example",
*   admin_permission = "administer site configuration",
*   entity_keys = {
```

www.drupalc.com

```
*       "id" = "id",
*       "label" = "label",
*    },
*    links = {
*       "edit-form" = "/admin/config/system/example/{example}",
*       "delete-form" = "/admin/config/system/example/{example}/delete",
*    }
* )
*/
class Example extends ConfigEntityBase implements ExampleInterface {
  /**
   * The Example ID.
   *
   * @var string
   */
  public $id;
  /**
   * The Example label.
   *
   * @var string
   */
  public $label;
  // Your specific configuration property get/set methods go here,
  // implementing the interface.
}
```

admin_permission 键自动允许用户的所有访问。若需要实现更多逻辑，可以定义一个自定义访问控制。类 Example 扩展至 ConfigEntiryBase 并实现了 ExampleInterface 接口，ConfigEntiryBase 是配置实体的基类。

Schema 配置文件

example/config/schema/example.schema.yml

```
example.example.*:
  type: config_entity
  label: 'Example config'
  mapping:
    id:
      type: string
      label: 'ID'
    label:
      type: label
      label: 'Label'
```

www.drupalc.com

实体控制器类

example/src/Form/ExampleForm.php

```
namespace Drupal\example\Form;
use Drupal\Core\Entity\EntityInterface;
use Drupal\Core\Entity\EntityForm;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Drupal\Core\Entity\Query\QueryFactory;
use Drupal\Core\Form\FormStateInterface;
class ExampleForm extends EntityForm {
  /**
   * @param \Drupal\Core\Entity\Query\QueryFactory $entity_query
   *   The entity query.
   */
  public function __construct(QueryFactory $entity_query) {
    $this->entityQuery = $entity_query;
  }
  /**
   * {@inheritdoc}
   */
  public static function create(ContainerInterface $container) {
    return new static(
      $container->get('entity.query')
    );
  }
  /**
   * {@inheritdoc}
   */
  public function form(array $form, FormStateInterface $form_state) {
    $form = parent::form($form, $form_state);
    $example = $this->entity;
    $form['label'] = array(
      '#type' => 'textfield',
      '#title' => $this->t('Label'),
      '#maxlength' => 255,
      '#default_value' => $example->label(),
      '#description' => $this->t("Label for the Example."),
      '#required' => TRUE,
    );
    $form['id'] = array(
      '#type' => 'machine_name',
      '#default_value' => $example->id(),
      '#machine_name' => array(
```

www.drupalc.com

```
        'exists' => array($this, 'exist'),
      ),
      '#disabled' => !$example->isNew(),
    );
    // You will need additional form elements for your custom properties.
    return $form;
  }
  /**
   * {@inheritdoc}
   */
  public function save(array $form, FormStateInterface $form_state) {
    $example = $this->entity;
    $status = $example->save();
    if ($status) {
      drupal_set_message($this->t('Saved the %label Example.', array(
        '%label' => $example->label(),
      )));
    }
    else {
      drupal_set_message($this->t('The %label Example was not saved.',
array(
        '%label' => $example->label(),
      )));
    }
    $form_state->setRedirect('entity.example.collection');
  }
  public function exist($id) {
    $entity = $this->entityQuery->get('example')
      ->condition('id', $id)
      ->execute();
    return (bool) $entity;
  }
}
```

example/src/Controller/ExampleListBuilder.php

```
namespace Drupal\example\Controller;
use Drupal\Core\Config\Entity\ConfigEntityListBuilder;
use Drupal\Core\Entity\EntityInterface;
/**
* Provides a listing of Example.
*/
class ExampleListBuilder extends ConfigEntityListBuilder {
  /**
```

www.drupalc.com

```
   * {@inheritdoc}
   */
  public function buildHeader() {
    $header['label'] = $this->t('Example');
    $header['id'] = $this->t('Machine name');
    return $header + parent::buildHeader();
  }
  /**
   * {@inheritdoc}
   */
  public function buildRow(EntityInterface $entity) {
    $row['label'] = $this->getLabel($entity);
    $row['id'] = $entity->id();
    // You probably want a few more properties here...
    return $row + parent::buildRow($entity);
  }
}
```

example/src/Form/ExampleDeleteForm.php

```
namespace Drupal\example\Form;
use Drupal\Core\Entity\EntityConfirmFormBase;
use Drupal\Core\Url;
use Drupal\Core\Form\FormStateInterface;
/**
* Builds the form to delete a Example.
*/
class ExampleDeleteForm extends EntityConfirmFormBase {
  /**
   * {@inheritdoc}
   */
  public function getQuestion() {
    return $this->t('Are you sure you want to delete %name?', array('%name'
=> $this->entity->label()));
  }
  /**
   * {@inheritdoc}
   */
  public function getCancelUrl() {
    return new Url('entity.example.collection');
  }
  /**
   * {@inheritdoc}
   */
```

www.drupalc.com

```
  public function getConfirmText() {
    return $this->t('Delete');
  }
  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state)
{
    $this->entity->delete();
    drupal_set_message($this->t('Category %label has been deleted.',
array('%label' => $this->entity->label())));
    $form_state->setRedirectUrl($this->getCancelUrl());
  }
}
```

www.drupalc.com

第 14 章 菜单系统

Drupal 8 的菜单系统与 Drupal 7 的菜单系统有较大差异，Drupal 7 的菜单与 URL 存在映射关系，在 Drupal 8 中 URL 的处理则是由路由系统完成，菜单链接与路由系统将会由相应的配置文件相关联，相比之下，Drupal 8 的处理方式更加灵活、文便。本章将比较 Drupal7 与 Drupal 8 的菜单系统，如何由模块提供菜单链接，定义菜单局部任务，系统 action 等。

14.1 Drupal 7 菜单API与Drupal 8 菜单API的异同

Drupal 7 菜单系统的核心是实现 hook_menu()，它提供了路径与页面回调函数(也叫页面控制器)之间的映射。它服务于各种菜单，如管理菜单，Tab 菜单，局部任务菜单，上下文链接等。它还提供访问检测，实体加载等等，那是一个很庞大的系统。

在 Drupal 8 中，因为引入了路由系统处理路径 URL，因此它与 Drupal 7 的处理方式有很大不同。每一个路径有相应的控制器，控制器对路径进行响应，包括访问控制等。菜单与 URL 路径的关联由相应的 yml 配置文件进行定义，由路由系统进行处理。Drupal 8 为菜单系统定义了一套新的 API，用来为模块处理菜单，action，局部任务，上下文链接等。

14.2 提供模块定义的菜单链接

与 Drupal 7 不同，Drupal 8 已删除 hook_menu，路径由路由系统处理。菜单链接定义在静态文件.yml 中。其命名规则为 module_name.links.menu.yml。例如，在开发设置(development settings)下提供了配置菜单，需要这样做:

```
example.admin
  title: ‘Example settings’
  description: ‘Manage example settings for your site ’
  parent: system.admin_config_development
  weight: 100
  #If menu_name is omitted, the “Tools” menu will be used.
  menu_name: devel
```

www.drupalc.com

只有 title 键是必须的。上面的例子定义了一个局部菜单链接，它使用 route_name 将菜单链接与路由关联在一起。菜单链接指向外部 URL，需要使用外部 URL 路径值。Description 键的值是对菜单项的描述，当鼠标停在菜单上时将出现此提示信息。Weight 将用于对菜单进行排序，值越大越朝后排。通过设置 parent 键指定父菜单项，并将其置于菜单树中。可以从 hook_menu_links_discovered_alter()文档中查看所有可使用的键。

确定菜单父链接可能有点困难。如果你知道父菜单项的路径，你需要在所有的*.routing.yml 文件中搜索，大多数的文本编辑器或 IDE 都有在所有文件中搜索的功能，找到与路径关联的路由名，然后，你需要在所有的*.links.menu.yml 文件中搜索路由名。与路由名相匹配的菜单链接就是你菜单的父链接。另外，如果你知道定义父菜单链接的模块，你直接可以到该模块中去搜索。

如果你的菜单链接文本是不清晰的(如'Extend'，'May'等)，你可以使用 title_content 键为菜单链接提供标题字符串。这些字符串将作为翻译文本。

视图模块创建的路径的路由名的格式为 view.name.display_id。

14.3 提供模块定义的局部任务

局部任务(Local task)用于在页面上生成 tab 菜单，一般出现在页面顶部。它大部份用于管理页面而不是前端页面，但用户注册、登录、密码修改页面也使用了局部任务。在 Drupal 8 中可以使用一个简单的 YAML 文件定义局部任务，但需遵循一定申明格式。

定义静态的局部任务

定义的大多数局部任务都是静态的，它被定义在你模块的 YAML 文件中，当然，文件的命名有一定的规则。例如，如果模块名为'example'，则文件是 example.links.task.yml 并且应把它放在模块的根目录下。

```
example.admin: # The first plugin ID
  title: 'Settings'
  route_name: example.admin
  base_route: example.admin
example.admin_3rd_party: # The second plugin ID
  title: 'The party services'
  route_name: example.admin_3rd_party
  base_route: example.admin
```

这个文件定义了两个 tab，第一个的路由为 example.admin，另一个的路由为 example.admin_3rd_party。在定义插件 ID(最顶层的键)时，建议路由名保持一

www.drupalc.com

致。如果你需要为不同的 tab 使用相同的路由名，那么请向路由名添加一个有意义的后缀。Tab 的标题将会显示用户界面上，并且基本路由(base_route)的标题将会是默认的标签页，键 base_route 的作用是将相关的 tabs 分组在一起，你可以为 tab 提供一个权重用以对它们进行排序，基本路由将会自动取得一个负的权重并排在其它 tab 的左边。

为了提供多级的 tab，可以使用 parent_id 将子 tab 与它的父 tab 关联在一起，使用 base_route 将 tab 分组。注意如果你提供了 parent_id，那么 base_route 的值将被忽略，因它将从它的父 tab 那里获得。

例如，下面是 block_content 模块定义的 local tasks:

```
entity.block_content.collection: # (1) Non-default tab by side of "Block layout" (which is on block.admin_display)
  title: 'Custom block library'
  route_name: entity.block_content.collection
  base_route: block.admin_display
block_content.list_sub: # (2) Default subtab, same route as the parent tab, so different tab ID.
  title: Blocks
  route_name: entity.block_content.collection
  parent_id: entity.block_content.collection
entity.block_content_type.collection: # (3) Non-default subtab alongside the "Blocks" default tab
  title: Block types
  route_name: entity.block_content_type.collection
  parent_id: entity.block_content.collection
  weight: 1
entity.block_content_type.edit_form: # (4) Default edit tab on content block.
  title: 'Edit'
  route_name: entity.block_content_type.edit_form
  base_route: entity.block_content_type.edit_form
entity.block_content.delete_form: # (5) Non-default delete tab on content block.
  title: Delete
  route_name: entity.block_content.delete_form
  base_route: entity.block_content.canonical
```

前三个 tab 出现在区块管理页面，最后两个出现在编辑内容区块页面。

最后，你可以为使用 title_context 键为 tab 的标题提供字符串文本，有些字符串是不清晰的，这些字符串可以帮助翻译。

动态产生局部任务

www.drupalc.com

有时静态地列出局部任务是不够的。例如，视图(Views)、内容翻译(Content translation)和配置翻译(Configuration translation)在 Drupal 中向不同的页面添加局部任务。要做到这一点，需要使用 driver 键来指向一个用于生成动态局部任务的类。例如可以在 example.links.task.yml 这样进行定义:

```
example.local_tasks:
  deriver: 'Drupal\example\Plugin\Derivative\DynamicLocalTasks'
  weight: 100
```

产生局部任务的派生类应放在 src/Plugin/Derivative/DynamicLocalTasks.php 中:

```
/**
* @file
* Contains \Drupal\example\Plugin\Derivative\DynamicLocalTasks.
*/
namespace Drupal\example\Plugin\Derivative;
use Drupal\Component\Plugin\Derivative\DeriverBase;
/**
* Defines dynamic local tasks.
*/
class DynamicLocalTasks extends DeriverBase {
  /**
   * {@inheritdoc}
   */
  public function getDerivativeDefinitions($base_plugin_definition) {
    // Implement dynamic logic to provide values for the same keys as in
example.links.task.yml.
    $this->derivatives['example.task_id'] = $base_plugin_definition;
    $this->derivatives['example.task_id']['title'] = "I'm a tab";
    $this->derivatives['example.task_id']['route_name'] = 'example.route';
    return $this->derivatives;
  }
}
```

在这个例子中，我们没有做任何事，但你可以使用这样的结构来动态地产生你需要的局部任务。

自定义局部任务行为

通过扩展 LocalTaskDefault，你可以自定义局部任务的行为。举例，你能够提供一个动态的标题。你需要在 class 键中提供实现自定义局部任务所使用的类名。例如，UserTrackerTab 依赖于路径中当前用的 ID，它定义于 tracker.links.task.yml 中:

```
tracker.users_recent_tab:
  route_name: tracker.user_recent_content
  title: 'My recent content'
  base_route: tracker.page
  class: '\Drupal\tracker\Plugin\Menu\UserTrackerTab'
```

最后，你可以使用 hook_menu_local_tasks_alter 修改已存在的局部任务。

14.4 提供模块定义的局部动作

在 Drupal 8 中，局部动作(local actions)已经从 hook_menu()系统中移除，并且它与局部任务非常相似。使用局部动作来定义局部操作，比如添加新菜单、添加新内容等。

局部动作也是在 YAML 文件中定义，命名规则是基于模块。如 menu_ui.links.action.yml 是基于 menu_ui 模块:

```
menu_ui.link_add:
  route_name: menu_ui.link_add
  title: 'Add link'
  appears_on:
    - menu_ui.menu_edit
menu_ui.menu_add:
  route_name: menu_ui.menu_add
  title: 'Add menu'
  appears_on:
    - menu_ui.overview_page
```

www.drupalc.com

这些 action 像下图这样显示:

正如局部任务，最好的方式是局部动作和它的路由名使用相同的机器名。使用 route_name 键指定路由，使用 title 键指定标题，它将会出现在命令按钮上。你还可以指定 title_content 键用于翻译。

最后，使用 appears_on 键定义局部动作将出现的页面，你可以以列表的形式指定多个路由。

动态生成局部动作

实现动态生生局部动作与实现动态生成局部任务很相似。使用 deriver 键定义一个基于 DerivativeBase 的类来实现动态生成局部任务。

自定义局部动作行为

在 LocalActionDefault 中实现了默认的局部动作行为。你能够通过扩展这个类来实现修改局部动作的行为。如实现动态的标题:

```
/**
* @file
* Contains \Drupal\example\Plugin\Menu\LocalAction\CustomLocalAction.
*/
namespace Drupal\example\Plugin\Menu\LocalAction;
use Drupal\Core\Menu\LocalActionDefault;
/**
* Defines a local action plugin with a dynamic title.
*/
class CustomLocalAction extends LocalActionDefault {
  /**
   * {@inheritdoc}
   */
  public function getTitle() {
    return $this->t('My @arg action', array('@arg' => 'dynamic-title'));
  }
}
```

在 example.links.action.yml 文件中关联这个类:

```
example.content.action:
  route_name: example.content.action
  title: 'Example dynamic title action'
  weight: -20
  class: '\Drupal\example\Plugin\Menu\LocalAction\CustomLocalAction'
  appears_on:
    - example.content
```

最后，使用 hook_menu_local_actions_alter 修改已存在的局部动作。

14.5 提供模块定义的上下文链接

上下文链接是指显示在页面上某个位置的菜单链接，如点击一个区块的右上角会显示配置区块菜单链接。上下文链接是为了方便管理，从而减轻管理员的管理站点的压力。

在 Drupal 8 中上下文链接已经从 hook_menu()系统中删除，它与局部任务非常相似。使用上下文链接可以在前端为 Drupal 对象提供链接选项。

申请上下文链接

上下文链接也是在 YAML 文件中定义，基于模块名进行命名。例如 block 模块，上下文链接的 YAML 文件名为 block.links.contextual.yml，其内容如下:

```
block_configure:
  title: 'Configure block'
  route_name: 'block.admin_edit'
  group: 'block'
```

上下文链接的是通过 route_name 定义的路由名进行响应，当用户点击该上下文链接时，系统转到相应的路由对它进行处理。Title 键为上下文链接提供显示用的标题，你可以使用 title_content 键为其指定翻译文本。Group 键的作用是对上下文链接进行分组，具有相同的 group 的显示在一起，你可以通过 weight 键来对上下文链接排序。

渲染上下文链接

上下文链接的渲染需要在渲染数组中使用#contextual_links 键。例如 BlockViewBuilder 提供的上下文链接列表如下:

```
public function viewMultiple(array $entities = array(), $view_mode = 'full',
$langcode = NULL) {
    $build = array();
    foreach ($entities as $key => $entity) {
      // ...
      $build[$entity_id] = array(
        '#theme' => 'block',
        // ...
        '#contextual_links' => array(
          'block' => array(
            'route_parameters' => array('block' => $entity->id()),
          ),
        ),
        // ...
      );
    }
    // ...
  }
```

在渲染数组中#contextual 键下的 block 键用以建立'block'分组。'block'分组下的所有链接将会显示为上下文链接。该键的值是一个数组，它带有一个路由参数，

这个路由参数是可以转化与该链接相关的路径(区块标识将作为路由的一个动态组件)。

例如上面提到 block.admin_edit 路由在 block.routing.yml 中定下如下:

```
block.admin_edit:
  path: '/admin/structure/block/manage/{block}'
  defaults:
_entity_form: 'block.default'
_title: 'Configure block'
  requirements:
_entity_access: 'block.update'
```

在#contextual_links 数组中定义的'block'路由参数允许系统解析到正确的路径，它是通过替换路由中的{block}参数为区块的 id 来实现。

上下文链接存储在 title_suffix 中。这个变量需要打印到 html.twig 文件，就像下面这样:

```
<div{{   attributes   }}>
    {{ title_suffix }}
</div>
```

修改上下文链接

为了修改已存在的上下文链接，需实现 hook_contextual_links_view_alter()。

为了向还没有上下文链接分组的站点组件添加上下文链接，你需要通过修改 build 数组以建立一个新的分组，定义一个新的键并提供充足的路由参数。这个处理过程基于在哪里如何对渲染数组进行处理。下面是一个修改渲染数组的例子:

```
function menu_ui_block_view_system_menu_block_alter(array &$build,
BlockPluginInterface $block) {
  // Add contextual links for system menu blocks.
  $menus = menu_list_system_menus();
  $menu_name = $block->getDerivativeId();
  if (isset($menus[$menu_name])) {
    $build['#contextual_links']['menu'] = array(
      'route_parameters' => array('menu' => $menu_name),
    );
  }
}
```

www.drupalc.com

通过在区块的 build 数组中包含新的'menu'分组，现在所有的上下文链接将会被添加到这个区块。下面的上下文链接被定义在 menu_ui.links.contextual.yml 中的 menu 分组中:

```
menu_ui_edit:
  title: 'Edit menu'
  route_name: 'menu_ui.menu_edit'
  group: menu
```

上面的代码导致'block'和'menu'上下文链接都会现面在区块的上下文链接菜单中。

动态生成上下文链接

动态生成上下文链接与动态生成局部任务相似。需在 example.links.contextual.yml 文件中使用 derivative 键定认一个扩展至 DerivativeBase 的类。请注意相关格式。

自定义上下文链接的行为

ContextualLinkDefault 定义了上下文链接的缺省行为，如果要进行更改，你需要扩展该类来修改上下文链接的缺省行为。例如让上下文链接使用动态标题，需要覆写 getTitle 方法:

```
example.customtitle:
  title: 'Custom title example'
  route_name: 'example.admin_edit'
  group: 'example'
  class: '\Drupal\example\Plugin\Menu\ContextualLink\CustomContextualLink'
```

www.drupalc.com

第 15 章 路由系统

路由系统简单地说是系统对 URL 路径的处理方式。在 Drupal 7 中路径的处理是通过菜单系统实现 URL 到函数功能的映射，也就是说，Drupal 7 的模块中定义了许多路径，并定义了相关的函数来对这些路径进行响应。Drupal 8 的路由系统有了很大的变化，它不再使用菜单系统来实现 URL 到函数的映射。它引入了功能强大的 Symfony2 框架来对路由进行处理，整个系统的路径将由路由来定义，路由则与相应的函数控制器或类来进行处理，它们之间的映射关系在 YAML 文件中定义。本章将详细讨论 Drupal 8 的路由系统。

15.1 Drupal 8 路由系统概述

在 Drupal 7 中使用 hook_menu()来实现路由，并使用 hook_menu()来创建菜单、标签页面、命令按钮、上下文链接等。Drupal 8 对此作了极大的调整，它不再使用 hook_menu 来实现路由，菜单、标签页等的定义由 menu API 来实现。

引入 Symfony

Drupal 8 的路由系统极大地依赖于 Symfony。Drupal 的路由系统可以完成 Symfony 的所有路由功能，并且可以使用与其相同的语法来定义路由。

要学习Drupal 8 路由系统的基本知识，你可以看Symfony的路由系统文档。这里的文档并不涉及Symfony的路由系统的所有方面，因此如果你卡在这里了，你最好先阅读一下Symfony的路由系统文档。

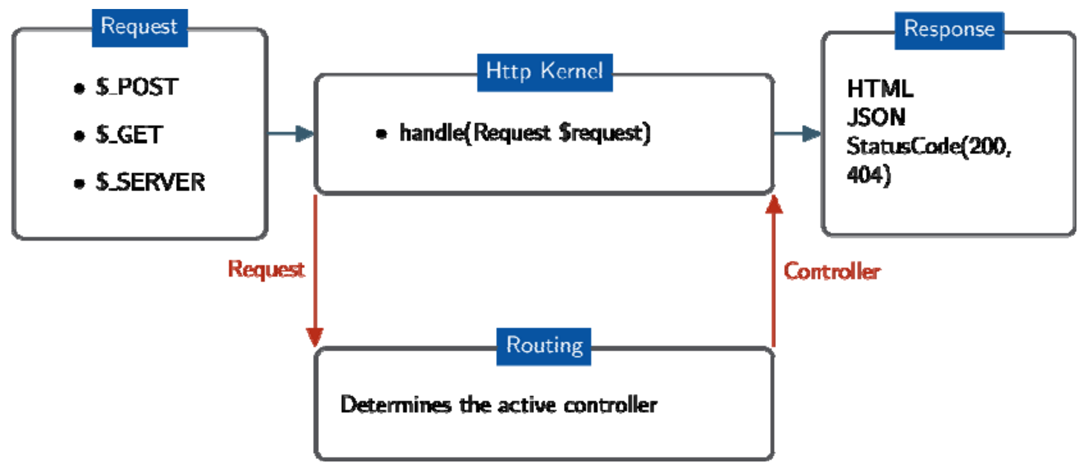
路由概述

路由实际上是由 Drupal 定义的一个路径并对它进行响应返回一些内容。例如，缺省的首页其路径为'/node'是一个路由。当 Drupal 收到一个请求后，它尝试寻找一个与该请求路径相匹配的路由。如果找到了，就交给该路由相关的控制器处理并返回内容。否则，Drupal 返回一个 404。

路由与控制器

Drupal 的路由系统与 Symfony HTTP Kernel 一起工作。然而，你并不需要知道 Symfony HTTP Kernel 是如何处理路由的。下面的图描述了各个组件间的关系:

www.drupalc.com



路由系统使用与路径相匹配的控制器进行响应，你可以使用路由来定义路径与控制器间的关联。在路由中你可以向控制器传递额外的信息。还可以在路由上进行访问控制。

路由中的参数

Drupal 8 的路由中可以使用占位符，这允行 URL 中包含动态参数。URL 中的参数将会以变量的形式传递给控制器中的方法。如下面例子:

```
example.name:
  path: '/example/{name}'
  defaults:
_controller: '\Drupal\example\Controller\ExampleController::content'
  requirements:
_permission: 'access content'
```

{name}元素即为 URL 中的动态参数，意思是说在{name}位置上的值是变化的，在控制器使用$name 变量引用它。

15.2 Drupal 8 路由与控制器引例

本节将以实例来讲讲 Drupal 8 的路由与控制器。如果你只是想修改或扩展已存在的功能，你不需要了解路由。但如果想进行模块开发，那么你需要掌握路由知识。

在 **routing.yml** 文件中定义路由

模块的信息文件命名形式为 module_name.info.yml。模块名是用来在 Drupal 核心中注册的机器名。你可以到 Drupal 核心的模块目录中了解细节。当 Drupal 系统遇到一个路径时，它便会加载 module_name.routing.yml 中定义的路由以对路径进行响应。例如，如果定义了一个名为 example 模块，则定义的路由 example.routing.yml 为:

```
example.content:
  path: '/example'
  defaults:
_controller: '\Drupal\example\Controller\ExampleController::content'
_title: 'Hello World'
  requirements:
_permission: 'access content'
```

这个文件定义了名为'example.content'的路由，命名带有模块名为前缀，它与路径 URI'/example'进行关联。当这一路径被访问时，先检查'access content'权限，如果权限允许，系统调用 ExampleController::content 方法并显示一个标题为'Hello World'的页面。

请注意 Drupal 使用了自动加载机制，它将加载文件 modules/example/src/Controller/ExampleController.php，因为当它在 _controller 中遇到机器名'example'时，它将会搜寻模块的 src 目录。如果文件没有自动加载，你可能要检查一个模块的机器名。

创建页面控制器

Drupal 8 命使用 Symfony HTTP Kernel 处理客户端请求，它接受客户端请求并寻找处理该路径的响应对象，然后将服务器输出返回给客户端。服务器输出由名为控制器的一段代码产生。从理论上讲，控制器可以是一个 PHP4 函数，一个对象方法或者一个匿名函数。在 Drupal 8 中最好使用一个控制器类。这个类存放于相应的 php 文件中。

上面例子中的类应存放于 example/src/Controller 目录下，并且命名为 ExampleController.php，因此它的完全路径为 example/src/Controller/ExampleController.php。

```
namespace Drupal\example\Controller;
use Drupal\Core\Controller\ControllerBase;
/**
* An example controller.
*/
class ExampleController extends ControllerBase {
  /**
```

```
   * {@inheritdoc}
   */
  public function content() {
    $build = array(
      '#type' => 'markup',
      '#markup' => t('Hello World!'),
    );
    return $build;
  }
}
```

只需一个路由文件和页面控制器，我们就可以在站点中定义/example 路径，当访问它时，输出页面"Hello World!"。

15.3 Drupal 8 路由系统的基本功能

Drupal 8 的路由系统建立在 Symfony 框架之上。定义与使用路由，你不需要学习 Symfony，但对于一些高级应用，了解一些 Symfony 还是很有帮助。

Drupal 8 使用了 Symfony HTTP Kernel，它接受请求，并要求其它系统对请求进行处理，并将结果返回客户端。处理结果由控制器完成，控制器是一段 PHP 代码，如是一个 PHP 函数或对象方法。

www.drupalc.com

Symfony 路由与 Drupal 路由的比较

作为一个模块开发者，你的工作是定义路由列表，并定义响应路由的控制器。例如:

```
taxonomy.autocomplete_vid:
  path: '/taxonomy/autocomplete_vid/{taxonomy_vocabulary}'
  defaults:
_controller:
'\Drupal\taxonomy\Controller\TermAutocompleteController::autocompletePerVid'
requirements:
   taxonomy_vocabulary: \d+
```

Symfony 文档使用'pattern'代替'path'，但是从 Symfony2.2 起，在路由文件中使用'path'键

控制器将会从系统中获取参数并将其传递给响应的方法，在这个例子中，参数是'taxonomy_vocabulary'。没有下划线的一切被视为控制器的参数。如果你想为参数指定缺省值，你应该将其定义在缺省的数组中。与控制器参数无关的属性被视为内部属性并以下划线开头。

Symfony 也允许你使用 requirements 键定义正则表达式，用来验证参数的有效性。上面的代码中的'\d+'表示匹配数字。

控制器解析

一旦 Symfony 确定了当前请求的控制器，它会要求控制器解析器(controller resolver)创建控制器的实例，然后通过 call_user_func_array()进行调用。

Drupal 对 Symfony 路由的扩展

上面谈了 Symfony 对请求处理的一些基本知识，Drupal 稍稍有些复杂:

你可以在路由上进行访问检测。例如在 Drupal 7 中经常调用 user_access()。

你想转换 taxonomy_vocabulary 到真正的实体对象。

你不想产生整页，而只是想产生主要内容。

下划线开头的键在系统间进行通知。

www.drupalc.com

15.4 路由数据结构

要定义路由需要创建一个 module.routing.yml 文件。每个路由的命名遵循 module_name.route_name 这样的形式(book.render)，并可以还有以下属性:

path(必须):路由的 URL，带有斜杠如(path:'/book')。你可以使用大括号将动态属性括起来。例如(path: '/node/{node}/outline')。这个参数将会被传递到控制器。注意路径的第一项不能是动态的，你可以在路径后面定义可选的参数。

defaults(必须):定义路由的缺省属性，下面列出了可能的键，你可以指定一个以指定如何产生输出:

_controller:它的值是一个可调用的类方法，格式为 classname::method，其返回值可以是以下之一:

可渲染数组(renderable array)。可渲染数组用于输出可主题化页面的主要内容。

Symfony\Component\HttpFoundation\Response 对象。它可以是非 HTML 内容如 XML 种子或 HTML 页面片段。这些内容会被真接发送，不会主题化或添加区块。

注意_controller 的值并非是一个路径，而是一个遵循 PSR-4 命名的类。以 PSR-4 命名的路径名必须映射成一个可响应的路径结构:

命名空间(namespace):格式为\Drupal\[模块名]\Controller\[类名]::[方法名]。例子，_controller: '\Drupal\test_mod\Controller\TestModController::build'。

路径(path):格式为[模块名]/src/Controller/[类名].php，例如 test_mod/src/Controller/TestModController.php，包含类 TestModController，在类里定义了 build()方法。

注意模块名必须小写可以使用下划线，类采用驼峰命名法。

_form:一个实现 Drupal\Core\Form\Forminterface 接口的类。详请请阅读 Drupal 8 表单 API。

_entity_view:它的值为 entity_type.view_mode，即实体类型的查看模式。这将会在路径中寻找实体并以给定的查看模式渲染实体。例如，_entity_view:node.teaser，将以返回{node}的摘要。

_entity_list:它的值是实体类型(entity_type)。使用 EntityListController 提供一个实体例表。例如:_entity_list:view_mode 返回视图列表的渲染数组。

_entity_form:与_from 相似，但它还会为实体添加编辑表单。在实体 metadata 中定义实体表单处理。例如，_entity_form: node.default 将显示缺省的节点表单。在 node.default 中的'node'与实体 ID 相关联，'default'与表单处理键相关联。

_title(optional):路由的页面标题，它是可选的。可以和菜单链接标题不同。

_title_context(optional):可选的，传递给 t()函数的上下文信息。

_title_callback(optional):标题回调函数(一般为 类名::方法名)为路由返回页面标题。

最后，请将需要向控制器传递的静态变量在 defaults 数组中定义。命名它们与你的控制器参数相匹配。可选的参数:如果你想在路径后面添加可选的参数，foo/bar/{baz}将会与 foo/bar 相匹配，baz 的值可在 defaults array 中设置。

www.drupalc.com

requirements(必须):访问路由需要满足的条件，它是一个数组，数组中可以含有下面的一个或多个键:

_permission:权限字符串(例如，_permission:'access content')。你可以指定多个权限，并用逗号','分隔它们。例如，_permission: 'access content'，access user profiles'。模块在 my_module.permissions.yml 文件中定义权限。

_role:指定用户角色，例如'administrator'。你可指定多个并用逗号分开。注意，因为不同站点有不同的角色，所以推荐基于权限进行访问控制。

_access: 设置它为"'TRUE'"(带有单引号)，在任何情况下都有权访问。

_entity_access:在这种情况，实体是路由的一部份，这样可以在授权访问以前就可以检查特定的访问等级(例如，_entity_access:'node.view')。你可以指定实体的验证方式，例如 node:\d+。如果你定义了路由/foo/{node}和/foo/bar，其中{node}表示节点的 id，则/foo/{node}不会匹配/foo/bar，因为 bar 不是数字。

_custom_access:自定义访问。细节请阅读路由访问检测。

_format:使用这个键检测请求类型。例如，_format: json，仅匹配请求头为 json 的请求。

_content_type_format:使用该键检测请求的内容类型。例如，_content_type_format: json 仅匹配 Content-type 头为 json 的请求。请阅读 ContentTypeHeaderMatcher.php 了解其工作原理。

_module_dependenices:这个键是可选键，它的作用是为路径指定需要的一个或多个模块。模块名以'+'连接表明是与关系，模块名以','连接表明是或关系。例子，_module_dependencies: 'node + search' 意为'node'和'search'模块都是必须的。_module_dependencies: 'node,search'意为'node'和'search'之一即可。如果模块已通过 info.yml 文件定义了依赖，则这里不需要再定义相关模块的依赖。对于可选的依赖，只能在路由中定义。

_csrf_token:如果 URL 实现了一些命令或操作，并且这些命令或操作没有使用表单回调，则应该将该键的值设为'TRUE'。

_method:可选键，它限制路由到指定的 http 方法。缺省限制到 GET 和 POST。可以设置多个方法，例如'GET|POST'。更新到 symfony 3 这个将会改变。

options(optional):路由交互方式的额外选项。常用选项如下:

_acess_mode:出于安全，这个选项已被删除。

_admin_route:指定路由是否使用管理主题。

_theme:路由使用的主题。

www.drupalc.com

no_cache:设为'TRUE'，表明路由的响应不被缓存。

parameters:如果来自路径的参数名没有匹配到任何实体名，你可以在这儿向实体传递参数。因为没有同名的参数，让我们看一下{user}，你能够映射没有匹配的实体。

second_user:路径中的参数名。

type:使用的实体类型，在这个例子中，实体为 user，这将映射参数到用户实体。

下面是一个更加复杂的例子:

book.routing.yml

```
# Each route is defined by a machine name, in the form of
module_name.route_name.
#
book.render:
  # The path always starts with a leading forward-slash.
  path: '/book'
  # Defines the default properties of a route.
  defaults:
    # For page callbacks that return a render array use _controller.
    _controller: '\Drupal\book\Controller\BookController::bookRender'
  # Require a permission to access this route.
  requirements:
    _permission: 'access content'
book.export:
  # This path takes dynamic arguments, which are enclosed in { }.
  path: '/book/export/{type}/{node}'
  defaults:
    # Because this route does not return HTML, use _controller.
    _controller: '\Drupal\book\Controller\BookController::bookExport'
  requirements:
    _permission: 'access printer-friendly version'
    # Ensure user has access to view the node passed in.
    _entity_access: 'node.view'
```

向控制器传递参数

在 defaults 部份下定义的没有以'_'开头的键将会被当作参数传递到控制器。适当地命名控制器参数是有好处的。如下面的例子:

example.content

```
  path: '/example'
  defaults:
_controller: '\Drupal\example\Controller\ExampleController::content'
    custom_arg: 12
requirements:
   _permission: 'access content'
```

上面的代码将会向控制器传递$custom_arg 变量，因此你可以在控制器方法中使用:

```
   //...
   public function content(Request $request,$custom_arg){
      //Now can use $custom_arg (which will get 12 here) and $request.
   }
```

15.5 路由访问检测

在上节中我们介绍了路由文件中可用的键，其的 requirements 键定义路由访问的条件，其下可以指定的键比较多，但一般简单的访问控制，可以使用 premissions/roles 进行控制。

多条件访问检测

如果一个路由要进行多个检测，可以使用 anif 操作将它们链在一起：所有的结果必须是 AccessResult::allowed，否则访问将会被拒绝。运行 AccessResult::neutral()和 AccessResult::forbidden()没有什么不同。但对于实体访问检测却有一些不同：其一是使用 orif 并且可以允许中性的结果。

自定义路由访问检测

有时使用 permissions 和 roles 进行访问控制是不够的，你需要自定义路由访问控制。要实要这一些，可以定义一个实现了 AccessInterface 结口的类来进行访问检测。例如 example 模块，这个类应该是这样 exaple/src/Access/CustomAccessCheck.php:

```
namespace Drupal\example\Access;
use Drupal\Core\Routing\Access\AccessInterface;
use Drupal\Core\Session\AccountInterface;
use Symfony\Component\Routing\Route;
use Symfony\Component\HttpFoundation\Request;
/**
* Checks access for displaying configuration translation page.
```

www.drupalc.com

```
*/
class CustomAccessCheck implements AccessInterface {
  /**
   * A custom access check.
   *
   * @param \Drupal\Core\Session\AccountInterface $account
   *   Run access checks for this account.
   */
  public function access(AccountInterface $account) {
    // Check permissions and combine that with any custom access checking
needed. Pass forward
    // parameters from the route and/or request as needed.
    return $account->hasPermission('do example things') &&
$this->someOtherCustomCondition();
  }
}
```

访问检测方法的参数解析与正常路由的参数解析是相同的，正常情况下仅仅使用$result 对象，这个方法可以使用$route,$route_match,$account。

请注意不要混淆来自于{user}的$user 变量和 AccountInterface 接口变量$account。后者完全限定的类名为 Drupal\Core\Session\AccountInterface，因为在 session 中的用户即为当前登录的用户。在 foo/bar{user}上运行 AccessResult::allowedIfHasPermission($user,'administer something'，路径的访问授权将会基于访客，这是不对的。正确的代码应该是 AccessResult::allowedIfHasPermission($account,'adinister something')。

别外，你还必须在.services.yml 文件中添加访问检测的类，使它成为一项服务，例如 example.services.yml 如下:

```
services:
  example.access_chécher:
    class: Drupal\example\Access\CustomAccessCheck
    arguments: ['@current_user']
    tags:
      - { name: access_check,applies_to: _example_access_check }
```

在路由中添加'_example_acess_check'标志，将会唤醒路由访问检测(通过在 applies()方法中匹配已实现的方法)。然后调用 access()方法实现自定义访问检测。你可以使用来自于第三方的数据服务、自定义数据检测等。

当然，如果你能确定正确的路由，就无需为访问检测器定义一个自定义的请求标志。

www.drupalc.com

有时，你添加了访问检测用于单个路由，这种情况，你可以极大地简化这一过程，你可以在路由定义中指定一个如方法一样的控制器。

```
example:
  path: '/example'
  defaults:
_controller: '\Drupal\example\Controller\ExampleController::content'
  requirements:
    _custom_access: '\Drupal\example\Controller\ExampleController::access'
/**
* @file
* Contains \Drupal\example\Controller\ExampleController.
*/
namespace Drupal\example\Controller;
use Drupal\Core\Session\AccountInterface;
use Symfony\Component\HttpFoundation\Request;
/**
* Builds an example page.
*/
class ExampleController {
  /**
   * Checks access for a specific request.
   *
   * @param \Drupal\Core\Session\AccountInterface $account
   *   Run access checks for this account.
   */
  public function access(AccountInterface $account) {
    // Check permissions and combine that with any custom access checking
needed. Pass forward
    // parameters from the route and/or request as needed.
    return AccessResult::allowedIf($account->hasPermission('do example
things') && $this->someOtherCustomCondition());
  }
}
```

在这种情况，访问方法是在同一个类上的控制器本身，因此无需定义单独的服务。可用的参数与服务定义的方法是一样的。

在控制器内容方法中进行访问检查

最后，出于某些原因，如果你需要或者想在控制器内容方法上完成访问检查，你可以在路由定义时设置_access:true，并在控制器内容方法上进行访问检查。如果你使用了这种方法，请注意当页面没有找到或者访问被拒绝时，必须返回一个合适的 Symfony 路由异常。

CSRF 访问检查

CSRF(跨站点的请求伪造)保护现在已集成到路由访问系统，它应该被用于任意完成某种没有使用表单回调的操作的 URL 上。Drupal 8 以前，必须在 URL 上添加一个查询参数动态令牌(token)，并在回调函数中检查令牌。现在你只须在路由定义中使用_csrf_token 请求就可以了。这样做将会自动地添加令牌作为查询字符串，并且这个令牌会被检查。

```
example:
  path: '/example'
  defaults:
    _controller: '\Drupal\example\Controller\ExampleController::content'
  requirements:
    _csrf_token: 'TRUE'
```

注意，为了添加 token，必须通过 url 生成器服务来产生链接，而不是手工构造一个路径。

www.drupalc.com

15.6 修改或增加已存在的路由

任意在 YAML 文件中定义的静态路由或者动态路由都能修改。你可以触发通过 RoutingEvents::ALTER 事件来修改路由集合。

修改已存在路由

在建立路由后(当启用模块或清除缓存)，RoutingEvents::ALTER 事件触发路由修改过程。\Drupal\Core\Routing\RouteSubscriberBase 类包含一个事件监听器用来监听这些事件。你可以通过实现该类的 alterRoutes(RouteCollection $collection)方法来修改已存在的路由。

下面的例子修改用户模块中的两个路由。模块中的 src/Routing/RouteSubscriber.php 文件如下:

```
namespace Drupal\example\Routing;
use Drupal\Core\Routing\RouteSubscriberBase;
use Symfony\Component\Routing\RouteCollection;
/**
* Listens to the dynamic route events.
*/
class RouteSubscriber extends RouteSubscriberBase {
  /**
   * {@inheritdoc}
```

```
  */
  public function alterRoutes(RouteCollection $collection) {
    // Change path '/user/login' to '/login'.
    if ($route = $collection->get('user.login')) {
      $route->setPath('/login');
    }
    // Always deny access to '/user/logout'.
    // Note that the second parameter of setRequirement() is a string.
    if ($route = $collection->get('user.logout')) {
      $route->setRequirement('_access', 'FALSE');
    }
  }
}
```

\Drupal\example\Routing\RouteSubscriber::alterRoutes 方法是一个事件签订者，因为它扩展至 RouteSubscriberBase。因此，这个类必须被注册成为事件签订服务。

在模块中定义 example.services.yml 文件，如:

```
services:
  example.route_subscriber:
    class: Drupal\example\Routing\RouteSubscriber
tags:
    - { name: event_subscriber }
```

www.drupalc.com

在已存在的动态路由上添加路由

你可以使用 alterRoutes()方法添加动态路由。如果动态路由是独立的，这个类没有实现 preferred 方法，并且应该用简化的路由回调替换事件签订。然而，如果动态路由是依赖于其它动态路由，你需要实现一个扩展至 RouteSubscriberBase 的类。并调整 getSubscribedEvents()方法中的事件权重。

15.7 路由参数

Drupal 8 的路由中可以使用占位符，占位符是包含在 URL 中的动态参数。如 /article/{id}，Id 就是一个动态参数，它的值会随着具体的 URL 确定如/article/50，Id 值为 50。动态的参数可以在控制器方法中使用，只需引用与占位符相同名称的变量即可，如$id。下面是一个 example.routing.yml 例子:

```
example.name
  path: '/example/{name}'
```

```
  defaults:
    _controller: '\Drupal\example\Controller\ExampleController::content'
  requirements:
    _permission: 'access content'
```

以上代码中的{name}就是一个占位符或路径参数，在控制器方法中可以使用$name 变量访问它的值。这与 Symfony 路由不同，Drupal 的请求分布在两个'/'之前，参数在最后一个'/'之后。

```
<?php
class ExampleController {
   //...
   public function content($name) {
      // Name is a string value
      // Do something with $name
   }
}
?>
```

在大多数 PHP 代码中变量名称并不那么重要，但这里变量名称必须和路由中的占位符相匹配，以便于占位符作为参数传入控制器，参数的顺序并不重要。

当在 URL 中使用节点 ID 时，这个 ID 将会自动地由 ParamConverter 系统转换成一个节点对象。具体请阅读路由参数转换一节。最多能使用 9 个参数，向变量传递 URL 组件，请阅读在路由中使用参数。

www.drupalc.com

15.8 路由参数转换

Drupal 8 的路由中可能包含占位符，占位符用于为 URL 中的动态参数提供一个占位。系统能转换占位符的值到真正的对象实例。例如，在路径'/node/{node}'中，{node}就是一个占位符。参数转换系统(ParamConverter system)会将路径中的参数自动转换为一个节点对象实例。

菜单参数转换意为着对菜单参数进行转换以便在控制器中使用。参数可以是一个对象或数组。

让我们看看在 Drupal 7 中 my_module 例子:

```
function my_module_menu() {
   $item['node/%my_menu/mytab'] = array(
      //...
      //...
```

```
  );
}
```

函数 my_module_menu 实现了 hook_menu()，它显示了一个带有参数 %my_menu 的菜单项。这里我们假定在 url 传递参数后，菜单项接收到一个对象并交由回调函数进行处理。例如，我们想通过点击 node/1/mytab 来加载一个节点 id 为 1 的节点。

为了在 Drupal 7 中实现上面所说，这需要我们创建一个加载函数，代码如下:

```
function my_menu_load($arg){
  //do whatever with argument and return your values
}
```

菜单的页面回调函数将会接收到 load 函数反回的值。

在 Drupal 8 中，已经不再使用 hook_menu()函数了，那又怎么实现的呢？Drupal 8 采用一种基于 YAML 格式的配置文件来定义。Drupal 8 构造了一个参数转换接口。为了实现上面的例子，我们需要完成下面几件事:

1.创建 my_module.routing.yml

2.创建 my_module.services.yml 描述自定义参数转换。

3.实现在 my_module.servies.yml 中定义的自定义参数转换类。

4.实现在 my_module.routing.yml 中定义的菜单回调。

my_module.routing.yml 如下:

```
my_module.mymenu:
  path: '/node/{my_menu}/mytab'
  defaults:
    _title: 'My Title'
    _form: '\Drupal\mymodule\Form\MyModuleformControllerForm'
  options:
    parameters:
      my_menu:
        type: my_menu
```

上面的例子是 Drupal 8 中一个典型的路由。以上描述的路由将会依据传递给它的 my_menu 参数来渲染一个表单。

在实体类型中，你不需要实现参数转换类。而是使用 entiry: entiry_type 来代替 type: my_menu。

在页面回调函数使用传递来的参数变量需与路由中的占位符的名称相同。例如，在路由文件中已定义的 my_menu 参数名，在回调函数中会接收到已转换类型的 $my_menu 变量。

创建 my_module.services.yml:

```
services:
  my_menu:
    class: Drupal\mymodule\ParamConverter\MyModuleParamConver
tags:
      - { name: paramconver }
```

确保 tags 的值中包含 paramconverter，这将有助于 Drupal 清除缓存并建立相应的服务。

创建参数转换回调

```
namespace Drupal\mymodule\ParamConverter;
use Drupal\Core\ParamConverter\ParamConverterInterface;
use Drupal\node\Entity\Node;
use Symfony\Component\Routing\Route;
class MyModuleParamConverter implements ParamConverterInterface {
  public function convert($value, $definition, $name, array $defaults) {
    return Node::load($value);
  }
  public function applies($definition, $name, Route $route) {
    return (!empty($definition['type']) && $definition['type'] == 'my_menu');
  }
}
```

上面类实现了由 Drupal 8 核心提供的 ParamConverterInterface 接口，并实现了以下两个函数:

1.public function convert():处理 URL 中参数的逻辑，上面的例子我们将参数转换为一个节点对象。

2.public function applies():这是一个验证函数，用以描述哪儿的参数转换将会被应用。定义变量接收在 routing.yml 文件中定义的参数。因为我们只想这一转换应用到 my_menu 类型的参数，因此作了相应的检测。

最后定义菜单的回调类，它位于 src/Controller/MyModuleControllerForm.php。如下:

```
Class MyModuleformControllerForm extends FormBase{
```

```
  public function buildForm(array $form, FormStateInterface $form_state,
NodeInterface $my_menu = NULL) {
    // $my_menu will be converted object from convert function above.
  }
}
```

上面的回调集中在 buildForm 函数中，因为那是接收已转找参数的地方。

注意:变量名必须与占位符参数{my_menu}相匹配。参数的返回类型依赖于在 MyModuleParamConverter.php 中定义的转换函数的返回值的类型。

15.9 在路由中使用实体参数

如果想在路径中使用与内容或配置实体类型相关联的参数则需要使用预定义的参数名。例如使用 user 对象:

```
example.user
  path: 'example/{user}'
  defaults:
    _controller: '\Drupal\example\Controller\ExampleController::content'
  requirements:
    _permission: 'access content'
```

预定义参数默认没有静化

这意着你应该在使用这些参数之前进行净化。你可以在 requirements 中使用正则表达式对参数进行验证，以降低因代码注入带来的危害。例如:

```
example.user
  path: 'example/{user}'
  defaults:
    _controller: '\Drupal\example\Controller\ExampleController::content'
  requirements:
    _permission: 'access content'
    user: ^[a-zA-X0-9_]+
```

这里将不会对 user 参数进行类型转换，因为 user 是一个实体类型名称，它将会自动地获得一个用户对象。在控制器方法上定义与占位符相同名称的参数即可获得用户对象传入:

```
use Drupal\Core\Session\AccountInterface;
use Symfony\Component\HttpFoundation\Request;
```

www.drupalc.com

```
class ExampleController {
  // ...
  public function content(AccountInterface $user, Request $request) {
    // Do something with $user.
  }
}
```

它是可以工作的，原因有二:

1.$user 是一个实现了 AccountInterface 接口的 Drupal\user\Entity\User 对象的实例。输入类型提示 Drupal\user\UserInterface 或者 Drupal\Core\Entity\EntityInterface 将会正常传递参数。

2.带有类型提示 Request 的对象会自动传入(不用管参数名)。

在上面的例子中，如果 URL 中包含的用户 ID 不存在，系统将自动返回一个 404 错误信息。

表单也可以使用这样的 URL 参数。下面的例子使用了账户对象作为参数传入:

```
example.user_form
  path: 'example/form/{user}'
  defaults:
    _form: '\Drupal\example\Form\ExampleForm'
  requirements:
    _permission: 'access content'
namespace Drupal\example\Form;
use Drupal\Core\Form\FormBase;
use Drupal\Core\Session\AccountInterface;
class BasicForm extends FormBase {
public function buildForm(array $form, FormStateInterface $form_state,
AccountInterface $user = NULL) {
    // Do something with $user in the form
  }
}
```

请注意:向表单提供实体数据的这些方法不用于实体表单，如添加、编辑、删除表单的实体数据。

如果同一个路由需要同类型的两个实体，控制器方法上的类型提示将不会工作，但上节讲的基于占位符名称的参数转换仍然实用，但需在配置文件中作小小的改动如下:

```
route_with_two_nodes:
  path: '/foo/{node1}/{node2}
```

```
  defaults:
    _controller: '\Drupal\example\Controller\ExampleController::foo'
  options:
    parameters:
      node1:
        type: entity:node
      node2:
        type: entity:node
```

```
use Drupal\node\NodeInterface;
class ExampleController {
  function foo(NodeInterface $node1, NodeInterface $node2) {
  }
}
```

虽然类型提示没有工作，但为了便于阅读，建议仍然写上类型提示。

可选参数

当为参数提供了默认值后，路由中的参数是可以省略的。想象一下，你写了一个表单控制器，它允许人们报告问题(例如 bug reports,feature requests and support requests)，并且如果省略了报告问题的类型，则它的类型是'support request'。则可以像下面这样写配置:

```
issue.report_form:
  path: 'report/{issue_type}'
  defaults:
    _controller: '\Drupal\issue\Controller\IssueController::report'
    issue_type: 'support-request'
  requirements:
    _permission: 'report issue'
```

如果我们发现了 report 的请求，相应的控制器将接收到$issue_type 参数，它的值为'support_request'。我们可以在 URL 中加上参数来覆写这一个默认值，如'report/bug'。

参数的默认值还可以用于帮助路由修复路径。一个假想的例子，我们的 SEO 专家发现我们提交 bug 报告的表单在路径'report-a-bug'上是可用的。我们可以重用前面例子中的控制器，并为它提供一个不同的默认值'issue_type'。这样路由器就知道这个参数已存在并将其传给控制器。代码如下:

```
issue.report_a_bug:
  path: 'report-a-bug'
  defaults:
    _controller: '\Drupal\issue\Controller\IssueController::report'
```

```
    issue_type: 'bug'
  requirements:
    _permission: 'report issue'
```

15.10 存取路由参数中的原始值

如果想获取 URL 路径参数中的原始数据，而不是转换后的数据，例如想想获取用户 ID，你可以在参数转换以前调用 $request->attributes-get('_raw_variables')->get('user')来访问用户 ID。更好的方式是调用$route_match->getRawParameters('user')获取原始数据。

与$request 对象相似，你可以通过在控制器方法上传递$route_match 参数来获取 route match 对象。其它情况可使用'current_route_match'服务。

15.11 实现自定义参数转换

一般来说是不需要进行自定义参数转换的，因为所有的内容和配置实体都会自动转换，你应该尽可能地使用系统提供的自动转换。如果你非要进行自定义转换，请继续阅读。

参数转换由 ParamConverterManager 进行管理。为了实现新的参数转换，请实现 ParamConverterInterface 接口。

一个简单的例子是语言转换器的实现。

在你的 convert()方法中，包含需要加载的对象的代码。也可以直接进行数据库查询。在 applies()方法中指定你的参数转换用于哪个路由。$definition['type']是你在 routing 文件中申请的类型。如下面的例子:

```
example.myroute:
  # {example_object} declares the parameter.
  # This is the parameter "label".
  # @see https://www.drupal.org/node/2186285
  path: '/example/{example_object}'
  options:
    parameters:
      # The parameter label. Corresponds to the string
      # used in "path" above.
      example_object:
```

```
    # The type name of the parameter.
    # This value is passed as-is to your applies() method,
    # in $definition['type'].
    type: 'example-type'
```

参数转换属于一项服务，因此你应该在 example.services.yml 文件中登记参数转换用到的类。如下:

```
services:
  example.param_converter:
    class: Drupal\example\Routing\ParamConverter
    tags:
      - { name: paramconverter }
```

Paramconverter 标签指出这是一个路由参数转换服务。

www.drupalc.com

15.12 验证路由参数

Drupal 8 的路由参数值在路由系统中使用前应该进行验证，以减少因非法的参数值给系统带来的危害。参数值可能是字符串值、整型值、或其它值。我们可以为其指定一个正则表达式用以验证参数的合法性，如果参数验证失败，就会返回页面未找到。

我们使用正则表达式来约束路由中占位符的值，如下面的例子中，占位符{name}的值只能包含大写和小写字母。

```
example.user
  path: '/example/{name}'
  defaults:
    _controller: '\Drupal\example\Controller\ExampleController::content'
  requirements:
_permission: 'access content'
    name: '[a-zA-Z]+'
```

15.13 提供动态路由

在 YAML 中定义路由是最简单的，这种方式定义的路由称为静态路由。事实上，不是所有的路由都是静态的，出于某些原因，可能需要定义动态路由，例如视图

模块，在添加视图时会定义视图的路径，这个路径并不是在 YAML 中定义好的，这需要动态附加。

实现动态路由

为了实现动态路由，我们需要在路由定义文件(.routing.yml)中使用 route_callbacks 键，这个键用来定义路由回调。出此之外，还可以在服务中使用这一个键定义路由回调。

例子:

路由回调:

```
route_callbacks:

  - '\Drupal\example\Routing\ExampleRoutes::routes'
```

定义成一项服务:

```
route_callbacks:

  - '\example.service:routes'
```

请注意:route_callbacks 在路由文件中属于最顶层的键，下面是一个例子:

```
views.ajax:
  path: '/views/ajax'
  defaults:
    _controller: '\Drupal\views\Controller\ViewAjaxController::ajaxView'
  options:
    _theme: ajax_base_page
  requirements:
    _access: 'TRUE'
route_callbacks:
  - 'views.route_subscriber:routes'
```

从这个例子中看出 route_callbacks 是最顶层的键，从 8.2.0 开始已不再需要 _theme: ajax_base_page 了。

要定义动态路由，需在路由回调中返回\Symfony\Component\Routing\Route 类的实例或者\Symfony\Comonent\Routing\RouteCollection 类的实例，前者定义单个路由，后者定义路由集合。这两个类的实例都可以使用 new 关键字创建，前者的参数与 YAML 文件中的键差不多，只不过这里使用变量或数组代替。请看一个实例:

```
/**
```

www.drupalc.com

```
* @file
* Contains \Drupal\example\Routing\ExampleRoutes.
*/
namespace Drupal\example\Routing;
use Symfony\Component\Routing\Route;
/**
* Defines dynamic routes.
*/
class ExampleRoutes {
  /**
   * {@inheritdoc}
   */
  public function routes() {
    $routes = array();
    // Declares a single route under the name 'example.content'.
    // Returns an array of Route objects.
    $routes['example.content'] = new Route(
      // Path to attach this route to:
      '/example',
      // Route defaults:
      array(
        '_controller' =>
'\Drupal\example\Controller\ExampleController::content',
        '_title' => 'Hello'
      ),
      // Route requirements:
      array(
        '_permission'  => 'access content',
      )
    );
    return $routes;
  }
}
```

RouteCollection 对象因是一个路由集合，因此需先定义好 Route 对象，然后使用 RouteCollection 类的 Add 方法将其添加到路由集合中。

```
/**
* @file
* Contains \Drupal\example\Routing\ExampleRoutes.
*/
namespace Drupal\example\Routing;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\RouteCollection;
```

www.drupalc.com

```
/**
* Defines dynamic routes.
*/
class ExampleRoutes {
  /**
   * {@inheritdoc}
   */
  public function routes() {
    $route_collection = new RouteCollection();
    $route = new Route(
      // Path to attach this route to:
      '/example',
      // Route defaults:
      array(
        '_controller' =>
'\Drupal\example\Controller\ExampleController::content',
        '_title' => 'Hello'
      ),
      // Route requirements:
      array(
        '_permission'   => 'access content',
      )
    );
    // Add the route under the name 'example.content'.
    $route_collection->add('example.content', $route);
    return $route_collection;
  }
}
```

一般来说，可以使用上面的例子代码来定义动态路由。视图(Views)、图像(Image)、RESTful Web 服务等模块就是使用这种方式来附加动态路由。下面是 Drupal 核心中图像模块动态产生的路由的一段代码:

```
/**
* @file
* Contains \Drupal\image\EventSubscriber\RouteSubscriber.
*/
namespace Drupal\image\Routing;
use Symfony\Component\Routing\Route;
/**
* Defines a route subscriber to register a url for serving image styles.
*/
class ImageStyleRoutes {
  /**
```

www.drupalc.com

```
   * Returns an array of route objects.
   *
   * @return \Symfony\Component\Routing\Route[]
   *   An array of route objects.
   */
  public function routes() {
    $routes = array();
    // Generate image derivatives of publicly available files. If clean URLs are
    // disabled image derivatives will always be served through the menu
system.
    // If clean URLs are enabled and the image derivative already exists, PHP
    // will be bypassed.
    $directory_path =
file_stream_wrapper_get_instance_by_scheme('public')->getDirectoryPath();
    $routes['image.style_public'] = new Route(
      '/' . $directory_path . '/styles/{image_style}/{scheme}',
      array(
        '_controller' =>
'Drupal\image\Controller\ImageStyleDownloadController::deliver',
      ),
      array(
        '_access' => 'TRUE',
      )
    );
    return $routes;
  }
}
```

基于其它动态路由的动态路由

有时在定义动态路由时，需要用到其它动态路由，例如你想在使用视图模块创建的页面添加同样的 tab。在这种情况，使用上面的方式将不行。你需要实现响应 onAlterRoutes 事件的服务。细节请阅读　修改或增加已存在的路由。

15.14 路由对象Route CurrentRouteMatch RouteMatch Url

在 Drupal 8 中与路由相关的对象包括 Route，CurrentRouteMatch，RouteMatch，Url 等。

Route 对象只不过是 YML 形式的路由的 PHP 形式，它没有什么方法可供调用，除了 getter，像这样的对象称为值对象。大多数情况它用作参数。

RouteMatch(\Drupal\Core\Routing\RouteMatchInterface)对象，返回路由匹配的结果。它包含路由名、Route 对象、与原生参数相关的参数。例如，如果它匹配到/node/123，那么路由对象是/node/{node}，原生参数是[1=>123]，与它相关的参数是[1=>Node::load(123)]。RouteMatch 没有什么有用的方法，除了 getter。

CurrentRouteMatch(\Drupal\Core\Routing\CurrentRouteMatch) 对象，返回路由匹配的结果。它与 RouteMatch 对象差不多，只不过 CurrentRouteMatch 包含当前对象。它可以通过 current_route_match 服务来访问它并且它被发送到任何 hook_help 方法。别外你可以使用 getter 方法。

Url 对象包含路由名和参数以及产生 Url 的选项。它包含了一些有用的方法，比如获取系统路径或者获取渲染数组等。

www.drupalc.com

第 16 章 插件开发

引入插件主要是为了实现 Drupal 的可扩展性，Drupal 的模块或子系统可以定义插件接口，其它模块可以实现这些插件接口来为 Drupal 添加一些小功能。这些小功能的开启或关闭都是很方便的。在 Drupal 内核系统中也有很多地方使用了插件，例如在区块系统中，区块类型即为插件，在实体/字段系统中，实体类型、字段类型、字段格式化器、字段部件等都是插件，在图像处理系统中，图像样式和图像工具都是插件，在搜索系统中，搜索页类型是插件。

插件按照插件类型来分组，每种类型定义成一个接口。使用插件管理器服务来管理插件类型，它使用插件 discovery 方法来发现插件类型并使用 plugin factory 来实例化它们。本章将深入讨论 Drupal 插件理论以及例子。

16.1 插件开发概述

插件是一些小功能，插件类型是对插件的分类，同种类型的插件的功能相似。Drupal 有许多种不同的插件类型和许多插件。例如'Field widget'是一个插件类型，这个类型的不同插件是一个具体的插件。管理员可以在字段管理界面中选择字段使用的插件。

D8 的插件系统为模块开发者提供了许多可重用的代码组件，这些组件允许开发者在自已的代码中使用并支持在用户界面中管理插件。插件是由模块定义的，一个模块提供不同种类的插件，不同的模块可以提供某一种类的插件。

插件系统包含三个基本元素:

1.插件类型(Plugin Types)

插件类型实际上是一个控制类，它定义如何发现并实例化这个插件类型。插件类型描述了具有这种类型的所有插件的功能目的；如后端缓存、图像操作、区块等。

2.插件发现(Plugin Discovery)

插件发现是从可用的插件仓库中寻找适合于当前使用的插件类型的过程。

3.插件工厂(Plugin Factory)

插件工厂响应插件的实例化。

另外插件系统还包含了几个有用的组件:

插件派生(Plugin Derivatives)

www.drupalc.com

插件派生允许一种插件在多个地方工作。当一个用户使用插件输入数据时，这是很有用的。例如，如果在屏幕上使用一个插件放置了一个菜单，当管理员创建一个新菜单时，放置菜单仍然是可用的，而不需要再使用一个插件。插件派生允许在用户界面上的同一个地方显示多个插件。插件派生的主要目的是提供部份配置的插件，因为在用户界面中插件的第一个实体与其它实体是无法区分的，这减轻了管理员使用这些插件的负担。

发现封装器(Discovery Decorators)

发现封装器是另一种可用的寻找插件的方法，它是对已存在的发现方法的封装。当前的核心提供了 CacheDecorator，它会缓存选择插件类型的过程。如果需要，这个类也可以进行扩展。

插件映射(Plugin Mappers)

插件映射你对插件实例进行映射，如映射一个字符串到一个插件实例。插件类型可以使用它的方法返回已配置的并实例化的插件，而不用开发者使用 API 手工实例化和配置插件实例。

www.drupalc.com

16.2 为什么要使用插件

插件实际上是由 PHP 定义的接口，接口规定了同类插件的标准。插件系统发现每一个实现了接口的类，并处理它们(默认是提供一个注释)并为插件类型提供一个工厂类。

插件还是服务?

插件

插件通过一个公共的接口实现不同的行为。例如，考虑图像变换。图像变换一般有缩放、裁剪、去色等。每种变换类型基于相同的数据和相同的方式。它接受一个文件，完成图像变换，返回修改后的图像。然而，每种变换效果是不同的。

服务

服务提供相同的功能，并且是可交换的，不同点仅仅在于它们的内部实现。考虑缓存，缓存应该提供获取(get)、设置(set)、过期(expire)等方法。用户只需要一个缓存，从一种缓存换成另一种缓存没有任何功能上的差别。至于缓存中的方法的实现机制用户是不会管的。所以缓存应该设计成一种服务。

如果你提供了一个 UI 想让用户进行配置，你应该使用插件系统。

D8 插件与 D7 info hook

使用 D7 的 hook 功能可以完成许多事，但那不是面向对象的开发方法。D8 的插件系统替换了 D7 的 info hook 功能。因为它提供了更加强大的机制来替换功能逻辑，这在以前是无法完成的。

在 Drupal 7 中，区块使用 hook_block_info()，定义了大量的的区块 hook，现在它已经被区块插件类型替换，所有这些 hook 都被定义成了相应的插件方法。我们创建了一个抽象的区块类为核心中的区块提供了默认的行为，区块插件只需对某些方法进行覆盖，就可以完成相应的功能。一个很好的例子是 hook_block_view 和 BlockBase::blockBuild()。blockBuild 方法看起来和 hook_block_view 差不多，但是类中的每一个方法都是一种行为的响应，而不需要在模块的主控函数中使用 switch 分支语句来完成响应。同时，模块可以实现 hook_block_alter()函数，这个替换了以前的 hook_block_info_alter()函数。它允许你更好的控制区块数据表现。这个函数允许你置换当前区块的类，来达到完全控制它的目的。

16.3 基于注释的插件

Drupal 8 的插件系统使用文件中的注释来注册插件，并描述它们的基本信息。Drupal 8 的核心提供了区块插件、字段格式化器、字段控件、视图插件、条件插件、迁移等。在阅读本文前，你可以先看看它们的代码以便更好地理解插件。

PSR-4 标准

插件基于 PHP 文件注释的注册方法符合 PSR-4 标准，为了注册一个插件需将插件的文件放置在模块目录下的 src/Plugin/插件类型/目录下。例如:core/modules/ckeditor/src/Plugin/CKEditorPlugin/Internal.php。为了告诉系统你定义的是一个插件，需在类定义前定义插件注释：

```
/**
* @Plugin(
*
*)
*/
```

比如
core/modules/user/src/Plugin/Validation/Constraint/UserNameUnique.php 就使用了这种注释方式。注释包含了 ID 和 label，如下:

```
/**
* Check if a user name is unique on the site.
*
* @Plugin(
```

www.drupalc.com

```
*   id = "UserNameUnique",
*   label = @Translation("User name unique", content   = "Validation"),
* )
*/
class UserNameUnique extends Constraint {
..
}
```

为什么要使用注释

与其它的发现机制不同，文件中的插件注释是实现插件的类的一个不可缺少的部份。这使发现一个插件变得更加轻松，通过复制代码就可以轻易地创建一个插件。

插件注释也可以使用更加复杂的结构化数据，并指出哪些字符串可被翻译。一般来说，插件应该有一个与之相关联的自定义注释类，它可以用来编写文档和为插件的基本数据设置缺省值。

另外，这种方式会获得更好的性能，因为使用这种机制 Drupal 找到一个插件使用更少的内存。原来的实现方式需要在每个类中定义一个 getinfo()方法。这意味着每一个这样的类都需要加载到内存中才可以获取信息，这样就增加了 PHP 对内存的需求。而这种方式只需解析 PHP 文件中的简单标记，不需要执行整个 PHP 文件，使内存使用最小化。

插件注释的语法

注释使用固定的键值描述结构化数据，并支持嵌套。注释语法源自于 Doctrine 项目，但它的风格与 Drupal 有细微的不同。

1.首先是插件 ID，相当于 Drupal 7 中的机器名，它是插件类型的唯一 ID。
2.根键可以使用双引号。
3.子键必须使用双引号。
4.不要使用单引号，使用单引号将抛出一个异常。
5.可用的数据类型有:

String:字符串类型必须使用双引号例如"foo"。如果字符串中包含双引号则需再使用一对双引号以转义，例如 "The ""On"" Value"。

Numbers:不能使用引号例如 21，使用引号将被解析成字符串。

Booleans:不能使用引号例如 TRUE 或者 FALSE，使用引号将被解析成字符串。

Lists:形如数组，例如:

```
base = {
  "node",
  "foo",
}
```

请注意列表中的最后一个元素后的逗号；这不是一个打印错误！它有助于防止解析错误，比如把另一个元素被放置到　列表的结尾。

Maps:数据映射，形如数组。例如:

```
edit = {
    "editor" = "direct",
}
```

允许使用常量

自定义注释类

新插件类型应该总是使用自定义的注释类。让我们看一过原生的例子 plaintext 格式化器，它在 text/src/Plugin/field/formatter/TextPlainFormatter.php 中定义。它使用了自已的注释类 FieldFormater，它扩展至 \Drupal\Component\Annotation\Plugin。

```
/**
* Check if a user name is unique on the site.
*
* @Plugin(
*   id = "UserNameUnique",
*   label = @Translation("User name unique", content  = "Validation"),
* )
*/
class UserNameUnique extends Constraint {
..
}
```

在你自已的插件类型中使用注释

如果你想在你自己的插件中使用注释，你只需扩展 DefaultPlguinManager 并使用 AnnotatedClassDiscovery，请看下面的例子。

DefaultPluginManager 构造函数的第一个参数是指定插件的搜索路径，在这个例子中，将会在$module/src/Plugin/Field/FieldFormatter 目录中搜索插件。最后一个参数是上面定义的自定义注释类。

```
use Drupal\Component\Plugin\Factory\DefaultFactory;
use Drupal\Core\Cache\CacheBackendInterface;
use Drupal\Core\Extension\ModuleHandlerInterface;
use Drupal\Core\Plugin\DefaultPluginManager;

class FormatterPluginManager extends DefaultPluginManager {

  /**
   * Constructs a FormatterPluginManager object.
   */
```

www.drupalc.com

```
  public function __construct(\Traversable $namespaces,
CacheBackendInterface $cache_backend, ModuleHandlerInterface
$module_handler, FieldTypePluginManagerInterface $field_type_manager) {
    parent::__construct('Plugin/Field/FieldFormatter', $namespaces,
$module_handler, 'Drupal\Core\Field\FormatterInterface',
'Drupal\Core\Field\Annotation\FieldFormatter');

    $this->setCacheBackend($cache_backend,
'field_formatter_types_plugins');
    $this->alterInfo('field_formatter_info');
    $this->fieldTypeManager = $field_type_manager;
  }
}
```

注入的名字空间来自于依赖注入容器，例如 FieldBundle:

```
use Drupal\Component\Plugin\Factory\DefaultFactory;
use Drupal\Core\Cache\CacheBackendInterface;
use Drupal\Core\Extension\ModuleHandlerInterface;
use Drupal\Core\Plugin\DefaultPluginManager;

class FormatterPluginManager extends DefaultPluginManager {

  /**
   * Constructs a FormatterPluginManager object.
   */
  public function __construct(\Traversable $namespaces,
CacheBackendInterface $cache_backend, ModuleHandlerInterface
$module_handler, FieldTypePluginManagerInterface $field_type_manager) {
    parent::__construct('Plugin/Field/FieldFormatter', $namespaces,
$module_handler, 'Drupal\Core\Field\FormatterInterface',
'Drupal\Core\Field\Annotation\FieldFormatter');

    $this->setCacheBackend($cache_backend,
'field_formatter_types_plugins');
    $this->alterInfo('field_formatter_info');
    $this->fieldTypeManager = $field_type_manager;
  }
}
```

为了调用自定义的插件管理器，你需要注入 Drupal 的名字空间到类的构造函数中。

www.drupalc.com

```
$type = new
CustomPluginManager(\Drupal::getContainer()->getParameter('container.namespaces'));
```

16.4 视图中基于注释的插件

在 Drupal 核心的大部份代码中，视图使用基于注释的插件发现机制。

注释类

对于不同的视图插件有不同的插件注释类。在视图插件中定义注释类，例如，视图显示类型"Block"，使用了注释类"ViewsDisplay"，它在下面的代码中由@ViewDisplay 指出。

```
/**
* The plugin that handles a block.
*
* @ingroup views_display_plugins
*
* @ViewsDisplay(
*   id = "block",
*   title = @Translation("Block"),
*   help = @Translation("Display the view as a block."),
*   theme = "views_view",
*   register_theme = FALSE,
*   uses_hook_block = TRUE,
*   contextual_links_locations = {"block"},
*   admin = @Translation("Block")
* )
*
…
```

注释的参数语法

注册一个新插件时，注释的参数语法也是很重要的。例如，视图参数处理插件(注释类为@ViewArgument)只需在注释区块中定义插件的机器名。假如下面定义了"formula"参数，则应该写成@ViewArgument("formula")。添加其它的信息将会导至一个 Fatal Error。

```
/**
* Abstract argument handler for simple formulae.
*
```

www.drupalc.com

```
* Child classes of this object should implement summaryArgument, at least.
*
* Definition terms:
* - formula: The formula to use for this handler.
*
* @ingroup views_argument_handlers
*
* @ViewsArgument("formula")
*/
```

确定哪些细节可以包含在插件注释中，可以看看 AnnotationBase 类。在上面的例子中注释类是"ViewsHandlerAnnotationBase"，它扩展至"PluginBase"类，并且没有作出改变，不支持使用键值对来定义属性。ViewStyle 插件定义了更多的细节，下面是 ViewStyle 类:

```
<?php
…

/**
* Defines a Plugin annotation object for views style plugins.
*
* @see \Drupal\views\Plugin\views\style\StylePluginBase
*
* @ingroup views_style_plugins
*
* @Annotation
*/
class ViewsStyle extends ViewsPluginAnnotationBase {
/**
   * The plugin ID.
   *
   * @var string
   */
  public $id;

  /**
   * The plugin title used in the views UI.
   *
   * @var \Drupal\Core\Annotation\Translation
   *
   * @ingroup plugin_translatable
   */
  public $title = '';
```

www.drupalc.com

```
/**
 * (optional) The short title used in the views UI.
 *
 *
 * @var \Drupal\Core\Annotation\Translation
 *
 * @ingroup plugin_translatable
 */
public $short_title = '';

/**
 * A short help string; this is displayed in the views UI.
 *
 * @var \Drupal\Core\Annotation\Translation
 *
 * @ingroup plugin_translatable
 */
public $help = '';

/**
 * The theme function used to render the style output.
 *
 * @var string
 */
public $theme;

/**
 * The types of the display this plugin can be used with.
 *
 * For example the Feed display defines the type 'feed', so only rss style
 * and row plugins can be used in the views UI.
 *
 * @var array
 */
public $display_types;

/**
 * The base tables on which this style plugin can be used.
 *
 * If no base table is specified the plugin can be used with all tables.
 *
 * @var array
 */
public $base;
```

www.drupalc.com

```
  /**
   * Whether the plugin should be not selectable in the UI.
   *
   * If it's set to TRUE, you can still use it via the API in config files.
   *
   * @var bool
   */
  public $no_ui;

}
```

上面的 ViewStyle 类定义了一些属性，这些属性可以在插件注释区块中使用，注释区块中的参数可以使用该类定义的属性。例如 Views Grid style 插件包含的注释区块如下:

```
<?php

/**
* @file
* Contains \Drupal\views\Plugin\views\style\Grid.
*/

namespace Drupal\views\Plugin\views\style;

use Drupal\Component\Utility\Html;
use Drupal\Core\Form\FormStateInterface;

/**
* Style plugin to render each item in a grid cell.
*
* @ingroup views_style_plugins
*
* @ViewsStyle(
*   id = "grid",
*   title = @Translation("Grid"),
*   help = @Translation("Displays rows in a grid."),
*   theme = "views_view_grid",
*   display_types = {"normal"}
* )
*/
class Grid extends StylePluginBase {
…
```

www.drupalc.com

可以看到注释区块中引用了@ViewStyle 类，其参数包括
id,title,help,theme,display_types。

16.5 创建插件管理器

插件管理器是一个主控类，它定义了如何发现和实例化某种类型的插件。这个类可以由任何模块调用。阅读这篇文档需要理解 PSR-4。

定义插件管理器

要定义一个插件管理器，你需要定义一个 discovery 方法，还需要定义一个工厂(factory)。推荐使用 DefaultPluginManager 作为基类，因为它已经内建了语言缓存、基于注释的插件发现机制以及其它一些基本功能。

定义服务

插件管理器应该定义成一项服务，最好在服务名前加上 plugin.manager，这样便于阅读。例如 MODULE.services.yml 列出了一项插件管理服务:

```
services:
  plugin.manager.archiver:
    class: Drupal\Core\Archiver\ArchiverManager
    parent: default_plugin_manager
```

插件管理类

```
namespace Drupal\Core\Archiver;

use Drupal\Component\Plugin\Factory\DefaultFactory;
use Drupal\Core\Cache\CacheBackendInterface;
use Drupal\Core\Extension\ModuleHandlerInterface;
use Drupal\Core\Plugin\DefaultPluginManager;

/**
* Provides an Archiver plugin manager.
*
* @see \Drupal\Core\Archiver\Annotation\Archiver
* @see \Drupal\Core\Archiver\ArchiverInterface
* @see plugin_api
*/
class ArchiverManager extends DefaultPluginManager {
```

www.drupalr.com

```
  /**
   * Constructs a ArchiverManager object.
   *
   * @param \Traversable $namespaces
   *   An object that implements \Traversable which contains the root paths
   *   keyed by the corresponding namespace to look for plugin
implementations.
   * @param \Drupal\Core\Cache\CacheBackendInterface $cache_backend
   *   Cache backend instance to use.
   * @param \Drupal\Core\Extension\ModuleHandlerInterface
$module_handler
   *   The module handler to invoke the alter hook with.
   */
  public function __construct(\Traversable $namespaces,
CacheBackendInterface $cache_backend, ModuleHandlerInterface
$module_handler) {
    parent::__construct(
      'Plugin/Archiver',
      $namespaces,
      $module_handler,
      'Drupal\Core\Archiver\ArchiverInterface',
      'Drupal\Core\Archiver\Annotation\Archiver'
    );
    $this->alterInfo('archiver_info');
    $this->setCacheBackend($cache_backend, 'archiver_info_plugins');
    $this->factory = new DefaultFactory($this->getDiscovery());
  }
}
```

上面的代码告诉系统，我们将寻找任何可用的 Plugin/Archiver 注释类，使用它来发现插件，并支持插件派生。其它模块可以通过 hook_archiver_info_alter() 来修改插件定义。系统使用缓存 id 为 archiver_info_plugins 对该定义进行缓存。

注意 DefaultFactory 类的第一个参数是 DiscoveryInterface 类型，但是我们传给它的是 PluginManagerBase 的扩展类，在大多数情况下这样做是对的，主要有以下两个原因，其一，PluginManagerInterface 是扩展至 DiscoveryInterface、FactoryInterface 和 MapperInterface 接口的，参数的类型可以是这些接口类型。其二，插件类型实际上是代理这些接口定义的方法，并添加一些额外的逻辑以适应更复杂的情况。

每一个工厂类需要定义构造函数，因为 FactoryInterface 没有定义__construct() 函数。这对于开发者自己写插件工厂类是很有益处的。

发现封装器

发现封装器是对一个定义发现的类的封装，它实际上也是一个类，它实现了所有相同的方法，并可以提供其它的方法以作一些额外的处理。使用这种机制实际上是对 Drupal 的插件机制的扩展，这样可以使它更易扩展，更加灵活。例如扩展至 DefaultPluginManager 类可以使用下面的方法来派生发现。

```
protected function getDiscovery() {
    if (!$this->discovery) {
      $discovery = new AnnotatedClassDiscovery($this->subdir,
$this->namespaces, $this->pluginDefinitionAnnotationName,
$this->additionalAnnotationNamespaces);
      $this->discovery = new
ContainerDerivativeDiscoveryDecorator($discovery);
    }
    return $this->discovery;
  }
```

使用插件管理器

假定我们的插件管理器已可以使用，首先我们必须调用该插件类:

```
$type = \Drupal::service('plugin.manager.archiver');
```

获取可用的插件列表

```
$plugin_definitions = $type->getDefinitions();
```

获取指定的插件

```
$plugin_definition = $type->getDefinition('plugin_id');
```

创建一个插件实例

```
$plugin = $type->createInstance('plugin_id',['of' => 'configuration values']);
```

www.drupalc.com

16.6 D8 插件发现机制

插件发现是指Drupal系统发现某种插件的处理过程。每一种插件类型都需要实现插件发现方法，具体解释请阅读上一节插件管理器。

插件发现组件实现了 DiscoveryInterface 接口，该接口定义了任何插件发现的类必须具有的方法。下面是这个接口的代码:

```
/**
```

```
* @file
* Contains \Drupal\Component\Plugin\Discovery\DiscoveryInterface.
*/

namespace Drupal\Component\Plugin\Discovery;

/**
* An interface defining the minimum requirements of building a plugin
* discovery component.
*
* @ingroup plugin_api
*/
interface DiscoveryInterface {

  /**
   * Gets a specific plugin definition.
   *
   * @param string $plugin_id
   *   A plugin id.
   * @param bool $exception_on_invalid
   *   (optional) If TRUE, an invalid plugin ID will throw an exception.
   *
   * @return mixed
   *   A plugin definition, or NULL if the plugin ID is invalid and
   *   $exception_on_invalid is FALSE.
   *
   * @throws \Drupal\Component\Plugin\Exception\PluginNotFoundException
   *   Thrown if $plugin_id is invalid and $exception_on_invalid is TRUE.
   */
  public function getDefinition($plugin_id, $exception_on_invalid = TRUE);

  /**
   * Gets the definition of all plugins for this type.
   *
   * @return mixed[]
   *   An array of plugin definitions (empty array if no definitions were
   *   found). Keys are plugin IDs.
   */
  public function getDefinitions();

  /**
   * Indicates if a specific plugin definition exists.
   *
   * @param string $plugin_id
```

www.drupalc.com

```
 *   A plugin ID.
 *
 * @return bool
 *   TRUE if the definition exists, FALSE otherwise.
 */
public function hasDefinition($plugin_id);
}
```

上面的接口定义了三个方法 getDefinition，getDefinitions，hasDefinition。第一个方法返回指定的插件定义，第二个方法返回插件定义集合，第三个方法指出插件定义是否存在。

Drupal 核心中包含四种不同的插件发现类型:

1.StaticDiscovery

StaticDiscovery 允许在插件发现类中直接注册插件。在类中的 protected 变量 $definitions 存储了所有通过方法 setDefinition()注册的插件定义。任何通过这种方法定义的插件可以被唤醒。

2.HookDiscovery

HookDiscovery 类允许使用 Drupal 的 hook_comonent_info()/hook_comonent_info_alter()进行插件发现。使用这种插件发现机制，插件管理器将会调用 info hook 来返回可用的插件列表。

3.AnnotatedClassDiscovery

AnnotatedClassDiscovery 即注释类发现，该发现机制允许你在定义插件的类的文件中插入注释，在注释中按照相关的格式定义插件注释类，AnnotatedClassDiscovery 会对这些文件进行扫描，从而发现插件。这种方式内存占用小。

4.YamlDiscovery

YamlDiscovery 允许你在 yaml 文件中定义插件类型。Drupal 核心主要将它用在 local task 和 local action 上。

16.7 插件定义

插件在用户界面组件中扮演了一个重要的角色，然而插件还可用于其它情景。所有组件结合在一起就可以起到很好的作用。插件定义可以看作是对 Drupal 以前版本的 info hook 的模拟。然而插件系统已经可以当作一个插件抽象层，我们叫它 Discovery。这是另一个主题，但插件定义在这个抽象层中也是很重要的。

www.drupalc.com

定义插件的基本方法是简单地定义一个数组，在数组中定义插件 id 作为键，在该键上再定义一个数组以指出该类插件的属性。如我们定义一个菜单区块插件，它的定义如下:

```
[
  ‘menu_block’ => [
    ‘class’ => ‘\Drupal\system\Plugin\block\block\MenuBlock’,
   ],
];
```

上面的代码定义了插件使用的类，当实例化该类插件时会使用到这个类，这对我们很有用，但它不支持用户界面。我们应该为它至少定义一个标题和一个简短描述，就像下面这样:

```
[
  'menu_block' => [
    'title' => t('Menu Block'),
    'description' => t('A block for displaying system or user generated menus.'),
    'class' => 'Drupal\system\Plugin\block\block\MenuBlock',
  ],
];
```

如果插件发现方式是基于 hook 的，假定我们定义了 hook_block_info()，它的代码大概像这样:

```
function system_block_info() {
  return [
    'menu_block' => [
      'title' => t('Menu Block'),
      'description' => t('A block for displaying system or user generated
menus.'),
      'class' => 'Drupal\system\Plugin\block\block\MenuBlock',
    ],
  ];
}
```

有很多种插件定义的方式。StaticDiscovery 允许你在插件类型中直接注册，HookDiscovery 发现方式正如本文所述，YamlDiscovery 通过解析 yml 文件中的插件定义发现插件，AnnotatedClassDiscovery 类解析文档注释区块来发现插件。也可以创建其它的插件发现机制，只需定义相应的解析类即可。

16.8 插件上下文环境

www.drupalc.com

有时插件需要其它对象来完成操作，这就是插件上下文。大多数需要一定条件的插件都需要插件上下文。让我们来看一个 NodeType 插件的定义。

```
/**
* Provides a 'Node Type' condition.
*
* @Condition(
*   id = "node_type",
*   label = @Translation("Node Bundle"),
*   context = {
*     "node" = @ContextDefinition("entity:node", label =
@Translation("Node"))
*   }
* )
*
*/
```

在插件定义中有三个键，第一个键是必须的即 id，第二个是'label'用于用户界面，最后一个是'content'数组。'content'键定义了一个数组，该数组定义了需求的上下文以实现'evaluate()'方法。在 NodeType 这个例子中，我们需要使用节点来检测它的类型。NodeType::evaluate()方法演示了它的工作过程:

```
/**
   * {@inheritdoc}
   */
  public function evaluate() {
    if (empty($this->configuration['bundles']) && !$this->isNegated()) {
      return TRUE;
    }
    $node = $this->getContextValue('node');
    return !empty($this->configuration['bundles'][$node->getType()]);
  }
```

ContextAwarePluginBase::getContextValue()方法接收一个 name 参数用以响应在上下文数组中定义的键。在这个例子中 name 参数是一个结点实体。该方法支持任何已定义的数据类型。ContextDefinition 注释会传递到 ContextDefinition 类。第一个字符串传递的是上下文定义中的 id，上面的例子为'node'，这个字符串会直接传递不会带有关联键。其它的子参数传递时需带有关联键。正如 NodeType 上下文中定义的，上下文中定义的 label 带有一个'label'键。上下文定义支持以下键:

- label:上下文标签，用于 UI
- required:指定是否为必须，默认为 true
- multiple:指定是否有多个条件，默认为 false
- description:上下文的描述，用于 UI

www.drupalc.com

- default_value:默认为 false
- class:默认\Drupal\Core\Plugin\ContextDefinition 类。如果你提供你自己的类，你必须实现 \Drupal\Core\Plugin\Context\ContextDefinitionInterface 接口。上下文定义可以不使用 default_value 键。

16.9 插件派生

插件系统的功能之一是它有能力通知用户界面组件，并且允许用户选择要使用的插件选项。以这种方式呈现的选项与插件选项是一对一的关联关系。然而，这种关联不是必须的，当需要通过用户输入信息时这种关联并不方便。让我们来讨论一下区块。

所有的区块都可以通过用户界面放置并使用。大多数区块都可以通过一个 UI 来呈现它。例如，"Powered by Drupal 区块通过一个区块插件进行配置就能工作得很好。然而，菜单区块也使用相同的逻辑，只是有些 UI 选项不同，如区块标题、描述等。再者，站点创建者建立了一个自定义区块，这个区块可以通过 UI 进行放置。插件派生可以完成这些工作。

插件派生提供了一种简单的方式来扩展单个插件，以使它能在用户界中表现为多个插件。这是通过创建一个单独的类来完成的，这个类可以在插件定义中进行引用。例如，系统菜单区块这一插件定义 \Drupal\system\Plugin\Block\SystemMenuBlock。让我们看一下它的注释:

```
/**
* Provides a generic Menu block.
*
* @Block(
*   id = "system_menu_block",
*   admin_label = @Translation("Menu"),
*   category = @Translation("Menus"),
*   deriver = "Drupal\system\Plugin\Derivative\SystemMenuBlock"
* )
*/
class SystemMenuBlock extends BlockBase implements
ContainerFactoryPluginInterface {
  // … the class definition goes on here.
}
```

这个定义仅仅是一个例子，除了'deriver'键以外，其它键都可以不同。原理上，这个派生类是可以跨插件重用的。实际上，插件的派生类是相似的。让我们看看实际的类:

/core/modules/system/src/Plugin/Derivative/SystemMenuBlock.php:

```
namespace Drupal\system\Plugin\Derivative;

class SystemMenuBlock extends DeriverBase implements
ContainerDeriverInterface {
  /* ...[snip]... */
  /**
   * {@inheritdoc}
   */
  public function getDerivativeDefinitions($base_plugin_definition) {
    foreach ($this->menuStorage->loadMultiple() as $menu => $entity) {
      $this->derivatives[$menu] = $base_plugin_definition;
      $this->derivatives[$menu]['admin_label'] = $entity->label();
      $this->derivatives[$menu]['config_dependencies']['config'] =
array($entity->getConfigDependencyName());
    }
    return $this->derivatives;
  }
}
```

修改插件类型

这个类为 Drupal 中的每一个菜单返回一个派生定义，包括系统产生的和用户创建的都被视为单独的插件定义。在写派生类时，我们只需一些简单的修改就可以支持插件派生功能。插件目前的代码如下:

```
namespace Drupal\block\Plugin\Type;

use Drupal\Component\Plugin\PluginType;
use Drupal\Core\Plugin\Discovery\HookDiscovery;
use Drupal\Component\Plugin\Factory\DefaultFactory;

class BlockPluginType extends PluginType {
  public function __construct() {
    $this->discovery = new HookDiscovery('block_info');
    $this->factory = new DefaultFactory($this);
  }
}
```

www.drupalc.com

我们需要添加一个新的 use 表达式以引入 DerivativeDiscoveryDecorator 名字空间，以便在$this->discovery 中使用。

```
namespace Drupal\block\Plugin\Type;
use Drupal\Component\Plugin\Discovery\DerivativeDiscoveryDecorator;

use Drupal\Component\Plugin\PluginType;
use Drupal\Core\Plugin\Discovery\HookDiscovery;
use Drupal\Component\Plugin\Factory\DefaultFactory;

class BlockPluginType extends PluginType {
  public function __construct() {
    $this->discovery = new DerivativeDiscoveryDecorator(new
HookDiscovery('block_info'));
    $this->factory = new DefaultFactory($this);
  }
}
```

这两处修改就可以使我们的区块插件系统支持任何所需类型的插件派生功能，正如菜单区块的派生处理过程一样。

16.10 创建能在主题中定义的插件

一般来说插件应在模块中定义而不应该在一个类中定义，但有时需要在自定义主题中定义插件，例如 BreakPoints 插件和 Layout 插件。

要创建一个可以在主题中使用的插件，需要在模块的 module.services.yml 文件中传递主题处理器。例如 layout_plugin 模块:

```
services:
  plugin.manager.layout_plugin:
    class: Drupal\layout_plugin\Plugin\Layout\LayoutPluginManager
    arguments: ['@container.namespaces', '@cache.discovery',
'@module_handler', '@theme_handler']
```

在插件管理器文件(src/Plugin/Layout/MyPluginManager.php)中添加主题处理器属性。

```
/**
   * The theme handler.
   *
   * @var \Drupal\Core\Extension\ThemeHandlerInterface
```

```
   */
  protected $themeHandler;
```

在构造函数中给主题处理器赋值:

```
$this->themeHandler = $theme_handler;
```

像下面这样定义 providerExists 方法:

```
  /**
   * {@inheritdoc}
   */
  protected function providerExists($provider) {
    return $this->moduleHandler->moduleExists($provider) ||
$this->themeHandler->themeExists($provider);
  }
```

更好的例子是阅读 BreakPoints 模块和 LayoutPlugin 模块的代码。

www.drupalc.com

第 17 章 实体系统

实体是一个抽象概念，实体有类型和实例之分，例如文章是一个实体类型，而每一篇文章是一个实体对象。实体类型包含一系列的字段，字段又包含了一系列的基本类型的数据。Drupal 8 使用内容实体处理内容，使用配置实体处理配置信息，并有一套功能完备的 API。本章将介绍这些知识。

17.1 Drupal 8 实体概述

实体系统是在 Drupal 7 开发的后期引入的，它定义了加载一个实体的基本标准。贡献模块 entity.module 扩展了实体 API，它支持保存和删除实体以及其它很多增强功能。

这些大量的功能现在已经包含在 Drupal 8 中。现在实体验证有它自己的 API，通过 REST 验证实体。

实体是带有方法的类型化类，通常的方法如$entity->id()，其它方法如$node->getTitle()。这些类型在接口中定义并文档化。

实体需要 handler 支持。

Storage hander:支持加载、保存和删除实体。它定义了对内容修订、翻译、字段等的默认支持。另外还有一些如访问控制、查看、列表、表单等的 handler。

核心中包含了两种实体

配置实体:用于配置系统。支持翻译并可以为安装提供默认值。

内容实体:由基本字段组成，提供内容修订和翻译支持。

17.2 实体类型

Drupal 7 实体是标准的类对象，Drupal 8 实体是类型化对象，每种实体类型(Entity type)定义一个类用于实例化该种实体。模块要提供实体类型，必须把实体类放在模块的实体子目录下，例如\Drupal\[module_name]\Entity。这表明实体类的 PHP 文件可以到模块的/src/Entity 目录下寻找。

www.drupalc.com

类的文档注释区必须包含实体类型注释，它定义了实体类型的基本信息。包括实体类型的标签、控制器、表等。实体类型可用的属性列表请参考\Drupal\Core\Entity\Annotation\EntityType 类的文档注释。

如果实体类型名称与模块名不相同，实体类型的命名应将模块名作为其前缀。添加模块名作为前缀也不是必须的，因为它是在模块的命名空间中定义的，一般来说取一个有意义的名称即可。例如，分类术语实体类型的名称为 taxonomy_term，其类名为 Drupal\taxonomy\Entity\Term。

Drupal 8 建议你在函数或方法中使用接口而不是类作为类型提示。例如，hook_entity_insert(EntityInterface $entity)中类型提示是 EntityInterface，又如 hook_node_insert(NodeInterface $node)中类型提示是 Nodeinterface。

实体的字段/属性名称是很短的，难以从它们的名称上看出它的作用。另外内容实体根本不会为它们的字段使用定义的属性。因此，建议提供一个文档化的接口作为类型提示，这样更易理解。这需要遵循一些规则:

- get/set/is 类方法带有相同的前缀:如 getSomething()，setSomething($value)，isSomething()。
- 仅仅为支持其它代码修改的属性添加方法。节点的最后修改日期不支持修改，因此只有$node->getChangedTime()方法，但是没有$node->setChangedTime()方法。
- 使用具有描述性的的方法名，例如访问节点状态的方法叫$node->isPublished()。

可发现性

为了寻找定义实体类型的模块，需要在模块的子命名空间中的类中寻找@EntityType 注释，它定义了实体类型的基本信息，它的 id 键即是实体类型的名称。

当尝试寻找一个实体类型定义时，首先是看实体类型的前缀。如果实体类型没有采用模块名作为前缀的命名规则，可以通过搜索 id=$type 来寻找。如果已知它是类或接口而不是实体类型，则命名空间可以提供它定义的地方。

core/modules/node/src/Entity/Node.php 是一个例子，有兴趣不妨看看。

17.3 实体常用操作

本节主要介绍常用的实体操作，包括实体检测、获取实体信息、创建实体、加载实体、保存实体、删除实体、访问控制等。

www.drupalc.com

实体检测

//检测对象是否是一个实体的实例

if ($object instanceof \Drupal\Core\Entity\EntityInterface) {

}

//内容实体实例检测

if ($entity instanceof \Drupal\Core\Entity\ContentEntityInterface){

}

//获取实体类型

$entity->getEntityTypeId();

//节点实例检测

if($entity instanceof \Drupal\node\NodeInterface){

}

//使用 entityType 获取动态实体类型

$needed_type = 'node';

if ($entity->getEntityTypeId() == $needed_type){

}

www.drupalc.com

获取实体或实体方法的信息

使用一些方法可以获取实体的信息，如实体的 ID，实体 bundle，修订 ID 等等。请看 EntityInterface 接口获取更多的细节。

//获取实体 id

$entity->id();

//获取实体 bundle

$entity->bundle();

//检测实体是否为新实体

$entity->isNew();

//获取实体的标签

$entiry->label();

//获取实体 URI

$entity->uri();

//创建实体的副本

$duplicate = $entity->createDuplicate();

创建实体

$node = entity_create(‘node’,array(

 ‘title’ => ‘My node’,

 ‘body’ => ‘The body content. This just works like this due to the new Entity Field API. It will be assigned as the value of the first field item in the default language.’

));

//你也可以使用静态的创建方法

$node = Node::create(array(‘title’ => ‘The node title’));

//使用实体管理器(entity manager)

$node = \Drupal::entityTypeManager()->getStorage(‘node’)->create(array(‘type’=>’article’,’title’=>’Another node’));

加载实体

//使用静态方法

$node = Node::load(1);

//动态实体类型，entity_load()加载新单个实体，在 7.x 中 entity_load()已经重命名为 entity_load_multiple()。

$entity = entity_load($entity_type,$id);

www.drupalc.com

```
//使用存储控制器

$entity = \Drupal::entityTypeManager()->getStorage($entity_type)->load(1);

//加载多个实体使用 entity_load_multiple()

$entity =
\Drupal::entityTypeManager()->getStorage($entity_type)->loadMultiple(array
(1,2,3));
```

为了更新实体，可以先加载它，然后修改，最后保存。

保存实体

```
//保存一个实体

$entity->save();
```

这一方法既可用于新实体，也可用于已存在的实体，可以从实体信息中得知实体是否是一个新实体。一般地，对于内容实体，会提供一个实体 ID。为了保存一个新实体(例如导入数据)，可以强制使用 isNew 标志，请看以下代码。

```
//下面的代码试图插入一个节点 ID 为 5 的节点，如果这个节点 ID 已存在则会失败

$node->nid->value = 5;

$node->enforceIsNew(TRUE);

$node->save();
```

删除实体

```
//删除单个实体

$entity = \Drupal::entityTypeManager()->getStorage('node')->load(1);

$entity->delete();

//一次删除多个实体

\Drupal::entityTypeManager()->getStorage($entity_type)->delete(array($id1
=> $entity1,$id2 => $entity2));
```

访问控制

www.drupalch.com

access()方法可以用于检测谁有权访问实体。这个方法支持很多操作，标准的操作有 view、update、delete 和 create，create 有点特殊，请看下面。

访问检测会被转发给访问控制器。

//检测实体的查看权限，默认会对当前登录用户进行这一检测

if ($entity->access('view')){

}

//检测一个用户是否可以删除一个实体

if ($entity->access('delete',$account)){

}

当检查实体创建权限时，通常还没有实体。这个权限只是为了检测某人是否有权创建该实体，开销较大，因此应直接使用访问控制器进行检查。

\Drupal::entity::entityTypeManager()->getAccessController('node')->createAccess('article');

如果已经有一个实体，$entity->access('create')也可以工作，它会被转到 createAccess()方法，同样地其它操作会被转到访问控制器的 access()方法上。

注意:有些资料上使用\Drupal::entityManager()，但已不提倡，并会在 9.x 中删除。因此请用\Drupa::entityTypeManager()代替\Drupal::entityManager()。

www.drupalc.com

17.4 Bundles

在 Drupal 8 中，bundles 是一个信息容器，它包含字段或者设置定义。它有时叫作子类型(sub-types)。Bundles 是可选的，它存在于某种实体的结构化层次中。

Entity Variants(内容和配置实体以及其它实体)

实体类型(Entity types)

Bundles;有时叫子类型，是可选的

下面是一些例子

Entity Variants:

1.Content(内容)

Content Entity Types(内容实体)

1.Node(节点)

Node Bundles，Content Types

1.Article

2.Basic page

2.Taxonomy(分类)

Taxonomy Bundles，Vocabulary

1.<Vocabulary A>

2.<Vocabulary B>

3.Blocks(区块)

Custom Block Bundles，Custom Block Types

1.<Block Type L>

2.<Block Type M>

4.User(没有子 bundles)

5.<Custom content enity type X>

2.Configuration(配置)

Configuration Entity Types:

1.Custom Block types(没有子 bundles)

www.drupalc.com

17.5 配置实体

配置实体使用新的 Drupal 8 实体 API 处理系统(模块、主题等)配置数据，并将其存入数据库中，使用 Drupal 8 的配置实体 API，你可以创建新的配置实体，并实现实体的 CURD 操作。

它是不同于内容实体的，它集成了 CMI API 用于导出实体，配置实体不可字段化，配置实体的所有属性使用 Schema 文件定义，内容实体使用 hook_schema()。

大多数配置实体是与 Drupal 的核心 config_object 类型进行交互，在 13 章中讨论了 Drupal 8 的配置系统已对配置实体作了介绍。它还可以用于其它地方，如菜单、视图显示、表单显示、联系表单、tours 等。

17.6 内容实体

内容实体用于管理网站内容，它有很多类和接口以及方法，具体可以看一下源代码文件。内容实体从基本实体继承了很多方法。请阅读上一章使用实体 API 查看常用的功能。

内容实体提供了基本的字段定义，可以通过字段 UI 模块对实体的字段进行配置。还可以对字段进行设置以及控制它的显示方式。

处理字段值

使用内容类型中的 baseFieldDefinitions 方法创建自定义字段。其过程为

1.读取实体实例到局部变量$Custom_Entity.

2.定义名为"custom_field"的自定义字段

3.保存数据.

例子:

```
$custom_field_value = $Custom_Entity->custom_field->value;

//完成多种类型的数据处理

$Custom_Entity->custom_field->value = $custom_field_value;

$Custom_Entity->save();
```

内容实体的其它例子请看节点、评论、用户等模块代码。

17.7 创建自定义内容实体

www.drupalc.com

在 Drupal 8 中创建一个没有包（bundle）的内容实体，这意味着这个实体没有实现字段 API，它完全在代码中定义。但我觉得还是应该好好计划一下，以便于以后我们引入更加复杂的数据。

假定我们的模块名为 advertiser，内容实体类型名为 advertiser。其字段表中的基本字段为 UUID 和 ID。

实体的目录结构

我们的模块放在 modules/custom/advertiser 下面，则实体所在的目录结构应像下面这样:

```
web/modules/custom/advertiser$
├─── advertiser.info.yml
└─── src
    └─── Entity
        └─── Advertiser.php
```

我们需要在 advertiser 模块中定义模块信息文件。像下面这样:

```
name: Advertiser
type: module
description: ‘Barebones advertiser entity’
package: custom
core: 8.x
```

Advertiser 类和相关的 schema 定义在 src/Entity/Advertiser.php 中。第一件事是定义 Advertiser 实体类的名字空间。

```
namespace Drupal\advertiser\Entity;
```

现在可以定义实体了，只需在注释中定义就行。注意，这是实体类型真正定义的地方，它可以被 Drupal 识别并被缓存，因此对它作出修改后需要清空缓存。

```
<?php

/**
 * Defines the Advertiser entity.
 *
 * @ingroup advertiser
 *
 * @ContentEntityType(
 *   id = "advertiser",
```

```
 *   label = @Translation("Advertiser"),
 *   base_table = "advertiser",
 *   entity_keys = {
 *     "id" = "id",
 *     "uuid" = "uuid",
 *   },
 * )
 */
?>
```

因为这只是一个例子，所以只是定义了几个属性并且没有像 Access 模块那样带有处理器(handler)。我们虽然定义了模块和内容实体，但是与实体相关的'advertiser'表在数据库还没有创建。那是因为我们的实体类没有任何方法来定义数据库中的表。我们需要对实体所需的数据表进行描述。

为此我们需要做两件事，其一是扩展 ContentEntityBase 类，另外需实现 ContentEntityInterface 接口，代码如下:

```
class Advertiser entends ContentEntityBase implements
ContentEntityInterface{
```

但我们仍需要实现一些有用的方法来存取数据库。

ContentEntityBase 类的 baseFieldDefinitions 方法，它带有一个参数:实体类型定义($entity_type)，它返回实体类型的字段定义，键为字段名。

因此我们可以像下面这样来实现它。

```
<?php

public static function baseFieldDefinitions(EntityTypeInterface $entity_type) {

    // Standard field, used as unique if primary index.
    $fields['id'] = BaseFieldDefinition::create('integer')
      ->setLabel(t('ID'))
      ->setDescription(t('The ID of the Advertiser entity.'))
      ->setReadOnly(TRUE);

    // Standard field, unique outside of the scope of the current project.
    $fields['uuid'] = BaseFieldDefinition::create('uuid')
      ->setLabel(t('UUID'))
      ->setDescription(t('The UUID of the Advertiser entity.'))
      ->setReadOnly(TRUE);

    return $fields;
```

www.drupalc.com

```
  }

?>
```

值得注意的是:

BaseFieldDefinitions:为实体类型提供基本的字段定义。它是 FieldableEntityInterface 接口的公共静态方法。

BaseFieldDefinition:定义实体字段的类，定义所需的创建字段，添加约束等...

下面是所有代码:

```
<?php

namespace Drupal\advertiser\Entity;

use Drupal\Core\Entity\ContentEntityBase;
use Drupal\Core\Field\BaseFieldDefinition;
use Drupal\Core\Entity\EntityTypeInterface;
use Drupal\Core\Entity\ContentEntityInterface;

/**
 * Defines the advertiser entity.
 *
 * @ingroup advertiser
 *
 * @ContentEntityType(
 *    id = "advertiser",
 *    label = @Translation("advertiser"),
 *    base_table = "advertiser",
 *    entity_keys = {
 *       "id" = "id",
 *       "uuid" = "uuid",
 *    },
 * )
 */

class Advertiser extends ContentEntityBase implements ContentEntityInterface
{

  public static function baseFieldDefinitions(EntityTypeInterface $entity_type)
{

    // Standard field, used as unique if primary index.
```

www.drupalc.com

```
    $fields['id'] = BaseFieldDefinition::create('integer')
      ->setLabel(t('ID'))
      ->setDescription(t('The ID of the Advertiser entity.'))
      ->setReadOnly(TRUE);

    // Standard field, unique outside of the scope of the current project.
    $fields['uuid'] = BaseFieldDefinition::create('uuid')
      ->setLabel(t('UUID'))
      ->setDescription(t('The UUID of the Advertiser entity.'))
      ->setReadOnly(TRUE);

    return $fields;
  }
}
?>
```

现在'advertiser'表已经添加到数据库中了。

```
$ drush sqlc
mysql> SHOW TABLES;
```

www.drupalc.com

17.8 创建自定义内容类型

要创建自定义内容类型，我们需要先创建一个自定义模块，假定我们创建的自定义模块的名称为 foobar。

正如我们在引例中所述，创建一个自定义内容类型可以通过创建几个 YAML 文件来包含所需的配置信息。在本例中，我们将创建一个名为 Car Brand 的内容类型，它包含两个默认的字段:body 和 title。下面我们来看看这几个 YAML 文件。

foobar/config/install/node.type.car_brand.yml，这个文件通知 Drupal，它创建的是一个内容类型。我们建议向 foobar 模块添加一个强制依赖，当我们卸载这个模块时，Drupal 会删除这个内容类型。当站点创建者决定不再需要这个模块时，我们不想任何人再使用这种内容类型。其代码如下:

```
# node.type.car_brand.yml
langcode: en
status: true
dependencies:
  enforced:
    module:
      - foobar # This is the name of the module we're using for this example
```

```
name: 'Car Brand'
type: car_brand
description: 'Content type that can be used to provide additional information on <em>Car Brands</em>'
help: ''
new_revision: false
preview_mode: 1
display_submitted: true
```

foorbar/config/install/field.field.node.car_brand.body.yml，这个文件向 Car Brand 内容类型添加 body 字段，该字段依赖 node.body。其代码如下:

```
# field.field.node.car_brand.body.yml
langcode: en
status: true
dependencies:
	config:
		- field.storage.node.body
		- node.type.car_brand
	module:
		- text
id: node.car_brand.body
field_name: body
entity_type: node
bundle: car_brand
label: Body
description: 'More specific information about the car brand.'
required: false
translatable: true
default_value: {  }
default_value_callback: ''
settings:
	display_summary: true
field_type: text_with_summary
```

foobar/config/install/core.entity_view_display.node.car_brand.teaser.yml，这个文件告诉 Drupal 这种内容类型的摘要要怎样显示。其代码如下:

```
# core.entity_view_display.node.car_brand.teaser.yml
langcode: en
status: true
```

www.drupalc.com

```
dependencies:
    config:
        - core.entity_view_mode.node.teaser
        - field.field.node.car_brand.body
        - node.type.car_brand
    module:
        - text
        - user
id: node.car_brand.teaser
targetEntityType: node
bundle: car_brand
mode: teaser
content:
    body:
        label: hidden
        type: text_summary_or_trimmed
        weight: 101
        settings:
            trim_length: 600
        third_party_settings: {  }
    links:
        weight: 100
hidden: {  }
```

foobar/config/install/core.entity_view_display.node.car_brand.default.yml，这个文件告诉 Drupal 该内容类型的默认显示方式。其代码如下:

```
# core.entity_view_display.node.car_brand.default.yml
langcode: en
status: true
dependencies:
    config:
        - field.field.node.car_brand.body
        - node.type.car_brand
    module:
        - text
        - user
id: node.car_brand.default
targetEntityType: node
bundle: car_brand
mode: default
content:
    body:
```

www.drupalc.com

```
        label: hidden
        type: text_default
        weight: 101
        settings: {   }
        third_party_settings: {   }
    links:
        weight: 100
hidden: {   }
```

foobar/config/install/core.entity_form_display.node.car_brand.default.yml，这个文件告诉 Drupal 当创建这种类型的新节点时表单的显示。其代码如下:

```
# core.entity_form_display.node.car_brand.default.yml
langcode: en
status: true
dependencies:
    config:
        - field.field.node.car_brand.body
        - node.type.car_brand
    module:
        - text
        - user
id: node.car_brand.default
targetEntityType: node
bundle: car_brand
mode: default
content:
    body:
        label: hidden
        type: text_textarea_with_summary
        weight: 101
        settings: {   }
        third_party_settings: {   }
    links:
        weight: 100
hidden: {   }
```

启用 **Car Brand** 内容类型

现在我们已经创建好了 Car Brand 所需的配置文件，我们需要通知 Drupal 启用这种内容类型。首先应安装 foorbar 模块，然后启用这个模块。如果你的模块早

www.drupalc.com

已启用，你应该卸载该模块，并重新安装和启用它。现在到创建内容类型页面，你将会发现你可以创建'Car Brand'这种类型的内容。

17.9 在内容类型中附加自定义字段

有时你想从自定义模块中提取一个内容类型，并想把内容类型中的某些字段提取出来添加到新的内容类型中。有两种方式向你的代码添加这些字段。首先，假定你已有一个自定义模块 foorbar，另外在该模块中已经存在内容类型 Car Brand。

向内容类型附加字段

向内容类型附加字段有两种方式。你可以通过字段 UI 创建字段并将其导出到文件，你也可以自己写这个文件。我喜欢使用前者，因为导出的文件是不会出现任何错误的，但是导出的字段是以'field_'开头而不是'foobar_car_brand_'与我们这个内容类型名称相关的开头。本节我们会通过手工编码的方式配置字段。

手工编码字段

如前所述，这种方法需要更多的知识，并且需要知道字段是做什么的。你添加的每个字段都需要使用两个 YAML 文件来定义字段配置信息。它们的命名如下:

field.field.node.car_brand.field_brand_information.yml
field.storage.node.field_brand_information.yml

如果你已经启用了 foorbar 模块，请卸载它。

foobar/config/install/field.storage.node.field_brand_information.yml，这个文件通知 Drupal 它创建了一个字段。其代码如下:

```
# field.storage.node.field_brand_information.yml
langcode: en
status: true
dependencies:
  module:
    - node
    - text
id: node.field_brand_information
field_name: field_brand_information
entity_type: node
```

```
type: text_with_summary
settings: {   }
module: text
locked: false
cardinality: 1
translatable: true
indexes: {   }
persist_with_no_fields: false
custom_storage: false
```

foobar/config/install/field.field.node.car_brand.field_brand_information.yml，这个文件通知 Drupal 它向内容类型附加字段。其代码如下:

```
# field.field.node.car_brand.field_brand_information.yml
langcode: en
status: true
dependencies:
  config:
    - field.storage.node.field_brand_information
    - node.type.car_brand
  module:
    - text
id: node.car_brand.field_brand_information
field_name: field_brand_information
entity_type: node
bundle: car_brand
label: 'Brand Information'
description: 'More specific information about the car brand'
required: false
translatable: false
default_value: {   }
default_value_callback: ''
settings:
  display_summary: false
field_type: text_with_summary
```

foorbar/config/install/core.entity_form_display.node.car_brand.default.yml 或者 foobar/config/install/core.entity_view_display.node.car_brand.default.yml，这两个文件因为添加了两个字段，因此需要更新它们，这里添加的两个字段名为′additional_field_1′和′additional_field_2′。如果你不想创建这两个附加字段，请在下面的代码中删除依赖信息。它们的代码如下:

www.drupalc.com

```
# core.entity_view_display.node.car_brand.default.yml
langcode: en
status: true
dependencies:
  config:
    - field.field.node.car_brand.field_brand_information
    - field.field.node.car_brand.field_additional_field_1
    - field.field.node.car_brand.field_additional_field_2
    - node.type.car_brand
  module:
    - file
    - text
    - user
_core:
  default_config_hash:
Nfnv6VMugBKl6EOqi_U0I_LQ1ZQpbNDN3a9GXHWUBz4
id: node.car_brand.default
targetEntityType: node
bundle: car_brand
mode: default
content:
  field_brand_information:
    weight: 101
    label: above
    settings: {  }
    third_party_settings: {  }
    type: text_default
  field_additional_field_1:
    weight: 103
    label: above
    settings:
      link_to_entity: false
    third_party_settings: {  }
    type: string
  field_additional_field_2:
    weight: 102
    label: above
    settings: {  }
    third_party_settings: {  }
    type: file_default
  links:
    weight: 100
hidden: {  }
# core.entity_form_display.node.car_brand.default.yml
```

www.drupalc.com

```
langcode: en
status: true
dependencies:
  config:
    - field.field.node.car_brand.field_brand_information
    - field.field.node.car_brand.field_additional_field_1
    - field.field.node.car_brand.field_additional_field_2
    - node.type.car_brand
  module:
    - file
    - path
    - text
_core:
  default_config_hash:
qZE-qJ04DTTNggVVQdVOPQmpE_I69GQ_LqB32kXivVg
id: node.car_brand.default
targetEntityType: node
bundle: car_brand
mode: default
content:
  created:
    type: datetime_timestamp
    weight: 2
    settings: {   }
    third_party_settings: {   }
  field_brand_information:
    weight: 7
    settings:
      rows: 9
      summary_rows: 3
      placeholder: ''
    third_party_settings: {   }
    type: text_textarea_with_summary
  field_additional_field_1:
    weight: 6
    settings:
      size: 60
      placeholder: ''
    third_party_settings: {   }
    type: string_textfield
  field_additional_field_2:
    weight: 8
    settings:
      progress_indicator: throbber
```


www.drupalc.com

```
    third_party_settings: {  }
    type: file_generic
  path:
    type: path
    weight: 5
    settings: {  }
    third_party_settings: {  }
  promote:
    type: boolean_checkbox
    settings:
      display_label: true
    weight: 3
    third_party_settings: {  }
  sticky:
    type: boolean_checkbox
    settings:
      display_label: true
    weight: 4
    third_party_settings: {  }
  title:
    type: string_textfield
    weight: 0
    settings:
      size: 60
      placeholder: ''
    third_party_settings: {  }
  uid:
    type: entity_reference_autocomplete
    weight: 1
    settings:
      match_operator: CONTAINS
      size: 60
      placeholder: ''
    third_party_settings: {  }
hidden: {  }
```

www.drupalc.com

启用容类型

现在你需要启用 foobar 模块，现在你到创建内容(Create content)页面，你会发现你可以创建"Car Brand"类型的节点，并且它现在包含新的字段"Brand Information"。

17.10 用UI导出自定义字段的代码

正如上一节所提到的可以使用字段 UI 来向内容类型添加字段并导出其配置信息，本节我们将讲述这些内容。接上一节的例子，假定模块为 foobar，内容类型为 Car Brand。

使用 UI 创建字段

如前所述，这种方法比手工编写文件更好，因为这种方式既方便又准确并且可以导出其配置信息以便重复使用。

启用 Car Brand

如果你还没有启用 foobar 模块，请先启用它。启用它以后到创建内容页面，你应该能看到可以创建"Car Brand"类型的内容。

地址栏输入/admin/structure/types/manage/car_brand/fields，会转到编辑 Car Brand 类型的字段界面。你可以在这一页向该类型添加字段。现在我们添加一个名这"Brand Information"的字段，其机器名为 field_brand_information，字段类型为 Text(formatted,long,with summary)，并删除主体(Body)字段。其它保持默认设置。

www.drupalc.com

使用 UI 导出字段配置

你现在已经向 Car Brand 添加了 Brand Information 字段，你可以到 Configuration Synchronization 页面导出字段配置信息。这一页的上面有三个 tab，请选择"Export"或输入/admin/config/development/configuration/single/export 转到导出配置页面，接下来选择"Single item"。

获取导出的配置信息

在单个导出界面你需要选择配置类型，包括 Field Storage,Field,Entity View Display,Entity Form Display 等，在下面有一个 textarea 用来显示配置信息，并在它下面显示了这个配置的文件名信息，你只需在相应目录下创建同名文件并将 textarea 中的内容复制出来并粘贴即可创建好。

最后需要卸载 foobar 模块并重新安装和启用它，然后你会发现可以创建 Car Brand 类型的内容了，并且它带有 Brand Information 字段。

17.11 实体API实现类型化数据API

现在实体 API 已经实现了类型化数据 API，在新的实体 API 实现中，一切都是基于相同 API 的字段，因此实体是可以预见的和一致的。

理解 Drupal 的数据模型

在深入类型化数据 API 以前，有必要理解怎样使用 Drupal 的数据模型(实体 API)。这很重要，因为这是数据类型化 API 的源头，实体 API 只不过是为它设计的一个系统而已。

实体是一个复杂的数据对象，它复合了其它数据如字段。字段也是复杂的，它复合了更多的数据如文本字符串值以及输入格式等。然而复杂性一直到数据能被描述为像字符串或者整数等基本类型才会停止。

下面是 Drupal 7 中的一个简化的例子:

```
//实体是复杂的，它包含其它数据
$entity;
//属性是不复杂的，它仅仅包含一个字段列表
$entity->image;
//字段项是复杂的，它包含了其它数据，它是可翻译和可访问的(有权限控制)
$entity->image[0];
//文件 ID 是原始数据类型整数
$entity->image[0]['fid'];
//图像的替换文本是原始数据字符串
$entity->image[0]['alt'];
```

把它们放在一起

实体 API 从类型化数据 API 中实现了多个接口，实体 API 扩展了这些接口并添加了所需的方法。下面的表达式的值为真。

```
//实体是复杂的
$entity instanceof ComplexDataInterface;
//属性是不复杂的，它们仅仅是一些列表项
```

```
$entity->get('image') instanceof ListInterface;
//项是复杂的
$entity->get('image')->offsetGet(0) instanceof ComplexDataInterface;
//类型化数据对象呈现图像替换文本
$entity->get('image')->offsetGet(0)->get('alt') instanceof
TypedDataInterface;
//图像替换文本是原始数据类型字符串
is_string($entity->get('image')->offsetGet(0)->get('alt')->getValue());
```

下面的代码显示了实体 API 是如何扩展类型化数据 API 的:

```
interface EntityInterface extends ComplexDataInterface, TranslatableInterface,
AccessibleInterface {
 // ...
}

interface FielditemListInterface extends ListInterface {
 // ...
}

// Note that this extends two interfaces. Explanation below.
interface FieldItemInterface extends ComplexDataInterface,
TypedDataInterface {
 // ...
}

// Below follows some actual implementations.
// Extends an abstract class with some common logic.
class ImageItem extends FieldItemBase {
 // ...
}

// Extends an abstract class with some common logic.
class String extends TypedData {
 // ...
}
```

上面的代码需要注意两点:

1.EntityInterface 扩展了一些实用的结口，如翻译(TranslatableInterface)和访问能力(AccessibleInterface)。上面的代码已作了很好的解释。

www.drupalc.com

2.FieldItemInterface 扩展了 ComplexDataInterface 和 TypedDataInterface 接口。正如前面如述，字段项是复杂的，它包含了更多更复杂的数据，比如文本字符串值以及文本格式等。同时一个字段项有可能是'类型化数据'本身。因此需定义它自身的数据类型。

总结起来，下面的表达式都是真:

```
$entity instanceof EntityInterface;
$entity->get('image') instanceof FieldItemListInterface;
$entity->get('image')->offsetGet(0) instanceof FieldItemInterface;
$entity->get('image')->offsetGet(0)->get('alt') instanceof String;
is_string($entity->get('image')->offsetGet(0)->get('alt')->getValue());
```

实体 API 定义了一些魔术方法如__get()，主要是为了方便存取字段值。使用实体 API 提供的魔术方法是很简单的就像直接存取对象的属性一样。例如获取图像替换文本的值的代码如下:

```
//冗长方式
$string = $entity->get('image')->offsetGet(0)->get('alt')->getValue();
//使用实体 API 提供的魔术方法
$string = $entity->image[0]->alt;
//使用实体 API 魔术方法获取列表中第一项的值
$string = $entity->image->alt;
```

上面的例子演示了如何调用魔术方法。下面的例子演示了实体 API 中的数据来自哪里:

```
//返回所有字段和它们定义的关联数组。例如'image'字段。
$property_definitions = $entity->getFieldDefinitions();

//返回所有属性和它们的定义的关联数组。例如'file_id'和'alt'属性。
$property_definitions = $entity->image
  ->getFieldDefinition()
  ->getFieldStorageDefinition()
  ->getPropertyDefinitions();

//仅返回'alt'属性的定义
$string_definition = $entity->image
  ->getFieldDefinition()
  ->getFieldStorageDefinition
  ->getPropertyDefinition('alt');
```

基于上面获取的数据，我们现在可以完成数据序列化或者其它数据管理。我们可以丰富实体 API 以实现数据传输接口，就像 JSON-LD 以便于其它系统能理解我们的数据。

www.drupalc.com

17.12 定义和使用内容实体字段

内容实体必须在实体类中明确地定义它们的所有字段。字段的定义是基于类型化数据 API(请参见实体的实现方式)。

字段定义

实体类型在实体类的静态方法中定义它们的基本字段。基本字段是实体类型中不可配置的字段，比如节点的标题(node title)、节点创建日期和节点修改日期等。实体管理器(entity_manager)可以通过唤醒 hook_entity_field_info()和 hook_entity_field_info_alter()来修改其它模块提供的可配置的字段和不可配置字段。字段的配置可以通过字段 UI 添加或配置。

字段定义是实现了 FieldDefinitionInterface 接口的简单对象，而基本字段通常由 FieldDefinition 类创建，可配置字段直接实现这一接口以及它们各自的配置实体。

字段定义也可以定义字段项的约束或字段项的属性。能够使用所有的字段类型插件。字段是一个字段项列表，这意味着 FieldItemList 类封装了 FieldItem 类，目的是呈现字段项列表。

所有字段使用字段控件和格式化器来显示和编辑它们。

基本字段

以下是一个简化了的节点实体类型的字段定义列表:

```
use Drupal\Core\Field\BaseFieldDefinition;

class Node implements NodeInterface {

  /**
   * {@inheritdoc}
   */
  public static function baseFieldDefinitions($entity_type) {
    // The node id is an integer, using the IntegerItem field item class.
    $fields['nid'] = BaseFieldDefinition::create('integer')
      ->setLabel(t('Node ID'))
      ->setDescription(t('The node ID.'))
      ->setReadOnly(TRUE);
```

```
// The UUID field uses the uuid_field type which ensures that a new UUID
will automatically be generated when an entity is created.
    $fields['uuid'] = BaseFieldDefinition::create('uuid')
      ->setLabel(t('UUID'))
      ->setDescription(t('The node UUID.'))
      ->setReadOnly(TRUE);

    // The language code is defined as a language_field, which, again, ensures
that a valid default language
    // code is set for new entities.
    $fields['langcode'] = BaseFieldDefinition::create('language')
      ->setLabel(t('Language code'))
      ->setDescription(t('The node language code.'));

    // The title is StringItem, the default value is an empty string and defines a
property constraint for the
    // value to be at most 255 characters long.
    $fields['title'] = BaseFieldDefinition::create('string')
      ->setLabel(t('Title'))
      ->setDescription(t('The title of this node, always treated as non-markup
plain text.'))
      ->setRequired(TRUE)
      ->setTranslatable(TRUE)
      ->setSettings(array(
        'default_value' => '',
        'max_length' => 255,
      ));

    // The uid is an entity reference to the user entity type, which allows to
access the user id with $node->uid->target_id
    // and the user entity with $node->uid->entity. NodeInterface also defines
getAuthor() and getAuthorId(). (@todo: check owner vs. revisionAuthor)
    $fields['uid'] = BaseFieldDefinition::create('entity_reference')
      ->setLabel(t('User ID'))
      ->setDescription(t('The user ID of the node author.'))
      ->setSettings(array(
        'target_type' => 'user',
        'default_value' => 0,
      ));

    // The changed field type automatically updates the timestamp every time
the
    // entity is saved.
```

www.drupalc.com

```
    $fields['changed'] = BaseFieldDefinition::create('changed')
      ->setLabel(t('Changed'))
      ->setDescription(t('The time that the node was last edited.'))
    return $fields;
  }
}
```

实体还可以通过包(bundle)为特定的包(bundle)提供字段或者修改它们。例如，每个包都有不同的节点标题。为了使包可以修改字段，必须在修改基本字段前克隆它们，因为修改基本字段的定义将会影响到所有的包。

```
use Drupal\node\Entity\NodeType;

  /**
   * {@inheritdoc}
   */
  public static function bundleFieldDefinitions(EntityTypeInterface
$entity_type, $bundle, array $base_field_definitions) {
    $node_type = NodeType::load($bundle);
    $fields = array();
    if (isset($node_type->title_label)) {
      $fields['title'] = clone $base_field_definitions['title'];
      $fields['title']->setLabel($node_type->title_label);
    }
    return $fields;
  }
```

www.drupalc.com

字段类型

Drupal 的核心提供了一些基本的字段类型，另外，模块能提供其它的字段类型以供其它模块使用。下面列出了 Drupal 提供的字段类型:

- string:字符串型
- boolean:布尔型
- integer:整型，可设置取值范围作为字段验证
- decimal:十进制数
- float:浮点数
- language:语言代码
- timestamp:unix 时间，以整数存储
- created:创建时间
- changed:当实体保存时使用当前时间更新

- date:日期存储一个 ISO 8601 字符串
- uri:包含一个 URI。
- uuid:UUID 字段类型
- email:emal 字段类型
- entity_reference:实体引用类型
- map:映射类型，可以包含任意数量的任意属性，存储为序列化的字符串。

可配置字段

另外可以使用 hook_entity_base_field_info()和 hook_entity_bundle_field_info()注册字段。下面是代码示例:

```
use Drupal\Core\Field\BaseFieldDefinition;

/**
 * Implements hook_entity_base_field_info().
 */
function path_entity_base_field_info(EntityTypeInterface $entity_type) {
  if ($entity_type->id() === 'taxonomy_term' || $entity_type->id() ===
'node') {
     $fields['path'] = BaseFieldDefinition::create('path')
       ->setLabel(t('The path alias'))
       ->setComputed(TRUE);

     return $fields;
  }
}

/**
 * Implements hook_entity_bundle_field_info().
 */
function field_entity_bundle_field_info(EntityTypeInterface $entity_type,
$bundle, array $base_field_definitions) {
  if ($entity_type->isFieldable()) {
     // Configurable fields, which are always attached to a specific bundle, are
     // added 'by bundle'.
     return Field::fieldInfo()->getBundleInstances($entity_type->id(),
$bundle);
  }
}
```

www.drupalc.com

字段存储

Entity Field API 实现了字段的数据库存储和更新。如果你的字段没有特殊的要求，使用默认的存段存储方式即可并且没有被标记为已计算(setComputed(TRUE))，如要自己写字段存储需特别指出(setCustomStorage(TRUE))。

假如你想给节点实体添加一个新字段，它包含一个用于高亮内容的布尔值。下面给出了示例

```
use Drupal\Core\Entity\EntityTypeInterface;
use Drupal\Core\Field\BaseFieldDefinition;

/**
 * Implements hook_entity_base_field_info().
 */
function MYMODULE_entity_base_field_info(EntityTypeInterface $entity_type)
{
  $fields = array();

  // Add a 'Highlight' base field to all node types.
  if ($entity_type->id() === 'node') {
    $fields['highlight'] = BaseFieldDefinition::create('boolean')
      ->setLabel(t('Highlight'))
      ->setDescription(t('Whether or not the node is highlighted.'))
      ->setRevisionable(TRUE)
      ->setTranslatable(TRUE)
      ->setDisplayOptions('form', array(
        'type' => 'boolean_checkbox',
        'settings' => array(
          'display_label' => TRUE,
        ),
      ))
      ->setDisplayConfigurable('form', TRUE);
  }

  return $fields;
}
```

现在需要运行 update.php 以更新数据库，它会在存储节点数据的表中添加一个'highlight'列，但是运行了 update.php 后，数据库节点表中并没有出现'highlight'列，还需要运行以下代码:

```
\Drupal::entityTypeManager()->clearCachedDefinitions();
\Drupal::service('entity.definition_update_manager')->applyUpdates();
```

以可以使用 Drush 来达到相同目的

```
drush updated –entity-updates
```

为了在开启或禁用你的模块时自运更新数据库，你可以创建事件监听器来触发，这需要在 hook_inistall()和 hook_uninstall()中实现事件响应。

```
/**
 * Implements hook_install().
 */
function MYMODULE_install() {
  // Create field storage for the 'Highlight' base field.
  $entity_manager = \Drupal::entityManager();
  $definition =
$entity_manager->getFieldStorageDefinitions('node')['highlight'];
  $entity_manager->onFieldStorageDefinitionCreate($definition);
}

/**
 * Implements hook_uninstall().
 */
function MYMODULE_uninstall() {
  $entity_manager = \Drupal::entityManager();
  $definition =
$entity_manager->getLastInstalledFieldStorageDefinitions('node')['highlight'];
  $entity_manager->onFieldStorageDefinitionDelete($definition);
}
```

www.drupalc.com

字段用法

因为实体是复杂数据，他们遵循 ComplexDataInterface 接口。从类型化数据角度来说，复杂数据对象中的所有的类型化的数据元素都是属性。这些限制/强制命名可能会被删除。

```
//检测实体是否拥有该字段
$entity->hasField('field_tags');

//返回所有字段及它们的定义
$field_definitions = $entity->getFieldDefinitions();
```

```
//返回字段项的属性和它们的定义。
$property_definitions =
$entity->image->getFieldDefinition()->getPropertyDefinitions();

//返回字段定义
Drupal::entityManager()->getFieldDefinitions($entity_type);

//从某个 bundle 返回字段集
Drupal::entity->Manager()->getFieldDefinitions($entity_type,$bundle);
```

字段控件和格式化器

因为字段需要进行配置，所以引入了字段控件和字段格式化器。字段控件和格式化器以及其它的设置应该在 FieldDefinition 类中指定。下面是例子代码:

```
use Drupal\Core\Field\BaseFieldDefinition;

// ...

    $fields['title'] = use Drupal\Core\Field\BaseFieldDefinition;
FieldDefinition::create('string')
      ->setLabel(t('Title'))
      ->setDescription(t('The title of this node, always treated as non-markup
plain text.'))
      ->setRequired(TRUE)
      ->setTranslatable(TRUE)
      ->setSettings(array(
        'default_value' => '',
        'max_length' => 255,
      ))
      ->setDisplayOptions('view', array(
        'label' => 'hidden',
        'type' => 'string',
        'weight' => -5,
      ))
      ->setDisplayOptions('form', array(
        'type' => 'string',
        'weight' => -5,
      ))
      ->setDisplayConfigurable('form', TRUE);
```

www.drupalc.com

上面的代码使用'字符串'格式化器和字段控件配置节点标题。
setDisplayConfigurable()的作用是让字段显示在字段 UI 界面中，它可以修改字段显示顺序和标签。在字段 UI 界面也可以修改字段控件和它们的设置。

17.13 实体翻译API

Drupal 8 的字段语言不再暴露给公共的 API，而是附加了语言感知实体对象，从而继承它们的语言。

这要做的优点有:

我们不必担心字段的可翻译性，因为这是实体对象内部完成的事情。

```
$translation = $entity->getTranslation($active_langcode);
$value = $translation->field_foo->value;
```

我们不需要传递活动语言，事实上我们只需传递翻译对象，它实现了 EntityInterface，它是原始对象的克隆，并带有不同的语言。这意味着在大多数情况下，由此产生的代码是具有语言感知能力的。

```
//初始化合适的翻译对象以传递到需要传递的地方。这是核心关心的事情。
//在多数情况下以及子系统不需要明确地返回翻译对象。
$language =
Drupal::languageManager()->getLanguage(Language::TYPE_CONTENT);
$translation = $entity->getTranslation($langcode);
entity_do_stuff($translation);

function entity_do_stuff(EntityInterface $entity){
$value = $entity->field_foo->value;
}
```

我们现在使用了一个可重用的实体语言判定 API，它用于决定适合特定内容的实体翻译。

```
    // Simplified code to generate a renderable array for an entity.
     function viewEntity(EntityInterface $entity, $view_mode = 'full',
$langcode = NULL) {
        // The EntityManagerInterface::getTranslationFromContext() method
will
        // apply entity language negotiation logic to the whole entity object
```

www.drupal.com

```
    // and will return the proper translation object for the given context.
    // The $langcode parameter is optional and indicates the language of
the
    // current context. If it is not specified the current content language
    // is used, which is the desired behavior during the rendering phase.
    // Note that field values are left alone in the process, so empty values
    // will just not be displayed.
    $langcode = NULL;
    $translation =
$this->entityManager->getTranslationFromContext($entity, $langcode);
    $build = entity_do_stuff($translation, 'full');
    return $build;
  }
```

我们也可以指定一个可选的$content 参数，它用来描述翻译对象将被应用到哪儿的内容。

```
  // Simplified token relpacements generation code.
  function node_tokens($type, $tokens, array $data = array(), array
$options = array()) {
    $replacements = array();

    // If no language is specified for this context we just default to the
    // default entity language.
    if (!isset($options['langcode'])) {
      $langcode = Language::LANGCODE_DEFAULT;
    }

    // We pass a $context parameter describing the operation being
performed.
    // The default operation is 'entity_view'.
    $context = array('operation' => 'node_tokens');
    $translation =
\Drupal::entityManager()->getTranslationFromContext($data['node'],
$langcode, $context);
    $items = $translation->get('body');

    // do stuff

    return $replacements;
  }
```

模块可以修改翻译对象判定逻辑，请看 LanguageManager::getFallbackCandidates()以了解更多细节。

www.drupalc.com

实际的字段数据在翻译对象与不可翻译的字段间是共享的，修改不可翻译字段的值会自动地修改翻译对象的值。下面是例子:

```
$entity->langcode->value = 'en';
  $translation = $entity->getTranslation('it');

  $en_value = $entity->field_foo->value; // $en_value is 'bar'
  $it_value = $translation->field_foo->value; // $it_value is 'bella'

  $entity->field_untranslatable->value = 'baz';
  $translation->field_untranslatable->value = 'zio';
  $value = $entity->field_untranslatable->value; // $value is 'zio'
```

在任何时候，可以从原始对象或其它翻译对象通过调用 EntityInterface::getTranslation()方法实例化一个翻译对象。如果明确地指定了活动语言，可以使用 EntityInterface::language()方法访问翻译对象。通过 EntityInterface::getUntranslated()方法返回原始对象。

```
$entity->langcode->value = 'en';

  $translation = $entity->getTranslation('it');
  $langcode = $translation->language()->id; // $langcode is 'it';

  $untranslated_entity = $translation->getUntranslated();
  $langcode = $untranslated_entity->language()->id; // $langcode is 'en';

  $identical = $entity === $untranslated_entity; // $identical is TRUE

  $entity_langcode = $translation->getUntranslated()->language()->id; //
$entity_langcode is 'en'
```

EntityInterface 定义了几个处理实体翻译的简易方法。如果一段代码需要活动在每个可用的翻译上，可以利用 EntityInterface::getTranslationLanguage():

```
foreach ($entity->getTranslationLanguages() as $langcode => $language) {
    $translation = $entity->getTranslation($langcode);
    entity_do_stuff($translation);
  }
```

下面的代码演示了如何添加、删除、检测翻译:

```
if (!$entity->hasTranslation('fr')) {
    $translation = $entity->addTranslation('fr', array('field_foo' => 'bag'));
```

```
  }

  // Which is equivalent to the following code, although if an invalid language
  // code is specified an exception is thrown.
  $translation = $entity->getTranslation('fr');
  $translation->field_foo->value = 'bag';

  // Accessing a field on a removed translation object causes an exception to
  // be thrown.
  $translation = $entity->getTranslation('it');
  $entity->removeTranslation('it');
  $value = $translation->field_foo->value; // throws
InvalidArgumentException
```

当实体翻译被添加或被删除，以下 hook 分别被引发:

hook_entity_translation_insert()

hook_entity_translation_delete()

通过在字段对象自身上调用适合的方法也可以返回字段语言:

```
$langcode = $translation->field_foo->getLangcode();
```

17.14 显示模式、查看模式和表单模式

显示模块为内容实体的查看和编辑操作提供不同的显示方式。有两种类型的显示模式即查看模式和表单模式。这两种类型的显示模式都是配置实体的例子。下面是一个导出的查看模式代码:

```
uuid: 15dc7aa9-13fd-4412-9c06-06d09f915d08
langcode: en
status: false
dependencies:
  module:
    - node
id: node.full
label: 'Full content'
targetEntityType: node
cache: true
```

www.drupalc.com

请注意主要的属性是'targetEntityType'。每一种显示模式都与唯一的一个实体关联。有些显示模式可以应用到多个内容实体，例如"Full"查看模式可以用于节点、自定义区块及评论等内容实体。

查看模式与查看显示

查看模式在/admin/structure/display-modes/view 进行管理。可以在下方为包(bundle)设置自定义显示设置，如/admin/structure/types/manage/page/display/teaser，'page'是一个节点实体，'teaser'是一个查看模式。

查看模式在 Drupal 8 中是一个概念。存在于目前的 Drupal 7 中。Drupal 6 有一个'build modes'概念。查看模式允许以指定的方式渲染实体引用字段。例如，假定'song'和'artist'都是节点类型，'song'引用了'artist'，'song'需要'full'显示，而'artist'可能会渲染成'teaser'显示。

如果站点创建者想将 artist 节点 bundle 以 teaser 查看模式显示，站点创建者可以到"管理显示"页面进行配置。在这一页，站点创建者可以自定义字段显示设置，不需要为每个包(bundles)进行自定义显示设置。这这个例子中只需对"RSS"和"Teaser"进行特殊设置，其它的查看模式缺省就好。每个实体类型包(bundle)和查看模式间的关联叫 ViewDisplay。

www.drupalc.com

表单模式和表单操作

管理的地址:/admin/structure/display-modes/form，可以为每个包(bundles)启用。正如查看模式一样，表单模式是为同种内容实体 bundle 创建不同的字段配置的方式。表单模式允许配置字段，如选择字段控件，对表单字段排序等。

另外，表单操作定义了使用哪个类来完成操作，如删除节点的类和编辑节点的类是不同的，在实体的注释中定义了这些操作。

下面是一个操作定义的例子，它在 MyEntityForm 表单上映射了两个自定义操作。它需要扩展 ContentEntityForm 类。

```
/**
 * @ContentEntityType(
 *    id = "myentity",
 *    handlers = {
 *       "form" = {
 *          "default" = "Drupal\myentity\Form\MyEntityForm",
 *          "add" = "Drupal\myentity\Form\MyEntityForm",
```

```
 *          "edit" = "Drupal\myentity\Form\MyEntityForm",
 *          "delete" = "Drupal\myentity\Form\MyEntityDeleteForm",
 *       },
 *    },
 * )
 */
```

如果你需要添加或修改实体表单操作，你可以使用 hook_entity_type_build 和 hook_entity_type_alter。

当前表单模式的操作必须明确设置，它不像查看模式可以设为'default'。这可能是一个 bug。

为了显示一个自定义模式的表单，需要在你的路由中使用_entity_form。例如，显示 MyEntity 实体的自定义"edit"表单，使用以下路由:

```
entity.myentity.edit_form:
 path: '/myentity/{myentity}/edit'
 defaults:
    _entity_form: myentity.edit
    _title: 'Edit MyEntity'
 requirements:
    _permission: 'edit myentity entities'
```

www.drupalc.com

17.15 字段类型、字段控件和字段格式

Drupal 8 提供了一个庞大的类库以帮助你处理内容。Drupal 8 的实体让你可以使用字段，字段是很重要的，因为它是用来存储实体数据的工具。Drupal 8 的核心已提供了多种字段类型，它们用来处理各种类型的数据。

Drupal 8 核心提供的基本类型

- boolean 布尔型
- changed 修改日期
- created 创建日期
- decimal 数字
- email 电子邮件
- entity_reference 实体引用
- float 浮点数
- integer 整型
- language 语言
- map 映射

- password 密码
- string_long 长文本
- timestamp 时间戳
- uri 统一资源定位符
- uuid 淮一 id
- comment 评论
- datetime 时间日期
- file 文件
- image 图像
- link 链接
- list_float 浮点数列表
- list_integer 整数列表
- list_string 字符串列表
- path 路径
- telephone 电话
- text 文本
- text_long 长文本
- text_with_summary 带摘要的文本

自定义字段类型

当 Drupal 提供的数据类型无法提供你的需求时，你可能想创建一个新的字段类型以表示你的数据。假如你想使用一个内容实体存储一些敏感数据。内容创建者允许特定用户通过密码访问实体。假如我们在数据库中创建数据如下:

entity_id	uid	password
1	1	'helloworld'
1	1	'goodbye'

正如上表所示，实体 id 为 1 的实体为两个用户有不同的密码。在不手工创建表的情况下怎样在 Drupal 中实现呢？答案是需要创建一个新的字段类型。

因为 Drupal 使用 Plugin 来实现字段，我们可以继承一个基类来使它工作。创建新的字段类型需要在模块中创建以下目录结构:

modules/custom/MODULENAME/src/Plugin/Field/FieldType

虽然这个路径有点长，但它能被 Drupal 更好地处理。

在这个例子中，我们创建的文件 EntityUserAccessField.php 如下:

```
/**
 * @file
```

www.drupalc.com

```
 * Contains
\Drupal\MODULENAME\Plugin\Field\FieldType\EntityUserAccessField
 */

namespace Drupal\MODULENAME\Plugin\Field\FieldType;

use Drupal\Core\Field\FieldItemBase;
use Drupal\Core\Field\FieldStorageDefinitionInterface;
use Drupal\Core\TypedData\DataDefinition;

/**
 * @FieldType(
 *   id = "entity_user_access",
 *   label = @Translation("Entity User Access"),
 *   description = @Translation("This field stores a reference to a user
and a password for this user on the entity."),
 * )
 */

class EntityUserAccessField extends FieldItemBase {
  /**
   * {@inheritdoc}
   */
  public static function
propertyDefinitions(FieldStorageDefinitionInterface $field_definition) {
    //ToDo: Implement this.
  }

  /**
   * {@inheritdoc}
   */
  public static function schema(FieldStorageDefinitionInterface
$field_definition) {
    //ToDo: Implement this.
  }
}
```

一个字段类型看就来就像一个内容实体。事实上它们之前没有什么不同。

首先是定义了字段类型的注释

- @FieldType:指出需调用 Drupal 类库中的字段类型注释类来解析。
- id:字段类型的机器名，应保持唯一。
- label: 用于用户界面的标签。

- description: 有时标签无法表达完整的意思，字段描述可以帮助理解字段的作用。

其次，我们的类扩展了 FieldItemBase，它可以让我们实现两个方法以访问字段类型的属性:

propertyDefinitions():这个方法和内容实体的 baseFieldDefinition 方法相似(但不是一个)。我们可以定义用在表单显示中的字段的数据。

因为写出这些方法并不清晰，让我们写一些代码以便使用:

```
public static function propertyDefinitions(FieldStorageDefinitionInterface $field_definition) {
  $properties['uid'] = DataDefinition::create('integer')
      ->setLabel(t('User ID Reference'))
      ->setDescription(t('The ID of the referenced user.'))
      ->setSetting('unsigned', TRUE);

  $properties['password'] = DataDefinition::create('string')
      ->setLabel(t('Password'))
      ->setDescription(t('A password saved in plain text. That is not save dude!'));

    // ToDo: Add more Properties.

    return $properties;
}
```

schema()是必须的，它让 Drupal 知道应该如何保存我们的代码。正如你看到的，我们添加了没有出现在 propertyDefinitions()中的第三列。

现在，我们已创建了一个新的字段类型。它现在还没有任何处理数据输入的逻辑，但它可以作为一个应用到内容实体的字段。如果想把它设为基本字段:

```
public static function schema(FieldStorageDefinitionInterface $field_definition)
{
  $columns = array(
    'uid' => array(
      'description' => 'The ID of the referenced user.',
      'type' => 'int',
      'unsigned' => TRUE,
    ),
    'password' => array(
      'description' => 'A plain text password.',
      'type' => 'varchar',
```

www.drupalc.com

```
    ),
    'created' => array(
      'description' => 'A timestamp of when this entry has been created.',
      'type' => 'timestamp',
    ),

    // ToDo: Add more columns.
  );

  $schema = array(
    'columns' => $columns,
    'indexes' => array(),
    'foreign keys' => array(),
  );

  return $schema;
}
```

BaseFieldDefinition::create():在 create()中必须使用字段类型的机器名

setCardinality(-1):基数意为实体可以有多少个字段数据。例如 2 表示只有两个用户能访问这个实体，3 表示 3 个用户可以访问这个实体等等。-1 表示不限。

www.drupalc.com

字段控件

你可能想自定义字段的数据表现，控件用于呈现获取用户输入数据的表单。例如:

- 需要表单输入一个整数，但用户可以通过勾选 checkbox 以实现输入。
- 你想用自动完成功能输入数据。
- 密码输入有特殊的 UI。

字段控件可以在 modules/custom/MODULENAME/src/Plugin/Field/FieldWidget 下找到。这个路径虽然比较长。但它确是清晰的，因为能够轻易看出哪个文件属于哪里。

我们在 EntityUserAccessWidget.php 中创建一个字段控件。

```
/**
 * @file
 * Contains
\Drupal\MODULENAME\Plugin\Field\FieldWidget\EntityUserAccessWidget
 */
```

```
namespace Drupal\MODULENAME\Plugin\Field\FieldWidget;

use Drupal\Core\Field\FieldItemListInterface;
use Drupal\Core\Field\WidgetBase;
use Drupal\Core\Form\FormStateInterface;

/**
 * Plugin implementation of the 'entity_user_access_w' widget.
 *
 * @FieldWidget(
 *    id = "entity_user_access_w",
 *    label = @Translation("Entity User Access - Widget"),
 *    description = @Translation("Entity User Access - Widget"),
 *    field_types = {
 *       "entity_user_access",
 *    },
 *    multiple_values = TRUE,
 * )
 */

class EntityUserAccessWidget extends WidgetBase {
  /**
   * {@inheritdoc}
   */
  public function formElement(FieldItemListInterface $items, $delta, array
$element, array &$form, FormStateInterface $form_state) {
     // ToDo: Implement this.
  }
}
```

你注意到了吗？Drupal 一次又一次的使用这种代码风格，如果你想实现这种功能。你需要定义一个注释并继承一个基类。这样 Drupal 才能使用它。

@FieldWidget:指出注释类

id:控件的机器名

field_types:这个字段能使用的字段类型数组

multiple_values:缺省为 FALSE。如果它是 TRUE 就可以让你向实体表单提交更多的值。

如果你想在你的字段中使用这个控件，你必须像下面这样编辑字段的注释:

```
// ...
```

```
    /**
     * @FieldType(
     *     id = "entity_user_access",
     *     label = @Translation("Entity User Access"),
     *     description = @Translation("This field stores a reference to a user
and a password for this user on the entity."),
     *     default_widget = "entity_user_access_w",
     * )
     */

    // ...
```

不用等待，什么事也不会发生，因为我们已经实现了 formElement()。

```
public function formElement(FieldItemListInterface $items, $delta, array
$element, array &$form, FormStateInterface $form_state) {
  $element['userlist'] = array(
    '#type' => 'select',
    '#title' => t('User'),
    '#description' => t('Select group members from the list.'),
    '#options' => array(
       0 => t('Anonymous'),
       1 => t('Admin'),
       2 => t('foobar'),
       // This should be implemented in a better way!
     ),
  );

$element['passwordlist'] = array(
    '#type' => 'password',
    '#title' => t('Password'),
    '#description' => t('Select a password for the user'),
  );

  return $element;
}
```

你现在打开带有这个控件的表单，你会看到至少有两个输入字段，一个是用于输入用户名，另一个用于输入密码。如果你想验证表单数据，你需要实现这个控件的验证方法。请阅读 Drupal 8 表单 API 以获取更多的信息。

现在你可以使用自定义字段做更多的事。如果你不知道应该做什么，你可以看看核心模块的代码以深入理解这些内容。

www.drupalc.com

字段格式插件

最后的还没有做的事情是呈现数据，这叫做实体的查看模式。一般地，控件是用于表单模式，字段格式用于实体字段的查看模式。

字段格式没有什么好说的，我们直接上代码。下面是 modules/custom/MODULENAME/src/Plugin/Field/FieldFormatter 下的 EntityUserAccessFormatter.php

```
/**
 * @file
 * Contains
\Drupal\MODULENAME\Plugin\Field\FieldFormatter\EntityUserAccessFormatter
.
 */

namespace Drupal\MODULENAME\Plugin\Field\FieldFormatter;

use Drupal\Core\Field\FieldItemListInterface;
use Drupal\Core\Field\FormatterBase;

/**
 * Plugin implementation of the 'entity_user_access_f' formatter.
 *
 * @FieldFormatter(
 *   id = "entity_user_access_f",
 *   label = @Translation("Entity User Access - Formatter"),
 *   description = @Translation("Entity User Access - Formatter"),
 *   field_types = {
 *     "entity_user_access",
 *   }
 * )
 */

class EntityUserAccessFormatter extends FormatterBase {
  /**
   * {@inheritdoc}
   */
  public function viewElements(FieldItemListInterface $items, $langcode)
{
    $elements = array();

    foreach ($items as $delta => $item) {
```


www.drupalc.com

```
            $elements[$delta] = array(
              'uid' => array(
                '#markup' =>
\Drupal\user\Entity\User::load($item->user_id)->getUsername(),
              ),
              // Add more content
            );
          }

          return $elements;
        }
      }
```

这个例子与控件的例子是比较相似的，因此我们不需要讲太多。viewElements()将我们的字段的用户名存入$elements 数组中。在实体中必须实现访问控制。这个实现将会显示有权访问实体的用户名。

www.drupalc.com

17.16 升级Code Snippets模块到Drupal 8:创建一个自定义字段

CodeSnippets 模块在 Drupal 7 中允许你在一个字段中存储代码片段。它带有一个名为"Snippets field"的字段，并且它会渲染三个表单元素描述、源代码和语法高亮(编程语言)。

现在是时候在 Drupal 8 中升级这个模块了。

在这个教程中，我会向你讲述如何在 Drupal 8 中创建一个自定义字段。我不会解释像 PSR-4、插件注释定义之类的，因为它们涉及到很多内容。我会添加详细讲述它们的链接。

在 Drupal 7 中，字段使用 hook 来实现。但在 Drupal 8 中，字段使用插件来创建。意思是说，以前的 hook 已经被现在的插件代替。插件会为字段项的控件和格式定义相应的类。Drupal 7 中的像 hook_field_schema、hook_field_is_empty 等已成为了这些类的方法。

第一步:实现字段项

首先要做的是定义一个字段项，字段项使用类来定义，在本例中它是 SnippetsItem，这个类扩展至 FieldItemBase 类。

1.在 Drupal 8 中，类的加载使用了 PSR-4，因此，为了定义 SnippetsItem 类，我们需要在 module_name/src/Plugin/Field/FieldType/目录下创建 SnippetsItem.php 文件，在该文件中定义 SnippetsItem 类。具体如下:

```
/**
 * @file
 * Contains \Drupal\snippets\Plugin\Field\FieldType\SnippetsItem.
 */

namespace Drupal\snippets\Plugin\Field\FieldType;

use Drupal\Core\Field\FieldItemBase;
use Drupal\Core\Field\FieldStorageDefinitionInterface;
use Drupal\Core\TypedData\DataDefinition;
```

2.现在我们需要定义字段细节，如字段 id、label、缺省控件和格式等等。这和在 Drupal 7 中实现 hook_field_info 差不多。

在 Drupal 8 中，这些 hook 已经被注释所替换。

```
/**
 * Plugin implementation of the 'snippets' field type.
 *
 * @FieldType(
 *   id = "snippets_code",
 *   label = @Translation("Snippets field"),
 *   description = @Translation("This field stores code snippets in the
database."),
 *   default_widget = "snippets_default",
 *   default_formatter = "snippets_default"
 * )
 */
class SnippetsItem extends FieldItemBase { }
```

在上面的代码中我们定义了一个插件用来替换 hook_field_info，这个插件的定义是在类上面的注释区块中按照一定的格式定义的。在注释中的@FieldType 表明这是一个字段类型插件，它的属性是定义在一个数组中，属性 id 表明插件名，label 用于在界面上显示的可翻译的字符串，@Translation 表明这是能够翻译的，description 用于在界面上显示的描述信息，default_widget 指出缺省的字段控件，default_formatter 指出缺省的字段显示格式。控件和格式都使用类进行定义。

3.现在，我们定义字段项的类，我们需要在类中定义一些方法。首先要定义的就是 schema()。

在 Drupal 7 中，当你创建了一个自定义的字段后，使用 hook_field_schema 定义字段的 schema。在 Drupal 8 中，我们需要通过实现 SnippetsItem 类的 schema()方法来定义这个 schema。

```
/**
 * {@inheritdoc}
 */
public static function schema(FieldStorageDefinitionInterface $field) {
  return array(
    'columns' => array(
      'source_description' => array(
        'type' => 'varchar',
        'length' => 256,
        'not null' => FALSE,
      ),
      'source_code' => array(
        'type' => 'text',
        'size' => 'big',
        'not null' => FALSE,
      ),
      'source_lang' => array(
        'type' => 'varchar',
        'length' => 256,
        'not null' => FALSE,
      ),
    ),
  );
}
```

4.现在，我们需要添加 isEmpty()方法，这个方法用来构造一个空的字段项。这个方法就像是 Drupal 7 中的 hook_field_is_empty。

```
/**
 * {@inheritdoc}
 */
public function isEmpty() {
  $value = $this->get('source_code')->getValue();
  return $value === NULL || $value === '';
}
```

5.最后我们向类添加一个 propertyDefinitions()属性定义方法。

```
/**
 * {@inheritdoc}
 */
```

www.drupalc.com

```
static $propertyDefinitions;

/**
 * {@inheritdoc}
 */
public static function propertyDefinitions(FieldStorageDefinitionInterface
$field_definition) {
    $properties['source_description'] = DataDefinition::create('string')
      ->setLabel(t('Snippet description'));

    $properties['source_code'] = DataDefinition::create('string')
      ->setLabel(t('Snippet code'));

    $properties['source_lang'] = DataDefinition::create('string')
      ->setLabel(t('Last comment timestamp'))
      ->setDescription(t('Snippet code language'));

    return $properties;
  }
```

这个方法用来定义已存在的字段值的数据类型。"Snippets"字段已经有三个值:description、code 和 language。因此我在属性定义中把这些值设为字符串型。

第二步:实现字段控件

现在，我们已经定义了字段项，是时候定义字段控件了。为此，我们需要定义一个名为 SnippetsDefaultWidget 以扩展 WidgetBase 类。

1.创建 SnippetsDefaultWidget.php 文件并将它放在 module_name/src/Plugin/Field/FieldWidget/SnippetsDefaultWidget.php。

```
/**
 * @file
 * Contains \Drupal\snippets\Plugin\Field\FieldWidget\SnippetsDefaultWidget.
 */

namespace Drupal\snippets\Plugin\Field\FieldWidget;

use Drupal\Core\Field\FieldItemListInterface;
use Drupal\Core\Field\WidgetBase;
use Drupal\Core\Form\FormStateInterface;
```

请注意其命名空间为 Drupal\snippets\Plugin\Field\FieldWidget，并且引用以下三个命名空间。

www.drupalc.com

2.接下来，我们需要使用注释来定义这个控件。这相当于使用 Drupal 7 中的 hook_field_widget_info。

```
/**
 * Plugin implementation of the 'snippets_default' widget.
 *
 * @FieldWidget(
 *    id = "snippets_default",
 *    label = @Translation("Snippets default"),
 *    field_types = {
 *       "snippets_code"
 *    }
 * )
 */
class SnippetsDefaultWidget extends WidgetBase { }
```

在上面的代码中，@FieldWidget 表是这是一个控件定义，它的属性定义在一个数组中，id 为控件名，label 为在界面显示的控件名，field_types 定义引用的字段类型。在本例，它引用了上面定义的 snippets_code 字段。

3.最后，我们需要实现这个控件。这是通过在 SnippetsDefaultWidget 类中添加 formElement()方法来实现的。这个方法与 Drupal 7 中的 hook_field_widget_form 相似。

```
/**
 * {@inheritdoc}
 */
public function formElement(FieldItemListInterface $items, $delta, array
$element, array &$form, FormStateInterface $form_state) {

  $element['source_description'] = array(
          '#title' => t('Description'),
          '#type' => 'textfield',
          '#default_value' => isset($items[$delta]->source_description) ?
$items[$delta]->source_description : NULL,
        );
  $element['source_code'] = array(
          '#title' => t('Code'),
          '#type' => 'textarea',
          '#default_value' => isset($items[$delta]->source_code) ?
$items[$delta]->source_code : NULL,
        );
  $element['source_lang'] = array(
          '#title' => t('Source language'),
          '#type' => 'textfield',
```

www.drupalc.com

```
          '#default_value' => isset($items[$delta]->source_lang) ?
$items[$delta]->source_lang : NULL,
        );
  return $element;
}
```

第三步:实现字段的显示格式

最后，让我们来实现字段的显示格式，这需要定义一个名为 SnippetsDefaultFormatter 类，它需扩展 FormatterBase 类。

1.在 module_name/src/Plugin/Field/FieldFormatter/目录下创建 SnippetsDefaultFormatter.php 文件。其代码如下:

```
/**
 * @file
 * Contains \Drupal\snippets\Plugin\field\formatter\SnippetsDefaultFormatter.
 */

namespace Drupal\snippets\Plugin\Field\FieldFormatter;

use Drupal\Core\Field\FormatterBase;
use Drupal\Core\Field\FieldItemListInterface;
```

注意它的命名空间和引用表达式。

2.接下来，我们需要在注释中定义字段格式。这一过程和定义字段与定义字段控件类似，这相当于使用 hook_field_formatter_info。

```
/**
 * Plugin implementation of the 'snippets_default' formatter.
 *
 * @FieldFormatter(
 *   id = "snippets_default",
 *   label = @Translation("Snippets default"),
 *   field_types = {
 *     "snippets_code"
 *   }
 * )
 */
class SnippetsDefaultFormatter extends FormatterBase { }
```

3.现在让我们定义 viewElements()方法并定义真正的字段格式。这个方法与 Drupal 7 中的 hook_field_formatter_view 相类似。

```
/**
 * {@inheritdoc}
 */
public function viewElements(FieldItemListInterface $items) {
  $elements = array();
  foreach ($items as $delta => $item) {
    // Render output using snippets_default theme.
    $source = array(
      '#theme' => 'snippets_default',
      '#source_description' => $item->source_description,
      '#source_code' => $item->source_code,
    );

    $elements[$delta] = array('#markup' => drupal_render($source));
  }

  return $elements;
}
```

上面的代码说明，这种格式使用了一个名名 snippets_default 的模板来渲染输出 snippets。使用这个模板来渲染是因为我不想在 viewElements()方法中实现放置 HTML 代码的逻辑。

结论

正如前面所说，Drupal 8 最大的改变是它的字段使用 Plugin API 来实现，而不是采用 hook 机制。一旦你理解了这个原理，在 Drupal 8 中创建字段就像在 Drupal 7 中创建字段一样轻松。Drupal 8 中的很多类方法都是与 Drupal 7 中的 hook 相匹配的。

如果你想测试 Code Snippets，请下载 8.x-dev 开发版。

17.17 使用computed字段属性实现动态字段值

有时，必需在字段中添加"computed"属性，以表明此字段的值是已计算过的。例如，Drupal 核心中的文本字段，它用来存储用户输入的文本，这些文本是存储在数据库中的，我们把它叫做"原生值"。因为用户输入的值可能存在一些安全方面的问题，因此我们需使用文本过滤器对其进行过滤，我们把过滤后的内容叫做"已处理/已计算"的值。这样做的好处是这些文本只需过滤或计算一次。就能存储在缓存中以供用户以后使用。

Drupal 7 – 原来的处理方式

www.drupalc.com

在 Drupal 7 中，给字段添加 computed 属性是通过 hook_field_load()实现的。你可以看一下 text_field_load()，它对文本格式进行处理。

Drupal 8 – 新的处理方式

在 Drupal 8 中，已经删除除了 hook_field_load()函数，取而代之地是使用 computed 字段属性。

Computed 字段项的属性

- TextItemBase 是处理文本字段的基类，它定义了三个字段属性:
- value:存放在数据库中的原生值。
- format:用于过滤文本值的过滤格式。
- processed:用来存储已经计算过的文本值。

请看 TextItemBase::propertyDefinitions()方法，你将会看到"processed"属性的定义与其它属性定义有点不同:

```
<?php

  public static function propertyDefinitions(FieldStorageDefinitionInterface
$field_definition) {
      $properties['value'] = DataDefinition::create('string')
        ->setLabel(t('Text'))
        ->setRequired(TRUE);

      $properties['format'] = DataDefinition::create('filter_format')
        ->setLabel(t('Text format'));

      $properties['processed'] = DataDefinition::create('string')
        ->setLabel(t('Processed text'))
        ->setDescription(t('The text with the text format applied.'))
        ->setComputed(TRUE)
        ->setClass('\Drupal\text\TextProcessed')
        ->setSetting('text source', 'value');

      return $properties;
  }

?>
```

主要添加了三项:

- setComputed(TRUE):告诉字段 API，这个字段是已经计算了的，不需要在数据库中查询它。

- setClass('\Drupal\text\TextProcessed'):用来定义一个产生属性值的类。这个类需要实现 TypedDataInterface 接口。
- ->setSeting('text source','value'):定义字段属性的设置。

TextProcessed 类的主要目的是定义"processed"字段属性是如何被计算的。这是在 TextProcessed::getValue()方法中完成的(请看下面的代码)。处理过程的其中一部份涉及到加载它的父字段项并获取其值,然后使用这些来生成已计算的属性值。

```
/**
 * A computed property for processing text with a format.
 *
 * Required settings (below the definition's 'settings' key) are:
 *   - text source: The text property containing the to be processed text.
 */
class TextProcessed extends TypedData {

  /**
   * Cached processed text.
   *
   * @var string|null
   */
  protected $processed = NULL;

…

  /**
   * Implements \Drupal\Core\TypedData\TypedDataInterface::getValue().
   */
  public function getValue($langcode = NULL) {
    if ($this->processed !== NULL) {
      return $this->processed;
    }

    $item = $this->getParent();
    $text = $item->{($this->definition->getSetting('text source'))};

    // Avoid running check_markup() or
    // \Drupal\Component\Utility\SafeMarkup::checkPlain() on empty strings.
    if (!isset($text) || $text === '') {
      $this->processed = '';
    }
    else {
```

www.drupalc.com

```
        $this->processed = check_markup($text, $item->format,
$item->getLangcode());
      }
      return $this->processed;
   }

…

}
```

TextProcessed 类还覆写 TypedData::__construct()以强制设置它。这是在 TextProcessed 类需求中指出的，在简单的计算字段实现中是不需要的。

已计算的字段项

这里给出一个使用 hook_entity_base_field_info()来定义已计算字段项的例子(注意:没有字段项属性)。

```
/**
 * Implements hook_entity_base_field_info().
 */
function
bix_profile_entity_base_field_info(\Drupal\Core\Entity\EntityTypeInterface
$entity_type) {
   if ($entity_type->id() === 'profile') {
      $fields = array();
      $fields['completeness'] = BaseFieldDefinition::create('float')
        ->setLabel(t('Complete Profile'))
        ->setDescription(t('User profile complete percentage, such as 0.40, i.e,
40%'))
        ->setDisplayOptions('view', array(
           'label' => 'above',
           'weight' => -5,
        ));
      $fields['current_company'] = BaseFieldDefinition::create('string')
        ->setLabel(t('Current company'))
        ->setDescription(t('Current job company'))
        ->setComputed(TRUE)
        ->setClass('\Drupal\bix_profile\Plugin\BixProfileCurrentCompany')
        ->setDisplayOptions('view', array(
           'label' => 'above',
           'weight' => -5,
        ));
```

www.drupalc.com

```
    $fields['current_title'] = BaseFieldDefinition::create('string')
      ->setLabel(t('Current title'))
      ->setDescription(t('Current job title'))
      ->setComputed(TRUE)
      ->setClass('\Drupal\bix_profile\Plugin\BixProfileCurrentTitle')
      ->setDisplayOptions('view', array(
        'label' => 'above',
        'weight' => -5,
      ));
    return $fields;
  }
}
```

在上面的例子中定义了三个虚字段。第一个是没有被计算的，但你可以认为它是一个已计算或动态的字段。这三个字段的主要不同是第一个字段需在数据库中存储，但第二个和第三个字段不需要数据库存储。你需要根据字段使用情况来决定是否需要存储在数据库中。

下面的代码更新数据库表'profile'以添加一个新列'completeness'。请将下面的代码放在.install 文件中。

```
/**
 * Implements hook_install().
 */
function bix_profile_install() {
  // Different approaches for this update, see
https://www.drupal.org/node/2078241.
  // Create field storage for the 'completeness' base field.
  $entity_manager = \Drupal::entityManager();
  $definition =
$entity_manager->getFieldStorageDefinitions('profile')['completeness'];
  $entity_manager->onFieldStorageDefinitionCreate($definition);
}

/**
 * Implements hook_uninstall().
 */
function bix_profile_uninstall() {
  $entity_manager = \Drupal::entityManager();
  $definition =
$entity_manager->getLastInstalledFieldStorageDefinitions('profile')['complete
ness'];
  $entity_manager->onFieldStorageDefinitionDelete($definition);
}
```

下面的代码演示了当保存 profile 实体时，如何计算"completeness"字段。

```
/**
 * Implements hook_ENTITY_TYPE_presave().
 */
function bix_profile_profile_presave(Drupal\Core\Entity\EntityInterface $entity)
{
  if ($entity->bundle() === 'profile') {
    // Add the profile entity's completeness field value.
    $completeness = 0;

    // Check expertise field.
    $field_profile_expertise =
$entity->get('field_profile_expertise')->isEmpty();
    if (!$field_profile_expertise) {
      $completeness += 0.2;
    }

    // Other field calculations.

    $entity->set('completeness', $completeness);
  }
}
```

下面的代码演示了如何获取已计算字段属性'current_company'的值。

```
namespace Drupal\bix_profile\Plugin;

use Drupal\Core\Field\FieldItemList;

/**
 * A computed property profile's current company.
 */
class BixProfileCurrentCompany extends FieldItemList {

  /**
   * Implements \Drupal\Core\TypedData\TypedDataInterface::getValue().
   */
  public function getValue($langcode = NULL) {
    // Get professional experience field.
    $profile = $this->getEntity();
    $position = bix_profile_get_current_position($profile);
    return $position['company'];
  }
}
```

www.drupalc.com

你还可以像下面这样获取动态字段的值。

```
$profile = entity_load('profile', 23);
$company = $profile->get('current_company')->getvalue();
$completeness = $profile->get('completeness')->getvalue();
```

视图集成

现在，视图还不支持已计算的字段，例如它支持'completeness'字段，但它不支持'current_company'和'current_title'字段。下面给出一个视图集成的例子。

首先，在视图中添加这个属性到 profile 表中:

```
/**
 * Implements hook_views_data_alter().
 */
function bix_profile_views_data_alter(array &$data) {
  if (isset($data['profile'])) {
    // Add computed fields to views: current job company & job title.
    $data['profile']['current_company'] = array(
      'title' => t('Current company'),
      'help' => t('Current job company'),
      'field' => array(
        'id' => 'bix_profile_view_current_company',
      ),
    );

    $data['profile']['current_title'] = array(
      'title' => t('Current title'),
      'help' => t('Current job title'),
      'field' => array(
        'id' => 'bix_profile_view_current_title',
      ),
    );
  }
}
```

下面是视图 ID 插件，注意，命名空间必须遵循视图命名规则。

```
namespace Drupal\bix_profile\Plugin\views\field;

use Drupal\views\ResultRow;
use Drupal\views\Plugin\views\field\FieldPluginBase;
```

www.drupalc.com

```
/**
 * A handler to provide proper displays for profile current company.
 *
 * @ingroup views_field_handlers
 *
 * @ViewsField("bix_profile_view_current_company")
 */
class BixProfileViewCurrentCompany extends FieldPluginBase {

  /**
   * {@inheritdoc}
   */
  public function render(ResultRow $values) {
    $relationship_entities = $values->_relationship_entities;
    $company = '';
    // First check the referenced entity.
    if (isset($relationship_entities['profile'])) {
      $profile = $relationship_entities['profile'];
    }
    else {
      $profile = $values->_entity;
    }

    $type = get_class($profile);
    if ($type === 'Drupal\profile\Entity\Profile') {
      $company = $profile->get('current_company')->getvalue();
    }

    return $company;
  }

  /**
   * {@inheritdoc}
   */
  public function query() {
    // This function exists to override parent query function.
    // Do nothing.
  }
}
```

上面的代码完成两件事

www.drupalc.com

a.函数 query()覆盖了它的父查询函数以使 sql 查询不包括已计算的字段'current_company'。这个字段根本就不在数据库中。如果在 sql 查询中包括它，将会获得一个 fatal error。

b.函数 render()考虑两种情况。这个视图字段通过 profile relationship 渲染。另一种，这个字段通过 profile 基本表渲染。返回任何已计算的字段值。

缓存

在 D7 中，hook_field_load()允许开发者在字段被缓存以前存储已计算字段的信息。这种方式已被 Drupal 8 删除。

17.18 处理器 Handlers

处理器(Handlers)以前叫控制器(Controllers)。因为在某些情况下，实体的控制器与路由的控制器是不同的，基于这一点，把实体的控制器改为处理器更合适。当实体表示数据时，处理器用来响应对它们的操作。实体处理器可以通过 entity_type.manager 服务来访问。

www.drupalc.com

存储(Storage)

存储处理器实现了 EntityStorageInterface 接口并且缺省为 ContentEntityStorageBase 类，在这个类中提供了标准的方法以实现 CRUD 操作

访问(Access)

访问处理器实现了 EntityAccessControlHandlerInterface 接口，默认使用 EntityAccessControlHandler 类。

List Builder

列表建立器实现了 EntityListBuilderInterface 接口，默认使用 EntityListBuilder 类。

View Builder

View Builder 实现了 EntityViewBuilderInterface 接口，默认使用 EntityViewBuilder 类。

表单(Form)

表单处理器实现了 EntityFormInterface 接口。因为 EntityForm 类已经实现了这个接口，你可以直接对该类进行扩展，以实现如 add、edit 和 delete 等操作。然后你可以通过 routing.yml 文件访问这些操作。请看 Drupal 配置实体的例子，如 src/Entity/Robot.php。

```
$storage = \Drupal::entityManager()->getStorage('node');

$storage->load(1);
$storage->loadMultiple(array(1, 2, 3));
// Equaivalent to $node->save().
$storage->save($node);
$new_node = $storage->create(array('title' => 'My awesome node'));
```

www.drupalc.com

17.19 使实体可修订

使实体可修订是相当地简单，你只需要定义实体的 revision 表并且使字段成为可修订的就行。下面提供了一个如何使实体可修订的例子。这里使用了内容实体 Foo 作为例子。

1.定义 revision 表

无论是配置实体还是内容实体都需要使用注释来定义 revision 表，将 revision_table 信息添加到你的实体注释中。

File: foo_module/src/Entity/Foo.php

```
*    revision_table = "foo_revision",
```

2.定义修订 ID 键

修订通过它的 ID 被引用，因此每一个内容修订都应该有一个修订 ID。在实体注释中添加 revision_id 键。

File: <code>foo_module/src/Entity/Foo.php

```
…

 *     entity_keys = {
 *       "id" = "id",
 *       "revision" = "revision_id",
 *       "label" = "name",
 *       "uuid" = "uuid",
 *       "uid" = "user_id",
 *       "status" = "status",
 *     },

…
```

注意:如果它是一个内容实体，修订 ID 字段将会自动地被它的父类 ContentEntityBase 或 RevisionableContentEntityBase 创建。

3.使实体字段可修订

在使实体字段可修订以前，请确保你从父类 ContentEntityBase 或 RevisionableContentEntityBase 继承 baseFieldDefinitions。

```
/**
   * {@inheritdoc}
   */
  public static function baseFieldDefinitions(EntityTypeInterface $entity_type)
{
    $fields = parent::baseFieldDefinitions($entity_type);

…
```

为使你的实体字段可响应，请使用方法 setRevisionable(TRUE)。

File: foo_module/src/Entity/Foo.php

```
 /**
   * {@inheritdoc}
   */
  public static function baseFieldDefinitions(EntityTypeInterface $entity_type)
{
    $fields = parent::baseFieldDefinitions($entity_type);
```

www.drupalc.com

```
    $fields['created'] = BaseFieldDefinition::create('created')
      ->setLabel(t('Created'))
      ->setRevisionable(TRUE)
      ->setDescription(t('The time that the entity was created.'));

…
```

4.在字段保存上设置新的修订

在实体保存时你可以创建一个新的修订，在实体表单的 save 方法上使用 $entity->setNewRevision()方法创建修订。

File: foo_module/src/Form/FooForm.php

```
 /**
   * {@inheritdoc}
   */
  public function save(array $form, FormStateInterface $form_state) {
    $entity = $this->entity;

    // Set new Revision.
    $entity->setNewRevision();

…
```

上面的实现是基于扩展至基类 ContentEntityBase 的一个内容实体。这个类仅设置 revision_id 字段。如果你想为修订保存其它的信息，你的实体应该扩展类 RevisionableContentEntityBase，它带有字段修订的 Created time、Revision user、Revision log messages。对于 Drupal 8.3，你可以启用 revision UI 以向实体表单添加修订控件。

17.20 实体注释结构

配置实体类型和内容实体类型都是使用注释来定义，下面是核心中的一个例子，core/modules/user/src/Entity/User.php:

```
/**
 * Defines the user entity class.
 *
 * The base table name here is plural, despite Drupal table naming standards,
```


www.drupalc.com

```
 * because "user" is a reserved word in many databases.
 *
 * @ContentEntityType(
 *   id = "user",
 *   label = @Translation("User"),
 *   handlers = {
 *     "storage" = "Drupal\user\UserStorage",
 *     "storage_schema" = "Drupal\user\UserStorageSchema",
 *     "access" = "Drupal\user\UserAccessControlHandler",
 *     "list_builder" = "Drupal\user\UserListBuilder",
 *     "view_builder" = "Drupal\Core\Entity\EntityViewBuilder",
 *     "views_data" = "Drupal\user\UserViewsData",
 *     "route_provider" = {
 *       "html" = "Drupal\user\Entity\UserRouteProvider",
 *     },
 *     "form" = {
 *       "default" = "Drupal\user\ProfileForm",
 *       "cancel" = "Drupal\user\Form\UserCancelForm",
 *       "register" = "Drupal\user\RegisterForm"
 *     },
 *     "translation" = "Drupal\user\ProfileTranslationHandler"
 *   },
 *   admin_permission = "administer user",
 *   base_table = "users",
 *   data_table = "users_field_data",
 *   label_callback = "user_format_name",
 *   translatable = TRUE,
 *   entity_keys = {
 *     "id" = "uid",
 *     "langcode" = "langcode",
 *     "uuid" = "uuid"
 *   },
 *   links = {
 *     "canonical" = "/user/{user}",
 *     "edit-form" = "/user/{user}/edit",
 *     "cancel-form" = "/user/{user}/cancel",
 *     "collection" = "/admin/people",
 *   },
 *   field_ui_base_route = "entity.user.admin_form",
 * )
 */
class User extends ContentEntityBase implements UserInterface {
…
```

www.drupalc.com

配置实体使用@ConfigEntityType 并且需扩展
Drupal\Core\Config\Entity\ConfigEntityBase 类。内容实体使用
@ContentEntityType 并且扩展 Drupal\Core\Config\Entity\ContentEntityBase
类。

上面的注释使用了下面的键

id:实体类型唯一标识符。
label:实体标题用于 UI，@Translation 表明其可被翻译。
handlers:处理器，实现实体的各种功能。
admin_permission:实体的管理权限。
base_table:实体的基本表。
data_table:实体的数据表。
translatable:实体是否可被翻译。
entity_keys:实体键的数组。
links:使用 URI 模板语法的 Link 模板，它是一个数组。
field_ui_base_route:字段 UI 的基本路由。

实体类型注释可以使用的属性请查看 EntityType API 文档。

17.21 实体验证

Drupal 8 的实体验证被移到了实体验证 API 中并且没有结合表单验证。这允许验证实体与表单提交无关。新的验证 API 是基于 Symfony 验证器实现的。

验证是通过设置一系列约束条件来控制内容，如文本最大长度或者限制允许的值。Symfony 提供了很多有用的约束，Drupal 对其进行了扩展添加了更多的约束。Drupal Symfony 验证器已经与 Typed Data API 集成，以便于在字段定义中定义验证约束，它支持任何类型化的数据。

使用验证 API

在任何类型化的数据对象上调用 validate()方法实施验证，请看下面的例子:

```
$definition = DataDefinition::create('integer')
    ->addConstraint('Range', ['min' => 5]);
$typed_data = \Drupal::typedDataManager()->create($definition, 10);
$violations = $typed_data->validate();
```

它返回一个验证列表，如果传递一个空的验证，则验证失败。

www.drupalc.com

```
if ($violations->count() > 0) {
    // Validation failed.
}
```

验证是对象，它提供一个已翻译的信息给调用者:

在这个例子中，在数据定义中指定了一个 Range 约束。然而这个类可能自动生成一些其它验证，或添加一些数据类型默认的验证约束。举一个例子，一个 e-mail 类型会添加字符串验证以及有效 e-mail 地址验证。可通过调用 $type_data->getConstraints()返回所有已应用的验证约束。

在类型化数据对象上调用 validate()获取一个 Symfony 验证器并验证数据是很快捷的，因为 Symfony 验证器对象已经配置为类型化数据。$type_data->validate()相当于:

```
return \Drupal::typedDataManager()->getValidator()->validate($typed_data);
```

验证实体

实体字段和字段项是类型化数据对象，因此能够被验证，如下面的例子:

```
$violations = $entity->field_text->validate();
```

验证实体如:

```
$violations = $entity->validate();
```

$violations 变量包含了验证失败的属性的路径，这个路径相对于验证开始的地方。例如，如果字段的文本值($field[0]->value)验证失败，在第一个例子中$violation->getPropertyPath()的属性路径为"0.value"，在第二个例子中它是"field_text.0.value"。

在字段项属性中设置约束

实体字段定义能轻易地放置约束，这可以通过调用字段项的setPropertyConstraints()方法。在下面的例子中，字段项放置了一个最大长度的约束:

```
$fields['name'] = BaseFieldDefinition::create('string')

  ->setLabel(t('Name'))
```

www.drupalc.com

```
->setPropertyConstraints('value',array('Length' => array('max'=>32)));
```

Symfony validator 与 drupal 的关系

Drupal 使用了 Symfony 的约束类和它的的验证逻辑，这是通过 Drupal 8 的插件系统来集成它们，这样做可以使用基于注释的插件发现机制来寻找它们。其结果是在类型化数据定义中使用插件 ID 来引用约束。例如，上面的 Range 例子，引用了 Range 这个约束插件类。

Symfony 验证器使用 Drupal 的翻译类，以使验证信息正确地通地 t()运行。当 Symfony 约束信息使用一个{{ key }}语法作为占位符时，这些信息被转换成 Drupal 的表单占位符%key。出于一致性考虑，这种方式应该继续使用。

www.drupalc.com

第 18 章 表单系统

在本章中，我们将会讨论与 Drupal 表单相关的各种主题包括创建表单；使用 HTML5 元素；验证表单数据；处理表单提交的数据；修改其它表单。

Drupal 提供了一个 Form API 用于创建和管理表单，使用它你无需写任何 HTML 代码。Drupal 处理表单的创建、验证、提交。Drupal 处理创建表单的请求或者处理 HTTP POST 请求。这允许开发者只需简单地在一个表单中定义元素，提供表单的验证方式，然后通过特殊方法处理成功提交的数据。

本章包含了各种与表单相关的主题，在 Drupal 8 中，表单和表单状态都是对象。

18.1 创建一个表单

在这一节中，我们将会创建一个表单，可以通过一个菜单路径访问它。这需要创建一个路由以告诉 Drupal 如何调用表单并向用户显示。

表单使用一个类来定义，它实现了\Drupal\Core\Form\FormInterface 接口，\Drupal\Core\Form\FormBase 是表单的基类，它定义了很多有用的方法，我们可以扩展这个类来创建一个表单。

www.drupalc.com

准备开始

我们需要写代码，因此需要先创建一个自定义模块。在 Drupal 8 中创建一个自定义模块并不复杂，只需在模块目录下创建一个模块信息文件(modulename.info.yml)。对于这个例子，我们需要在/modules 下创一个名为 drupalform 的目录。drupalform 就是模块名。

在 drupalform 目录下，创建 drupalform.info.yml 文件用来描述这个模块。它的内容如下:

```
name: Drupal form example
description: Create a basic Drupal form, accessible from a route
type: module
version: 1.0
core: 8.x
```

name 指定模块名称，描述将会在模块列表页面列出。type 键指出这是一个模块。

下面是具体步骤:

1.在模块目录下创建 src 目录，在这个目录下，创建 Form 目录，这个目录用于存放定义表单的类文件。

2.接下来，创建一个名为 ExampleForm.php 的文件并把它放在 src/Form 目录下。Drupal 使用 PSR4 来发现和自动加载类。它要求每个文件只定义一个类，文件名与类名相同。目录结构模拟命名空间。

3.编辑 ExampleForm.php 文件并添加合适的命名空间，需使用的类以及类自身。下面是代码结构:

```
<?php
/**
* @file
* Contains \Drupal\drupalform\Form\ExampleForm.
**/

namespace Drupal\drupalform\Form;
use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;
class ExampleForm extends FormBase {
}
```

命名空间定义了这个类在你的模块的 Form 目录下。自动加载器将会在 drupalform 模块的路径中搜索，并从 src/From 目录加载 ExampleForm 类。

Use 表达式允许我们在引用 FormBase 类时仅仅使用类名，对于 FormStateInterface 也一样。否则我们在使用一个类时需要使用完全限定的命名空间路径。

4.\Drupal\Core\Form\FormBase 是一个抽象类，需要我们实现在接口中定义的四个方法:getFormId、buildForm、ValidateForm 和 submitForm。后面两个方法会单独讲，在这里只是简单地定义。下面是代码:

```
class ExampleForm extends FormBase {

    /**
     * {@inheritdoc}
     */

    public function getFormId() {
        return 'drupalform_example_form';
    }

    /**
     * {@inheritdoc}
```

www.drupalc.com

```
 */

public function buildForm(array $form, FormStateInterface $form_ state) {
    // Return array of Form API elements.
}

/**
 * {@inheritdoc}
 */

public function validateForm(array &$form, FormStateInterface
$form_state) {
    // Validation covered in later recipe, required to satisfy interface
}

/**
 * {@inheritdoc}
 */

public function submitForm(array &$form, FormStateInterface
$form_state) {
    // Validation covered in later recipe, required to satisfy interface
}
}
```

这段代码充实了前一步的初始类定义。FormBase 提供了很多实用的方法，但是它并没有实现接口中的方法。我们把它们定义在这里，因为它们对于每个表单定义都是唯一的。

getFormID 方法返回一个唯一的字符串以标识表单，例如，site_information。你可能会遇到以_form 结尾的表单 ID。这并不是必须的，它只不过是以前的 Drupal 版本中的一种命名规则而已。

下一步将会介绍 buildForm 方法。至于 validateForm 和 submitForm 方法是在表单处理期间调用，将会在后面讲述。

5.调用 buildForm 方法将会返回表单 API 元素，它们会渲染给最终用户。我们添加了一个文本输入字段要求用户输入公司名称和一个提交按钮。下面是代码:

```
/**
* {@inheritdoc}
*/
public function buildForm(array $form, FormStateInterface $form_ state) {
    $form['company_name'] = array(
        '#type' => 'textfield',
```

```
        '#title' => $this->t('Company name'),
    );

    $form['submit'] = array(
        '#type' => 'submit',
        '#value' => $this->t('Save'),
    );
    return $form;
}
```

我们已经将表单元素定义添加到了 form 数组中。在定义"company_name"时使用'#type'键以指定元素类型，使用'#title'键指出用于界面的标题，使用 t()函数表明它是可翻译的。在定义['submit']时使用'#type'定义类型，使用'#value'定义按钮文本，使用 t()表明可翻译。

6.为了能访问这一个表单，我们将会在模块目录下创建一个路由文件(drupalform.routing.yml)。在文件中定义一个路由来通知 Drupal 使用\Drupal\Core\Form\FormBuilder 创建和显示我们的表单。代码如下:

```
drupalform.form:
path: '/drupal-example-form'
defaults:
  _title: 'Example form'
  _form: '\Drupal\drupalform\Form\ExampleForm'
requirements:
  _access: 'TRUE'
```

在 Drupal 中，所有的路由都有一个名称，在这个例子中，它是 drupalform.form。这个路由定义了一个 path 属性并覆写了缺省的变量。路由修改了它的标题，指出它是一个表单，并给出了与表单相关联的完全限定的类名。

路由需要传递一个 requirements 属性以提出访问的条件，如果不满足条件，访问就会被拒绝。

7.访问 Extend 页面并启用这个例子模块。

8.访问/drupal-example-form，页面显示了一个表单。

18.2 创建表单的相关知识

通过 Drupal 的表单 API 创建表单涉及到许多构成表单的组件，我们这里介绍一些。

www.drupalc.com

表单元素定义

一个表单由许多表单元素组成，表单元素在 Drupal 8 中由插件提供。在 Drupal 8 中插件是一较小的可插拨的功能模块。插件与插件开发请参见第十六章。下面列出一些常用的表单元素。

weight:这个用于指出表单元素在表单中的位置。缺省，表单元素的显示顺序根据它们被添加到表单元素数组的顺序。定义一个 weight 允许开发者控制表单元素的位置。

default_value:这个是表单元素的默认值。例如，当建立配置表单时已经存在一些数据。

placeholder:这是 Drupal 新加的。Drupal 8 提供了对 HTML5 的支持，这个属性将会在输出表单时设置它的占位符。例如，我们常常看到输入密码的文本框中显示"请输入密码"，当你真正输入时，它会显示你输入的值。

表单状态(form state)

\Drupal\Core\Form\FormStateInterface 对象呈现了表单和数据的当前状态。表单状态包含用户提交的数据和创建状态信息。表单提交后的重定向也是通过 form state 处理的。在表单验证和表单提交一节还会遇到 form state。

表单缓存

Drupal 为表单使用一个缓存表。它存储了通过表单创建的标识符。这允许 Drupal 在 AJAX 期间验证表单并且当需要时轻松建立它们。这对表单的持久存储是很有用的；否则可有会发生如表单数据丢失或表单验证失效等。

18.3 使用html5 元素

随着 Drupal 8 的发布，Drupal 终于进行了 HTML5 领域。表单 API 现在允许你直接利用 HTML5 的元素。包含下面这些元素类型:

- tel
- email
- number

www.drupalc.com

- date
- url
- search
- range

这允许你的 Drupal 表单使用这些方法并带有本地验证支持。

准备开始

本节教你向 Drupal 表单添加一些新元素。你需要创建一个自定义模块，就像创建表单一节所讲的一样。

步骤

1.为了输入电话号码，你需要在 buildForm 方法中添加一个新的表单元素定义 tel:

```
$form['phone'] = array(
    '#type' => 'tel',
    '#title' => t('Phone'),
);
```

2.为了输入 Email，你需要在 buildForm 方法中添加一个新的表单元素定义 email。Form API 将会验证 e-mail 地址的有效性。

```
$form['email'] = array(
    '#type' => 'email',
    '#title' => t('Email'),
);
```

3.为了输入数字，你需要在 buildForm 方法中添加一个新的表单元素 number。FormAPI 会验证它是否为数字和限制它的取值范围。

```
$form['integer'] = array(
    '#type' => 'number',
    '#title' => t('Some integer'),
    // The increment or decrement amount
    '#step' => 1,
    // Miminum allowed value
    '#min' => 0,
    // Maxmimum allowed value
    '#max' => 100,
);
```

www.drupalc.com

4.为了输入日期，你需要在 buildForm 方法中添加一个新的表单元素定义 date。你可以传递#date_date_format 选项以修改输入的格式:

```
$form['date'] = array(
      '#type' => 'date',
      '#title' => t('Date'),
      '#date_date_format' => 'Y-m-d',
);
```

5.为了输入 URL，你需要在 buildForm 方法中添加一个新的表单元素定义 url。这个元素有一个验证器以检测 URL 格式:

```
$form['website'] = array(
      '#type' => 'url',
      '#title' => t('Website'),
);
```

6.为了输入搜索关键词，你需要在 buildForm 方法中添加一个新的表单元素定义 search。你需要指定一个路由名，搜索字段将会查询自动完成选项。

```
$form['search'] = array(
      '#type' => 'search',
      '#title' => t('Search'),
      '#autocomplete_route_name' => FALSE,
);
```

7.为了使用 range 输入，你需要在 buildForm 方法中添加一个新的元素定义 range。它扩展了 number 元素并接受 min、max、step 属性控制输入的值与范围。

```
$form['range'] = array(
      '#type' => 'range',
      '#title' => t('Range'),
      '#min' => 0,
      '#max' => 100,
      '#step' => 1,
);
```

18.4 Drupal表单元素的工作原理

www.drupalc.com

每种类型引用了一个\Drupal\Core\Render\Element\FormElement 的扩展类。它提供了元素的定义和其它函数。每种元素在类中定义了一个 prerender 方法，元素的类定义了元素类型(type)属性以及其它属性。

每种 input 定义它的主题作为 input__TYPE，允许你复制 input.html.twig 到 input.TYPE.html.twig 作为它的模板。然后解析模板属性并渲染成 HTML。

有一些元素如 e-mail，元素自身提供了验证器。e-mail 元素定义了 validateEmail 方法。下面是核心中(\Drupal\Core\Render\Element\Email::valdateEmail)的一个 eamil 验证例子:

```
/**
* Form element validation handler for #type 'email'.
*
* Note that #maxlength and #required is validated by _form_ validate()
already.
*/

public static function validateEmail(&$element, FormStateInterface
$form_state, &$complete_form) {
	$value = trim($element['#value']);
	$form_state->setValueForElement($element, $value);
	if ($value !== '' && !\Drupal::service('email.validator')- >isValid($value)) {
		$form_state->setError($element, t('The email address %mail is not
valid.', array('%mail' => $value)));
	}
}
```

这段代码会在表单提交并验证提供的 e-mail 时执行。它通过获取当前值并截掉首尾空格，并使用表单状态对象(form state object)更新表单的值。调用 email.validator 服务以验证 e-mail。如果这个方法返回 false，表单状态对象会设置错误，以防止提交表单，并将表单返回给用户修改这个值。

还有更多

表单元素由 Drupal 的插件系统提供，在下一节讲述其细节。

元素属性

元素具有一些属性以及一些验证方法。可以通过阅读 Drupal 的核心的源代码学习每种元素的属性及定义的方法，这些类在 命名空间

\Drupal\Core\Render\Element 下，类相关文件位于 /core/lib/Drupal/Core/Render/Element。

创建新元素

表单使用的每一个元素都是扩展\Drupal\Core\Render\Element\FormElement 类，它是一个插件。模块可以通过在它们的插件中添加类来提供新元素，插件的有关知识请阅读第十六章。

18.5 验证表单数据

表单 API 要求所有的与表单相关的类实现\Drupal\Core\Form\FormInterface 接口。这个接口定义了一个表单验证方法。一个表单一旦提交就会调用 validateForm 方法。表单验证方法会对表单数据进行验证，如果需要的话它会中止数据处理。表单状态对象提供了标记错误字段的方法，并提供一个用户体验工具以指出哪里出错并帮助用户修改错误。

本节内容将会基于创建一个表单一节的自定义模块。我们将会验证字段的长度。

步骤

1.打开并编辑模块目录/src/Form 目录下的 \Drupal\drupalform\Form\ExampleForm 类。

2.在验证 company_name 值以前，我们需要检查它的值是否是空值，这需使用 \Drupal\Core\Form\FormStateInterface 对象的 isValueEmpty()方法。

```
/**
* {@inheritdoc}
*/
public function validateForm(array &$form, FormStateInterface $form_state) {
    if (!$form_state->isValueEmpty('company_name')) {
        // Value is set, perform validation
    }
}
```

3.\Drupal\Form\FormStateInterface::isValueEmpty 方法的参数是表单元素的键名。例如，$form['company_name']。

4.接下来，我们将检测元素的值的长度是否大于 5:

```
/**
* {@inheritdoc}
*/

public function validateForm(array &$form, FormStateInterface $form_state) {
    if (!$form_state->isValueEmpty('company_name')) {
        if (strlen($form_state->getValue('company_name')) <= 5) {
            // Set validation error.
        }
    }
}
```

getValue 方法以表单元素的键为参数并返回它的值。因为我们已经验证了这个值为非空，我们可以获取这个值。

请注意，现在表单状态是一个对象而不是一个数组。

5.如果检测逻辑发现表单元素'company_name'的值的长度小于或等于 5 个字符，它将会抛出一个错误并阻止表单提交:

```
$form_state->setErrorByName('company_name', t ('Company name is less
than 5 characters'));
```

我们可以在长度检测逻辑中使用 setErrorByName 方法来抛出错误。如果这个字符串小于或等于 5 个字符，会在这个元素上设置一个错误。该方法的第一个参数是元素的键，第二参数是向用户显示的消息。

6.整个验证方法与下面的代码相似:

```
/**
* {@inheritdoc}
*/

public function validateForm(array &$form, FormStateInterface $form_state) {
    if (!$form_state->isValueEmpty('company_name')) {
        if (strlen($form_state->getValue('company_name')) <= 5) {
            $form_state->setErrorByName('company_name', t('Company
name is less than 5 characters'                ));
        }
    }
}
```

www.drupalc.com

7.当表单提交时，Company name 文本字段的值将大于 5 个字符或为空才可以提交。

18.6 添加多个表单验证器

在表单创建服务调用表单对象的提交表单(submitForm)方法以前，它调用对象的 validatedForm 方法。在验证方法中，表单状态可以检查表单元素的值并完成检查逻辑。如果某项被视为无效就会在相应的表单元素上设置错误，表单将不会提交并且会向用户显示错误。

当向一个元素添加一个错误时，表单上的错误计数是递增的。如果表单有任何错误，表单创建服务将不会执行提交方法。

多个验证处理器

一个表单可以有多个验证处理器。缺省情况下，所有表单至少有一个验证器，就是它自身的 validateForm 方法。表单默认只执行::validateForm 方法和所有的元素验证器。其实还可以添加更多方法，你可以调用其它类或其它表单的静态方法。

如果一个类提供了 method1 和 method2，这两个方法都要执行，下面的代码把它们添加到 buildForm 方法:

```
$form_state->setValidateHandlers([
      ['::validateForm'],
      ['::method1'],
      [$this, 'method2'],
]);
```

上面的代码设置了一个验证器数组用来执行默认的 valitedateForm 方法和其它两个方法。你可以用(::)和方法名来引用当前类的方法。也可以使用一个由类实例和需调用的方法组成的数组。

访问多维数组

表单数组支持表单元素嵌套。默认实现\Drupal\Core\Form\FormStateInterface 接口的\Drupal\Core\Form\FormState 类支持访问多维数组。你应该传递一个数组而不是一个字符串，这个数组呈现了表单元素的层次结构。如果元素定义在 $form['company']['company_name']，然后我们向表单状态的方法传递 array('company','company_name')。

www.drupalc.com

元素验证方法

表单元素拥有它们自己的验证器。表单状态对象将会聚合所有元素验证方法并把它们传递给表单验证服务。使它运行时带有表单验证。

设置 limit_validation_errors 选项，使错误传递无效。这个选项允许你绕过表单上的特定元素。这个属性在提交按钮上定义，它被称为表单状态触发的元素。它是由表单元素键所组成的数组。

18.7 处理提交的表单数据

表单的目的是为了收集一些数据并同这些数据一起完成一些事情。所有表单需要实现\Drupal\Core\Form\FormInterface 接口。这个接口定义了一个 submit 方法。一旦表单 API 调用类的验证方法后，就可运行 submit 方法。

本节将继续使用在 8.1 中创建的自定义模块。我们将会把表单转换成\Drupal\Core\FormConfigBaseForm，并允许存储字段元素。

步骤

1.在你的模块目录下，创建一个 config 目录，并在 config 目录下创建一个名为 install 的目录。

2.在 install 目录下创建 drupalform.shema.yml；这个文件告诉 Drupal 需要存储的配置项。

3.在 drupalform.schema.yml 文件中添加下面的配置 schema 定义:

```
drupalform.company:
type: config_object
label: 'Drupal form settings'
mapping:
company_name:
type: string
label: 'A company name'
```

这个定义告诉 Drupal 我们的配置名为 drupalform.company，它有一个有效的选项 company_name。

www.drupalc.com

4.替换 FormBase 的 use 表达式以使用 ConfigFormBase 类:

```
<?php
/**
* @file
* Contains \Drupal\drupalform\Form\ExampleForm.
**/
namespace Drupal\drupalform\Form;
use Drupal\Core\Form\ConfigFormBase;
use Drupal\Core\Form\FormStateInterface;
```

5.更新 ExampleForm 类以扩展 ConfigFormBase:

```
class ExampleForm extends ConfigFormBase
```

这允许我们重用 ConfigFormBase 类的方法，并且使我们的实现方法更简单。

6.因为 ExampleForm 扩展至 ConfigFormBase，需要实现 getEditableConfigNames 方法以满足\Drupal\Core\Form\ConfigBaseTrait 这个 trait:

```
/**
* {@inheritdoc}
*/

protected function getEditableConfigNames() {
    return ['drupalform.company'];
}
```

这个函数定义了配置名称，它将通过表单进行编辑。这使得通过由 ConfigFormBaseTrait 提供的方法访问表单时，在 drupalform[company]下的配置将可以被编辑。

7.删除 submit 表单元素。更新 buildForm 方法以从它的父方法那里返回数据而不是从$form 自身。我们还需要向 company_name 添加一个#default_value 选项以使它在下次加载我们的表单时已有一个默认值。

```
/**
* {@inheritdoc}
*/

public function buildForm(array $form, FormStateInterface $form_ state) {

    $form['company_name'] = array(
        '#type' => 'textfield',
```

```
            '#title' => t('Company name'),
            '#default_value' => $this->config('drupalform.company')-
>get('company_name'),
        );

        return parent::buildForm($form, $form_state);

}
```

ConfigFormBase 基类实现了 buildForm 方法以提供一个可重用的 submit 按钮。它使各个 Drupal 的配置表单的表现一致。

8.ConfigFormBase 基类提供了一个配置 factory 方法。我们将向表单元素添加一个 default_value 属性以存储当前配置。在元素的定义中添加#default_value 键。它调用由 ConfigFormBaseTrait 提供的 config 方法以加载我们的配置组并且访问一个特殊的配置值。

9.最后一步是在 submitForm 方法中的保存配置:

```
/**
  * {@inheritdoc}
  */

  public function submitForm(array &$form, FormStateInterface $form_state) {
        parent::submitForm($form, $form_state);
      $this->config('drupalform.company')
            ->set('name', $form_state->getValue('company_name'));
  }
```

通过指定我们的配置组'drupalform.company'调用 config 方法。然后，我们使用 set 方法来保存 company name 字段的值。

10.当你编辑表单并且点击 submit 按钮，你在 Company name 字段输入的值将会被保存在配置之中。

18.8 添加多个提交处理器

ConfigFormBase 利用 ConfigFormBaseTrait 提供的访问配置的 factory。这个类实现的 buildForm 也会向表单添加 submit 按钮和主题样式。submit 的处理器显示了一个保存的配置信息，但依懒于实现一个模块来保存配置。

www.drupalc.com

表单将它的数据保存在 drupalform.company 命名空间下。company name 的值存储在 name 中并且能通过 drupalform.company.name 来访问。注意配置名不必匹配表单元素的键。

一个表单可以有多个提交处理器。一般地，所有表单都会实现一个提交处理器，这就是它自己的 submitForm 方法。表单将会自动执行::submitForm 和任何定义在触发元素上的方法。可以添加很多这样的方法。然而，这允许你调用在其它类或其它表单上的静态方法。

如果一个类提供了 method1 和 method2，要想执行它们。可以在 buildForm 方法中添加下面的代码:

```
$form_state->setSubmitHandlers([
    ['::submitForm'],
    ['::method1'],
    [$this, 'method2']
]);
```

这一系列的提交处理器执行默认的 submitForm 方法和其它两个方法。你可以引用当前类定义的方法使用两个冒号和方法名。或者你可以使用由类实例和方法组成的数组。

18.9 修改其它表单

Drupal 的表单 API 不仅仅是提供了创建表单的方法而且还可以修改核心模块和贡献模块提供的表单。使用这种技术，可以向表单添加新元素，也可以改变表单元素的缺省值或隐藏元素以简化用户界面。

表单修改不是在自定义类中进行，而是在模块文件中定义修改表单的 hook，我们将使用 hook_form_FORM_ID_alter() hook 向站点的配置表单添加一个 telephone 字段。

步骤

1.在你的 Drupal 站点的目录下，创建命名为 mymodule 的目录。

2.在 mymodule 目录下，创建一个 mymodule.info.yml 文件，它包含下面的代码:

```
name: My module
description: Custom module that uses a form alter
type: module
```

www.drupalc.com

```
core: 8.x
```

Next, create a .module file in your module's directory:

```
<?php
/**
* @file
* Custom module that alters forms.
*/
```

为便于阅读和理解代码，在文件头应该加入文档区块以描述文件的意图。

3.在文件中添加 mymodule_form_system_site_information_settings_alter() hook。通过查看表单的类和检查 getFormID 方法找到表单 ID。

```
/**
* Implements hook_form_FORM_ID_alter().
**/

function mymodule_form_system_site_information_settings_ alter(&$form,
\Drupal\Core\Form\FormStateInterface $form_state) {
    // Code to alter form or form state here
}
```

Drupal 调用这个 hook 并传递当前的 form 数组和表 form state 对象。表单数组通过引用传递，允许我们修改数组而不会返回任何值。$form 参数要以引用传递需要在前面添加'&'。在 PHP 中，所有的对象都是通过引用传递的。

当我们在一个普通文中调用一个类时，比如模块文件，你需要使用完全限定的类名或者在文件的开头添加 use 表达式。在这个例子中我们添加 use \Drupal\Core\Form\FormStateInterface。

4.下一步，我们向表单添加 telephone 字段以使它能被显示和保存:

```
/**
* Implements hook_form_FORM_ID_alter().
*/

function mymodule_form_system_site_information_settings_ alter(&$form,
\Drupal\Core\Form\FormStateInterface $form_state) {

    $form['site_phone'] = array(
        '#type' => 'tel',
        '#title' => t('Site phone'),
        '#default_value' => Drupal::config('system.site')- >get('phone'),
```

www.drupalc.com

```
    );

}
```

如果已经设置了 phone 的值，则我们可以从 system.site 配置中获取它的当前值。

5.访问 Extend 页面并启用创建的 My module 模块。

6.在配置下面访问站点信息(Site Information)的设置表单并测试站点的电话号码(telephone)设置。

7.我们需要添加一个 submit 处理器以保存我们新字段的配置。我们将需要向表单添加一个提交处理器并且一个提交表单处理器的回调:

```
/**
* Implements hook_form_FORM_ID_alter().
*/

function mymodule_form_system_site_information_settings_ alter(&$form,
\Drupal\Core\Form\FormStateInterface $form_state) {

    $form['site_phone'] = array(
        '#type' => 'tel',
        '#title' => t('Site phone'),
        '#default_value' => Drupal::config('system.site')- >get('phone'),
    );

    $form['#submit'][] = 'mymodule_system_site_information_phone_
submit';

}

/**
* Form callback to save site_phone
* @param array $form
* @param \Drupal\Core\Form\FormStateInterface $form_state
*/

function mymodule_system_site_information_phone_submit(array &$form,
\Drupal\Core\Form\FormStateInterface $form_state) {

    $config = Drupal::configFactory()->getEditable('system.site');
    $config
        ->set('phone', $form_state->getValue('site_phone'))
        ->save();
```

www.drupalc.com

```
}
```

上面的代码修改了['#submit']添加了表单的提交处理器(回调函数)。这允许我们的模块与提交的表单信息交互。表单提交回调函数 mymodule_system_site_information_phone_submit 接受表单数组(&$form)和表单状态对象($form_state)。我们加载当前的 factory 以获取已编辑过的配置。然后，我们加载 system.site 并保存 phone 的值。

8.提交表单并且验证已保存的数据。

18.10 修改表单的原理及表单验证

\Drupal\system\Form\SiteInformationForm 类扩展至 \Drupal\Core\Form\ConfigFormBase 类，它用来处理表单元素的单个配置值。然而，它没有将配置的值自动地写进 form state。在这里，我们需要添加一个提交处理器(表单提交回调函数)以手动保存我们添加的字段。

表单数组(&$form)通过引用传递，允许在 hook 中修改表单的原始数据。我们可以通过表单修改 hook 添加表单元素或者修改一个已有元素，如修改标题、描述等。

www.drupalch.com

添加其它验证处理器

使用一个表单修改 hook，我们能向一个表单添加其它验证器。可以先从表单状态对象上获取当前的验证器，然后将需要添加的其它验证器附加到验证器数组上。下面是代码:

```
$validators = $form_state->getValidateHandlers();
$validators[] = 'mymodule_form_validate';
$form_state->setValidateHandlers($validators);
```

首先我们从表单状态对象$form_state 那里加载当前的验证器到$validators 变量中。然后在它的后面以数组元素形式添加了另一个回调函数。修改了$validators 变量后，我们通过使用表单状态对象的 setValidateHandlers 方法修改表单状态的验证器数组。

你可以使用 PHP 数组处理函数对你的验证器进行排序。例如，可以使用 array_unshift 函数将会把验证器放在数组的开头，这样会先执行你的验证器。

添加其它的 submit 处理器

使用表单修改 hook，我们能轻松地向一个表单添加多个 submit 处理器。方法是先加载当前的 submit 处理器，然后以数组方式添加其它的 submit 处理器。下面是代码:

```
$submit_handlers = $form_state->getSubmitHandlers();
$submit_handlers [] = 'mymodule_form_submit';
$form_state->setSubmitHandlers($submit_handlers );
```

代码先通过表单对象的 getSubmitHandlers()方法获取当前的 submit 处理器，然后对$submit_handlers 变量进行修改以加入其它的 submit 处理器，最后执行表单对象的 setSubmitHandlers 函数设置 submit 处理器数组。

同样地我们可以重排 submit 处理器的执行顺序。如使用 array_unshift 将会把回调函数放在数组的开头以使它能尽早执行。

www.drupalc.com

第 19 章 渲染系统

Drupal 的渲染系统大概由两个部份组成:

1.渲染管线----支持常见格式(HTML,json,以及其它)

2.渲染数组----它用于呈现一个 HTML 页面。允许它们嵌套(像 HTML)和修改(Drupal 是高定制的)。

Drupal 通过渲染系统将用户的请求转换成 HTML 响应传送给用户，Drupal 的渲染系统涉及到缓存以及 Symfony 的一些知识，其结构比较复杂。本章将详细讨论其细节。

19.1 渲染数组与缓存

渲染数组决定了向用户显示什么。因此，这个数组也决定了如何缓存一个结果。

如果代码动态地产生渲染数组(通常所说的，如果使用了很多 if 表达式)，这意味着 Drupal 不能简单地缓存从渲染数组渲染的 HTML，它需要在适当的时候调用这些代码(所有的 if 表达式)。

换句话说:Drupal 需要知道你的动态代码。如果不知道，它可能向不同的用户发送相同的缓存的 HTML。

当你要生成一个渲染数组时，需要思考是否对它们进行缓存，使用下面 5 步:

1.我要渲染一些东西。这意味着我必须思考渲染结果的可缓存性。

2.渲染的东西是否要耗费大量的资源，是否值得缓存?如果答案是″yes″，那么就要考虑使用什么来标识渲染的东西。这些标识符被称为缓存的键(cache keys)。

3.我渲染的结果对权限、每个 URL、每种界面语言以及其它是否有不同?这些我们称为缓存上下文(cache contexts)。注意缓存上下文是通过设置 HTTP 的 Vary 头完成的。

4.什么会导致我渲染的内容过时?例如，它依赖于哪些东西，如果这些东西改变时，我的渲染又当如何?这些是 cache tags。

5.我渲染的内容什么时候过时?例如，数据是否在有限的时间内有效？那是 max-age。它默认为持久缓存(Cache::PERMANENT)。当数据只在有限的时间内有效时，请设置 max-age，单位为秒。0 表示不缓存。

www.drupalc.com

让我们来举一个概念性的例子:

cache keys:当以摘要查看模式渲染一个节点时，缓存键如['node',5,'teaser']。

cache contexts:节点摘要显示节点标题，写作日期，作者，作者的头像以及正文，这依赖节点，作者与用户实体相关联，作者的头像与文件实体相关联，正文字段与文本格式相关联。如果它们中有些有改变，那么缓存的 node 5 teaser 的 HTML 必须重新产生。因此 cache tags 应该是:['node:5','user:3','file:4','config:filter.format.basic_html']。

cache max-age:节点摘要直到节点修改后才失效，并不是一个有限的时间。因此，不需要设置 max-age，默认 Cache::PERMANENT 就行。

注意 Drupal 会自动地设置以上的缓存 metadata，实体查看建立器和字段格式会完成它们。上例只不过是一个用来理解缓存概念的例子而已。

注意，每个实体和配置对象都实现了 CacheableDependencyInterface 接口。这个接口提供了所有的缓存元数据以确保当一个实体/配置对象被修改时它的渲染数组会失效。

请看一个直观的例子:

```
$renderer = \Drupal::service('renderer');
$config = \Drupal::config('system.site');
$current_user = \Drupal::currentUser();

$build = [
    '#prefix' => '

', '#markup' => t('Hi, %name, welcome back to @site!', [ '%name' =>
$current_user->getUsername(), '@site' => $config->get('name'), ]), '#suffix'
=> '

', '#cache' => [ 'contexts' => [ // The "current user" is used above, which
depends on the request, // so we tell Drupal to vary by the 'user' cache context.
'user', ], ], ]; // Merges the cache contexts, cache tags and max-age of the
config object // and user entity that the render array depend on.
$renderer->addCacheableDependency($build, $config);
$renderer->addCacheableDependency($build,
\Drupal\user\Entity\User::load($current_user->id()));
```

我们产生的 markup 包含用户名和站点名。用户名绑定到用户实体，因此我们想把它与缓存 tags 相关联。并且输出随用户的不同而不同(这是对用户的欢迎信息)，我们必须指定缓存上下文'user'。站点名存储在配置系统中，因此我们想把配置对象与缓存 tags 相关联，这样输出就会依赖于它。

www.drupalc.com

让我们来看看如何使欢迎信息不个性化，而是一个通常的欢迎信息。

```
$renderer = \Drupal::service('renderer');
$config = \Drupal::config('system.site');
$build = [

  '#prefix' => '

', '#markup' => t('Hi, welcome back to @site!', [ '@site' =>
$config->get('name'), ]), '#suffix' => '

', ]; $renderer->addCacheableDependency($build, $config);
```

生成的标签只是轻微不同，但代码显得更加简洁。上面的代码表示这个 markup 能用于所有的用户，因此只需使用少量的缓存 metadata。特别地，没有使用缓存上下文。

19.2 Drupal怎样使用缓存优化渲染

Drupal 提供了页面缓存和动态页面缓存，它会自动地使缓存的页面失效，显示过时的内容，显示基于权限的内容。自动地缓存渲染数组，并依据缓存上下文自动地创建缓存变体以确保某些字段不会向读者显示，但可以向新闻编辑显示。自动为页面的动态部份创建占位符，以获得更好的缓存性能。

为什么 addCacheableDependency()方法的$dependency 的参数没有 CacheableDependcyInterface 这一类型提不，难道还能传递其它么?

它的文档解释了这是为何:

```
* @param \Drupal\Core\Cache\CacheableDependencyInterface|mixed
$dependency

*    The dependency. If the object implements CacheableDependencyInterface,

*    then its cacheability metadata will be used. Otherwise, the passed in

*    object must be assumed to be uncacheable, so max-age 0 is set.
```

如果对象实现了 CacheableDependcyInterface 接口，它的缓存元数据将会使用。否则，传入的对象被视为不可缓存的，因此设置 max-age 为 0。

www.drupalc.com

最形象的例子是:AccessResultInterface 没有扩展 CacheableDenpendencyInterface，你可以实现没有提供缓存元数据的访问结果。在这种情况，我们必须将 max-age 设为 0，即不可缓存。

因为你的渲染数组依赖用户输入数据，这些数据是不可缓存的，因此渲染数组也不能被缓存。换句话说:渲染数据依赖于输入的任何数据都应该被传递到那个方法。否则，你必须手动确保存在正确的缓存元数据。

19.3 可冒泡的元数据

上一节解释了通常的缓存。这一节介绍一下渲染数组冒泡缓存的细节。

缓存元数据

缓存上下文:是指可能会影响渲染的上下文，如用户角色和语言等，当没有指定缓存上下文时，这表明渲染数组不会因为任何上下文而有所不同。

缓存 tags:渲染依赖的数据，如节点或用户账户等。当这些实体内容改变时，能使我们的缓存内容自动失效。如果数据由实体组成，你能使用 \Drupal\Core\Entity\EntityInterface::getCacheTags()成生适当的 tags；配置对象也有类似的方法。

缓存 max-age:渲染数组缓存的最大过期时间。默认是持久缓存即 \Drupal\Core\Cache\Cache::PERMANENT。

冒泡的元数据:缓存元数据 + #attached + #post_render_cache 回调。#attached 键用于指定附件，这里的附件是指附加的资源，包括 CSS、JS 和 HTML 标签等。

默认的渲染缓存上下文:每一个渲染数组使用#theme 来指定要使用的主题模板，几乎每个渲染数组都包含字符串翻译(即将文本放在 t()函数内)。这就是为什么 Drupal 添加了对缓存上下文('theme' 和'languages:'.LanguageInterface::TYPE_INTERFACE)的响应。默认的缓存上下文是在 required_cache_contexts 键中的指定的。注意，对于一些高级用法，站点可以选择覆写这些默认的上下文需求。

在渲染元素时要考虑很多因素，比如缓存是否命中，命中又怎么做，没命中又怎么做。另外渲染数组是嵌套的，因此渲染是递归进行的，就像爬树一样自上而下，这里就涉及到元素的冒泡过程，即从孩子元素冒泡到它的父元素。具体的渲染过程很复杂，有兴趣的可阅读 RenderInterface 接口的相关文档。

www.drupalc.com

19.4 自动占位符

Drupal 8 的渲染 API 会自动地为页面中变化率比较高的部份设置占位符，以提高缓存的效率(尽可能少的上下文)。

为什么要使用占位符

一些缓存上下文有比较大的基数，因此它们会有许多变体，会花费大量的时间。一个好的例子是'user'的缓存内容。

某些 tags 的失效率很高:我们事先知道这些 tags 会经常失效，不值得缓存它们。

缓存内容的过期时间比较小，有些甚至是 0，这样根本就不需要缓存它们。但是对于某些站点，一些内容更新非常频繁，比如 max-age=1 或者 max-age=5，缓存的内容一会儿就过期了。依赖于你站点的服务器架构与需求，可以不值得缓存这些 max-age 比较低的内容。

换句话说，上面所说的三个种缓存情况可能会导致缓存经常处于冒泡状态，从而影响缓存效率，这种情况还会传递给其它页面。

检测页面高动态内容和延迟渲染它们的处理过程被称为自动占位。如果某个渲染数组被延迟建立(使用了#lazy_builder 回调)并且带有'user'缓存上下文，Drupal 会将它的渲染推迟到最后。在页面上的高动态部份放置一个占位符，在渲染的最后，它会被替换为真正的内容。

但是 Drupal 仍然可以:

渲染缓存的区块、实体等，尽管它们是高动态内容，很可能不值得缓存。

使用动态页面缓存(Dynamic Page Cache)缓存整个页面。

自动占位是按照自动占位器的条件设置来完成，这些条件在 renderer.config 中指定:

```
renderer.config:
  auto_placeholder_conditions:
max-age: 0
contents: ['session', 'user']
tags: []
```

www.drupalc.com

可以自定义这些条件以满足你站点的需求。正如你所看到的，默认 Drupal 将会自动地为 max-age=0 或者与 session 或者当前用户有关的页面提供占位符。没有指定任何高失效率的缓存 tags。这是因为这些是特定于站点的。

在 Drupal 中一个 HTML 页面可以被视为一棵大的渲染树，树的根部就代表整个页面，第一级是区域，第二级是区块，第三级是区块内容等等。

因此，任何子树将会自动占位，如果:

1. 它是通过#lazy_builder 定义而不是通过一个渲染数组。
2. 它满足了自动占位的条件，这可能是以下两种方式:
    1. 它带有一个缓存值，这个属性满足了自动占位符的条件之一。
    2. 在执行和渲染它的#lazy_builder 回调时，在冒泡缓存元数据的过程中遇到自动占位的条件之一。第一次遇到时，冒泡的元数据将会被缓存以使将来命中缓存时不需要执行和渲染#lazy_builder 就能知道它应该要自动占位。

例子

请看 CommentDefaultFormatter，BlockViewBuilder，NodeViewBuilder，StatusMessages 等。

19.6 Drupal 8 渲染管线

Drupal 是怎样渲染页面的呢?首先你需要懂得一些路由概念，请先阅读路由控制器这一节。

这一节解释了 Drupal 8 渲染页面的过程，使用了一幅渲染流程图，并作了文本解释。由于这幅图非常大，建议你下载它并打印出来，以便于学习 Drupal 8 的渲染过程。

https://www.drupal.org/files/d8_render_pipeline_0.pdf

控制器 VIEW 事件 主内容渲染

路由控是器返回一个 Response 对象绕过上面的管线。它们直接依赖 Symfony 的渲染管线。路由控制器自动地返回主内容的渲染数组，并且有能力以多种方式响应请求:它可以渲染成一种特定的格式(HTML,JSON ...)并且在一个特定的方式中(例如主内容在区块中)。

www.drupalc.com

渲染管线

Drupal 的渲染管线实际上是嵌入在 Symfony 的渲染管线中。

1. 当控制器返回一个渲染数组后，HttpKernel 对象会触发 VIEW 事件，因为控制器返回的结果不是一个 Response 对象，而是一个渲染数组。
2. MainContentViewSubscriber 订阅了 VIEW 事件。它会检查控制器结果是否为一个数组，如果是，由它生成一个 Response 对象。
3. 下一步，MainContentViewSubscriber 检查是否支持请求的格式:
    1. 实现了 MainContentRendererInterface 接口的任何格式都是支持的。
    2. 如果不支持协商请求的格式，会生成一个 406JSON 响应，它会列出支持的格式(可阅读的机器名)。
4. 否则，当支持协商请求的格式时，初始化响应主内容的服务。通过调用服务对象上的 MainContentRendererInterface::renderResponse()方法产生一个响应。

www.drupalc.com

主内容渲染

每项主内容渲染服务能选择如何实现它的 renderResponse()方法，如果主内容是一个复杂的渲染，可以添加一个 protected 的辅助方法以提供更复杂的渲染结构。

Drupal 8 集成了下面的主内容数组(支持渲染任何一个下列格式/MIMI 类型的渲染数组)。

- HTML:HtmlRender(text/html)
- AJAX:AjaxRenderer(application/vnd.drupal-ajax)
- Dialog: DialogRenderer(application/vnd.drupal-dialog)
- Modal: ModalRenderer(application/vnd.drupal-modal)

HTML 主内容渲染

HTML 主内容渲染管线嵌入在 Drupal 的的渲染管线中。HTML 主内容渲染(HtmlRenderer)是最常用的，现在让我们来看一看:

1. HtmlRenderer::prepare()接受主内容渲染并确保'#type' => 'page'在渲染数组中('它用来响应<body>):

    1. 如果主内容渲染数组的类型为页面('#type' => 'page')，则不需重建立页面渲染数组。
    2. 否则，我们需要建立'#type' => 'page'这样的渲染数组。随后发出 RenderEvents::SELECT_PAGE_DISPLAY_VARIANT 事件用于选择一个页面进行显示。
    3. 默认使用 SimplePageVariant，它不使用任何页面装饰器。
    4. 但当区块模块启用后，则使用 BlockPageVariant，它允许站点创建者将区块放在页面的任意区域，并在此处布置主内容。
    5. Drupal 8 的模块也可以订阅这个事件，并使用一个不同的页面变体，甚至在每个路由上实现事件(Panels，页面管理器等等。都能实现这一事件)。
2. HtmlRenderer::prepare()的作用是保证渲染数组的类型为页面('#type' => 'page')。它调用 hook_page_attachments()和 hook_page_attachments_alter()获取附加到页面上的资源。
3. HtmlRenderer::renderResponse()将上一步返回的渲染数组包含在'#type' => 'html'中，并调用 hook_page_top()和 hook_page_bottom()。
4. 在'#type' => 'html'渲染数组上调用 Renderer，它使用 html.html.twig 模板并返回 HTML 文档字符串。
5. 这些 HTML 字符串作为一个 Response 对象返回(特别地，一个 HtmlResponse 对象，它是 Response 的子类)。

第 1-2 步产生<body>(page.html.twig)。第 3-4 步产生<html>(html.html.twig)。第 5 步发送一个包含<html>的响应对象 Response。

在限制的环境中渲染 HTML

Drupal 可以在限制的环境中渲染 HTML，如当你安装和更新 Drupal 的页面，维护模式的页面，或者发生错误时的页面，这些页面都是非常简单的，可以使用 BareHtmlPageRenderer 类处理它们的渲染过程。

19.5 渲染数组

渲染数组或者可渲染数组用于渲染 Drupal 页面。渲染数组是一个关联数组，它符合 Drupal 渲染 API 规定的标准数据结构。渲染 API 与主题 API 集成在一起。

绝大多数情况，用于建立页面或其中一部份的数据一直保持为结构化的数组，直到最后一步生成一个响应。这为扩充、修改或者完全覆写页面提供了极大的灵活性。

注意:渲染数组与被表单 API 使用的数组共享一些元素、属性和结构，有很多表单元素的属性对表单 API 是有意义的，但对渲染 API 却是无意义的。表单 API 数组通过 FormBuilder 方法转换为渲染数组，直接将表单数组传给渲染 API 可能会导致一个不可预知的错误。

什么是渲染?

渲染在 Drupal 中是指将结构化的渲染数组转换成 HTML。请阅读 HTML 主要内容渲染一节。

什么是渲染数组?

渲染数组是一个结构化的数组，它提供渲染的数据并带有一些属性以指出怎样渲染(属性如#type)。一个页面数组看起来像:

```
$page = [
  '#type' => 'page',
  'content' => [
    'system_main' => [...],
    'another_block' => [...],
    '#sorted' => TRUE,
  ],
  'sidebar_first' => [
    ...
  ],
];
```

www.drupalc.com

在 Drupal 7 中，表单可以使用 hook_form_alter()来修改，但是有很多东西需要使用模块或主题来进行修改。在 Drupal 8 中，模块或主题可以使用 hook_block_view_alter() 来修改渲染数组，使用 hook_entity_view_alter()来修改实体的渲染数组等等。

模块可以定义元素(渲染元素)类型，元素类型实际上就是预先将渲染数组打包。定义一个新的渲染类型，需创建一个 RenderElement 插件，它需实现 ElementInterface 接口。

渲染数组的属性

可以给一个渲染数组添加许多属性，渲染数组的属性将指出要如何渲染数组元素。这里列出一些比较常用的属性。渲染数组的所有属性可以阅读 RenderElement API。

#type:元素的类型。比如表单(form)，'submit'等。元素的类型是通过插件实现的，可以定义新类型。

#cache:标出数组是可以缓存的并指出它的过期时间等等。一旦渲染数组已经被渲染，就不需要再次渲染。如果缓存的内容过期或无效，就会再次渲染。这个属性还有以下子属性:

- key
- contexts
- tags
- bin

#markup:指出直接提供 HTML 的数组。除非标签非常简单，如一个段落标签，这时使用#theme 或#type 代替会更好，这样使用 theme 能自定义标签。注意这些值是通过\Drupal\Component\Utility\Xss::filterAdmin()传递，它会去除那些 XSS 标签即<script>和<style>是不允许出现在标签中。

#plain_text:指出数组提供的文本需要转义，这些值优先于#markup。

#prefix/#suffix:指出渲染元素的前缀和后缀字符串。

#pre_render:在渲染数组渲染以前，可以使用一个数组函数来修改真正的渲染数组。可以重排渲染数组或是删除一部份，设置#prointed = TRUE 以防止进一步渲染等等。

#theme:用来响应渲染数组元素(包含它的子数组)的主题函数或者是模板，它预先知道了元素的结构。

#theme_wrappers:一个主题 hook 的数组，当渲染数组的子数组渲染完成后，它用来添加渲染元素并将其放置到#children 中。典型的应用是给一个子数组添加 HTML wrappers，还用于子数组使用它们自已的主题信息的递归渲染过程。

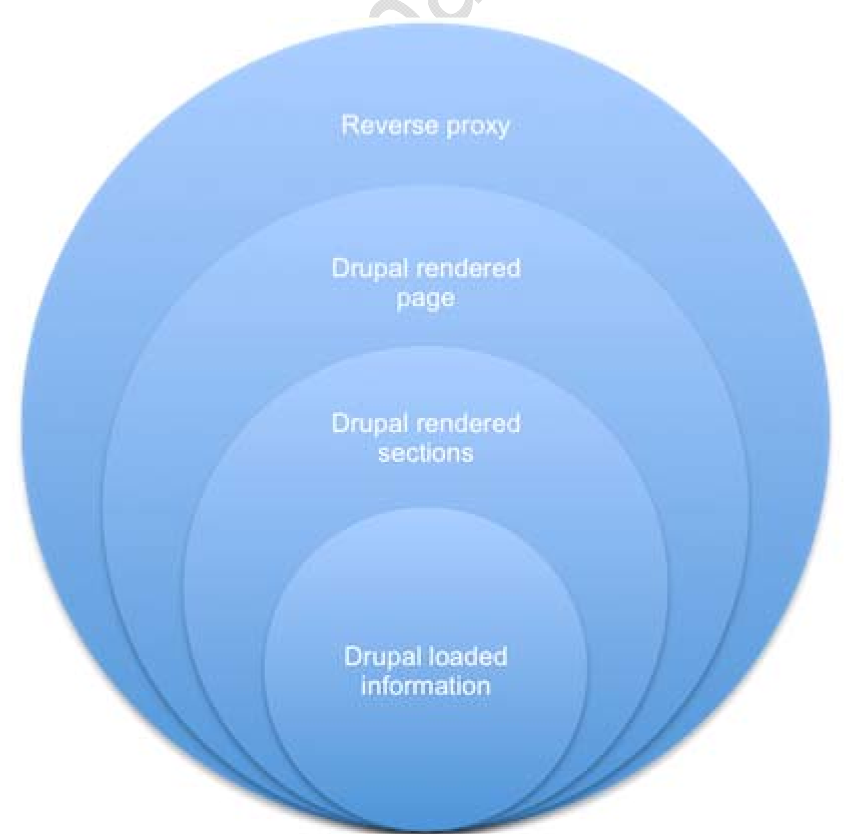
www.drupalc.com

第 20 章 缓存系统

Drupal 8 引入 Symfony 框架，集成了许多第三方库，这使得它的体积十分庞大。人们不禁要问？这么一个庞然大物，其性能可能不好吧？其实不然，Drupal 8 引入了很多特性来解决性能问题。当然 PHP 的逻辑越复杂，执行起来肯定会慢。为了解决这一个问题，Drupal 8 加强了缓存系统，它完全采用面向对象的编程方法为缓存设计了很多接口和类。并将缓存设计为系统服务，这就允许开发者能轻易地自定义缓存并注册为缓存服务。只要把渲染的结果缓存起来，用户访问的是缓存内容，因为这些内容已经计算好，所以会加快页面显示。本章将讨论一下 Drupal 8 的缓存机制。

20.1 Drupal缓存系统架构

为了获得 Drupal 的最大性能，我们需要理解 Drupal 的分层架构，Drupal 是由许多层构成的，有许多层甚至在 Drupal 之外。下面是一个简易的分层结构图：



从这个结构图中可以看出，首先是 Drupal 要加载信息，然后要对信息进行渲染，页面由页面上各个元素组成，渲染完各部份信息后就可以渲染页面，渲染页面后

就可以发送给客户端。当然如果设置了逆向代理服务器，信息还会经过这一层。这个分层的结构中，有许多层都会缓存信息，以提高性能。

对于 Drupal 来说有两种不同类型的用户，它们看到的页面是不一样的，也就带来了不同的性能因素。

一是注册用户，他们所看到的是动态页面，页面对于他们来说是唯一的，这就给缓存信息带来了挑战。

二是匿名用户，在 Drupal 中每一个访问都是一个用户，没有注册的是被视为匿名用户，它们看到的是静态页面。所有的匿名用户看到的页面都是一样的，这时缓存的作用就大了，直接将页面缓存起来，当他们访问时直接从缓存读取，这样速度就会很快。

如果你的站点用户不多，使用站点的基本配置完全可以满足要求。你的站点若有大量的用户可能需要很复杂的配置才能处理，包括配置强大的缓存、代理服务器、服务器优化等。

www.drupalc.com

基本配置

Drupal 已经考虑到了中小流量的站点，在管理配置页面可进行性能管理。具体是 Administration > Configuration > Performance > Development > Performance (admin/config/development/performance)。

Drupal 8 已经默认为你开启了动态页面缓存和静态页面缓存。当你对站点的某些方面作出修改后，一定要清除缓存才能看到修改后的效果。使用 Devel 模块可以快速地清除缓存，Devel 模块是 Drupal 开发的必备模块，也是非常好用的模块，其操作界面非常友好。也可以使用 Drush，一个十分强大的命令行工具，drush cc all 使用这一简单的命令就能清除缓存，熟练地使用它，会使你更有效地管理站点。在站点开发期间，最好禁用缓存，以免总是去清除缓存。对于生产环境的站点，应该启用缓存以获得最佳的性能。

页面缓存与区块缓存有时是失效的。有些页面组件被标记为不可缓存。某些模块也会使缓存无效，如验证码模块。

Opcode 缓存(PHP 的高情能 CPU)

PHP 是一种解释型的脚本语言，这使它在 web 服务器上占用较高的 CPU。其执行速度相比编译型语言来说会差不少。为了解决这个问题，就出现了多种类型的 PHP 的代码缓存。代码缓存就是将 PHP 源文件编译为字节码(byte code)后缓存在内存中以提供高性能并减少对 CPU 的占用。根据实测使用 PHP 的 Opcode 缓

存会带来 50%到 100%的性能提升。不过悲催的是一些入门级的 web 空间因为内存占用的问题是没有开 Opcode 的，看来要获得高性能还得硬件支持。

数据库缓存

数据库缓存是将数据缓存到数据库中，当访问数据时直接从数据库缓存表中返回，不需要额外的计算。它也使用分布式的内存缓存系统即 Memcache。Memecache 将数据库查询存储到内存中，当请求的数据在内存中时，不需要进行数据库查询，从而缓解数据库的压力。当内存用完时，它将会使用 LRU 策略来调度内存，它还会保持数据库与内存的一致性。使用 Memcache 会极大地提升网站的访问性能。

Web 代理服务器缓存

HTTP 加速器或者 Web 代理服务器缓存，将会大大减少资源需求和页面加载时间。Varnish 缓存就一个款用得非常广的 HTTP 加速器。

HTTP 加速是由逆向代理服务器处理。一个逆向代理服务器是代理服务器的一种，它会优先为 web 站点的访问者获取资源(页面、图像、文件等)。这些资源存放在虚拟内存中以便于当出现请求时能快速获取。逆向代理服务器优化了数据获取。

贡献模块

为了获得更好的性能，有时需要安装一些贡献模块。对于 Drupal 7 已经发布几年了，与网站性能相关的模块非常多，如 Boost，Authcache，Varnish，Memcache 等等。而 Drupal 8 发布的时间尚短，这方面的模块相对较少，但由于 Drupal 8 对缓存系统作了改进，其本身的性能已经非常不错了。

20.2 后端缓存

Drupal 8 已经为我们提供了几种可用的后端缓存，包括 APCu、Chain、ChainedFast、Database、Memory、PHP 这些后端缓存，下面作简要介绍。

APCu

www.drupalc.com

使用 APCu 作为后端缓存，APCu 是由 APC 发展而来，APC 是将 PHP 编译为字节码存放在内存中，以加速 PHP 执行。从 PHP 5.5 开始，Zend 官方发布了更加优秀的 OPcode 缓存，APC 就开始没落了，APC 中的系统缓存被删除，保留了用户缓存即 APCu。如果要使用这个缓存需要安装这个扩展，如果没有安装开启它是会出错的。

Chain

定义一个链式缓存实现，用来联合多个后端缓存。它能被用于将两个或多个缓存后端结合在一起以表现出一个缓存后端的行为。

举一个例子，一个很慢的、持久的存储引擎可以与一个快的、不稳定的存储引擎相结合。当从缓存中获取内容时，将会优先从快的后端缓存中获取，只有当内容不在快的后端缓存时才在慢的后端缓存中获取。一个缓存项不在快的缓存中而在慢的缓存中找到，则会把它传到快的缓存中以便于下一次请求时能快速地返回。缓存的设置和删除，两个后端缓存都会进行以确保缓存内容的一致性。

ChainedFast

定义一个后端缓存，它带一个快速的和一致的后端链。

为了减少获取缓存的时间，这个缓存允许你将一个快速缓存后端放在一个低速缓存后端的前面。典型地，快速缓存后端像 APCu，能够绑定单个 web 节点，获取一个缓存项不需要网络往返。快速缓存后端通常不是一致的(它只看到一个 web 节点的修改)。低速缓存后端如 MySQL，Memcached 或者 Redis，它们缓存所有的 web 节点，因此它是一致的，但是每次获取缓存内容都需要网络往返。

除了站点运行在多个 web 节点上有用以外，这个后端运行在单个 web 节点上也是有用的，web 节点使用快速后端缓存(例如 APCu)，它不可在 web 与 CLI 处理间共享。单个节点没有那个限制可以直接使用快速后端缓存。

我们总是使用快速后端缓存来读取缓存项，但在最后修改的内容被设置到这个缓存后端以前需检测它们是否已创建。在修改前以创建的缓存项会被丢弃，但我们使用它们的缓存 ID 来从低速缓存后端读取，同时我们更新快速缓存后端，以使它下一次能命中快速缓存后端。即我们能保证我们返回的缓存项是已更新的内容，并最大化地利用快速缓存后端。这个缓存后端使用并维护一个"last write timestamp"以确定哪些缓存项应该被放弃。

因这这个后端会为每一次写入 bin 时标记 bin 中所有的缓存项为过时数据，因些它比较适合较少修改的 bin。

www.drupalc.com

注意，这个后端是为结合一个快速不一致的缓存后端与一个低速的一致的缓存后端而设计的。为了正确的工作，它需要做一致性检测(请看"last write timestamp"逻辑)。这与\Drupal\Core\Cache\BackendChain 相反，它假设两个缓存链是一致的，因此不存在一致性检测。

Database

定义一个数据库缓存后端，它是 Drupal 缓存的缺省实现，也就是说所有的缓存数据都是存放在数据库中的。每一个缓存 bin 由数据库中的缓存表来响应，表名与 bin 名是一致的。

Memory

定义一个内存缓存实现，它会将缓存的项以 PHP 数组的形式存放在内存中，应该用作单元测试和特殊用例，一般不为每个请求存储缓存项。这个后端确实很快，但你得有足够的内存。

PHP

定义一个 PHP 缓存实现。将缓存的项存储在一个 PHP 文件中，PHP 文件存储实现了 Drupal\Component\PhpStorage\PhpStorageInterface。这是非常快的，因为 PHP 的 opcode 缓存机制。一旦一个 PHP 文件内容存储到 PHP 的 opcode 缓存，包含它后不需要从文件系统读取。相反，PHP 将会使用存储在内存中的已编译的 opcode。想像一下，它在内存中，并且已编译，又没有各种逻辑，那有多快。

www.drupalc.com

20.3 缓存bin与缓存配置

缓存的存储被分离到"bins"中，每一个 bin 包含了各种缓存项。每个 bin 可以单独地进行配置，在核心服务文件中定义了缓存 bin。

当你请求一个缓存对象时，你可以在\Drupal::cache()中指定 bin 的名称。你可以通过从服务容器中获取"cache.nameofbin"来请求一个 bin。缺省的 bin 叫"default"，服务名称为"cache.default"，它用来存储公共的和频繁使用的缓存。

其它常用的缓存 bin 如下:

- **bootstrap**:请求从开始到结束的大部份数据，对数据变化有严格限制，一般是很少失效的。
- **render**:包含缓存的 HTML 字符串，如页面缓存和区块缓存，缓存会变得很大。
- **data**:包含依据路径或者上下文不同的数据。
- **discovery**:包含缓存的发现数据，如发现了一个插件就要缓存它，避免再次发现它，又如发现的 YAML 数据等。

一个模块可以定义一个缓存 bin，需要在模块目录下的 modulename.services.yml 文件中定义服务，假定缓存 bin 的名称为'nameofbin'，其定义的代码如下:

```
cache.nameofbin:
  class: Drupal\Core\Cache\CacheBackendInterface
  tags:
    - { name: cache.bin }
  factory: cache_factory:get
  arguments: [nameofbin]
```

配置缓存

Drupal 8 的默认缓存数据是存储于数据库中的，这个配置是可以改变的。可以将整个缓存数据或者是单个缓存 bin 配置到其它的后端缓存，比如 APCu 或者 Memcache 等等。

在 setttings.php 文件中，你可以覆盖这项服务以配置其它的后端缓存。如果你的缓存服务实现了\Drupal\Core\CacheBackendInterface 接口并名为 cache.custom，下面这一行将让 Drupal 为缓存 bin"cache_render"使用 cache.custom 缓存服务。

```
$settings['cache']['bins']['render'] = 'cache.custom';
```

另外，也可以让所有的缓存都使用你提供的缓存服务即:

```
$settings['cache']['default'] = 'cache.custom';
```

配置文件缓存

当我们的数据库不够用时，我们无法再将缓存数据缓存到数据库中，另外数据库也比较慢。基于以上原因，我们可以使用文件缓存，Drupal 8 已经为我们提供了 PHP 文件缓存，下面我们来配置它。

```
$settings['cache']['default'] = 'cache.backend.php';
```

www.drupalc.com

20.4 访问检测与缓存

路由访问检测、hook_entity_access()和其它任何返回 AccessResultInterface 接口的对象都应该添加合适的缓存元数据。

如果你不清楚缓存元数据，请阅读缓存标签(cache tags)、缓存上下文和最大过期时间。

访问检查的参数

访问检查将会接受几个参数，至少一个用户账户(AccountInterface 接口)和一个实体。然后它决定这些参数的属性。

如果修改其中一个参数的属性会导致修改访问结果，那么需要给参数添加缓存依赖。如下面的例子:

```
$access_result = AccessResult::allowedIf($node->isPublished())
  ->addCacheableDependency($node);
  // Access result depends on a property of the object that might change: it is a cacheable dependency.
```

当访问结果依赖于一个不能修改的属性(如 ID,UUID)时的其它用法。例如，如果给出的用户账户是一个对象的所有者就允许访问:

```
$access_result = AccessResult::allowedIf($node->getOwnerId() === $account->id())
  // Access result depends on the node's owner, the owner might change.
  ->addCacheableDependency($node);

// Access result also depends on a user account, and the ID of the user account can never change. Hence we don't need to add $account as a cacheable dependency.

// But, if $account is the current user, and not some hardcoded user, we also need to make sure we vary this by the current user, so that we don't run this access check once and then reuse its result for all users.

if ($account->id() === \Drupal::currentUser()->id()) {
  $access_result->cachePerUser();
}
```

www.drupalc.com

20.5 缓存API

Drupal 8 对缓存系统有所加强，核心提供了许多与缓存相关的接口与类。本节将讨论缓存的一些细节。

缓存元数据

直接渲染或者用于确定渲染什么的一切都提供了缓存元素据，它们的顺序从访问结果到实体和 URL。缓存元数据由三个属性组成，它们分别是:

- cache tags:由 Drupal 管理的依赖数据，像实体&配置
- cache contexts:缓存的上下文。
- cache max-age:缓存有效期。

Cache API 的典型用法

一般来讲，你的代码到渲染(区块，实体等等)就结束了并且你的控制器将返回渲染数组或响应。因此，你一般不需要直接与缓存 API 交互。而是使用:渲染缓存和响应缓存。

渲染缓存

渲染 API 将缓存元数据嵌入到渲染数组以完成缓存。因此，缓存 API 不会与渲染缓存交互(既不获取缓存项也不创建新的缓存项)。

响应缓存

缓存元数据通过渲染 API 冒泡所有方面到响应对象(通常是 HtmlResponse)，它实现了 CacheableResponseInterface 接口。在这些响应对象上的缓存元数据允许 Drupal 8 使用页面缓存和动态页缓存，这两个模块默认是开启的。

20.6 缓存上下文

缓存上下文可以定义需缓存的内容的依赖关系。通过定义缓存上下文，使创建的缓存更易于阅读和理解。并且相同的逻辑不会在每个地方重复。例如，有些昂贵计算的数据依赖于活动主题:不同的主题有不同的结果。也就是说通过设置

www.drupalc.com

theme 作为缓存上下文来指出它们的不同。当创建一个显示个性化消息的渲染数组时，渲染数组随用户的不同而不同，缓存上下文是 user。通常，当一些昂贵计算的信息随环境而改变时，需设置缓存上下文。

缓存上下文是一个字符串，它关联到一个可用的缓存上下文服务。缓存上下文以字符串集合进行传递，因此它们的类型提示为 string[]。为什么类型为 string[]呢？因为单个缓存项可能有很多依赖的缓存上下文。

典型地，缓存上下文源自请求的上下文对象。Web 应用的大多数环境是源自请求的上下文。产生的 HTTP 响应的很大部份依赖于触发它的的 HTTP 请求的属性。但是，这并不意味着缓存上下文必须产生于请求，他们也可能依赖例如布署的代码，例如 deployment_id 缓存上下文。

其次，缓存上下文是结构化分层的。最清晰的例子:当缓存内容随用户而改变时，就无需随用户的权限而改变，因为用户已经是很细小的实体了。一个用户有许多权限，因此每个用户的缓存也指出了每种权限的缓存。

有趣的是：如果页面的一部份随用户而变，另一部份随权限而变，那么 Drupal 需要将这两者结合在一起，让它只针对用户而变。在这里就需要 Drupal 利用分层的信息以防止创建不必要的变体。

www.drupalc.com

语法

- 分离双亲与孩子。
- 缓存上下文可以指定参数；方法是用冒号加参数(如果没有指出参数，则捕捉所有可能的参数，如所有查询参数)。

Drupal 8 核心上下文

下面是 Drupal 8 核心的上下文的分层结构:

```
cookies
  :name
headers
  :name
ip
languages
  :type
request_format
route
  .book_navigation
```

```
  .menu_active_trails
    :menu_name
  .name
session
  .exists
theme
timezone
url
  .path
    .is_front // Available in 8.3.x or higher.
    .parent
  .query_args
    :key
    .pagers
      :pager_id
  .site
user
  .is_super_user
  .node_grants
    :operation
  .permissions
  .roles
    :role
```

www.drupalc.com

在缓存上下文使用的每个地方填充，会列出整个分层结构，这样做有 3 个好处:

1. 不管缓存上下文用在什么地方，它的父缓存上下文都是很清晰的。
2. 对照缓存上下文变得更加容易:如果出现了 a.b.c 和 a.b，显然 a.b 包括了 a.b.c，这样 a.b.c 就能省略，它能够被包括在它的双亲中。
3. 不需要保证树中的每一个级在整个树中唯一。

申请缓存上下文的一些例子:

- theme(随主题不同而不同)
- user.roles(随用户角色不同而不同)
- user.roles:anonymous(当前用户是否为匿名用户)
- languages(随语言类型不同而不同:界面，内容等)
- languages:language_interface(随语言界面不同而不同，LanguageInterface::TYPE_INTERFACE)
- languages:language_content(随语言内容不同而不同，LanguageInterface::TYPE_CONTENT)
- url(随整个 URL 不同而不同)
- url.query_args(随给出的查询字符串不同而不同)

- url.query_args:foo(随查询字符串?foo 不同而不同)

优化缓存上下文

Drupal 会自动地使用分层的信息来简化缓存上下文。例如，当页面一部份随着用户的不同而不同(user 为缓存上下文)，并且页面的其它部份随权限的不同而不同(user.permissions 为缓存上下文)，但是页面的最终结果并不依赖于每种权限，因为用户已经够细小了。换句话讲:optimize([user,user.permissions]) = [user]。

虽然 user 确实暗示了 user.permissions，但是那样做确实把事情过份简化了一点。如果我们那样优化 user.permissions，任何权限改变都不会影响到缓存上下文。这意为着如果权限改变了，我们仍然继续使用相同的缓存版本，甚至是权限修改时认为它应当改变。这就是为什么依赖于随时间而改变的配置的缓存上下文能关联缓存元数据:cache tags 和 max-age。当这样优化缓存上下文时，它的缓存 tags 与缓存项相关联。今后无论什么时候当赋给的权限改变时，缓存项也无效。

请记住，缓存的基本意义是避免不必要的计算。因此，优化上下文可以被当作是缓存 getContext()方法的结果。在这种情况，缓存是隐性的(值被丢弃而不是被存储)，但是效果是一样的;一个缓存被命中，就不会调用 getContext()方法，即避免计算。当我们缓存内容时，我们就关联这些内容的缓存；在缓存上下文这种情况，我们关联 tags 和 max-age。

更高级的用法例子是节点授权(node grants)。节点授权应用到特定的用户，因此缓存上下文是 user.node_grants，高动态的节点授权除外(它们是时间依赖，修改频繁)。它依赖于节点授权 hook 实现。因此出于安全，节点授权缓存上下文指定 max-age = 0，意为它不被缓存。即 optimize([user,user.node_grants]) = [user,user.node_grants]。

有些站点覆盖了缺省的节点授权上下文实现并指定 max-age = 3600 代替，这指出所有的节点缓存 hook 允许访问结果被缓存最多 1 小时。在这些站点，optimize([user,user.node_grants]) = [user]。

如何重新认识、发现和创建缓存上下文?

缓存上下文是被标记为 cache.content 的服务。任何模块都可以添加更多的缓存上下文，他们需实现\Drupa\Core\Cache\Context\CacheContextInterface 接口或者\Drupal\Core\Cache\Context\CalculatematedCacheContextInterface 接口。

你可以找出所有可用的缓存上下文，这可以通过 IDE 来寻找它的所有实现。(在 PHPStorm 中:Type Hierarchy -> Subtypes Hierarchy)。在你找到的每个类中，你将会在\Drupal\Core\Cache\Context\UserCacheContext 看到一个评论:

Cache context ID: 'user'。

这意为'user'是一个你可以在代码中真正指定的缓存上下文。另外在 *.services.yml 中寻找这个类使用的地方，并查看它的服务 ID。

服务 ID 有标准化的命名约定，它总是带有 cache_content.，接下来是它的双亲缓存下上文，最后是缓存上下文的名字，即 cache_context(强制性前缀)+route(parents)+book_navigation(缓存上下文的名称)。

以上代码定义了 route.book_navigation 缓存上下文。

```
cache_context.route.book_navigation:
    class: Drupal\book\Cache\BookNavigationCacheContext
    arguments: ['@request_stack']
    tags:
      - { name: cache.context }
```

www.drupalc.com

调试

上面所讲的对调试缓存很有用，这儿在说一说带有缓存键如['foo','bar']和缓存上下文如['languages:language_interface','user.permissions','route']。响应的缓存项将会带有一个 CID(cache ID)缓存在特定的缓存 bin 中。

foo:bar:[languages:language_interface]=en:[user.permissions]=A_QUITE_LONG_HASH:[route]=myroute.ROUTE_PARAMS_HASH

换言之:

- 在提供的顺序中首先列出缓存键。
- 其次按字母顺序列出缓存上下文，CID 各个部分形如 []=[<cache_context_name>]=<cache_context_value>。
- 使用":"将 CID 各个部份组合在一起。

这样做使分析与调试缓存变理更加容易。

Headers 调试

很容易看出一个响应依赖于哪一个缓存上下文，只需看看 X-Drupal-Cache-Contexts 头就会明白。注意:如果你没有看到这个头，你需要配置 Drupal 的开发环境。

20.7 缓存max-age

缓存 max-age 即缓存的有效期，它提供了一种创建时间依赖缓存的申明方式。有一些数据仅仅在一个有限的时间段内有效，在这种情况下，你想指定一个最大响应时间。然而，在 Drupal 8 的核心中，没有任何数据是在一个有效时间内有效；我们通常永久地缓存内容，缓存失效依赖缓存 tags。

缓存 max-age 是一个正整数，单位为秒。

例子:

60 意为着缓存 60 秒。

100 意味着缓存 100 秒。

0 意味着不缓存。

\Drupal\Core\Cache\Cache::PERMANENT 意味永久缓存，缓存失效依赖于缓存 tags。

如果你不想缓存一个渲染的区块，你应该将其指定为 max-age=0。

www.drupalc.com

20.8 缓存tags

缓存 tags 用来跟踪哪些缓存项依赖 Drupal 的数据管理。这对于一个像 Drupal 这样的内容管理系统/框架是必要的，因为相同的内容能以许多方式重用。换句话说:事先知道一些内容将用在哪里这是很重要的。在内容使用的任何位置都有可能被缓存，这意味着相同的内容在很多位置都能被缓存。然后我们可以对缓存内容进行引用，但这里有两个困难的问题:缓存失效和命名。也就是你如何使缓存的内容无效?

注意:Drupal 7 提供了三种使缓存项失效的方式:指定一个 CID，使用一个 CID 前缀，或者使缓存 bin 中所有都失效。这三种方法没有一种能使实体被修改时使缓存项失效。实体在什么时候修改是无法预先知道的。

缓存 tag 是一个字符串。缓存 tags 是以字符串集合进行传递，因此它们的类型提示为 string[]。为什么是 string[]，因为一个缓存项依赖许多缓存 tags。

按照约定，缓存 tags 形如 thing:identifier 并且一个 thing 没有多个实例概念，它的形式是 thing。这里仅仅的规则是它不能包含空格。这里没有严格的语法。

例子:

- node:5----为节点 5 缓存 tag(当它被修改时失效)
- user:3----为用户 3 缓存 tag(当它被修改时失效)
- node_list----为节点实体列出缓存 tag(当任何节点被更新、删除或创建时无效，即节点列表需要改变时)
- config:system.performance----为 system.performance 配置缓存 tag。
- library_info----为资源库缓存 tag。

Drupal 8 核心缓存 tags

Drupal 的数据管理主要分为三类:

1. 实体----缓存 tags 形式<entity type>:<entity ID>
2. configuration----缓存 tags 形式 config:<configuration name>
3. custom(例子 library_info)

Drupal 自动地为实体&配置提供缓存 tags----请看实体基类和配置基类。(所有的实体类型和配置对象继承它们)。使用::getCacheTages()获取它们的缓存 tags，例如$node->getCacheTags()，$user->getCacheTags(),$view->getCacheTags()等。EntityTypeInterface::getListCacheTags()用于当创建、更新或删除实体类型时使缓存实体类型的列表失效。

使用 cache_tags.invalidator:invalidateTags()让缓存 tags 失效(或者当你不能注入 cache_tags.invalidator 服务:Cache::invalidateTages()时)，它接受一个缓存 tags 集合(string[])。

注意这个缓存失效是针对所有的缓存 bin 的，它对于单个缓存 bin 是没什么意义的，因为数据已经被修改，缓存 tags 已经失效了。

调试

综上所述对调试缓存是有益的。但是这里再提一件事:让我们看一下带有缓存 tags['foo','bar']的缓存。然后响应缓存的项将会有一个 tags 列(这里假定使用数据库作为后端缓存)会有下列值:

www.drupalc.com

bar foo

另一方面:

- 缓存 tags 是由空格分开。
- 缓存 tags 以字母顺序存储。

这样做是为了便用分析&调试缓存!

缓存头

很容易看出一个响应依赖于哪些缓存 tags:只需看看 X-Drupal-Cache-Tags 头!这也是为什么会禁止空格，因为 X-Drupal-Cache-Tags 头就像许多 HTTP 头，使用空格分离值。注意:如果你没有看到这些头，你需要配置 Drupal 的开发环境。

集成逆向代理服务器

如果不想在 Drupal 中缓存响应结果并使带有缓存 tags 的缓存失效，你可以将响应结果缓存到逆向代理服务器中(Varnish，CDN...)，并可以使这些缓存失效，使用缓存 tags 与这些响应缓存相关联。为了让逆向代理服务器知道每个响应与哪些缓存 tags 相关联，你需要发送一个缓存 tags 头。

就像 Drupal 8 可以发送 X-Drupal-Cache-Tags 头用于调试一样，它也可以发送一个 Surrogate-Keys 头(使用空格分隔)被 CDN 接受，或者发送一个 Cache-Tag 头(使用逗号分隔)被其它 CDN 接受。你自己的服务器也可以作为逆向代理服务器运行，而不需要其它的商业 CDN 服务。

根据经验，推荐你的 web 服务器和逆向代理服务器支持 16KB 的头。

1.HTTP 是基于文本的。缓存 tags 也是基于文本的。逆向代理服务器可以将内部不同的数据结构表示成缓存 tags。选择 16KB 的头基于两个因素:A.为了适用大多数情况；B.什么都可以实现。典型的 web 服务器(Apache)和典型的快速 CDN 都支持 16KB 头。这意味着可以支持大约 1000 个缓存 tags，这对于 99%应用已经足够。

2.缓存 tags 的数量根据不同的站点和不同的响应是不同的。如果一个响应依赖其它很多东西，那它将会有很多缓存 tags。但在一个响应中超过 1000 个缓存头是少有的。

3.当然，这一个指导线(每个响应 1000 个缓存头足够)将有可能过时，因为我们看到有许多真实的 web 应用在使用它，一些特定的应用的缓存头容量已超过这个数。

www.drupalc.com

最后，任何超过 1000 缓存 tags 可能指出一个严重的问题:响应过于复杂，以致于它应该被分割。没有什么能阻止你在 Drupal 中超过这个数，但它可能需要手工调整。一些极其复杂的用例是可以接受的。那样的例子甚至远小于 1000 个缓存 tags。

20.9 缓存依赖接口

为了更好地处理缓存元数据(缓存 tags，缓存上下文和缓存有效期)，Drupal 8 定义了缓存依赖接口(CacheableDependencyInterface)。

想像一下，在渲染数组中手工为每个实体和配置对象构造缓存 tags(或者进行其它计算)。并且在多语言站点上，手工添加必要的缓存上下文(比如翻译实体和配置对象的语言覆盖)。

不仅仅是实体&配置(entities&configuration)，还包括访问结果(access results)，区块插件(block plugins)，菜单链接(menu links)，上下文插件(context plugins)，条件插件(condition plugins)，等等，所有这些最终都会渲染。

在 Drupal 8 的早期开发中，这是常见的。但这是站不住脚的，并且很容易出错。

这就是引入缓存依赖接口(CacheableDependencyInterface)的原因。正如它的名字一样:实现这个接口的对象能自动获得缓存依赖。

例如，当创建一个渲染<p>Hi，%user，welcome to %site!</p>的渲染数组，需要依赖当前用户 user 实体和 system.site 配置。当那个渲染数组缓存时，它的缓存依赖是 user 实体和配置对象。

任何值对象都能实现 CacheableDependencyInterface 接口(即对象表示为一个数据逻辑)。如果你去看它的 API 文档，你将会看到在 Drupal 8 的核心中有许多值对象实现了这个接口。事实上，在写 Drupal 8 的代码时，与你交互的大部份对象实现它将是安全的。

有两种经常会遇到的极端对立的情况，出于方便 Drupal 提供了相应的 traits:永久缓存一个不会改变的对象(UnchangingCacheableDependencyTrait 总是返回 max-age === permanent)；一个对象总是会动态地计算永远不会缓存(UnchangingCacheableDependencyTrait 总是返回 max-age === 0)。

CacheableDependencyInterface 接口仅仅处理继承、规范一个对象的缓存元数据。有时，一个对象有多个变体。

最明显的例子是实体翻译(相同的实体，相同的实体 ID，只是在不同的翻译中)和配置翻译(相同的配置对象，相同的配置名，只是带有一个语言覆写)。在这两种情况，在原对象(未翻译)

的缓存元数据仍然可用。例如，node:5 缓存 tag。但在实体的用例中，必须要使用内容语言缓存上下文('languages:' . LanguageInterface::TYPE_CONTENT)，为了传递，这个实体是原来的实体的变体，它们随着内容语言缓存上下文的不同而不同。与此类似，在配置对象的用例中，必须要使用界面语言缓存上下文('languages:' . LanguageInterface::TYPE_INTERFACE)，为了传递，配置对象是原配置对象的变体。它们根据界面语言缓存上下文的不同而不同。

在以上的例子中，我们需要重新定义对象的缓存元数据以指出加载一个变体。出于这个原因，我们添加了 RefainableCacheableDependencyInterface 接口，使用它可以添加缓存 tags、缓存上下文和更新缓存有效期。

为了更好地实现这个接口，系统定义了一个方便的 trait:RefinableCacheableDependencyTrait。

实体与配置对象

在 Drupal 8 中(核心、贡献&自定义)的所有实体都实现 EntityInterface 接口，这个接口扩展了 CacheableDependencyInterface 和 RefinableCacheableDependencyInterface。除了这些接口，Drupal 8 的核心实体还扩展 Entity 这一抽象类，并鼓励贡献/自定义的实体也这样做。这意味着在 Drupal 8 中与你交互的实体都能自动地获得缓存 tags(形如<entity type>:<entity ID>，例如 node:5 和 user:3)和与翻译相关的缓存上下文。

在 Drupal 8 的核心中的所有配置对象都扩展了 ConfigBase 这一抽象基类，这个类实现了 CacheableDependencyInterface 接口和 RefinableCacheableDependencyInterface 接口。这意味着在 Drupal 8 中每个与你交互的配置对象都能自动地获得缓存 tags(形如 config:<configuration name>，例如 config:system.performance)和与配置覆写相关的缓存上下文。

最后，在 Drupal 8 中所有的实体和配置对象将自动获得内容/界面语言缓存上下文，这多亏了 Entitymanager::getTranslationFromContext()和 LanguageConfigFactoryOverride::getCacheableMetadata($name)。

使用缓存信赖对象

渲染是最普遍的依赖一个对象的例子，这个对象是一个缓存依赖。为了简化它，我们定义了

RendererInterface::addCacheableDependency($build,$dependency)方法----这里的$build 变量是一个渲染数组，$dependency 是它依赖的对象。这个对象的缓存上下文将自动被渲染数组吸收，这意味着，当对象的缓存 tag 失效时，渲染数组将会失效，如果使用了不同的翻译，将会导致缓存不同版本的内容(即内容语言缓存上下文映射到一个不同的语言)，如果依赖对象的 max-age 不为永久，缓存内容将会自动过期。

另一个很好的例子是访问检测，它返回 AccessResult 对象，它也有一个 AccessResult::addCacheableDependency($dependency)方法。注意，在这里只有$dependency 参数，因为我们能在访问结果对象自身上存储传入$dependency 的缓存元数据(带有渲染数组的渲染器例外)。

20.10 外部缓存Varnish

Varnish 缓存是一个 web 应用加速器，它也叫做逆向代理服务器。Varnish 用于数千个 Drupal 站点以提升站点性能，它也可用于缓存 tags 并能轻松地使缓存无效。

为了集成基本缓存 tag，你需要做三件事以使 Varnish 在 Drupal 中能正常工作:

1. 更新你的 Varnish VCL 以使它能处理 BAN 请求。
2. 为每一个请求发送一个缓存 tag 头(例如 X-Cache-Tags)，包含一个使用空格分隔的页面缓存 tags 列表。
3. 当与缓存 tags 相关的内容或页面更新后导致页面过期，能发送一个带有这些缓存 tags 的 BAN 请求。

更新 Varnish VCL

Symphony 的 FOSHttpCache 包含了一些更新 VCL 的文档，但是 Varnish 4.x 在这方面有一些细微的改变:

```
vcl_recv:
sub vcl_recv {
    …
    # Only allow BAN requests from IP addresses in the 'purge' ACL.
    if (req.method == "BAN") {
        # Same ACL check as above:
        if (!client.ip ~ purge) {
            return (synth(403, "Not allowed."));
        }
```

```
        # Logic for the ban, using the X-Cache-Tags header.
        if (req.http.X-Cache-Tags) {
          ban("obj.http.X-Cache-Tags ~ " + req.http.X-Cache-Tags);
        }
        else {
          return (synth(403, "X-Cache-Tags header missing."));
        }

        # Throw a synthetic page so the request won't go to the backend.
        return (synth(200, "Ban added."));
    }
}

vcl_backend_response:
sub vcl_backend_response {
    # Set ban-lurker friendly custom headers.
    set beresp.http.X-Url = bereq.url;
    set beresp.http.X-Host = bereq.http.host;
    …
}

vcl_deliver:
sub vcl_deliver {
    # Remove ban-lurker friendly custom headers when delivering to client.
    unset resp.http.X-Url;
    unset resp.http.X-Host;
    # Comment these for easier Drupal cache tag debugging in development.
    unset resp.http.X-Cache-Tags;
    unset resp.http.X-Cache-Contexts;
    …
}
```

在进行了适当的修改后，请重启你的 Varnish 服务器。

发送一个缓存 tags 头

Drupal 贡献模块 Purge 能自动地为每一页配置 Purge-Cache-Tags 头，因此当你启用了 Purge 模块，它将会自动地发送。

当内容或配置修改时发送一个 BAN 请求

www.drupalc.com

使用 Generic HTTP Purger 模块，你可以到 Purge 的配置页面
(admin/config/development/performance/purge)添加一个 HTTP Purger。然后
输入你的 Varnish 服务器信息(主机名，端口号，路径等)，在'Headers'配置下方
对 header 进行配置:

Header: X-Cache-Tags
Value: [invalidation:expression]

一旦你保存这个 Purger 配置,再配置一个处理 Purge 队列(drush p-queue-work)
的 cron 任务，当 purge 队列触发了 bans 时，Varnish 可以禁止页面。

www.drupalc.com

第 21 章 服务与依赖注入

现代的 web 应用程序把一切都看作对象，然后定义一个对象管理器来对对象进行中央处理。这有点像以前的全局变量和全局函数，但那并不是面向对象的，代码的维护比较困难。Drupal 8 引入了 Symfony 框架，并引入了它的服务概念。服务就是执行全局任务的类，如系统缓存、e-mail 发送等都是服务。由于一个系统包含多项服务，由此出现了服务容器这一概念。服务容器就是一个服务管理器，由它对系统的各项服务实施中央处理。本节将介绍 Drupal 8 的服务细节。

21.1 Drupal 8 的服务与依赖注入机制

在 Drupal 8 中，一项服务是一个由服务容器管理的任意对象。Drupal 8 引入服务这一概念是为了降低可重用功能的耦合性，并使服务可插拨可替换，只需注册一个新的服务容器就可实现服务替换。作为一个开发者，通过 Drupal 提供的服务容器来访问任意服务这是最佳的方式。与服务有关的基本概念在 Symfony2 的文档中已作出了很好的解释。

从开发者的角度说，服务是用于完成像访问数据库、使用缓存或是发送 e-mail 这样的操作。在访问数据库时，我们不使用 PHP 提供的原生 MySQL 函数，而是使用核心提供的数据库服务来完成对数据库的访问，而不需要担心底层的数据库是 MySQL 还是 SQLlite。这样保持了代码的独立性、标准性。同样地，使用这种机制发送 e-amil 时我们不必关心使用 SMTP 服务器还是其它。

www.drupalc.com

核心服务

核心服务定义在 CoreServiceProvider.php 和 core.services.yml 中。请看一段例子:

```
…
  language_manager:
    class: Drupal\Core\Language\LanguageManager
    arguments: ['@language.default']
  …
  path.alias_manager:
    class: Drupal\Core\Path\AliasManager
    arguments: ['@path.crud', '@path.alias_whitelist', '@language_manager']
  …
  string_translation:
    class: Drupal\Core\StringTranslation\TranslationManager
```

```
…
breadcrumb:
  class: Drupal\Core\Breadcrumb\BreadcrumbManager
  arguments: ['@module_handler']
…
```

每一项服务能依赖于其它的服务。在上面的例子中，别名管理服务 path.alias_manager 依赖于 path.crud、path.alias_whitelist 和 language_manger 服务，这可以从变量列表中看出。给服务定义一个依赖只需在服务名称前加上@标志，像@language_manager。当 Drupal 其它地方的代码请求 path.alias_manager 服务时，服务容器保证优先请求 path.crud、path.alias_whitelist 和 language_manager 服务，然后再将它们依次传递给 path.alias_manager 服务的构造函数。同样地，language_manager 又依赖于 language.default 等等。

Drupal 8 提供了大量的服务，可以通过查看 CoreServiceProvider.php 和 core.services.yml 文件了解这些服务。

一个服务容器(或者依赖注入容器)是一个用来管理服务实例化的 PHP 对象。Drupal 的服务容器是建立在 Symfony 2 服务容器之上并记录了这个文件的结构、特殊字符、可选依赖等。这一切都能在 Symfony2 服务容器文档中找出。

www.drupalc.com

在对象中使用依赖注入访问服务

在 Drupal 8 中，依赖注入是访问和使用服务的最好的方式，应该尽可能这样做。而不是调用全局服务容器，服务应该以参数的形式传入到构造函数中或者通过 setter 方法注入。Drupal 核心中的模块提供了许多控制器和插件类，它们使用这种模式来提供良好的服务。

全局类在全局函数中使用。然而，Drupal 8 的基本理念是循环使用在控制器、插件等中定义的类。最好的方法就是不要调用全局服务容器而是将需要的服务以参数的形式传入构造函数或者将需要的服务通过 setter 方法注入。

在服务中显示地传递一个依赖的对象叫做依赖注入。在有些情况下，依赖被显示地传入构造函数，例如，路由访问检测在服务创建时获取当前用户注入，并在检查访问时传递当前请求。你也可以使用 setter 方法设置一个依赖。

在全局函数中访问服务

全局 Drupal 类提供了静态方法来访问几种公共服务。例如，Drupal::moduleHandler()将返回模块处理服务，Drupal::translation()将返回字符

串翻译服务。如果没有指定方法来访问你想访问的服务，你可以使用 Drupal::service()方法来获取已经定义的服务。

如访问数据库服务通过专门的\Drupal::database()。

```
// Returns a Drupal\Core\Database\Connection object.
$connection = \Drupal::database();
$result = $connection->select('node', 'n')
  ->fields('n', array('nid'))
  ->execute();
```

通过通常的\Drupal::service()访问日期服务:

```
// Returns a Drupal\Core\Datetime\Date object.
$date = \Drupal::service('date');
```

理论上，你应该在全局函数中使代码最小化，并在依赖注入的地方进行重构，如控制器、监听器、插件等等。

www.drupalc.com

定义你自己的服务

你可以使用一个 module_name.services.yml 文件定义你自己的服务，module_name 是定义服务的模块名。这个文件的结构与 core.services.yml 的文件结构相同。

有几种子系统需要你定义服务。如，自定义路由访问检测类，自定义参数转换，自定义插件管理器，这些都需要将你的类注册成为一项服务。

应该尽可能地使用$GLOBALS['conf']['container_yamls']来添加更多的 YAML 文件以发现服务，虽然这样使用很少见。

Drupal 7 的全局函数与 Drupal 8 的服务

我们以代码调用模块的 hook 作为例子来说明 Drupal 7 与 Drupal 8 的区别。在 Drupal 7 中，你将使用 module_invoke_all('help')来调用所有的 hook_help()实现。因为我们直接在代码中调用了 module_invoke_all()函数，所以在不改变核心函数的情况下难以改变 Drupal 调用模块的方式。

在 Drupal 8 中，module_*函数已经被替换为 ModuleHandler 服务。因此在 Drupal 8 中，你将使用\Drupal::moduleHandler()->invokeAll('help')来达到相同效果。在这个例子中，\Drupal::moduleHandler()在服务容器中定位已注册的模块处理服务并调用它的 invokeAll()方法。

这种方案比 Drupal 7 的解决方案更好，因为它允许 Drupal 发行版或者主机提供商又或者其它模块覆写调用模块的方式，它们可以提供自己的类来进行模块处理，并将它注册成模块处理服务。这一修改对于 Drupal 的其它代码来说是透明的。这意味着 Drupal 的很多部份是可以交换出去而不需要修改它的核心。代码依赖能更好地文档化，它们的关系能更好地分离。最终各项服务能使用它们自己的界面进行单元测试，与集成测试相比，单元测试更快更紧凑。

Drupal 7 全局变量与 Drupal 8 服务

在 Drupal 7 中的全局变量像 global $language 和 global $user 等也可以通过 Drupal 8 的服务来访问。如 Drupal::languageManager->getLanguage(Language::TYPE_INTERFACE)以及 Drupal::currentUser()等。

21.2 修改已有服务 提供动态服务

服务容器有许多优点。访问和实例化每项服务只需使用一个字符串键和一个已定义的接口，通过接口的不同实现可以将这些服务交换出去。为修改一项已有服务，需实现一个扩展 ServiceProviderBase 的类并实现 alter()方法。

注意:虽然交换服务是很容易的，但在使用这项服务时应该谨慎。如果多个模块交换相同的服务，每个模块都无法预测它们自已能交换成功。如果一项服务使用事件、hooks 或其它来提供模块扩展，这肯定能提高它对其它模块的兼容性。

注意:如果你想让修改服务自动生效，你的类需要遵循特定的命类规则，module_nameServiceProvider，module_name 是模块名，在这个类中需要遵循驼峰命名法，随后是 ServiceProvider。命名空间应该在模块的最上层即 Drupal\module_name，它还必须实现 \Drupal\Core\DependencyInjection\ServiceModifierInterface 接口 (ServiceProviderBase 就实现了它)。

例如，如果模块名为 my_module，请定义 my_module/src/MyModuleServiceProvider.php:

```
/**
 * @file
 * Contains Drupal\my_module\MyModuleServiceProvider
 */

namespace Drupal\my_module;
```

www.drupalc.com

```
use Drupal\Core\DependencyInjection\ContainerBuilder;
use Drupal\Core\DependencyInjection\ServiceProviderBase;

/**
 * Modifies the language manager service.
 */

class MyModuleServiceProvider extends ServiceProviderBase {

  /**
   * {@inheritdoc}
   */

  public function alter(ContainerBuilder $container) {
    // Overrides language_manager class to test domain language negotiation.
    $definition = $container->getDefinition('language_manager');
    $definition->setClass('Drupal\language_test\LanguageTestManager');
  }
}
```

可能用到交互服务的其它地方如交换字符串翻译服务。一个常见的功能需求是 Drupal 的核心没有提供方语言支持或者是非正式语言/正式语言，它们之前的翻译差别非常小。通过交换字符串翻译服务，贡献模块能轻易地解决这个问题。

最后，也可以定义 register()方法用来动态地注册服务，但这用得非常少。

21.3 表单的依赖注入

需要一个 Drupal 服务或者一个自定义服务的表单应该使用依赖注入来访问服务。下面介绍一个使用'current_user'服务来获取当前用户的 uid 的表单例子(这与在 Drupal 8 表单 API 中使用表单类似)。如果模块为/module/example，则应创建文件/module/example/src/Form/ExampleForm.php 其内容如下:

```
<?php
/**
 * @file
 * Contains \Drupal\example\Form\ExampleForm.
 */
namespace Drupal\example\Form;

use Drupal\Core\Form\FormBase;
```

www.drupalc.com

```
use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Session\AccountInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Implements an example form.
 */
class ExampleForm extends FormBase {
  /**
   * @var AccountInterface $account
   */
  protected $account;

  /**
   * Class constructor.
   */
  public function __construct(AccountInterface $account) {
    $this->account = $account;
  }

  /**
   * {@inheritdoc}
   */
  public static function create(ContainerInterface $container) {
    // Instantiates this form class.
    return new static(
      // Load the service required to construct this class.
      $container->get('current_user')
    );
  }

  /**
   * {@inheritdoc}.
   */
  public function getFormID() {
    return 'example_form';
  }

  /**
   * {@inheritdoc}.
   */
  public function buildForm(array $form, FormStateInterface $form_state) {

    // Get current user data.
```

www.drupalc.com

```
    $uid = $this->account->id();

    // ...
  }

  /**
   * {@inheritdoc}
   */
  public function submitForm(array &$form, FormStateInterface $form_state)
{
    // ...
  }
}
?>
```

创建表单(create)方法是一个工厂方法，它返回一个 ExampleForm 对象的新实例。ExampleForm::create 加载了一个或多个服务。这里的服务可以是任意定义在 core.services.yml 或者任意定义在*.services.yml 中的服务。

ExampleForm::__construct 使用 ExampleForm::create 加载的服务并将它存储在类的属性中。在 ExampleForm::create 中加载服务的顺序必须与 ExampleForm::__construct 中参数的顺序一致。

create 方法在 Drupal\Core\DependencyInjection\ContainerInjectionInterface 接口中定义，它允许控制器被实例化并带有一个服务。
Drupal\Core\Form\FormBase 已经实现了这个接口。任何扩展至 Drupal\Core\Form\FormBase 的表单都具有依赖注入的能力。诸如 ConfigFormBase 和 ConfirmFormBase 等等。

添加表单控制器如何能自动地插入$route 信息到你的 buildinfo()方法，这需要定义一个 Request 类对象作为参数。

例如:

buildForm(array $form, FormStateInterface $form_state, **Request $request = NULL**)

21.4 服务标签 Service Tags

关于服务标签 Service Tags

有些服务带有服务标签(Service Tags)，它是在服务定义中定义。服务标签用于将相关的服务分组在一起，或者指出服务的相同行为。典型地，如果你给服务定义了标签，你的服务类必须实现一个响应接口。一些公共例子如下:

access_check:指定一个路由检测服务。

cache.bin:指定一个高速缓存 bin 服务。

event_subscriber:指出一个事件订阅报务。事件订阅可以用于动态路由和路由修改以及其它目的。

needs_destruction:如果这个服务已实例化，指出需要在最后调用 destruct()方法。

context_provider:指定一个区块上下文提供者，用于条件区块。它必须实现\Drupal\Core\Plugin\Context\ContextProviderInterface。

http_client_middleware:指出该服务提供了一个 guzzle 中间件。请阅读 https://guzzle.readthedocs.org/en/latest/handlers-and-middleware.html 以获得更多信息。

www.drupalc.com

收集服务

Drupal 有一个 API 用于定义一个服务收集器，它是一种特殊类型的服务，它通过 tag 来收集其它服务。服务收集器的工作方式是定义一个服务收集器和一个或多个已收集的服务(定义它们的类和*.services.yml)。当使用收集器服务时，Drupal 将实例化它(和其它服务一样，使用依赖注入)，然后定位到已收集的服务，并实例化它们，并将它们的实例传递给服务收集器类上的一个方法便于进行处理。

下面是 string_translation 服务的例子，它的 services.yml 定义如下:

```
# core.services.yml:

string_translation:
  class: Drupal\Core\StringTranslation\TranslationManager
  arguments: ['@language_manager']
  calls:
    - [initLanguageManager]
  tags:
    - { name: service_collector, tag: string_translator, call: addTranslator }
```

正如在这个例子中看到的，该服务定义了一个 tags，它的名称是 service_collector，它会告诉 Drupal 这是一个服务收集器。在 tags 定义中定义了

tag 属性，它会告诉服务收集器什么样的 tag 名称可以被收集。当使用服务 ID 作为 tag 名称时，这个可以省略。

tags 上的 call 属性告诉 Drupal 应该为每一项已收集的服务调用 TranslationManager::addTranslator()方法一次。下面是这个方法的代码:

```
// \Drupal\Core\StringTranslation\TranslationManager:
public function addTranslator(TranslatorInterface $translator, $priority = 0) {
  $this->translators[$priority][] = $translator;
  // Reset sorted translators property to trigger rebuild.
  $this->sortedTranslators = NULL;
  return $this;
}
```

收集方法接收了一个已收集的服务作为它的第一个参数。另外，如果收集服务包含参数 id 或 priority，真正完成收集服务的类会保证将这些属性传递到方法中。

这里有一个可以被 string_translation 服务收集的服务:

```
# core.services.yml:

string_translator.custom_strings:
  class: Drupal\Core\StringTranslation\Translator\CustomStrings
  arguments: ['@settings']
  tags:
    - { name: string_translator, priority: 30 }
```

这个服务被标签分组到 string_translator，这就是说要把它收集到 string_translation 服务中。因为 tags 有一个 priority 属性，它的值为 30。当服务被收集时，类 CustomStrings 的实例将会被传递给 TranslationManager::addTranslator()。

21.5 服务的文件结构

服务是在 example.services.yml 中定义的(这里假定模块名为'example')。当这个文件被放置到模块的根目录下时，它将会自动地被 Drupal 检测到。

下面是一个 example.services.yml 文件的例子:

```
services:

  # Defines a simple service of which requires no parameter for its constructor.
  example.simple:
```

www.drupalc.com

```
  class: Drupal\Example\Simple

# Defines a service which requires the module_handler for its constructor.
example.with_module_handler
  class: Drupal\Example\WithModuleHandler
  arguments: ['@module_handler']
```

在 core.services.yml 或者其它模块的服务文件中可以找到更多例子。所有的定义被聚合并作为一个序列化的数组保存在数据库中。

服务的属性

- **abstract**:这个服务定义将不会产生一项真正的服务。服务假定使用'parent'.Values:'true'=service 将会抽象；'false'(default)=service 将会实例化。
- **alias**:服务的别名
- **arguments**:参数用于工厂方法('factory_class')或者是类构造函数('class')。'@'指出其它服务，然后将服务名放在后面，并在它自已的 services.yml 文件中定义。
- **calls**:使用 setter 注入。定义其它方法来调用已经实例化的服务。
- **class**:服务的类。
- **configurator**:一个可调用的配置服务。
- **factory_class**:这个类将会实例化服务的类。
- **factory_method**:实例化服务类的方法。
- **file**:在服务加载以前包含的文件。
- **parent**:一个服务定义的属性能被继承。请看'abstract'。
- **properties**:属性注入。
- **public**:设置服务作为公共或私有。私有服务仅仅能被用作其它服务的参数。'true'(default):服务是公共的。'false':服务是私有的。
- **scope**:确定被服务容器使用的服务实例的范围。'container'(default):总是使用相同的服务。'prototype':每次调用服务时都创建一个新实例。'request':每个请求都创建一个新的实例。
- **synchronized**:在每个 scope 改变时服务将会重新配置(从 Symfony2.7 起开始废弃)。
- **synthetic**:注入到容器中的服务替换由容器创建的服务。Values:'true'=synthetic 服务;'false'(default)=normal 服务。
- **tags**:指出服务分组的名称。Tags 用在 compiler passes 中和收集缓存 bins。

www.drupalc.com

第 22 章 数据库API

Drupal 的数据库抽象层提供了一个统一的数据库查询 API，它能工作在不同的数据库上。数据库 API 是建立在 PDO 之上(PHP Data Objects)，并继承了它的语法封装了它的操作。

除了提供一个统一的数据库查询外，数据库抽象层也提供了结构化的方式以构造复杂查询，并且它采取了一些措施以提高访问数据库的安全。本章将讨论一下数据库抽象层以及常用的数据库操作。

22.1 数据库API概述

Drupal 的数据库 API 提供了访问数据库服务的标准的与具体数据库无关的抽象层。一般来说，你不要直接调用数据库 API，除非你用它来开发核心 API。

这个 API 在设计时尽可能地考虑了 SQL 语法和性能，并且还具有以下功能:

- 轻易支持多个数据库服务器。
- 允许开发者使用更复杂的功能，如事务处理。
- 提供一个结构化的结口用来动态构造查询。
- 强制安全检测和其它好的实践。
- 给模块提供一个简单的接口用以解析和修改站点查询。

主要的数据库 API 文档直接源于代码的注释。本章内容增强了这些 API 文档，并提供一些实际的例子来演示模块开发者要如何与数据库系统交互，以及从管理角度看数据库系统。

Drupal 8 的数据库系统是建立在面向对象的基础之上，系统为此定义了很多接口、类、traits 等，因此涉及到面向对象开发的相关概念。这里假定你已经熟悉了这些概念，然而最常见的操作还有过程化的前端，开发者可能选择面向过程的编程风格，他们认为那样做可读性更好。

注意，本章并没有讲到 Drupal 数据库 API 的所有功能。

22.2 数据库基本概念

Drupal 的数据库抽象层是建立在 PHP 的 PDO 库之上。PDO 提供了访问不同数据库的统一的面向对象的 API，但是它并没有抽象出通过不同的数据库而使用不同的 SQL 专业用语。

www.drupalc.com

驱动(Drivers)

因为不同的数据库有不同的特性，Drupal 的数据库抽象层需要为每种数据库类型提供了一个驱动器。一个驱动由许多文件组成，它们位于 core/lib/drupal/core/database/driver 目录下，驱动表现为一个字符串，它代表驱动的唯一键。在大多数情况下，驱动的键是小写的数据库名，如″mysql″、″pgsql″或者″mycustomdriver″。

每个驱动由几个类组成，这些类由核心的数据库中的类派生。这些特定于驱动的类可以覆写父类的任何行为以支持它们自身的数据库类型。在 Drupal 7 中特定于驱动类的命名总是先加上父类的名称和下划线，最后者是驱动名。如数据库链接类 DatabaseConnection，MySQL 的数据库链接类为 DatabaseConnection_mysql。Drupal 8 引入了命名空间概念，不需要再实现像 Drupal 7 那样的命名规则。

连接(Connections)

一个数据库连接是一个数据库 Connection 对象，它继承了 PDO 类。每一个数据库连接都与 Drupal 中的数所库链接相关。连接对象必须是数据库 Connection 类的子类如 MySQL 的连接。

为了访问连接对象你可以使用 Database 类。

```
$conn = Database::getConnection($target,$key);
```

与数据库目标与连接键相关的知识，请阅读数据库配置文档。

访问当前的活动连接可以使用

```
$conn = Database::getConnection();
```

注意，在大多数情况下，你不需要直接获取连接对象，可以使用数据库全局函数代替。只要在你要完成复杂的数据库处理时才直接访问连接对象，况且你也不想修改激活的数据库连接。

修改活动连接请使用:

```
db_set_active($key);
```

虽然数据库 API 也为我们提供了常用的全局函数，但是从 Drupal 8 开始将逐渐放弃那些全局函数，渐渐使用面向对象的方法来代替。

www.drupalc.com

查询(Queries)

一个查询是将一个 SQL 表达式发送给数据库连接。当前的数据库系统支持多种类型的查询:如静态查询、动态查询、插入、更新、删除、合并等等。有些查询直接写成 SQL 字符串模板(准备查询的表达式),而其它的查询使用面向对象的方法调用，即查询对象。

查询表达式(Statements)

一个表达式对象是一个 select 查询的结果。它总是 Statement 类型，或者是它的子类，Statement 类扩展至\PDOStatement 类并实现 StatementInterface 接口。

Drupal 为所有的查询使用预备的表达式。一个预备的表达式是一个查询模板，在表达式中使用占位符，在执行表达式时将会替换占位符的值。可以将它看成一个 SQL 函数，在调用它时传递参数。

在 PDO 中，显示地预备一个表达式对象，然后将特定值绑定到查询中的占位符上并执行。遍历表达式对象得到一个结果集。事实上一个表达式与一个结果集是同意的，但必须在表达式执行之后。

Drupal 不会直接暴露预备的表达式。相反，一个模块开发者将使用一个查询对象或者一个一次性的 SQL 字符串来执行一个查询并返回一个 statement 对象，statement 对象和 result 对象几乎是同意的。

www.drupalc.com

22.3 数据库配置

数据库配置主要是定义数据库连接，它是通过在 settings.php 文件中定义$databases 数组变量来实现的。正如它的名称一样，$databases 允许定义多个数据库连接。它也支持定义多个目标。一个数据库连接直到在它上运行第一个查询时才会打开(一开始并没有创建这一连接)。

连接键

连接键(connection key)对于一个给定的数据库连接是一个唯一的标识符。连接键对于一个给定的站点必须是唯一的，总是有一个"default"的连接，它是 Drupal 的主数据库连接。在大多数站点上，只定义了这一个连接。

目标(Target)

一个给定的连接键可以有一个或多个目标。一个目标是指一个可以使用的数据库。每个连接键总是可以定义"default"目标。如果请求的目标没有定义，系统将会智能地回退到"default"。

定义数据库目标主要是想使用主/副数据库服务器。"default"目标是主 SQL 服务器。可以定义一个或多个副服务器(注意，在有些情况，只有副服务器才是有效的替换目标，例如静态查询)。主服务器饱和后，将会尝试使用副 SQL 服务器。如果有可用的副 SQL 服务器，将会在上面打开连接并运行查询。如果没有可用的副 SQL 服务器，查询仍然可以运行在主 SQL 服务器上。这个回退过程是透明的，因此在写代码时可以考虑使用副服务器，如果它们不可用则自动回退而不需要对代码作任何修改。

$databases 变量语法

$databases 是数据库配置变量，它是一个嵌套数组，至少有三个级别。第一级定义数据库的键。第二级定义数据库的目标。每个键值对定义了数据库的连接信息。请看下面例子:

```
$databases['default']['default'] = array(
  'driver' => 'mysql',
  'database' => 'drupaldb',
  'username' => 'username',
  'password' => 'secret',
  'host' => 'localhost',
);
```

以上的$databases 数组定义了一个连接键("default")和目标("default")。这个连接使用了一个 MySQL 数据库，driver 键指出数据库的类型，database 键指出连接使用的数据库，username 键指出数据库用户名，password 键指出数据库密码，host 键指出数据库服务器地址。上面的例子是一个典型的只使用单个 SQL 服务器的例子，对于绝大多数站点这样的配置已经够用了。

对于主/副数据库配置，请看下面的定义:

```
$databases['default']['default'] = array(
  'driver' => 'mysql',
  'database' => 'drupaldb1',
  'username' => 'username',
  'password' => 'secret',
```

www.drupalc.com

```
  'host' => 'dbserver1',
);
$databases['default']['slave'][] = array(
  'driver' => 'mysql',
  'database' => 'drupaldb2',
  'username' => 'username',
  'password' => 'secret',
  'host' => 'dbserver2',
);
$databases['default']['slave'][] = array(
  'driver' => 'mysql',
  'database' => 'drupaldb3',
  'username' => 'username',
  'password' => 'secret',
  'host' => 'dbserver3',
);
```

上面的代码定义了一个"default"的数据库服务器和两个副数据库服务器。注意 slave 键是一个数组。如果一个目标被定义成一个连接信息数组，页面请求将会被随机发送到已定义的服务器之一。

```
$databases['default']['default'] = array(
  'driver' => 'mysql',
  'database' => 'drupaldb1',
  'username' => 'username',
  'password' => 'secret',
  'host' => 'dbserver1',
);
$databases['extra']['default'] = array(
  'driver' => 'sqlite',
  'database' => 'files/extradb.sqlite',
);
```

上面的配置定义了一个 Drupal 主数据库和名为"extra"的外部数据库，它使用 SQLite 数据库类型。注意 SQLite 的连接信息结构与 MySQL 不同。每一种数据库驱动的配置信息会依赖于它自身的特性。

需求 PDO

因为 Drupal 的数据库抽象层是建立在 PHP 的 PDO 类库之上，因此你的主机需要支持 PDO 扩展，才能运行 Drupal。

www.drupalc.com

PDO 选项

PDO 选项和基于特定数据库类型的 PDO 选项可以在$databases 变量中指定，这需要使用 pdo 键，它的值是一个数据库选项数组。请看下面的例子:

```
$databases['default']['default'] = array(
  'driver' => 'mysql',
  'database' => 'drupaldb',
  'username' => 'username',
  'password' => 'secret',
  'host' => 'dbserver1',
  'pdo' => array(ATTR_TIMEOUT => 2.0, MYSQL_ATTR_COMPRESS => 1),
);
```

22.4 静态查询

在 Drupal 中的大多数 SELECT 查询是静态查询，使用 db_query 函数完成。静态查询实际上是调用 Database 类执行查询，请看 db_query 函数的实现。

例子:

```
$query = db_query("SELECT nid, title FROM {node}");
$records = $query->fetchAll();
foreach ($records as $record) {
  // Do something.
}
```

只有简单的 SELECT 查询应该使用静态查询机制。你可以使用动态查询构造更加复杂的查询。

不要使用这个函数来执行 INSERT、UPDATE 或者 DELETE 查询。这些查询应该通过 db_insert()、db_update()和 db_delete 这些各自的函数来完成。

db_query()需要三个参数:

- **$query**:需运行的查询字符串，在字符串中使用占位符代替变量使用，表名使用大括号。
- **$args**:一个占位符数组用来替换查询中的占位符。
- **$options**:一个控制查询如何执行的选项数组。

前缀(Prefixing)

www.drupalc.com

在静态查询中，所有的表格必须使用大括号{}。这样做是为了让数据库系统能向表名字符串附加合适的前缀。前缀允许从相同的数据库运行多个站点，或者在有限的例子中在站点间共享指定的数据库表。

占位符

占位符是在查询字符串中使用一个由冒号和一个有意义的字符串组成，在查询执行时它会被替换成真正的值。这样做是为了防止 SQL 注入攻击。

```
$result = db_query("SELECT nid, title FROM {node} WHERE created
> :created", array(
  ':created' => REQUEST_TIME - 3600,
));
```

上面的代码将会从节点表中选择出过去一小时创建的所有节点。占位符:created 将会在运行查询时被动态地替换成 REQUEST_TIME – 3600。

一个查询中可以有很多占位符，但是所有的占位符必须有唯一的名称，哪怕它们的值一样。根据具体的使用情况，占位符可以嵌入代码中(如上例)，也可以定义一个占位符数组变量并传入，数组中占位符的顺序不用管。

以"db_"开始的占位符是为内部系统预留的，请不要显示地指定。

注意占位符不能转义也不能使用引号。因为它们被单独地传递给数据库服务器，服务器能区分查询字符串和占位符的值。

```
// 错误 (:type 占位符使用了引号)
$result = db_query("SELECT title FROM {node} WHERE type = ':type'", array(
  ':type' => 'page',
));

// 正确 (:type 占位符没有使用引号)
$result = db_query("SELECT title FROM {node} WHERE type = :type", array(
  ':type' => 'page',
));
```

占位符不能用于列名和表名。如果表名来自于不安全的输入，应该使用 db_escape_table()，这个函数会返回一个安全的表名。

占位符数组

Drupal 的数据库层包含了占位符的一个额外功能。如果传递给占位符的是一个数组，它将会被自动地展成以逗号分开的列表以响应占位符。这意味着开发者不必担心它们需要多少个占位符。

请看下面的例子以便于更好地理解:

Drupal 7:

```
db_query("SELECT * FROM {node} WHERE nid IN (:nids)", array(':nids' =>
array(13, 42, 144)));
```

Drupal 8:

```
db_query("SELECT * FROM {node} WHERE nid IN (:nids[])", array(':nids[]' =>
array(13, 42, 144)));
```

上面的查询表达式与下面的查询表达式是一致的:

```
db_query("SELECT * FROM {node} WHERE nid IN (:nids_1, :nids_2, :nids_3)",
array(
  ':nids_1' => 13,
  ':nids_2' => 42,
  ':nids_3' => 144,
));

db_query("SELECT * FROM {node} WHERE nid IN (13, 42, 144)");
```

www.drupalc.com

查询选项(Query options)

db_query()函数(数据库连接的查询方法)的第三个参数是一个选项数组，它指出查询将怎么执行。大多数的查询将使用两个值，其它值大多在内部使用。

target 键指出使用的数据库目标，如果没有指定，则使用 default。目前还可以使用的有效值是 slave，它指出查询将会运行在副 SQL 服务器上。

fetch 键指出如何从查询中获取查询结果。合法的值包括 PDO::FETCH_OBJ、PDO::FETCH_ASSOC、PDO::FETCH_NUM、PDO::FETCH_BOTH，或者是一个表示类名的字符串。如果指定类名字符串，每一条记录将会被获取到那个类的对象中。被 PDO 定义的其它值将会把获取的结果作为标准对象(stdClass object)、关联数组、索引数组、以及关联数组和索引数组。请阅读 http://php.net/manual/en/pdostatement.fetch.php 以获取更多的细节。缺省为 PDO::FETCH_OBJ。

下面的例子将会在一个副服务器上执行查询，并且使用关联数组从查询的结果集中返回记录。

```
$result = db_query("SELECT nid, title FROM {node}", array(), array(
  'target' => 'slave',
  'fetch' => PDO::FETCH_ASSOC,
));
```

调用 db_query()函数会返回一个结果对象，结果对象包含了返回的所有行和列。下面的例子中，$result 变量存储了查询返回的所有行，同时使用了它的 fetchAssoc()方法获取一行并存储到$row 变量中。

```
$sql = "SELECT name, quantity FROM goods WHERE vid = :vid";
$result = db_query($sql, array(':vid' => $vid));
if ($result) {
  while ($row = $result->fetchAssoc()) {
    // Do something with:
    // $row['name']
    // $row['quantity']
  }
}
```

22.5 使用自定义类获取查询结果

查询结果能返回到基于自定义类的对象中。例如，如果我们有一个名为 ExampleClass 的类，下面的查询将会返回 exampleClass 类型的对象。

```
$result = db_query("SELECT id, title FROM {example_table}", array(), array(
  'fetch' => 'ExampleClass',
));
```

如果这个类定义了一个__construct()方法，在类对象被创建时，属性将会添加到对象中，然后会调用类的构造函数。例如，如果你有下面的类和查询。

```
class ExampleClass {
  function __construct() {
    // Do something
  }
}

$result = db_query("SELECT id, title FROM {example_table}", array(), array(
  'fetch' => 'ExampleClass',
));
```

www.drupalc.com

上面的例子将会创建 ExampleClass 的对象，id 和 title 属性将会添加到这个对象中，然后才执行__construct()构造函数。这样的执行顺序是由于 5.2 以前的 PHP 版本中的 bug 造成的，PHP 5.2 及以后出现了常量 PDO::FETCH_PROPS_LATE，其作用是在设置属性前调用构造函数。

如果对象上有一个构造函数并且需要在属性添加到对象之前执行它的构造函数，请使用下面的代码。

```
$result = db_query("SELECT id, title FROM {example_table}");
foreach ($result->fetchAll(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE,
'ExampleClass') as $record) {
  // Do something
}
```

通过给 fetchAll 传递一些参数能获得相同的 fetch 效果。PDO::FETCH_CLASS 告诉 fetchAll 将返回的结果集作为 ExampleClass 类型对象的属性。PDO::FETCH_PROPS_LATE 告诉 fetchAll 向对象添加属性应放在调用对象的构造函数之后进行。

22.6 动态查询

前面我们讲了简单的 SQL 查询，事实上 SQL 的查询是既强大又复杂的，这一节我们介绍一下使用动态查询来构造更复杂的查询，主要包括条件子句、统计、去除重复值、分组查询、表连接以及排序等等。这些都有相应的 SQL 语法，这一节会介绍这些 SQL 语法如何转换为 Drupal 查询。

22.6.1 动态查询介绍

动态查询指的是由 Drupal 动态地创建查询而不是提供一个明确的查询字符串。所有插入(Insert)、更新(Update)、删除(Delete)、合并(Merge)等查询必须是动态的。选择(Select)查询可以是静态的也可以是动态的。因此"dynamic query"通常指的是一个动态的 Select 查询。

注意:90%的 select 查询都是静态查询。基于性能考虑，你应该尽量使用 db_query()而不是 db_select()。只有当查询中存在动态的部分时才使用动态查询(如添加依赖于上下文的 WHERE 条件)或者它们可以被修改(例如:node access)。

www.drupalc.com

从 Drupal 7 开始，已不支持 db_rewrite_sql 函数，必须使用动态查询来完成相同的事情。例如，任何时候查询节点表，你应该使用"node_access"，如下:

```
$query = db_select('node', 'n')
  ->addTag('node_access');
```

所有动态建立的查询都是使用一个查询对象来构造。与静态查询一样，大多数的函数能用于查询对象。查询的后续指令是通过调用查询对象上的方法完成的。

动态 select 查询使用 db_select()函数开始:

```
$query = db_select('users','u',$options);
```

在这种情况，users 是一个查询的基本表；它是 FROM 表达式后的第一个表。注意，它不需要使用大括号。查询建立器将会自动地处理。第二个参数是表的别名。如果没有指定，则它为正在使用的表的名称。$options 数组是可选的，它与静态查询中的$options 数组是一样的。

db_select()函数的返回值是一个 Select 类型的对象。因此，$query 变量的值是一个 Select 类型的对象。这个对象有很多有用的方法如 fields()、joins()、group()等等。这些方法能在这个变量上进一步调用。

动态 select 查询可能非常简单也可能非常复杂。下面我们会看到一些简单的查询，在后面我们会看到一些高级使用如表连接。

这里提供一个与用户表相关的简单查询，我们想创建一个与下面的静态查询相同功能的动态查询:

```
$result = db_query("SELECT uid, name, status, created, access FROM {users}
u WHERE uid <> 0 LIMIT 50 OFFSET 0");
```

它的动态查询如下:

```
// Create an object of type SelectQuery
$query = db_select('users', 'u');

// Add extra detail to this query object: a condition, fields and a range
$query->condition('u.uid', 0, '<>');
$query->fields('u', array('uid', 'name', 'status', 'created', 'access'));
$query->range(0, 50);
```

上面的代码实现了与静态查询相同的功能，首先是通过 db_select()函数创建一个查询对象$query，然后在该对象上依次调用 condition、fields、range 方法以指定查询条件、查询的字段和范围。

www.drupalc.com

上面的代码分别调用了对象的三个方法显得有些冗长，可以使用一种更简洁的调用写法，即在查询对象上连着写多个方法，这被称为链式写法。请看下面的代码:

```
// Create an object of type SelectQuery
$query = db_select('users', 'u');

// Add extra detail to this query object: a condition, fields and a range
$query->condition('u.uid', 0, '<>')
  ->fields('u', array('uid', 'name', 'status', 'created', 'access'))
  ->range(0, 50);
```

这样写感觉简洁多了。但其实还可以进一步简化，直接在 db_select()后直接链式调用它的方法。写法如下:

```
// Create an object of type SelectQuery and directly
// add extra detail to this query object: a condition, fields and a range
$query = db_select('users', 'u')
  ->condition('u.uid', 0, '<>')
  ->fields('u', array('uid', 'name', 'status', 'created', 'access'))
  ->range(0, 50);
```

执行一个查询

一旦建立了一个查询，可以调用 execute()方法编译并运行查询。

```
$result = $query->execute();
```

这个方法将会返回一个结果集/表达式对象，这与 db_query 返回的结果是一样的。查询结果可以通过遍历以获取行。

```
$result = $query->execute();
foreach ($result as $record) {
  // Do something with each $record
}
```

注意当使用下面的方法并带有一个多列的动态查询时应该要小心:

- fetchField()
- fetchAllKeyed()
- fetchCol()

这些方法现在需要数字列索引(0,1,2 等)而不是列别名。然后查询建立器并不会保证返回的字段符合指定的顺序，因此数据列的顺序可能并不是你期望的。特别

地表达式总是在字段以后添加，甚至你先将它们添加到查询中(这个问题不会出现在静态查询中，它返回的数据列总是你期望的顺序)。

调试

你可以在查询对象的生存期间检查 SQL 查询，这需打印查询对象。为检查它的参数，你可以打印它的 arguments()方法返回的结果。

```
echo $query;
print_r($query->__toString());
print_r($query->arguments());
```

22.6.2 条件子句

条件子句使用一定的条件来限制查询返回的结果。在 SQL 中，它是指 SELECT、UPDATE、DELETE 查询的 WHERE 或 HAVING 部份。在 Drupal 的所有动态查询中，这些都是使用相同的机制来实现。除特别说明，下面的一切都能应用到这三种查询类型。

- **Conditional fragment**:一个 conditional fragment 是条件子句自身的一部份。
- **Conjunction**:每一个条件子句由一个或多个 conditional fragments 连接在一起。即使用 AND 或者 OR 将两个或多个条件表达式连接在一起。
- **Conditional object**:Drupal 将每一个 conditional fragment 看作 Condition 类的实例。一个条件对象也是那个类的实例。

下面分析一个查询字符串:

Query:SELECT FROM {mytable} WHERE (a = 1 AND b= ‘foo’ OR (c=’bar’))

- **Conditional clause**: WHERE (a = 1 AND b= ‘foo’ OR (c = ‘bar’))
- **Conditional fragments**: (a = 1 AND b = ‘foo’ OR (c = ‘bar’)) ，(c = ‘bar’)
- **Conjunctions**: AND，OR

在 Drupal 7 中，Select、Update、Delete 等查询对象实现了 QueryConditionalInterface 接口，这个接口提供了处理查询条件的标准方法，在它们内部包含一个 QueryConditinal 对象。QueryConditional 类能直接实例化。

在 Drupal 8 中，查询条件有些复杂。QueryConditionalInterface 接口改为了 ConditionalInterface 接口，这个接口定义了处理查询条件的标准方法。其中

Select 实现了 SelectInterface 接口，这个接口又扩展了 ConditionalInterface 接口。Insert 和 Upsert 查询包含一个 SelectInterface 类型的对象，用它来处理查询条件。Update 和 Delete 查询使用了 QueryConditionTrait 来复用查询条件的方法。

一个条件表达式中的每个条件由一个 conjunction 来进行连接(主要是 AND、OR)。一个条件对象由一个或多个条件组成，它们通过特定的 conjunction 进行连接。默认，使用 AND 连接。条件对象支持嵌套，这样可以构造出任意复杂的条件表达式。

有两个应用到所有条件对象的主要方法:

$query->condition($field,$value = null,$operator = '=')

condition()方法允许向一个条件添加一个标准的$field $value $operator。条件使用二进制比较，操作符包括=、<、>=、LIKE 等等。如果没有指定操作符，则使用 = 。这意味着绝大多数情况将使用 condition('myfield',$value)，其结果是 myfield = :value，这里的:value 将在查询运行时被替换为$value。

$query->where($snippet,$args = array())

where()方法允许向 SQL 查询字符串添加条件子句。$snippet 包括需添加的 WHERE 子句，如果它包含变量，必须使用一个已命名的占位符。$args 是一个包含占位符和它的值的数组。在运行时会把占位符替换为具体的值，这会确保开发者使用的 snippet 是有效的 SQL。

在大多数情况下应优先使用 condition()方法，除非你的查询条件表达式过于复杂，无法使用 condition()方法来构造，如使用 like 表达式，或者多个字段等等。这两个方法返回的是 ConditionInterface 接口类型，因此可以直接在它上面进行链式调用。

condition()也处理几种其它特殊例子。

不像 Update 和 Delete 查询，Select 查询有两种类型的条件:WHERE 子句和 HAVING 子句。HAVING 子句的行为等同于 WHERE 子句，只不过它使用 havingCondition()和 having()代替了 condition()和 where()。

$query->havingCondition($field,$value = NULL,$operator = '=')
$query->having($snippet,$args = array())

数组操作符

有些操作符需要工作在数组上。比如 IN 和 BETWEEN。对于 IN 来说，$value 是一个包含许多值的数组，这些值可能是字段与之相等的值。因此，下面的调用将会以这种方式来求值。

```
$query->condition('myfield', array(1, 2, 3), 'IN');
// Becomes: myfield IN
(:db_placeholder_1, :db_placeholder_2, :db_placeholder_3)
```

如果操作符是 BETWEEN，$value 变量的值是一个带有两个元素的数组，字段值必须在这两个元素之间。例如:

```
$query->condition('myfield',rray(5,10),'BETWEEN');
// Becomes: myfield BETWEEN :db_placeholder_1 AND :db_placeholder_2
```

支持 NOT IN

```
$query->condition('myfield', array(1, 2, 3), 'NOT IN');
// myfield NOT IN (:db_placeholder_1, :db_placeholder_2, :db_placeholder_3)
```

在这里使用'<>'操作符将返回一个错误。

嵌套的条件

condition()方法的第一个参数可以接受另一个条件对象。内部的条件对象使用圆括号与外部的条件对象组合。内部的条件对象可以使用与外部的条件对象不相同的条件连接符。使用这种方式可以建立复杂的嵌套的条件结构。

辅助函数 db_condition()将会返回一个新的条件对象。它只有一个参数 $conjunction，它的值为 AND、OR 或者 XOR。通常，辅助函数 db_and()、db_or()和 db_xor()将会覆盖几乎所有的情况。这允许在查询内部插入条件。

举一个例子:

```
$query
  ->condition('field1', array(1, 2), 'IN')
  ->condition(db_or()->condition('field2', 5)->condition('field3', 6))
// Results in:
// (field1 IN (:db_placeholder_1, :db_placeholder_2) AND (field2
= :db_placeholder3 OR field3 = :db_placeholder_4))
```

NULL 值

要检测数据库字段的值是否为 NULL，请使用下面的方法:

```
$query->isNull('myfield');
// Results in (myfield IS NULL)

$query->isNotNull('myfield');
// Results in (myfield IS NOT NULL)
```

这两个方法也支持链式书写，可以与 condition()和 where()链在一起。

注意:虽然在 Drupal 8 中也可以使用 condition('field',NULL)检测 NULL，但这种用户已经被废弃，应该使用 isNull 或 isNotNull 代替。

子查询

condition()函数的参数$value 也可以是一个 subselect。为了使用一个 subselect，首先通过 db_select 构造一个 Select 对象。然后将它传给 condition()方法的$value 变量。当这个查询执行时子查询会自动集成到主查询中。

Subselect 一般用于两种情况:子查询的结果是单行并且操作符为=、< 、>、<=、>=；或者当子查询返回单列信息并且操作符是 IN。其它的连接符将会导致一个语法错误。

注意:当前只能使用带有 IN 操作符的子查询，因它带有其它操作的子查询不能被包含在圆括号中，从而会导致一个错误。
注意:在有些数据库中，特别是 MySQL，在条件子句中的 subselect 是非常慢的，应尽可能使用连结(join)代替。

例子:

下面的例子可以帮你更清晰地使用条件查询。为清晰起见，与之等价的查询字符串显示在条件查询的下方，并把占位符替换为假设的值。

```
db_delete('sessions')
  ->condition('timestamp', REQUEST_TIME - $lifetime, '<')
  ->execute();
// DELETE FROM {sessions} WHERE (timestamp < 1228713473)
db_update('sessions')
  ->fields(array(
    'sid' => session_id()
  ))
  ->condition('sid', $old_session_id)
  ->execute();
```

www.drupalc.com

```
// UPDATE {sessions} SET sid = 'abcde' WHERE (sid = 'fghij');
// From taxonomy_term_save():
$or = db_or()->condition('tid1', 5)->condition('tid2', 6);
db_delete('term_relation')->condition($or)->execute();
// DELETE FROM {term_relation} WHERE ((tid1 = 5 OR tid2 = 6))
class=MsoNormal>db_delete('term_relation')->condition($or)->execute();
// DELETE FROM {term_relation} WHERE ((tid1 = 5 OR tid2 = 6))
```

22.6.3 统计行数(COUNT QUERIES)和DISTINCT

任何一个查询都可以响应"count queries"。Count 查询返回原查询的行数。为了获得一个Count查询，请使用countQuery()方法。该方法返回一个SelectInterface类型的对象。

```
$count_query = $query->countQuery();
```

$count_query 现在是一个新的 Select 查询对象，执行这个查询只返回一个值那就是查询的行数统计。它除了 COUNT(*)外是不会添加任何字段和表达式。因为它返回的也是一个 SelectInterface 类型的对象，因此也可以使用下面的链式写法。

```
$num_rows = $query->countQuery()->execute()->fetchField();
```

请注意 countQuery()只不过构造一个查询表达式，具体还是要调用 execute()才能执行。

www.drupalc.com

排除重复值 DISTINCT

在些 SQL 查询可能需要处理重复的结果。在这种情况下，在静态查询中排除重复值可以使用"DISTINCT"这一SQL关键字来过滤。在动态查询中则使用distinct()方法。

```
// Force filtering of duplicate records in the result set.
$query->distinct()
```

注意:DISTINCT 可能引起性能问题，尽量不要使用。

22.6.4 表达式 Expressions

Select 查询建立器支持在选择的字段中使用表达式(注意这里的表达式是指 Expressions 是查询字符串的一部份与条件表达式 Statement 是有区别的)。在表达式中可以使用运算符与 SQL 函数，有些 SQL 函数是数据库特有的，并不是所有的数据库都支持，因此在使用表达式时需要考虑数据库兼容性。模块开发者应该保证使用的表达式兼容所有的数据库。

使用 SelectInterface 对象的 addExpression()方法来向查询字符串添加表达式(Expression)。例子:

```
$count_alias = $query->addExpression('COUNT(uid)', 'uid_count');
$count_alias = $query->addExpression('created - :offset', 'timestamp',
array(':offset' => 3600));
```

第一行代码添加了表达式 COUNT(uid)，这里使用了 SQL 函数 COUNT()，'uid_count'为表达式的别名。这个方法的第二个参数是为字段别名。在少数情况下，别名可能已经存在，这个方法将会产生一个新的别名。如果没有指定别名，则缺省产生"expression"(或者 expression_2，expression_3 等等)。

第三个参数是一个占位符和其值的关联数组，它用作表达式的一部份，它是可选的。

注意:有些 SQL 表达式可以没有功能除非添加了 GROUP BY 子句，GROUP BY 由 $query->groupBy()添加。开发者应确保生成的查询字符串的有效性。

www.drupalc.com

22.6.5 扩展Select查询

Select 查询支持扩展(Extenders)。扩展可以在 Select 查询运行时为它添加功能。这些功能可以是额外添加的方法或者是修改已经存在的方法的行为。

对于面向对象设计来说，扩展是对已有类的封装。它们通过提供一个灵活可修改的子类来动态地响应对象，扩展以有的功能。

使用一个 Extender

为了使用一个 Extender，你必须首先有一个查询对象。在查询对象上调用 extend()方法将会返回一个新对象用来替换原有查询对象。例子:

```
$query = $query->extend('PagerDefault');
```

上面的这行代码使用查询对象的 extend 创建一个新的查询对象 PagerDefault，它包含原有查询对象，并返回一个新对象。$query 变量不仅能使用原有对象的方法而且可以使用一些额外的方法。

请注意原有查询对象$query 并没有修改。新的对象是从 extend()返回，如果你没有保存它，它将会丢失。举一个例子，下面的代码将不会完成你希望的工作:

```
$query = db_select('node', 'n');
$query
  ->fields('n', array('nid', 'title'))
  ->extend('PagerDefault')    // This line returns a new PagerDefault object.
  ->limit(5);                 // This line works, because the PagerDefault object is
what is called.

// The return from extend() was never saved to a variable, so $query is still just
the Select object.
$query->orderBy('title');

// This line executes the Select object, not the extender.  The extender no
longer exists.
$result = $query->execute();
```

为了避免这一个问题，推荐首先为一个 extender 申明一个查询对象。就像下面这样:

```
$query = db_select('node', 'n')->extend('PagerDefault')->extend('TableSort');
$query->fields(...);
// ...
```

这样$query 变里存储了由 extend()返回的对象，确保它能正常工作。

也要注意扩展的可能被连着写(就像上面那样)，并不是所有的扩展都与其它扩展兼容并且与调用它们的顺序也有关系。例如，一个查询扩展了两个分页器并且表格排序行为必须首先扩展 PagerDefault。

创建一个 extenders

一个 extender 是一个实现 SelectInterface 接口的类，它的构造函数有两个参数:一个 select 查询和一个数据库连接对象。这个类必须重新实现 SelectInterface 接口的方法并传递给类对象的构造函数，并在适当的地方返回它自已。

在大多数情况下，这些工作已经由 SelectExtender 类完成了。因此一个 extender 只需扩展 SelectExtender 类。类的名称应该是在查询对象的 extend()中指定的。

www.drupalc.com

实现 extender 类，需要添加或者覆盖一些方法。没有覆盖的任何方法都会透明地传递给包含的对象。当覆盖一个方法时，该 extender 可以或者不可以调用下层的查询对象，但它必须返回一个 SelectInterface 接口的对象。在大多数情况下，应该是查询对象自身，或者是 extender 对象自身。

下面是一个例子:

```
class ExampleExtender extends SelectExtender {

  /**
   * Override the normal orderBy behavior.
   */
  public function orderBy($field, $direction = 'ASC') {
    return $this;
  }

  /**
   * Add a new method of our own.
   */
  public function orderByForReal($field, $direction = 'ASC') {
    $this->query->orderBy($field, $direction);
    return $this;
  }
}
```

在上面的例子中覆盖了查询对象的 orderBy()方法，但是没有做任何事只是简单地返回对象自身。它添加了一个 orderByForReal()方法，它实现了真正的排序行为。注意在这两个方法中，都是返回 extender 对象自身。这样确保通过返回查询对象时 extender 不会丢失。

任何模块都可以申明一个 extender。Drupal 8 核心实现了两个 extender，它们是 PagerSelect 和 TableSort。你可以阅读这些类的 API 文档以查看如何在你自己的代码中利用它们。

22.6.6 查询字段

添加一个字段

在 Drupal 中数据库表中的列使用字段来表示。可以向查询添加一个字段，这需使用 addField()方法:

```
$title_field = $query->addField('n','titile','my_title');
```

www.drupalc.com

以上代码将会构造一个查询用来从别名为'n'的表中选择'title'列，并给它一个别名'my_title'。如果没有指定别名，它将会自动产生一个别名。在大多数情况下，产生的别名就是字段名。在这个例子中，别名将是'title'。如果这个别名已经存在，别名将会是表名+字段名。在这个例子中，应该是'n_title'。如果这个别名也存在，将会在它后面添加一个计数，如'n_title_2'。

注意:如果你正在创建和处理你自己的查询，但不能指定一个别名也不能使用默认的别名。几乎可以肯定你的代码存在一个 bug。如果你实现了一个 hook_query_alter()，但是你确不知道哪些别名已存在并在使用，因此你总是使用产生的别名。

添加多个字段

如果要选择多个字段，可以调用 addField()多次以达到目的。注意大多数情况下字段的顺序应该是不重要的，但这样做模块的业务逻辑可能会存在缺陷。

你可以使用 fields()方法替代 addField()，fields 允许你一次性添加多个字段:

```
$query->fields('n', array('nid', 'title', 'created', 'uid'));
```

上面的 fields()调用相当于你调用 addFields()四次，每一次添加一个字段。但是，fields()不支持为字段指定别名。它返回一个查询对象自身，因此你可以在它上面进行链式调用而不需要返回任何产生的别名。如果你想知道生成的别名，可以使用 addField()或者 getFields()访问原生的内部字段结构。

调用不带字段列表的 fields()方法蒋会导致一个'SELECT *'查询。

```
$query->fields('n');
```

这个查询将会包含所有字段即'n.*'。注意这样不会生成任何别名。如果使用 SELECT *的一张表包含其它表中的字段，这有可能会引起结果集中的字段名称冲突。这时，结果集中只包含一个具有相同名称的字段。基于以上原因，不推荐使用'SELECT *'。

使用 fetchField 只返回一个字段

使用查询对象的 fetchField 方法只返回一个字段。请看下面的例子:

```
$query = db_select('node', 'n');
  $query->condition('n.nid', 123);
  $query->addField('n', 'title');
  $result = $query->execute();
```

```
  return $result->fetchField();
```

22.6.7 分组查询 Grouping

在 SQL 查询中常常遇到分组查询(group by 子句)，Drupal 的查询接口提供了 groupBy()方法实现这一查询功能。

```
$query->groupBy('uid');
```

上面的代码将会构造一个通过 uid 字段的分组查询。注意，这里的字段名应该是由 addField() 或者 addExpression()方法创建的别名，因此在多数情况下，你将会使用这些方法的返回值以确保使用正确的别名。

为了获取按照 uid 分组的行数统计，可以使用以下代码:

```
$query->addExpression('count(uid)','uid_node_count');
```

如果需按多个字段分组，多次调用 groupBy()即可。

HAVING 子句

可以在聚合的值上添加一个条件。

```
$query->having('COUNT(uid) >= :matches',array(':matches' => $limit));
```

这个例子将会找到 uid>=$limit 的行。注意 having 的第一个参数在发送到数据库以前是不会筛选的，因此用户提供的值应该通过第二个参数传入。

GROUP BY 和 HAVING 的例子

下面是使用这两个方法的一些例子:

```
$query = db_select('node', 'n')
  ->fields('n', array('uid'));
$query->addExpression('count(uid)', 'uid_node_count');
$query->groupBy("n.uid");
$query->execute();
```

以上代码相当于:

SELECT n.uid,count(n.uid) as uid_nod_count FROM node n GROUP BY n.uid。

www.drupalc.com

其作用是为每个用户统计其节点数量。

```
$query = db_select('node', 'n')
  ->fields('n',array('uid'));
$query->addExpression('count(uid)', 'uid_node_count');
$query->groupBy("n.uid");
$query->having('COUNT(uid) >= :matches', array(':matches' => 2));
$results = $query->execute();
```

以上代码相当于:

```
SELECT n.uid,count(uid) as uid_node_count FROM node n GROUP BY n.uid
HAVING COUNT(uid) >=2。
```

其作用是为每个用户统计节点数，但节点数至少应大于等于 2。

www.drupalc.com

22.6.8 数据库连接操作

Drupal 8 的 Select 类实现了数据库表的各种连接方式，包括 join()、innerJoin()、leftJoin()和 rightJoin()等方法，这些方法与 SQL 连接相对应。它们的参数看起来差不多，这里以 join 为例简单地介绍一下。

join($table, $alias = NULL, $condition = NULL, $arguments = array()) : \Drupal\Core\Database\Query\The

参数

$table:需连接的表。可以是字符串表名或者是另一个 SelectQuery 对象。如果传递一个对象，它被用作一个子查询。除非表名以数据库/schema 名开始，否则"."被当作表前缀。

$alias:表的别名。在大多数情况下，它是表名的第一个字母，或者表名中每个单词的第一个字母。

$condition:连接表的条件。如果需要使用一个值，在查询中应该使用一个占位符并将它的值传递给第四个参数。连接中的第一个表是基本表，它会忽略这个值。在这个字符串能使用 token%别名，它们会被直正的别名替换。这对于通过数据库来修改别名是很有用的，例如连接同一个表时。

$arguments:用来替换连接条件中占位符的数组。

返回值:这个方法返回一个唯一的表别名。

下面是一个例子:

```
$query = db_select('node', 'n');
$table_alias = $query->join('users', 'u', 'n.uid = u.uid AND u.uid = :uid',
array(':uid' => 5));
```

这相当于:

```
SELECT * FROM node n JOIN users u ON n.uid=u.uid AND u.uid=5;
```

上面的代码使用 INNER JOIN(默认的连接类型)连接 user 表，其表别名为 u。连接的条件是'n.uid=u.uid AND u.uid = :uid'，:uid 是一个占位符，其真实值为 5。注意使用预备的表达式有利于数据库安全。请不要直接在查询中使用变量，它可能会引发 SQL 注入攻击。其它的连接操作与此类似，这里就不再赘述。

它的返回值是一个表别名。如果指定了一个别名它将会被使用，只有一些极少数情况别名已经被其它表使用。在这种情况，系统将会给它赋予一个不同的别名。

join()方法的第一个参数可以接受一个 SelectQuery 对象，如下面的例子:

```
$query = db_select('node', 'n');

$myselect = db_select('mytable')
  ->fields('mytable')
  ->condition('myfield', 'myvalue');
$alias = $query->join($myselect, 'myalias', 'n.nid = myalias.nid');
```

因为 join()的返回方法是一个表别名，并不是 SelectQuery 对象，因此不能链式书写，只能对它们进行单独调用。如果你将多个函数链在一起，可以像下面这样:

```
$query = db_select('node', 'n');
$query->join('field_data_body', 'b', 'n.nid = b.entity_id');
$query
  ->fields('n', array('nid', 'title'))
  ->condition('n.type', 'page')
  ->condition('n.status', '1')
  ->orderBy('n.created', 'DESC')
  ->addTag('node_access');
```

22.6.9 查询结果排序

为了动态地添加一个 order by 子句，请使用可链式书写的 orderBy()方法:

www.drupalc.com

```
$query->orderBy('title', 'DESC')
  ->orderBy('node.created', 'ASC');
```

上面的代码将会构造一个查询，它对 title 字段以降序排列，对 node.created 进行升序排列。第二个参数可以是'ASC'或者'DESC'，ASC 代表升序，DESC 代表降序，默认为'ASC'。这里的字段名可以是通过 addField()或者 addExpression()方法创建的别名，因为你可能想使用这些方法返回的别名以防止别名冲突。如果要给多个字段排序，只需简单地调用 orderBy()多次即可。

随机排序

查询的随机排序对于不同的数据库，其语法有轻微的差异。因此，最好使用动态查询来处理。 为了指定一个随机排序，需调用

orderRandom()方法。

```
$query->orderRandom();
```

orderRandom()方法可以链式书写，并且可以与 orderBy()一起用。像下面这样做是安全的:

```
$query->orderBy('term')->orderRandom()->execute();
```

上面的代码优先对 term 字段进行排序，然后对相同的 term 进行随机排序。

www.drupalc.com

22.6.10 查询修改 tagging

动态选择查询提供了一项重要功能，就是模块可以修改已有的查询。允许模块向查询注入新的约束条件，这可以修改已有模块的行为或者在查询运行时加入约束，例如节点访问(node access)约束。查询修改涉及到 tagging,meta data,hook_query_alter()三个方面。

只有那些被 tagged 的查询才能被修改。在 Drupal 中的大多数查询并没有 tagged，因此它们不能使用这种方式修改。

Tagging

任何动态选择查询可能使用一个或多个字符串 tag。这些 tag 指出了查询的类型，它允许 alter hooks 决定它们是否需要执行其它任务。Tags 字符串必须小写，其

它规则与 PHP 变量相同(由字母数字和下划线组成，必须以字母开头)。向一个查询添加 tag，可以使用查询对象的 addTag()方法:

```
$query->addTag('node_access');
```

检测一个查询对象是否带有一个特定的 tag，可以使用以下三个方法:

```
// TRUE if this query object has this tag.
//如果查询对象有'example' tag，则返回 TRUE
$query->hasTag('example');

// TRUE if this query object has every single one of the specified tags.
//如果查询对象有'example1'和'example2' tag，则返回 TRUE
$query->hasAllTags('example1', 'example2');

// TRUE if this query object has at least one of the specified tags.
//如果查询对象有'example1'和'example2' tag 之一，则返回 TRUE
$query->hasAnyTag('example1', 'example2');
```

hasAllTags()和 hasAnyTag()可以使用任意数量的 tags 作为它的参数，参数的顺序无关紧要。

并没有限定使用什么样的 tags，但有一些常用的标准 tags。下面列出一部份标准 tags:

- **node_access**:应该在查询上放置节点访问限制；所有返回一个向用户显示的节点列表的查询都应该有这个 tag。注意当节点模块修改带有这个 tag 的查询时，它不会检测节点的发布状态(已发布/未发布)，因此你的查询应该进行这一检测，以保证未发布的内容不要向普通用户(没有查看未发布内容权限的用户)显示。
- **entity_field_access**:应该在查询上放置实体字段访问限制。
- **translatable**:这个查询应该包含可翻译的列。
- **term_access**:应该在查询上放置基于分类术语的限制；所有返回一个向用户显示的分类术语列表都应该有这个 tag。
- **views**:这个查询应由 views 模块产生。
- **view_<view_name>**:创建查询的视图名称。

Meta data

查询可以含有 meta data 以向 alter hooks 提供额外的上下文信息。Meta data 可以是任意的 PHP 变量，它的键是一个字符串。

```
$node = node_load($nid);
```

www.drupalc.com

```
// ... Create a $query object here.
$query->addMetaData('node', $node);
```

Meta data 本身并没有什么实际的意义，它不会影响到查询。它的存在只是为了向 alter hooks 提供额外的信息，一般来说，它只用于当查询带有特定 tags 时。

可以使用查询对象的 getMetaData()方法来访问查询的 meta data。

```
$node = $query->getMetaData('node');
```

如果没有给 meta data 赋给那个键，这个方法将会返回 NULL。

hook_query_alter()

tagging 和 meta data 本身都不能做什么。它们的存在只是为了向 hook_query_alter 提供额外信息，这个 hook 可以修改一个选择查询。

所有的动态选择查询在执行前都能通过 hook_query_alter()来修改。这就给了模块修改已有查询的一个机会。hook_query_alter()接受一个参数:即选择查询对象自身。

```
/**
 * Implementation of hook_query_alter().
 */
function example_query_alter(AlterableInterface $query) {
  // ...
}
```

还有一个针对特定 TAG 的 alter hook 即 hook_query_TAG_NAME_alter()，在常规调用完成后，会为给定的选择查询对象调用每一个 tag 相关的 alter。下面是一个带有'sort_by_weight' tag 查询调用的例子:

```
/**
 * implements hook query alter to allow ordering by weight
 * @param QueryAlterableInterface $query
 */
function MYMODULE_query_sort_by_weight_alter(QueryAlterableInterface
$query) {
  $query->join('weight_weights', 'w', 'node.nid = w.entity_id');
  $query->fields('w', array('weight'));
  $query->orderBy('w.weight', 'ASC');
}
```

www.drupalk.com

使用 hook_query_alter()时有两个重要的方面需要注意。

1.它的参数$query 不以引用传递。因为$query 是一个对象，对象在 PHP 中处理时不会以任何方式复制，因此不需要显示地引用传递。这个 alter hook 没有返回值。

2.参数的类型显示地指定为 AlterableInterface。虽然没有严格要求指定类型，但这样做有助于正确地传递参数，以防上传递错误类型的参数发生错误。参数类型指定为 AlterableInterface 而不是 Select 这是为了提供向后兼容。

alter hook 可以在原有查询对象上执行任何查询，除非执行查询会导至无限循环。alter hook 可以使用查询带有的 tags 和 meta data 以决定要执行什么任务。模块开发者可以调用查询对象的其它方法以添加字段、进行连接、添加条件等到查询中，或者可以请求访问查询对象的内部数据结构以直接处理它们。应优先使用前者来向查询添加新信息，而后者允许 alter hook 删除查询中的信息以处理已经在队列中的指令。

```
$fields =& $query->getFields();
$expressions =& $query->getExpressions();
$tables =& $query->getTables();
$order =& $query->getOrderBy();
$where =& $query->conditions();
$having =& $query->havingConditions();
```

请注意以上方法以引用返回，以便在 alter hook 中访问与对象相同的数据结构。上面的所有方法都返回一个数组，数组的常用结构已经文档化在数据库 select.inc 中。

举一下例子，从查询中删除排序:

```
$order =& $query->getOrderBy();
unset($order['n.nid']);
```

上面的代码适用于 Drupal 7，由于 Drupal 8 重写了数据库引擎，以上方法不需要以引用返回，因为它们返回的是 SelectInterface 接口的对象。

22.6.11 范围查询

有些查询可能只想返回一部份结果。通常这叫做**范围查询**。在 MySQL 中，这是通过使用 LIMIT 子句实现的。在 Drupal 中依然可以使用静态查询，在查询中使用 LIMIT 子句。但出于某些原因，我们需要使用动态查询来限定查询结果的范围，

www.drupalc.com

可以使用查询对象的 range()方法。这个方法有两个参数:第一个参数指出首行偏移；第二个参数指出返回的行数。

在大多数情况下，我们想从开始返回 n 条记录。只需将第一个参数设为 0，第二个参数设为 n 就行了。如下面的代码:

```
// Limit the result to 10 records
// where 0 is offset and 10 is limit
$query->range(0, 10);
```

下面的代码构造了一个范围查询，它从第 6 条记录开始返回随后的 10 条记录。

```
$query->range(5, 10);
```

后调用的 range()方法将会覆盖先调用的 range()方法。调用它时如果不带参数将会删除查询的所有范围限制。

22.6.12 表格排序

Drupal 8 的核心提供了一个对查询结果进行表格排序(table sorting)的功能，通过这个功能，你可以实现单击表头中的某列实现对表格数据按升序或降序排列。Drupal 8 实现了一个 TableSortExtender 类来处理表格排序。要实现表格排序，需添加一个表头。值得注意的是一个 extender 不会返回查询对象，而是返回一个 extender 对象。

```
$query = $query
  ->extend('TableSort')
  ->orderByHeader($header);
```

以上代码实现查询结果按表格排序，$header 参数是表头的列数组。

22.7 结果集

选择查询返回一个结果集合，它包含 0 条或多条记录。从结果集中获取数据有多种方式，这根据于具体的情况。

大多数情况下，可以使用 foreach()来遍历集果集。如下例子:

```
$result = db_query("SELECT nid, title FROM {node}");
foreach ($result as $record) {
```

www.drupalc.com

```
  // Do something with each $record
  $node = node_load($record->nid);
}
```

根据具体的使用情况，可以使用其它方式获取记录。

如:

```
$record = $result->fetch();            // 缺省模式
$record = $result->fetchObject();  // 返回一个记录对象
$record = $result->fetchAssoc();   // 返回一个关联数组
```

如果结果集中没有记录，则会返回 FALSE。应尽量避免使用 fetch()而使用 fetchObject()和 fetchAsscoc，因为后者带有自身描述更易于理解。如果你使用其它的 PDO 支持的 fetch 模式，请使用 fecth()。

从结果集中获取某列请使用:

```
$record = $result->fetchField($column_index);
```

列索引$column_index 的默认值为 0，表示第一列。

统计 DELETE、INSERT 或者 UPDATE 的受影响行数，请使用:

```
$number_of_rows = $result->rowCount();
```

统计选择查询返回的行数请使用:

```
$number_of_rows =
db_select('node')->countQuery()->execute()->fetchField();
```

获取所有记录到数组中，请使用:

```
// Retrieve all records into an indexed array of stdClass objects.
//获取所有数据到索引数组。
$result->fetchAll();

// Retrieve all records into an associative array keyed by the field in the result specified.
//获取所有数据到关联数组，其键为字段名。
$result->fetchAllAssoc($field);

// Retrieve a 2-column result set as an associative array of field 0 => field 1.
//获取一个 2 列的结果集的关联数组。
$result->fetchAllKeyed();
// You can also specify which two fields to use by specifying the column numbers for each field
$result->fetchAllKeyed(0,2); // would be field 0 => field 2
$result->fetchAllKeyed(1,0); // would be field 1 => field 0
// If you need an array where keys and values contain the same field (e.g. for creating a 'checkboxes' form element), the following is a perfectly valid method:
```

www.drupalc.com

```
$result->fetchAllKeyed(0,0); // would be field 0 => field 0, e.g. [article] =>
[article]

// Retrieve a 1-column result set as one single array.
$result->fetchCol();
// Column number can be specified otherwise defaults to first column
$result->fetchCol($column_index);
```

fetchAll()和 fetchAllAssoc 使用的是默认的 fetch 模式(索引数组、关联数组或者对象)。这是可能通过设置 fetch 模式常量进行修改。对于 fetchAll()方法 fetch 模式对应第一个参数，对于 fetchAllAssoc()方法它对应第二个参数。例子:

```
// Get an array of arrays keyed on the field 'id'.
$result->fetchAllAssoc('id', PDO::FETCH_ASSOC);
// Get an array of arrays with both numeric and associative keys.
$result->fetchAll(PDO::FETCH_BOTH);
```

因为 PHP 在返回对象上支持链式方法调用，所以经常会跳过$result 变量。请看:

```
// Get an associative array of nids to titles.
$nodes = db_query("SELECT nid, title FROM {node}")->fetchAllKeyed();

// Get a single record out of the database.
$node = db_query("SELECT * FROM {node} WHERE nid = :nid", array(':nid'
=> $nid))->fetchObject();

// Get a single value out of the database.
$title = db_query("SELECT title FROM {node} WHERE nid = :nid", array(':nid'
=> $nid))->fetchField();
```

如果你想使用一个像 array(1,2,3,4,5)这样的简单数组，并且想设置成这样 array(1=>1,2=>2,3=>3,4=>4,5=>5)。你可以这样使用:

```
$nids = db_query("SELECT nid FROM {node}")->fetchAllKeyed(0,0);
```

22.8 插入查询 INSERT

在 Drupal 中 INSERT 查询操作必须使用一个查询构建对象。不同的数据库对 INSERT 的操作略有不同，特别是处理 LOB(大对象如 MySQL 的 TEXT 型)和 BLOB(二进制大对象)字段，因此需要数据库抽象层允许特定类型的数据库驱动来实现这些特定于数据库的处理。

INSERT 查询操作使用 db_insert()函数开始，如下:

```
$query = db_insert('node', $options);
```

上面的代码创建一个 INSERT 查询对象，它将会向节点表插入一条或多条记录。注意表名不需要使用大括号，因为查询构建器会自动地处理它。

INSERT 查询对象使用了流式 API，除了 execute()方法外，其它方法返回查询对象自身，并允许链式调用方法。在大多数情况下，这意味着查询对象根本不需要保存为中间变量。

INSERT 查询对象支持多种不同的使用方式以满足不同的需求。通常，需要指定待插入的字段以及它的值，然后执行插入操作。下面是一些推荐的用法。

紧凑形式

大多数 INSERT 查询操作使用紧凑形式:

```
$nid = db_insert('node')
  ->fields(array(
    'title' => 'Example',
    'uid' => 1,
    'created' => REQUEST_TIME,
  ))
  ->execute();
```

上面的查询代码与以下 SQL 语句相同:

```
INSERT INTO {node} (title, uid, created) VALUES ('Example', 1, 1221717405);
```

上面的代码将插入操作的各个部份链接在一起。

```
db_insert('node')
```

这一行为节点表创建一个新的 INSERT 查询对象

```
->fields(array(
    'title' => 'Example',
    'uid' => 1,
    'created' => REQUEST_TIME,
  ))
```

fields 方法接受多种形式的参数，但是最常用的是关联数组。数组的键对于表中的列，键值对于到需插入的列值。其结果是在指定表上执行单个插入操作。

```
->execute();
```

www.drupalc.com

execute()方法告诉 Drupal 运行这个查询。只有调用了这个方法，查询才会真正执行。

与 INSERT 查询对象上的其它方法返回查询对象自身不同，execute()返回一个自动增加的字段值(hook_schema()中的 serial 类型值)。在上面的例子中其返回值赋给了$nid。如果没有使用自动增加字段，execute()的返回值是没有定义的并且不应该相信。

一般来说，这种形式的 INSERT 查询是最常用的

冗余形式

```
$nid = db_insert('node')
  ->fields(array('title', 'uid', 'created'))
  ->values(array(
    'title' => 'Example',
    'uid' => 1,
    'created' => REQUEST_TIME,
  ))
  ->execute();
```

这种形式与前面的形式相比，虽效果一样，但显得更冗长。

```
->fields(array('title', 'uid', 'created'))
```

在调用 fields()时其参数使用一个索引数组而不是关联数组，它只是设置是 INSERT 查询用到的字段(数据表的列)并没有指定字段值。这主要用于运行一次性插入多条记录(后面讲)。

```
->values(array(
    'title' => 'Example',
    'uid' => 1,
    'created' => REQUEST_TIME,
  ))
```

调用这个方法指定了一个带有字段名和字段值的关联数组。这个 value()方法可以接受一个索引数组作为参数，如果使用索引数组，需注意数组元素的顺序必须与字段顺序相对应。如果使用关联数组，顺序可以随意。通常关联数组具有更高的可读性。

这种查询形式用得较少，因为紧凑形式更简洁。大多数情况下，只有运行 multi-insert 查询操作时使用。

www.drupalc.com

多行插入形式

INSERT 查询对象也可以一次性插入多条记录。只须多次调用 values()方法将它们加入到查询表达式队列并将它们组合在一起。不过这依赖于具体的数据库能力。对于大多数数据库，将会一次性插入多条记录以提高数据插入速度。在 MySQL 中，它将会使用 MySQL 多值插入语法。

```
$values = array(
  array(
    'title' => 'Example',
    'uid' => 1,
    'created' => REQUEST_TIME,
  ),
  array(
    'title' => 'Example 2',
    'uid' => 1,
    'created' => REQUEST_TIME,
  ),
  array(
    'title' => 'Example 3',
    'uid' => 2,
    'created' => REQUEST_TIME,
  ),
);
$query = db_insert('node')->fields(array('title', 'uid', 'created'));
foreach ($values as $record) {
  $query->values($record);
}
$query->execute();
```

上面的代码将会把插入的三条记录组合在一起执行，为了更有效地执行，这需要使用相应的数据库驱动的插入方法。注意上面的代码我们将查询对象保存在变量中以使我们通过循环调用 values()构造多行查询。

上面的代码与以下三个查询相等:

```
INSERT INTO {node} (title, uid, created) VALUES ('Example', 1, 1221717405);
INSERT INTO {node} (title, uid, created) VALUES ('Example2', 1,
1221717405);
INSERT INTO {node} (title, uid, created) VALUES ('Example3', 2,
1221717405);
```

执行多行查询的 execute()方法的返回值没有定义并且是不可信的，因为它随着数据库的不同而不同。

www.drupalc.com

基于 SELECT 查询的 INSERT

如果你想处理的一个表带有其它表数据，你可能需要在源表上执行 SELECT 查询，然后在 PHP 中遍历数据并将其插入新表，或者使用 INSERT INTO...SELECT 语句将 SELECT 返回的每行作为 INSERT 查询的数据源。

举一个例子，我们想建立一个"mytable"表，它包含节点 id(nid)和用户名(name)，这个表的数据来源于节点表和用户表，并且针对于特定的页面类型。

```
<?php
// Build the SELECT query.
$query = db_select('node', 'n');
// Join to the users table.
$query->join('users', 'u', 'n.uid = u.uid');
// Add the fields we want.
$query->addField('n','nid');
$query->addField('u','name');
// Add a condition to only get page nodes.
$query->condition('type', 'page');

// Perform the insert.
db_insert('mytable')
  ->from($query)
  ->execute();
?>
```

www.drupalc.com

缺省值

在正常情况下，如果你没有为给定的字段指定一个值，由 schema 定义的数据库表将智能地为其应用一个缺省值。然而有时，你需要明确地通知数据库应用一个默认值。比如你想为整个记录应用默认值。要明确地指定默认值，可以使用 useDefaults()方法。

```
$query->useDefaults(array('field1', 'field2'));
```

这一行通知查询使用数据库为 field1 和 field2 定义的默认值。注意在 useDefaults()和 fields()或 values()指定相同字段会导致一个错误，并抛出一个异常。

22.9 更新查询

在 Drupal 中更新查询必须使用一个查询建立对象。不同的数据库对大对象 LOB(如 MySQL 中的 TEXT)或者 BLOB(二进制大对象)的处理有所不同，因此数据库抽象层应该将这些交由相应的数据库驱动类处理。

更新查询使用 up_date()方法:

```
$query = db_update('node',$options);
```

上面的代码创建了一个 update 查询对象，将会更新节点表中的一条或多条记录。注意这里的表名不需要使用大括号，查询建立器将会自动处理。

UPDATE 查询对象使用流式 API。除了 execute()方法外，其它方法都返回查询对象自身，以便于链式调用查询方法。这意味着查询对象根本不需要保存为中间变量。

下面有一些 UPDATE 查询的用法:

```
/* This is a horrible example as node.status is pulled from node_revision.status table as well, updating it here will do nothing. */
$num_updated = db_update('node')
  ->fields(array(
    'uid' => 5,
    'status' => 1,
  ))
  ->condition('created', REQUEST_TIME - 3600, '>=')
  ->execute();
```

上面的查询代码将会更新节点表中过去 1 小时内创建的所有记录，并将它们的 uid 设为 5，status 设为 1。fields()方法接受一个指定需更新的字段和它的值的关联数组。这与 INSERT 查询不同，Update::fields()只能接受一个关联数组。因此字段的段序元关紧要。

上面的例子与下面的查询等价:

```
UPDATE {node} SET uid=5, status=1 WHERE created >= 1221717405;
```

execute()方法将会返回查询所影响的行数。注意受影响行数与匹配的行数是不同的。在上例，uid 为 5 status 为 1 的记录也会匹配到，但查询不会修改它们，因此它们是不受影响的。

```
<?php
$query = db_update('mytable');
// Conditions etc.
```


www.drupalc.com

```
$affected_rows = $query->execute();
?>
```

应用 where 条件:

```
$query = db_update('block')
  ->condition('module', 'my_module')
  ->where(
    'SUBSTR(delta, 1, 14) <> :module_key',
    array('module_key' => 'my_module-key_')
  )
  ->expression('delta', "REPLACE(delta, 'my_module-other_',
'my_module-thing_')")
  ->execute();
```

在条件中使用字符串函数

```
$query = db_update('block')
  ->condition('module', 'my_module')
  ->condition('SUBSTR(delta, 1, 14)', 'my_module-key_', '<>') // causes error.
  ->expression('delta', "REPLACE(delta, 'my_module-other_',
'my_module-thing_')")
  ->execute();
```

上面的代码使用了 condition()方法添加条件，根据 condition()方法的参数解释，它的第一个参数是字段名，在字段名中要使用操作符或函数请使用 where()。所以上面的代码 causes error。

22.10 删除查询

DELETE 查询也需要使用一个查询建对象。使用 db_delete()函数进行 DELETE 操作。

```
$query = db_delete('node', $options);
```

上面的代码创建一个 DELETE 查询对象，其功能是从节点表中删除记录。同样地，表名不需要使用大括号，查询建立器会自动处理。

DELETE 查询对象也使用流式 API，除 execute()方法外，其它方法返回查询对象自身，因此可以在它上面链式调用方法。这样不需要保留查询对象作为中间变量。

www.drupalc.com

DELETE 查询是非常简单的，它仅仅由 WHERE 子句组成。WHERE 子句的结构同 Conditional 子句。

下面是一个完整的 DELETE 查询:

```
$num_deleted = db_delete('node')
  ->condition('nid', 5)
  ->execute();
```

上面的查询将会从节点表中删除 nid 为 5 的行。与下面的查询等价:

```
DELETE FROM {node} WHERE nid=5;
```

execute()方法将会返回受影响的行数。

www.drupalc.com