

16 | 扩展与定制：如何实现插件系统并形成生态圈？

2022-04-18 陈旭

《说透低代码》

[课程介绍 >](#)



讲述：陈旭

时长 22:14 大小 20.36M



你好，我是陈旭。

我们已经在专栏中多次提到插件这个词了，那么插件到底怎么来实现呢？今天我们就来系统地梳理一下。

在第 9 讲中，我们解决了低代码编辑器的属性编辑器与 Web 组件的紧耦合问题，而且在第 12 讲的获取组件个性化数据的方法中，我们也采用了类似的思路，实现了应用定制化的动作与低代码平台松耦合的效果。核心功能与扩展功能的松耦合架构，是实现插件的关键基础。所以，我们可以将这两讲采用的方法进行归纳和抽象，形成一个允许应用团队在更大范围内定制和扩展的能力，我将这个能力称为插件系统。这就是我们今天这讲要解决的任务。

特别是第 9 讲中，我们细化到代码层面，进行一步步地设计和解耦，最终用一套代码架构同时支撑低代码平台内部实现和外部扩展。你可以复习一下这一部分，能帮助你更好地理解今天的内容

在我看来，对于一个通用型的低代码平台来说，插件系统是一个非常重要的功能，它能够解决通用型低代码平台的许多弊端。我们一步步来分析，先看看通用型低代码平台都有哪些弊端。

通用型低代码平台的弊端有哪些？

我们之前已经说过很多发展通用型平台的好处了，不过凡事都有代价。如果站在业务开发（即平台的用户）的角度来看，通用带来的问题主要包括这几个方面：

- 使用门槛居高不下；
- 效率无法最大化，高不成低不就；
- 平台过于“高冷”和“挑剔”；
- 容易与业务需求耦合，逐渐腐化；
- 平台容易积压需求，造成不满。

我们简单分析一下这几个问题。首先要明确的是，效率和赋能（降低门槛）是建设低代码平台的两大目标。如果我们的低代码平台达不到这个目标，那它必然不能算作是一个成功的低代码平台。即使我们把效率和赋能中的其中一项做到极致，也很难说已经获得了成功。

不过，从通用到具体场景，我们还需要做很多的额外工作，而且这些定制化工作的难度和工作量可能还不低，或者是开发过程比较麻烦。总之，刚刚接触低代码平台的用户和低技能者是很难驾驭这个过程的。

当然，我们可以通过内置模板的方式解决配置方面的问题，但是深度定制部分的工作，往往通过简单的配置是无法解决的，所以模板也不行。那这方面要怎么处理呢？

深度定制这部分的工作和应用开发的效率紧密相关，实施难度很大。我们回顾下第 8 讲的内容，当时我们在讲解应用开发三部曲的布局篇时说到，网格布局器是一个通用的布局器，能用来开发表单类 App，也可以用来开发 Dashboard 类 App 和其他多数分析类的 App。但是，网格布局在各个场景下的效率表现有很大的不同，它在精细化 UI 的布局中的效率更高，但是在做表单布局时的效率却会比较低。

从这个例子中，我们可以看到：**为特定场景提供定制化的能力，是提升效率的一个非常有效且直接的方法。**

所以说，通用型的平台实现的任何需求，都需要充分考虑各种各样应用场景下的情况。在应用团队看来我们就非常高冷了，应用团队把一个小需求、小修改提给平台，往往都需要漫长的等待和评估，而且，许多在应用团队看来理所当然的需求，却会被平台团队给拒绝掉。

当然，只有在低代码平台团队非常强势的时候，它才能做到如此高冷和挑剔。但当低代码平台没有这样的资本的时候，事情则会朝另一个方向演进：与业务逐渐耦合，最终失去通用性。

如果扩展性不够，但又不够强势时，迫于压力，我们难免会使用 **if else** 来解决，**if** 只能爽一时，但一个 **if** 一个坑，用不了多久，要么就 **if** 不下去，要么就由于 **if** 的情况考虑不足与应用业务产生硬耦合。当第一个 **if** 出现的时候，低代码平台的架构腐化就开始了，当再也 **if** 不下去的时候，已经病入膏肓。

如果低代码平台的代码架构，已经有一定的扩展能力，能从架构上隔离开业务定制性的需求和通用性需求的话，就可以避免与业务功能产生强耦合了。但这样还不够，由于应用团队无法直接参与需求的定制，这会导致应用的需求都积压到低代码平台团队中，漫长的交付周期要么把平台团队搞得精疲力尽，要么无法按期交付，导致业务团队的不满。

我们再换个角度，站在低代码平台角度看这些通用性的弊端，主要问题是需要有一个良好的代码架构和演进策略；其次是需要尽早规划尽早实施，越往后推历史负担越大；再者是实现难度比较大，任何修改都要能满足通用性和定制性解耦。虽然这些问题也不少，但是这些都是平台内部的问题，属于内部矛盾，解决起来相对容易。今天这讲就不深入讨论这个方向上的问题了。

那么，为什么说插件系统能够解决这些弊端呢？

首先，插件非常廉价。就凭这一点，插件就可以解决通用型低代码平台的各种弊端。廉价意味着开发成本低，也意味着可以有庞大的数量。数量庞大的插件可以像细沙一样，填满低代码平台各个大功能的覆盖盲区。

和平台团队接到需求时的各种瞻前顾后不同，用插件实现需求，就一句话：干就完了。也因为廉价，插件的试错成本比较低，搞错了大不了推倒重来。我们的许多需求可以先以插件的形式提供、试错，验证过后，再融入低代码平台中去。

廉价也意味着应用团队也可以参与，这样就能挡住不少原本应该提交给平台团队的需求。同时，鞋子好不好，只有脚知道，应用需要啥功能，应用团队自己最清楚。所以，这些自己为自

己量身定做的插件也可以大幅提升应用开发的效率。

插件系统的设计与实现

那么，现在我们具体说说如何设计和实现一个插件系统。我们可以从 SDK 的提取方法、可扩展的功能建议、插件二次开发和 manifest 设计，以及插件生命周期管理这几个方面来考虑。

SDK 的提取方法

啥是 SDK 呢？SDK 的全称是 Software Development Kit，也就是软件开发套件。在这一讲里，SDK 的作用是提供了一套实现插件的框架和必要的辅助功能。

在设计一个插件系统的时候，低代码平台要先定义好哪些需要扩展的功能，并在内部事先预留好扩展点。而 SDK 的一个主要作用，就是帮助二次开发者更方便地找到这些扩展点。而且，SDK 需要提供必要的类型定义和功能，这些能够大幅降低在扩展点上做开发的难度，从而帮助二次开发者更快、更容易地做出一个插件来。

通常来说，SDK 主要包括：

- **接口和类型定义**。这部分代码量可能还占不到 SDK 包所有代码的 1/10，但却是最重要的一部分。这些接口和类型就是插件的架构和框架，它们勾勒了整个插件系统的轮廓和概貌；
- **基类的定义**。扩展点通用部分的实现，需要我们尽可能完整地将各个功能实现出来，只留下尽可能少的抽象方法。而二次开发者主要的工作，就是补全所有的抽象方法、覆盖必要的父类方法；
- **调试工具和构建 & 部署脚本**。这是二次开发过程必须的专用工具，包括调试器、构建 & 部署的脚本、脚手架等。如果没有这些工具，二次开发基本就无法继续了，所以我们必须优先实现和提供；
- **辅助性、功能性工具类**。这些就是工具包，封装了常用的功能，目的是降低二次开发难度，提升二次开发效率。它们是辅助性的，没有它们也不会对大局产生多大影响。因此我们可以降低这部分的优先级，在资源允许之后，或者根据二次开发人员的所问所需，针对性地逐渐提供。

那么，我们应该如何设计一个扩展点，以及哪些代码需要挪到 SDK 中呢？

这里你可以复习一下第 9 讲，我通过在低代码的属性编辑器上做扩展的方式，非常详细地给出了如何设计扩展点以及哪些代码需要放到 **SDK** 中的方案。同时，你也可以复习一下第 12 讲，这是另一个案例，采用类似的方法，你就可以按照自己的需要，设计出新的扩展点来了。

那么，有哪些功能适合开放出去给应用扩展呢？

可扩展的功能

总的来说，我们至少有数据与数据模型、自定义组件、自定义交互动作，等等这些扩展点。我们展开分析看看。

扩展点一：数据与数据模型

数据存取是存量系统与低代码平台之间最主要的对接方式，这也是插件系统主要要解决的问题。

我们都知道，**App** 的职能可以分为两种：生产数据和消费数据，低代码平台自身能处理掉一部分数据，但是绝非全部，特别是对存量系统的数据的使用，是一个刚需。

但是喜新厌旧是人之常情，在企业里往往也是这样的。有了新系统，老系统的流量会逐渐切换到新系统，然后进入只读状态，但老系统里的数据是有价值的，可能由于数据结构、数据量等这样那样的原因导致无法将数据移植到新系统中，往往就必须新老系统并行一段时间（而且这个时间往往会很久），此时，我们就可以给老系统做一个插件用来与低代码平台做对接，是一个很好的选择。等以后老系统彻底下线了，直接把插件拿掉就行了，不会有残留。

还有另一种情况，也是我现在面对的情况，低代码平台与存量系统是共存关系，老系统数量众多，且依然继续在演进，不可能被低代码平台顶替。而低代码平台也不可能不顾一切地融入到某个存量系统（这样会大大降低低代码平台的价值），于是就出现了一个低代码平台需要对接许多存量系统的局面。而且每个存量系统都有自己的一套获取数据机制和迥异的数据模型。这样的情况下，唯有针对各个存量系统打造一个专用插件用来获取数据、提取数据模型这一条路可走。

扩展点二：自定义组件

自定义组件是低代码平台最主要的扩展点之一。一般来说，低代码平台内置的组件集都是通用的、常见的，而应用单位自行封装的业务组件，虽然通用性差，但在它适用的那一亩三分地

里，价值很高，所以我们需要允许业务团队开发和封装他们适用的业务组件。

另外，再牛的内置组件集也会有功能盲区，在特定场合下，应用团队可以利用这个扩展点来为低代码平台添加新的组件。当然了，作为低代码平台兜底策略的一部分，低代码平台应该要有一种能力能在应用团队无需封装插件的前提下，直接调用原生 **API**，快速使用第三方库。

扩展点三：自定义交互动作

我们前面也说过，可视化编程是可视化开发模式中最难的一个环节，因为可视化编程中，我们需要编排大量的逻辑。

业务组件也有类似场景，比如业务团队内部会积累一些程式化的交互动作。如果我们要用通用动作编排出这些逻辑，需要填写大量复杂、不好维护的参数。这时，我们就可以将这些逻辑封装成自定义交互动作，只暴露出若干输入框作为参数，这样一来，自定义动作卡的使用体验往往就会好许多。

除了封装复杂逻辑之外，自定义交互动作还可以对存量的复杂系统的用法进行场景化归纳，再根据归纳到的使用场景设计相应的参数。动作的使用者只要选定一个预设场景，正确填写所需参数后，低代码平台就可以按预设自动生成相应的代码了。

比如，我所在的产品线有一个 **WebGIS** 系统，它有上百个 **API**。有一个深度使用这个 **GIS** 系统的业务团队把它在产品线中的用法归纳了一下，整理出了栅格展示、小区展示、热力图等用法，然后将各个用法封装成一个个自定义动作，每个动作必填的参数很少，主要是在配置如何查询数据。完成了后，即使对这个 **GIS** 系统不熟悉的人，也能快速地在 **GIS** 上渲染出所需的图层和数据。

其他扩展点：导出、登录等

最后，我们这里再列举一下其他可做插件的扩展点。

首先，我们在导出应用数据时，如果有扩展点，就可以对导出的原始应用数据做一些转换，转为其他平台或者其他用途的数据。我给你介绍下我碰到过的两个实际场景：第一个是将应用数据一键导出成支持多种运行平台应用包，有的是物理机运行时，有的是 **Docker** 虚拟机运行时；另一个是将在线 **Web** 应用直接导出为离线报告，比如 **Word**、**PDF** 等格式。我们现在就有应用团队正在研究如何导出有交互能力的离线 **Web** 应用包。

另外，如果你的低代码平台需要被其他多个系统纳管，或者要部署到客户的系统中，那很可能需要支持多种不同场景的单点登录功能。这个时候，针对每个系统制作登录插件，按需安装，就是一个非常好的做法。

插件二次开发和 manifest 设计

除了前面说的 SDK 的提取和可扩展点的设置外，在插件系统的设计和实现上，我们还需要考虑插件二次开发和 manifest 设计的问题。

和多数其他系统的插件一样，我们需要有一个 manifest 文件来描述插件的信息，基本信息包括：插件的名字、插件的版本、所用 SDK 的主版本，以及插件功能描述等静态描述信息。

但更关键的是，Schema 必须给出这个插件实现了哪些扩展点、各个扩展点所在的路径，还有各个扩展点的个性化配置信息等。举个例子，如果我们添加了自定义组件，那可能需要为每个新增的组件配置一个图标。

比如下面这个插件 manifest 是我们的低代码平台 Awade 正在使用的：

 复制代码

```
1 {
2   "name": "datahub",
3   "module": "DataHubModule",
4   "path": "web/dist/@awade/plugin",
5   "serviceInfo": {
6     "services": [
7       {
8         "label": "DataHub",
9         "name": "datahub",
10        "remoteData": {
11          "class": "DataHubRemoteData", "import": "web/src/lib/compon
12        },
13        "renderer": {
14          "class": "DataHubRemoteDataRenderer", "import": "web/src/li
15        },
16        "initData": {
17          "style": {
18            "dataReviserHeight": "calc(90vh - 410px)", "paramBoxHei
19          }
20        }
21      }
22    ]
23  },
24  "actionInfo": {
```

```

25     "actions": [
26         {
27             "category": "事件与数据",
28             "type": "datahub",
29             "action": {
30                 "class": "DatahubAction", "import": "web/src/lib/components
31             },
32             "renderer": {
33                 "class": "DatahubActionRenderer", "import": "web/src/lib/cc
34             },
35             "initData": {
36                 "style": {
37                     "padding": "10px 10px 0 10px",
38                     "dataReviserBottom": "10px",
39                 }
40             }
41         }
42     ]
43 },
44 "metadatas": [{
45     "selector": "plx-table",
46     "class": "PaletxTable",
47     "import": "web/src/lib/components/table/index",
48     "category": "dataDisplay",
49     "label": "Paletx Table",
50     "desc": "Paletx Table",
51     "icon": "assets/icon/plugin-paletx-pro/table.svg"
52 },
53 {
54     "selector": "plx-badge",
55     "class": "PaletxBadge",
56     "import": "web/src/lib/components/status/index",
57     "category": "dataDisplay",
58     "label": "Paletx Badge",
59     "desc": "Paletx Badge",
60     "icon": "assets/icon/plugin-paletx-pro/status.svg"
61 }]
62 }

```

插件系统在拿到一个插件包之后，首先就要读取这个文件，通过它来获取插件的所有信息。因此，这个文件在插件包里的位置和名字必须固定，比如就放在插件包的根目录下，名为 `manifest.json` 就可以了。

而二次开发的最主要工作，就是在安装好了 SDK 包之后，按照平台的规范，正确编写各个扩展点的代码。插件的开发工作一般不会特别难，但万事开头难，因此我建议你的平台可以根据不同的扩展点给出一些 **Demo 插件**，这样应用团队就可以在对应的 Demo 插件包的基础上，依葫芦画瓢完成剩余的工作。

注意，给出的 **Demo** 插件一定要是可以安装和使用的，否则它的价值就大打折扣了。

按照我的经验来说，编写的图文文档一般没人看，多数人还是喜欢直接照抄。在照抄的同时有不明白的地方，会直接来问，没几个人有耐心去搜索文档。所以，你可以把一些常见问题的解决方案，直接通过注释的形式，写在 **Demo** 插件的示例代码中。必要时，还可以在注释中放上一个超链接，导航到更详细的图文文档中去，这样的效果最好。

插件生命周期管理

最后，我们还要关注插件的生命周期管理的问题。一个插件的生命周期，大概有这些主要阶段：上传、安装、激活、使用、去激活、迭代更新、卸载。**插件系统需要在各个阶段提供对应的通道和工具，支持插件更新和切换自己的状态。**

上传比较简单。我们直接在开发平台上开放一个插件上传通道，这样开发者就可以将他的插件上传到系统中来了。

接下来，插件系统就需要对插件包做静态校验，读取 **manifest.json** 文件，并检查所有必要的配置项是否合法有效。校验通过之后，就可以把插件安装到插件系统中来了。这个过程主要是文件拷贝，在我们的实践中，主要是 **js bundle**、**npm** 包以及其他静态文件的拷贝，插件系统此时不会去使用或执行相关的代码。完成之后，插件后台管理器会给界面推送一个安装完成的提示，收到这个提示之后，应用就可以激活这个插件了。

插件激活时，插件系统就会实际使用和执行插件包里的代码了，所以这个操作对低代码平台的安全性会构成一定的风险。

需要说明的是，我这里并未对系统的安全性做特别的关注，因为我们现在主要面对内网用户，没有面向不确定的公众开放，因此我们没有将插件系统的安全性提到特别高的优先级，目前只做到单一插件死掉不影响系统和其他插件的最低程度，不考虑插件开发者有破坏系统的主观恶意。因为在完全实名的前提下，我们通过日志是很容易抓住破坏者的。

如果你的系统需要**向不确定的公众开放**，那么系统的安全将是一个非常重要的议题，需要重点关注。

具体激活插件时要执行哪些操作，取决于低代码编译器的架构，一般包含前端和后端两部分。前端的部分要把 **JavaScript bundle** 从服务器下载到浏览器，然后 **eval** 一下就可以了，后端部

分则是执行插件包的初始化代码，把 **NPM** 包的入口注册到插件包的功能入口上。

可以看到，这两部分都是有安全风险的：前端的激活容易遭受 **XSS** 攻击，后端在执行插件的初始化脚本时，有可能会执行到恶意代码。但我这里暂时没有防范的经验可以分享，如果你有相关的经验，欢迎在评论区分享给我们。

插件激活的时候，插件系统会把插件所提供的各个功能植入到低代码平台的各个环节，比如组件列表中增加对应的业务组件，动作列表中增加对应的自定义动作，获取数据的功能列表里增加对应的数据获取通道，等等。这些功能都植入好了后，我们就只需要等着应用开发人员按需使用就好了。

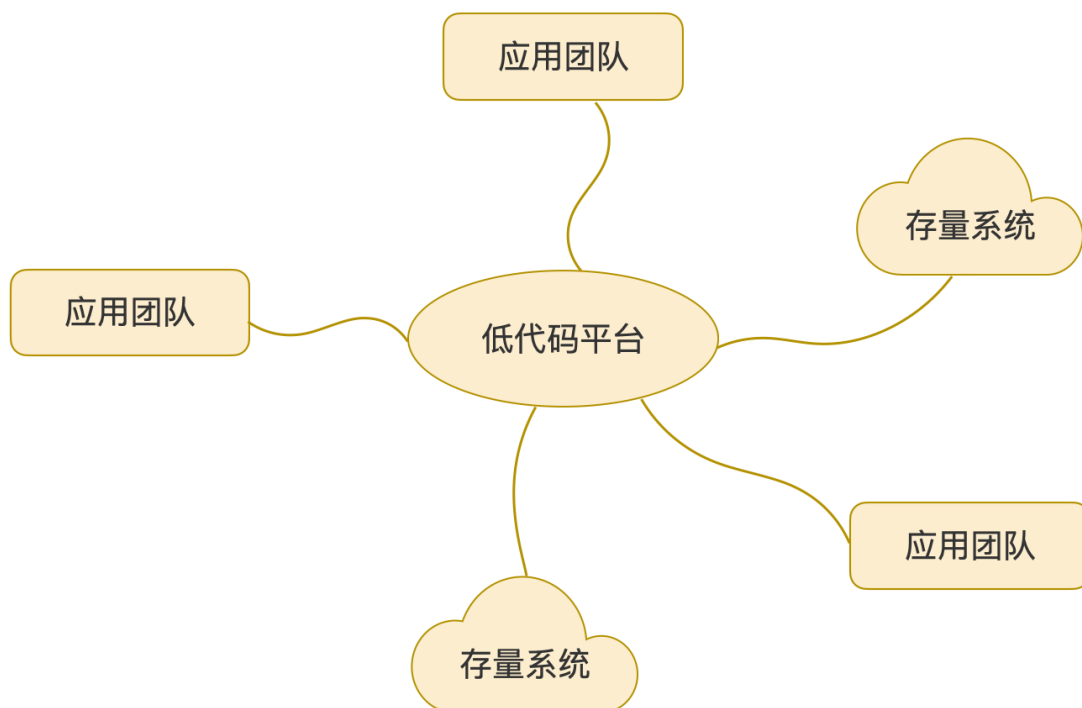
与激活和安装的流程相反，插件去激活和卸载的过程需要我们在相应的功能入口处删去植入的能力，删除插件相关的文件、包等内容。

最后，如果有余力，你还可以开发一个插件分发平台，用于集中管理插件，包括新增与删除插件、插件的在线自动发现、插件的版本升级，等等。如果你资源有限，又很需要这个功能，也不需要从零开始搭建，可以复用已有的功能，比如制品库和内部 **NPM** 镜像等。如果你们有内部网盘、论坛等，只要能托管文件，也可以加以利用，实在不行，搞个 **FTP** 服务器也成。

生态圈只是一个副产品

根据我们前面的分析，存量系统通过插件，可以将其数据与业务流程和低代码平台相连，应用团队通过插件，可以把业务组件、业务模板、方法和低代码平台连接。当连接的节点逐渐增多之后，你会发现原本相互隔离的人和数据之间，间接地产生了连接。

数据与数据之间、人与数据之间、人与人之间相互打通，形成了一个圈子。随着加入成员越来越多，每个成员可以从圈子里获得更多收益，同时也会吸引更多的成员加入，这是一个正向的增强回路。



这就是一个生态圈。一般生态这样字眼，会给人一种比较虚的感觉。但是我们这讲所谈及的插件系统是非常务实的，都是从实用以及如何解决存量系统之间的关系着手，也都是扎根于这样的目的。因为，我相信，在任何正常的企业中，低代码平台都绝不是第一个系统。在它之前，必然已经有形形色色的系统在跑，在用。

低代码平台不能仅仅是另一个新系统，而是要**成为企业的核心**，通过插件联通各个存量系统，贯通数据，让存量数据创造更大的价值。从这个角度看，低代码平台实际上起到的是中台的作用。这一点我在第一讲也说过，低代码平台的演进线路有相当一部分与中台是同向，甚至是重叠的，这两者可以、也必须放在一起考虑。

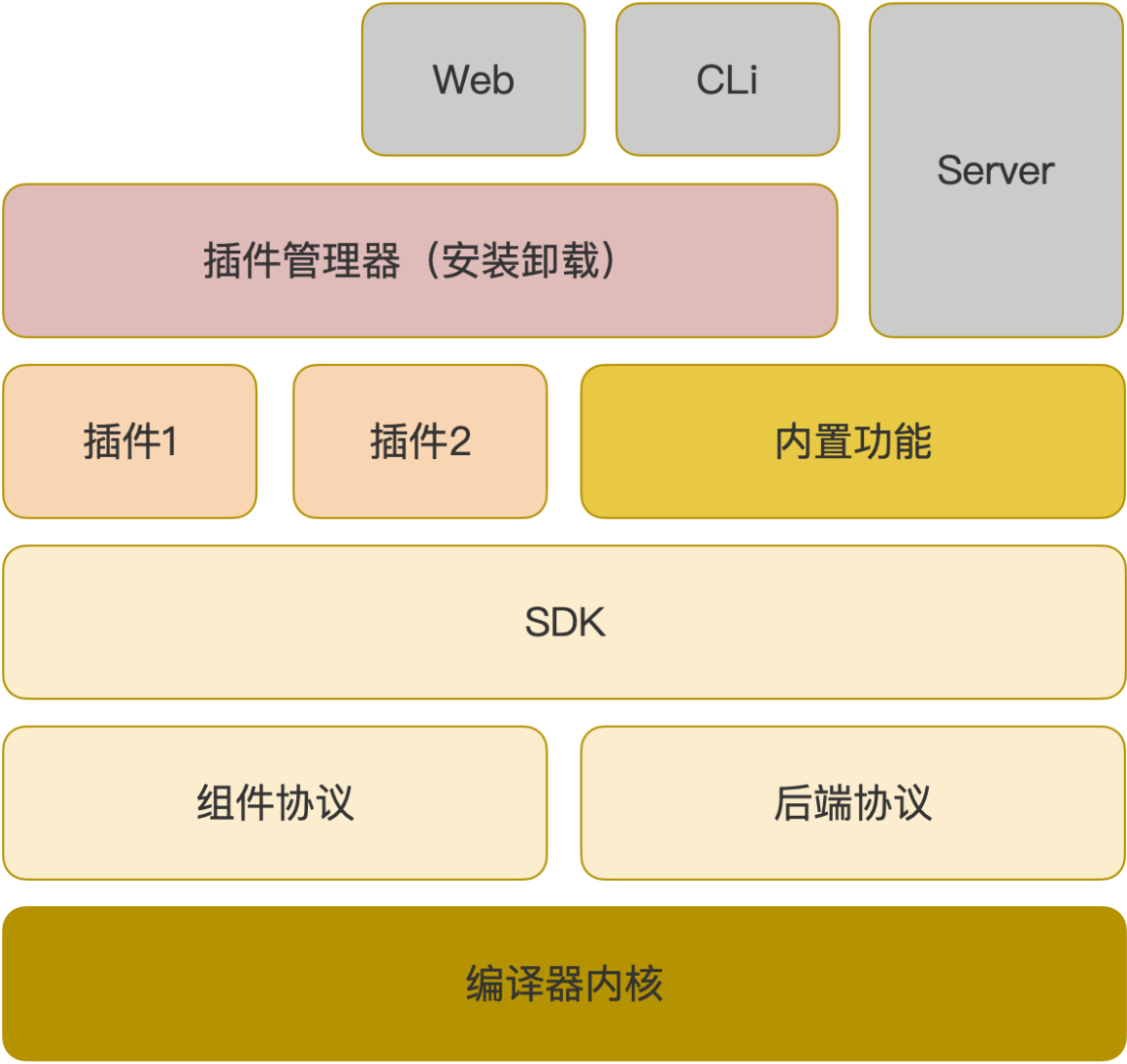
虽然我们不着重谈生态圈这样的虚头巴脑的内容，但是，依托于插件系统，实际上低代码平台和所有存量系统之间，自然而然就形成了一个生态。低代码平台能帮助存量系统发挥更大价值，而存量则一步步将低代码平台推到企业的核心。联通的系统越多，低代码平台对企业的价值就越大、越重要，这是一种良性共生关系。

在和其他系统打成一片的同时，又能独善其身，不与任何系统耦合，只有插件系统才有可能达到这样的目的。

总结

今天这讲，我们从全流程设计插件系统的角度，分析了插件系统的各个环节。由于内容比较多，我们没能仔细深入讲解一些环节。不过，其中插件系统的核心功能，也就是 SDK 的设计和 功能，我们在第 9 讲和第 10 讲里已经做了非常详细的阐述，而插件系统的其他功能，都是 围绕着 SDK 的架构设计和功能来打造的。

为了方便你更好地理解这些内容，我这里贴了一张 **Awade** 的插件系统架构关系图。我们一边 分析，一边回顾这些知识点：



你可以看到，在编译器内核之上，是一层协议层。它是编译器对外的抽象，所有的扩展点都是由协议层来定义和约束的。**SDK** 则是建立在协议层之上的，它提供了编译器协议的默认实现，以及所有扩展点的基类的定义。这些基类不仅能在减轻二次开发的难度的同时，更重要的是也约束了插件必须遵守的编译协议，插件的二次开发只能按照协议所画出的套路来实现。

SDK 之上，就是各个插件了。从架构图上可以看到，内置功能和插件一样，也必须遵守编译器协议。因此，从架构角度来说，内置功能与插件是平起平坐的，这样才能确保插件具有充分的扩展能力。当然，这只是从架构角度的设计，实际上插件能有多大能耐，还是取决于你的 SDK。这样就可以在实现层面上，保持内置功能的优势。

结合第 9 讲和第 12 讲的内容，以及上面的架构关系，相信你已经可以掌握 SDK 的架构和设计的方法了。在掌握了 SDK 的架构方法之后，你紧接着就需要考虑有哪些功能可以作为插件进行扩展。我认为插件系统可以在数据与模型，自定义组件，自定义动作等部位发挥显著的作用。其中，最重要的是数据与模型，通用型低代码平台要处理好存量系统的数据与模型的关系，插件是必须具备的能力。


思考题


我们假设，在你日常工作所要接触的各个系统和团队中，有这样一个插件系统可以打通所有的存量业务的数据。在这个前提下，互通的数据能创造出多少新的业务价值出来？

欢迎在评论区分享你的想法。我是陈旭，我们动态更新部分再见。

分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 低代码平台应该优先覆盖应用研发生命周期中的哪些功能？

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

