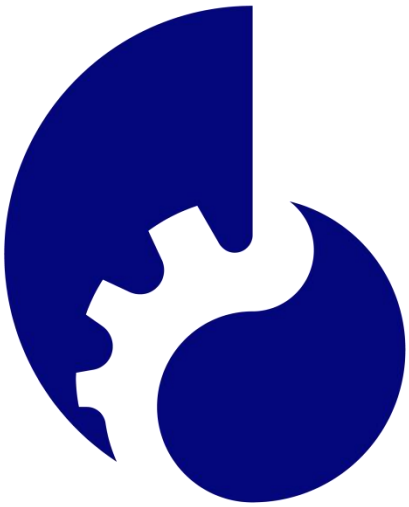

ONES

企业级研发管理工具

ONES 开放平台技术白皮书

2022 年 03 月

技术研发中心

文件更改记录

版本号	状态	修改描述	修订日期	修订人	批准人
V1.0	N	系统技术说明	2022.03.15	兰锦	

注：状态包括：N——新建，A——增加，M——修改，D——删除。

目录

1. 概述

1.1. 文档目标

本文介绍了 ONES 开放平台的目标、设计思路、以及实现过程的方式方法等。同时，详细描述了开放能力是如何依托开放平台，达到对系统功能的拓展、满足客户定制需求的。

1.2. 文档约定

系统：本文中的系统，如无特殊说明，即指 ONES 研发管理系统；
插件：指安装在 ONES 系统上的、基于开放平台开发的插件；

1.3. 目标读者

1. 希望基于 ONES 开放平台进行插件开发的工程师；
2. 希望了解 ONES 开放平台能力的业务、技术人员；
3. 其它对开放平台感兴趣的读者；

1.4. 项目目标

开放平台是 ONES 提供的基于主系统的二次开发平台。开发者可以基于此平台，开发 ONES 功能插件，达到修改现有系统表现、或增强标准系统功能的目的。

2. 开放平台概述

2.1. 目标与特性

开放平台在 ONES 系统的基础上，抽象出供二次开发使用的代码框架与对应的工具集，使得开发人员能够低成本、高效率的拓展 ONES 系统能力，满足用户的定制化需求。

开放平台解决的问题：

- 弥合核心系统产品化与用户需求个性化之间的矛盾
- 降低功能开发的难度，提高开发效率
- 打造完整的产品生态链，鼓励用户进行定制化

开放平台的特性，主要有如下几点：

- 低开发成本
  - 基于 react + node.js，使用一致的语言进行前后端开发定制；
  - 大量使用声明替代代码开发，进一步降低开发者学习成本和开发门槛；
- 高开发效率
  - 重视开发者体验，全方位提供开发者支持，提供全流程开发、调试、环境集成等工具；
  - 支持“所见即所得”的本地插件开发调试体验；
  - 提供“持续集成”相关的基础设施与工具，降低开发与测试集成成本；
- 强开放能力
  - 基于现有系统功能模块，提供全面的二次开发能力；
  - 进行了良好的能力模型抽象，易于理解的同时，满足深度定制；

2.2. 开放平台组成

开放平台包含一系列的组件、能力、开发环境与工具链等。

从开放平台实现的角度，主要包括基础设施实现，以及在此基础上完成的平台产品功能、开放能力、以及插件开发语言支持。

图 2.1 开放平台模块-平台实现

上述这些组件的实现细节，请参考后文的详细描述。

而从插件开发的角度，开放平台在提供插件开发的基础设施的同时，也提供了包括文档、

示例、培训视频在内的开发者支持。

这里的基础设施，包括插件开发工具、插件代码托管、以及开发过程的调试环境和持续集成环境。（注意，部分环境仅针对特定开发者提供）。

图 2.2 开放平台模块-插件开发

2.3. 插件使用

注意，若希望在您的系统上安装插件，需要系统启用了“开放平台”子系统。

2.3.1. 插件安装

管理员可以在插件管理界面通过点击“安装新插件”将插件安装到开放平台中。

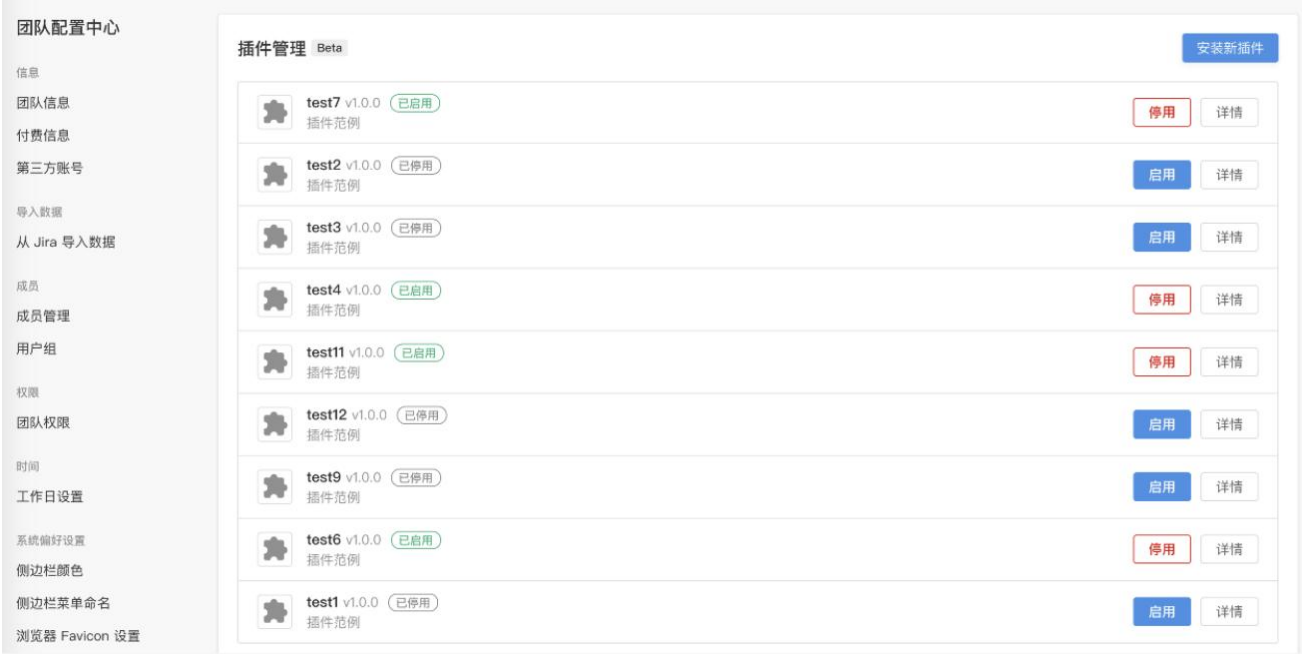

图 2.3 插件安装

2.3.2. 插件卸载

插件的卸载通过在插件的详情中点击“卸载”来进行。
在卸载前，需要确定插件处于停用状态。

图 2.4 插件卸载

2.3.3. 插件启用

新安装的插件需要被启用，通过在插件管理的插件列表中点击“启用”来进行。
在启用插件时，需要确定插件的授权说明，权限配置以及参数配置。

图 2.5 插件启用

2.3.4. 插件停用

启用状态的插件可以被停用，通过在插件管理的插件列表中点击“停用”来进行。

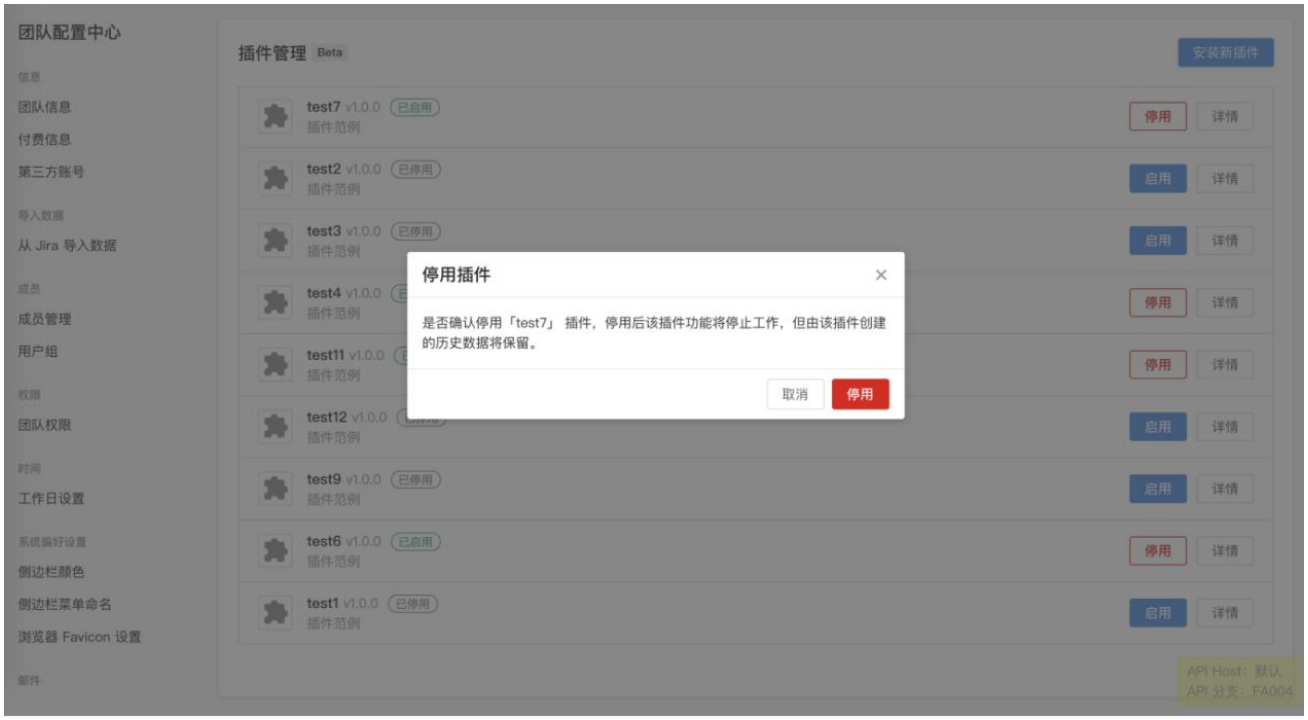

图 2.6 插件停用

2.3.5. 插件基本信息

插件开发者对插件功能等的一些说明，都会反映在插件的基本信息中。
这里也包括插件需要调用的接口范围的授权说明。

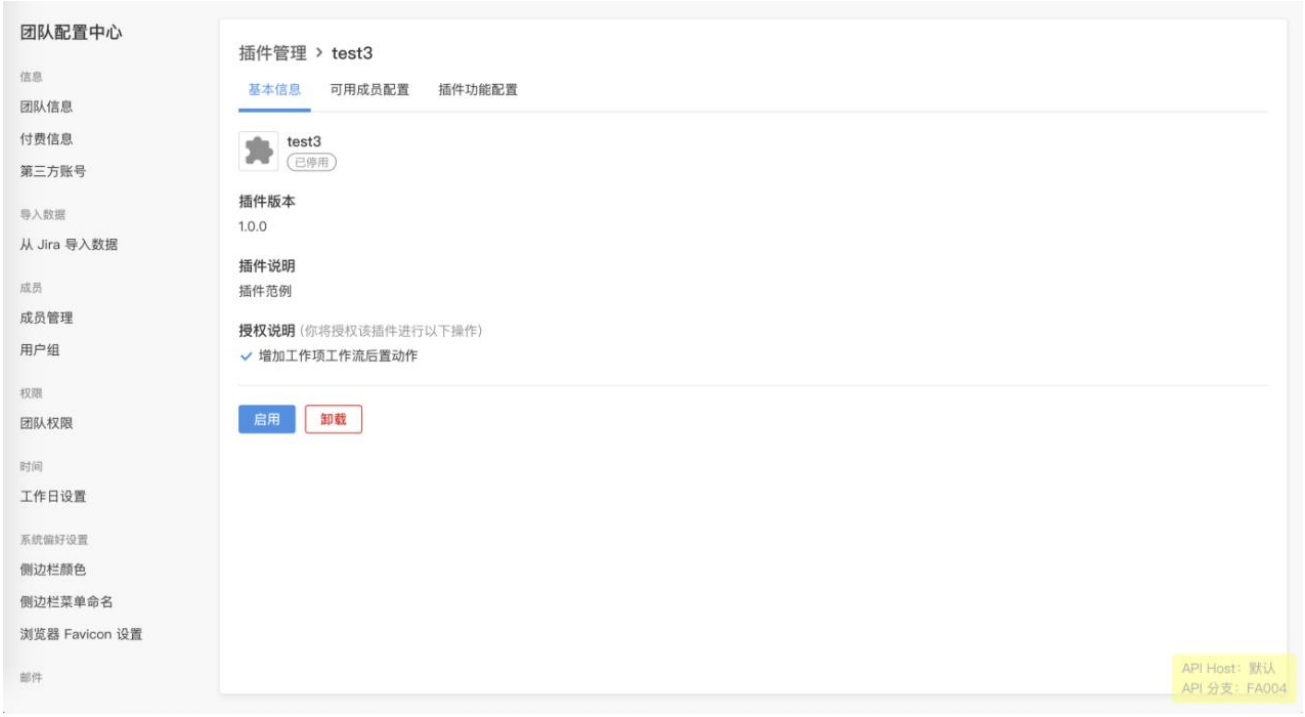

图 2.7 插件基本信息

2.3.6. 插件权限配置

插件需要遵循整个系统的权限规范。也就是说，插件只有被授权的团队成员才能使用。
权限配置在插件详情的可用成员配置中进行。

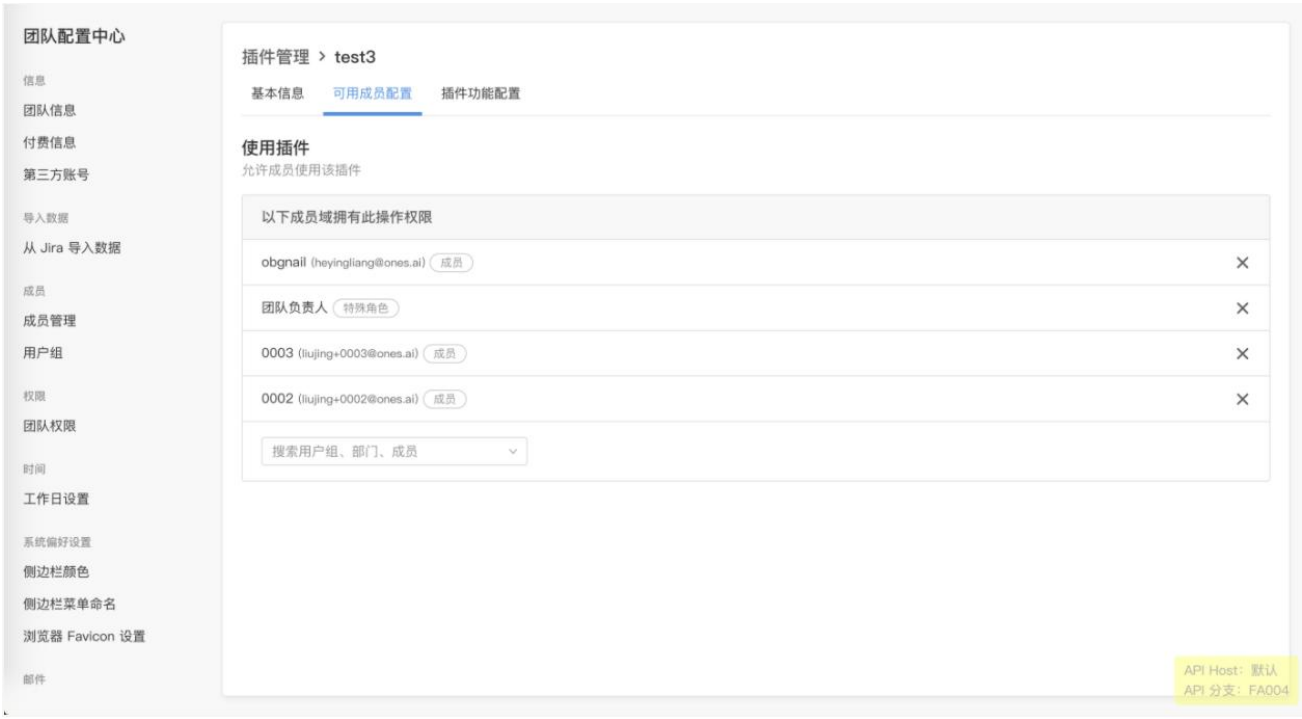

图 2.8 插件使用权限

2.3.7. 插件功能配置

插件在使用过程中，可能需要进行一些配置。这些配置由插件开发者申明，并且会在插件功能配置页面由管理员配置。

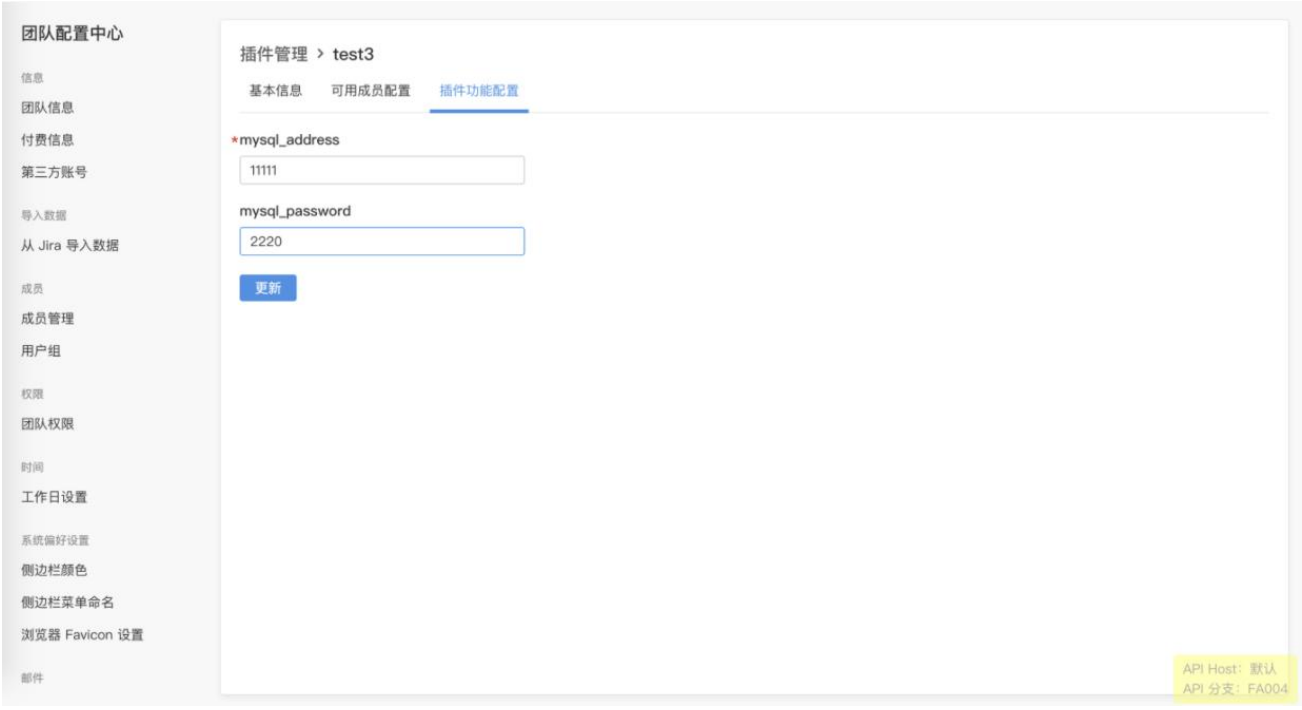

图 2.9 插件功能配置

2.3.8. 插件升级界面

管理员可以在插件管理界面通过点击“安装或升级”将某个插件的升级版本上传，上传以后，平台会根据插件的 appid 及其版本，选择对应的旧插件进行升级。

图 2.10 插件升级

3. 开放平台设计与实现

3.1. 模块与组件

开发平台是在 ONES 标准系统的基础上，对原有功能模块实现进行修改、重构等，使得系统的原有功能能够被（插件）动态的注入新的实现，从而达到使用插件对系统功能和表现进行二次开发的目的。

围绕这个目标，我们对标准系统进行了大量的修改与重构，同时增加了让插件能够“跑”起来的一系列新模块与组件。

如图所示，开放平台的组件包括：

1. 开放平台后端应用（OpenPlatform-API）；
2. 插件运行时环境（runtime/local development）；
   (1) 插件宿主机（Host）；
   (2) 插件宿主调度模块（Host Bot）；
3. 平台基础设施（Data Warehouse for Plugin）；
4. 标准系统的平台化模块：
   (1) 前端：插件前端能力层
   (2) 后端：平台服务（OpenPlatformService）；
5. 插件开发者支持体系；

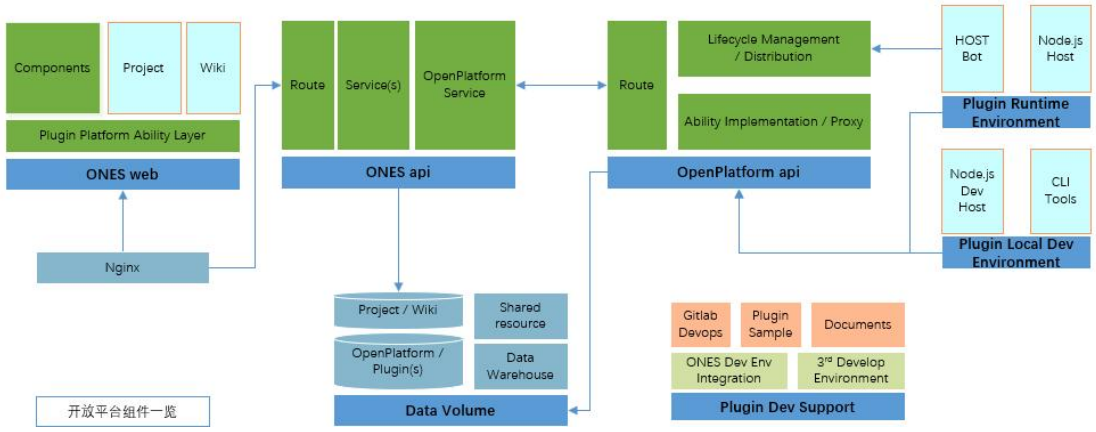

图 3.1 开放平台模块与组件

3.2. 插件工作原理

3.2.1. 标准系统实现

插件是基于 ONES 标准系统进行的二次开发的交付物。要了解插件是如何工作的，首先需要了解标准系统的构成。

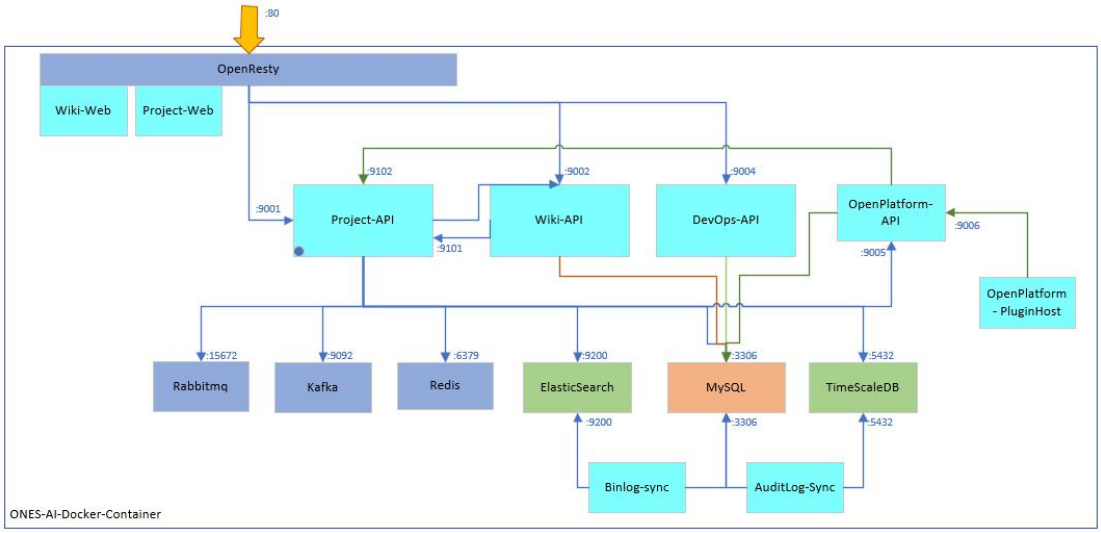

图 3.2 标准系统组件

如图，ONES 标准系统是前后端分离架构 + restful api 方式搭建的业务系统。这就表示：

- 用户看到的每个页面，都是由基于 react 的 web 项目代码，在客户浏览器渲染的；
- 而这些页面渲染过程中所需要的业务数据等，则是由浏览器请求系统后端 http 服务获取的；

3.2.2. 插件构成

标准系统是前后端分离的架构，因此基于标准系统运行的插件的代码，也包含前后端两个部分。

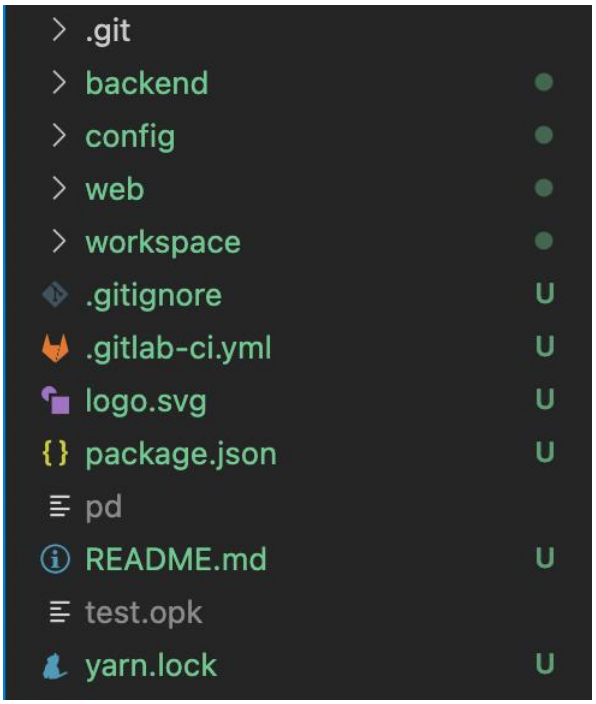

图 3.3 插件工程内容

同时，插件中还包括插件配置相关的问题，包括插件基本信息声明，以及插件版本升级声明等。

为了满足更多的自定义需求，插件还可以在打包时加入一些自定义的文件。这些文件会在安装时被释放到插件实例中，并且可以被插件访问和使用。详细情况可以参考对应的 插件文件操作 基础能力。

1. 插件前端
   - 插件前端组件相关代码编译产物；
2. 插件后端，插件后端包含这些内容：
   - 插件后端代码编译产物；
   - 插件的 workspace 中预先置入的文件；
3. 插件配置相关，插件配置部分，包括
   - 插件基础信息配置
   - 插件升级配置

3.2.3. 插件工作方式

我们将插件看成如下一些功能段的集合：

- 插件前端：
  - (纯前端插槽) 一些 react 组件。这些组件将会在其对应的声明中的位置 (插槽 id) 上被前端页面加载和渲染；
  - (带前端组件的业务开放能力) 一些 react 组件。这些组件将会在对应的 业务开放能力 被唤醒时，在前端页面被加载和渲染；
- 插件后端：
  - (生命周期方法) 与生命周期相关的预设的方法。这些方法将会在对应的插件生命周期阶段被调用，处理插件的一些数据准备、销毁等工作；
  - (接口相关) 插件注册的接口方法。这些方法根据注册方式的不同，会被 标准系统/插件 前端调用；
  - (带后端接口的业务开放能力) 大部分业务开放能力都会定义自己的接口（声明中的 Function 段）。这些接口需要插件实现，并在其能力被标准系统使用时被调用；
- 插件配置：
  - 插件配置用于帮助上述功能段的正常运行。

这里我们主要说明一下①和②的生效过程。

如图，从开放平台的角度，一个插件如果生效（被启用），其前后端分别会以不同的方式来工作。

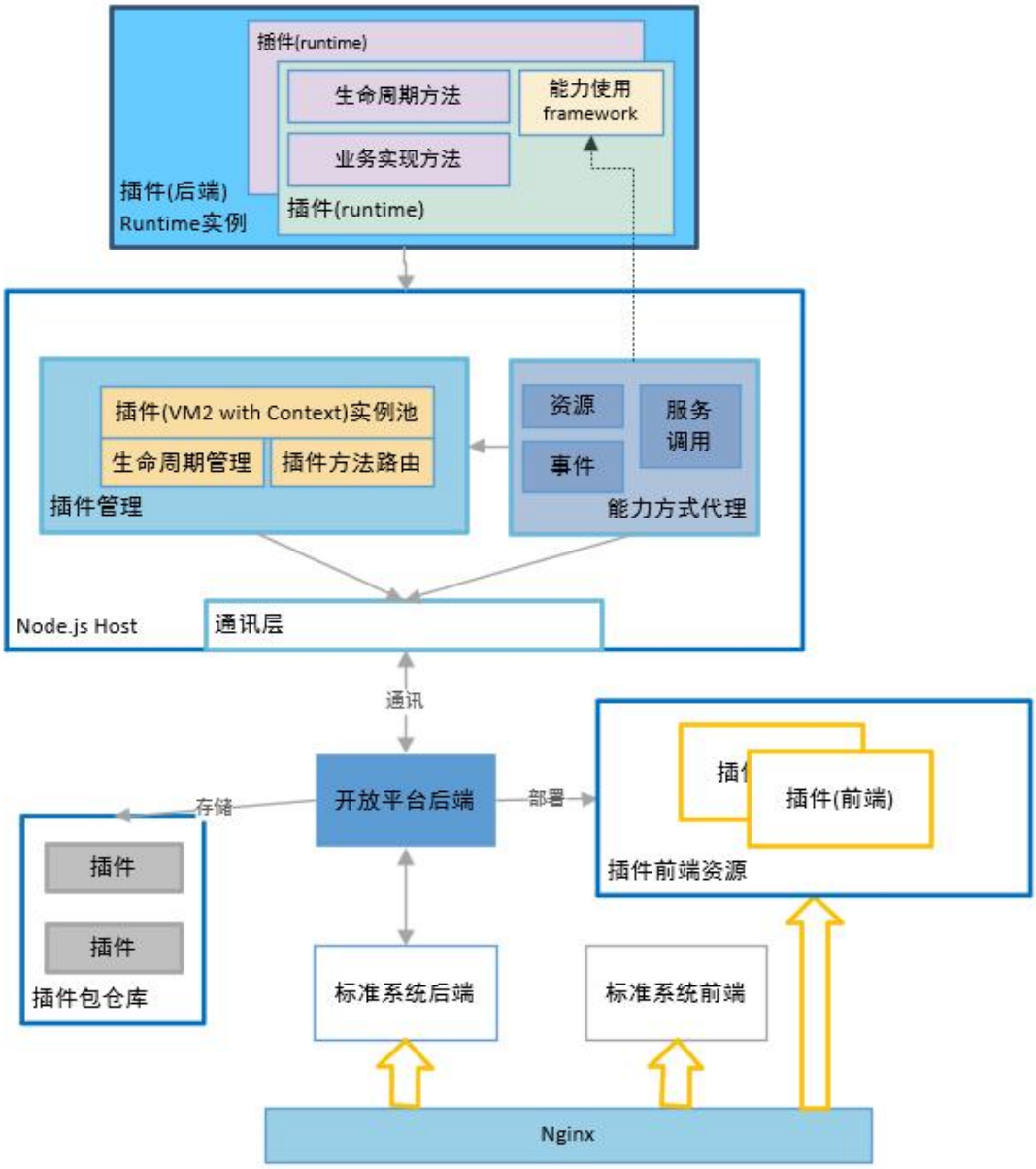

图 3.4 插件工作方式

插件前端的资源（打包好的组件、前端文件等等），都将会被部署到对应的插件前端资源目录中。此时 Nginx 可以直接访问这些资源。如果标准系统前端通过请求后端插件列表，并确定了需要在当前页面中生效的插件，那么它将会通过 Nginx 请求对应的资源并渲染出来（或者隐藏一些原有组件，视乎插件实现而定）。

插件后端，则将会被某个宿主进程（这里是 Nodejs Host）加载到自己的插件实例池中。前文提到过，插件后端代码本质上是一系列方法的集合。这些方法将会被加载到开放平台后端统一管理的路由表中，并可以被标准系统前后端访问。

3.3. 插件调度

在开放平台的基础架构层面，考虑到需要支持多语言、多版本的插件体系，我们引入插件调度。

3.3.1. 目标

开放平台中与调度相关的部分，包括自身组件的调度，与平台上安装、承载的插件实例的运行时分配与调度，这里我们主要讨论后者。

插件调度的主要目标，是在单节点 / 高可用模式下，保障插件能够被正确安装并运行。这里的插件指基于多种语言与多个能力版本（如果分的话）开发的插件。需要注意的是，当前插件前端主要基于 react 开发，以资源的方式被使用，所以我们的调度主要针对相对比较动态的后端部分。

3.3.2. 模块与责任

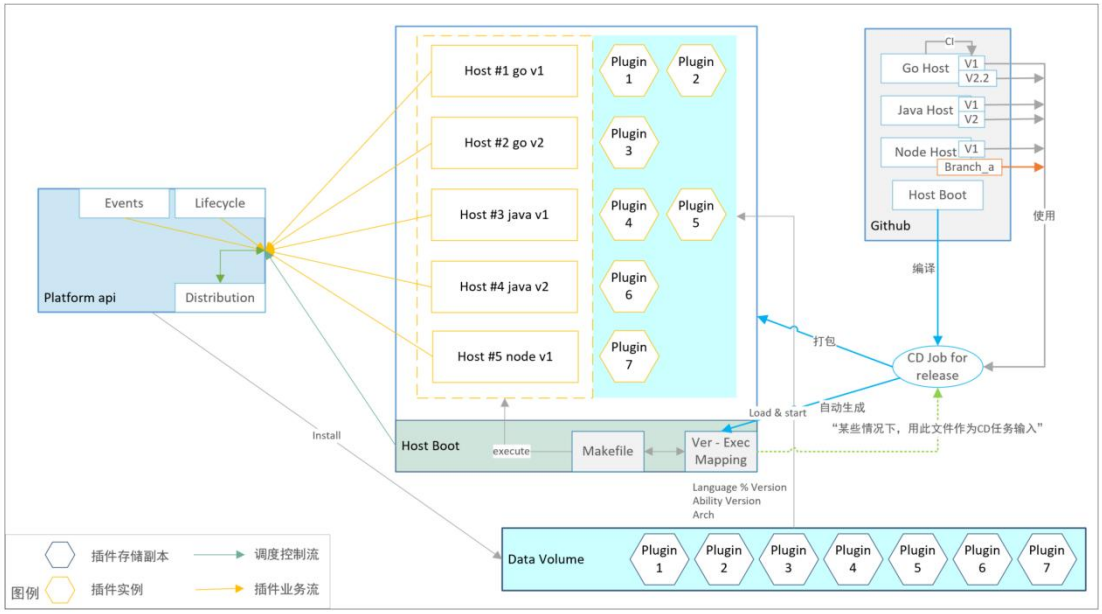

图 3.5　单节点插件调度：范围与模块

如图所示，整个插件调度算法涉及到平台本身、插件仓库、宿主机集合（包括各个语言、各个版本的宿主机，以及宿主机的启动器）。同时，也需要考虑宿主机集合的编译打包过程。

3.4. 插件生命周期

插件生命周期中的各个状态，及其触发的流程如图所示。

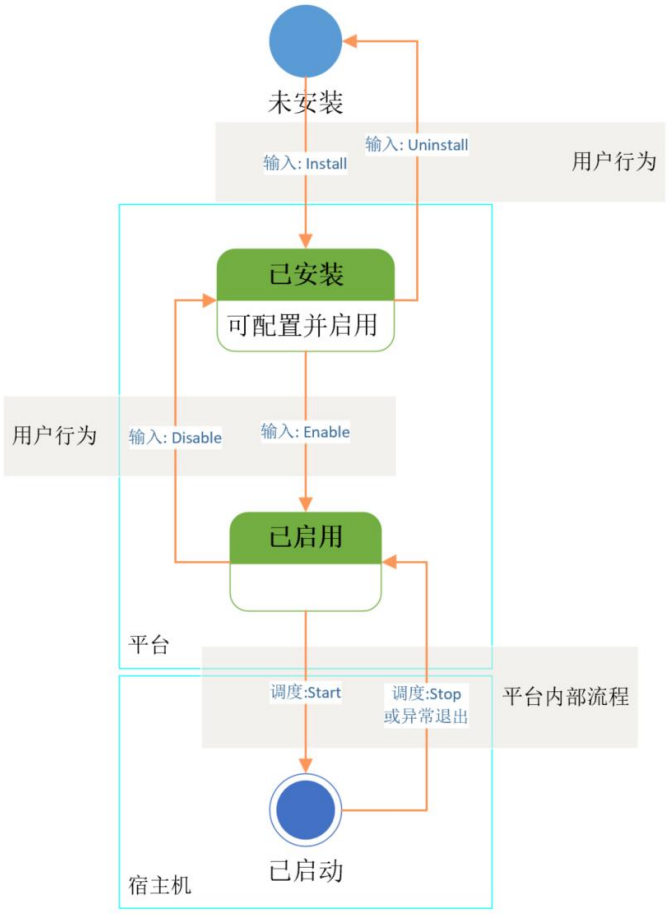

图 3.6 插件生命周期

插件生命周期动作，包括这样一些过程。

- 启用 | Enable
  - 启用过程实际上是使用 enable 参数启动进程；启用过程中，插件需要处理数据表的创建/文件创建等等；启用过程中，分配资源后插件应该启动自己的工作进程。
- 停用 | Disable
  - 当管理员在页面上停用插件时，插件需要响应的事件；停用过程中，插件需要释放自己的资源；然后插件需要关闭自己的工作进程；
- 启动 | Start
  - 当平台启动时，需要启动所有的已启用插件进程；
- 停止 | Stop
  - 当平台关闭时，需要向所有运行中插件发送停止消息；当插件升级时，也需要先关闭待升级的插件；
- 升级 | Upgrade
  - workspace 文件，workspace 中存放的是插件运行时可更改的文件。
  - 修改插件权限配置，通过跟原有的权限配置做对比，做新增操作，例如原定义了 A，B 权限，新插件有 A，B，C 权限，则新加 C 权限。
  - 修改插件自定义配置，通过跟原有的配置做对比，做新增操作，例如原定义了 A，B 配置，新插件有 A，B，C 配置，则新加 C 配置。
  - 替换插件运行文件。

对与插件开发人员，一般建议：

- Install：处理一些数据表结构的初始化等动作；这些数据将在 Uninstall 时被销毁；
- Enable：处理一些插件运行前的准备工作，如创建对应的运行时的标准系统数据；这些数据在 Disable 时应该去删除；
- Start：处理一些当前实例的运行时缓存等；这些缓存在 Stop 时应该被销毁；
- Upgrade：插件升级时需要处理的内容。

4. 开放能力设计与实现

4.1. 开放能力概述

在这个部分的内容中，我们来对开放平台的“开放”的承载者 - 开放能力，进行整体上的描述。

4.1.1. 什么开放能力

对于单个插件而言，无论其使用的语言和框架，我们都认为插件代码自身是无权限的。或者说，插件运行时环境，是从属于某个无权限账号启动的进程上的。插件不能访问系统资源，更不能访问其它插件的数据等。

即便不从插件进程角度考虑，我们也可以认为，插件与系统自身是在不同的端点、机器甚至机器上运行的；插件所能看到的资源，与平台认为的资源是不同的。

从这两方面考虑，插件在运行的时候，一旦需要调配资源，包括存储、网络甚至计算资源，来完成其自身的功能，都不能简单的像在本地开发运行代码一样，直接用某个库来访问系统 API。所以，我们可以简单的理解为，除了 CPU、内存相关的运算，插件做任何事情，都需要通过通讯来完成。

例如，当插件需要写入一个文件时，不能简单的通过 open 来获取 fd 并进行对应的读写操作，而是需要向平台发送一个封装好的数据包（请求消息）来获取文件标识，并基于此标识在后续进行读写。插件不能也不需要知道这个文件具体存储在哪台机器的哪个磁盘上。

通过这样的设计，我们将插件功能的实现和这些功能的使用者进行了全方位的解耦，使得底层架构能够更灵活的适配和调整。而这些请求，就是我们所说的开放能力。

以上这些论述，都是从技术层面说明一个开放能力所发挥的作用。而从业务上，我们的开放能力，则是插件开发者所能定义、修改系统功能的最小单位，以及所能使用、访问现有系统资源的最小单位。

同时需要注意到，一个纯前端的开放能力，与这些说明都不太一样。纯前端的开放能力，或者我们称之为“插槽”，是插件开发者对现有系统 UI 进行调整、修改的最小单位。

4.1.2. 如何使用开放能力

插件开发者使用开放能力的第一步，是了解我们有哪些开放能力，它们各自有哪些用处。因此，我们提供的能力文档中，不仅仅会包含能力描述本身，也会为能力提供对应的 sample。

为了让开发者更方便的使用能力，我们会在 CLI 工具中，提供开发者向一个插件工程增加一个能力使用规范的命令。通过这个命令，开发者可以很方便的将一个能力加入到工程中，包括能力的声明、配置以及对应的有一定业务意义的代码实现（如果需要的话）。

而对那些通过调用方法来访问现有系统功能的能力，我们会为开发者使用的语言封装对应的 framework。

4.2. 开放能力类型

4.2.1. 整体设计思路

从满足定制需求的角度，开放平台最重要的是提供更多的能力，为开发者满足定制需求服务。我们综合考虑现有系统的技术设计与实现，以及业务功能的开放性的评估，确定将开放的能力大致如图所示。

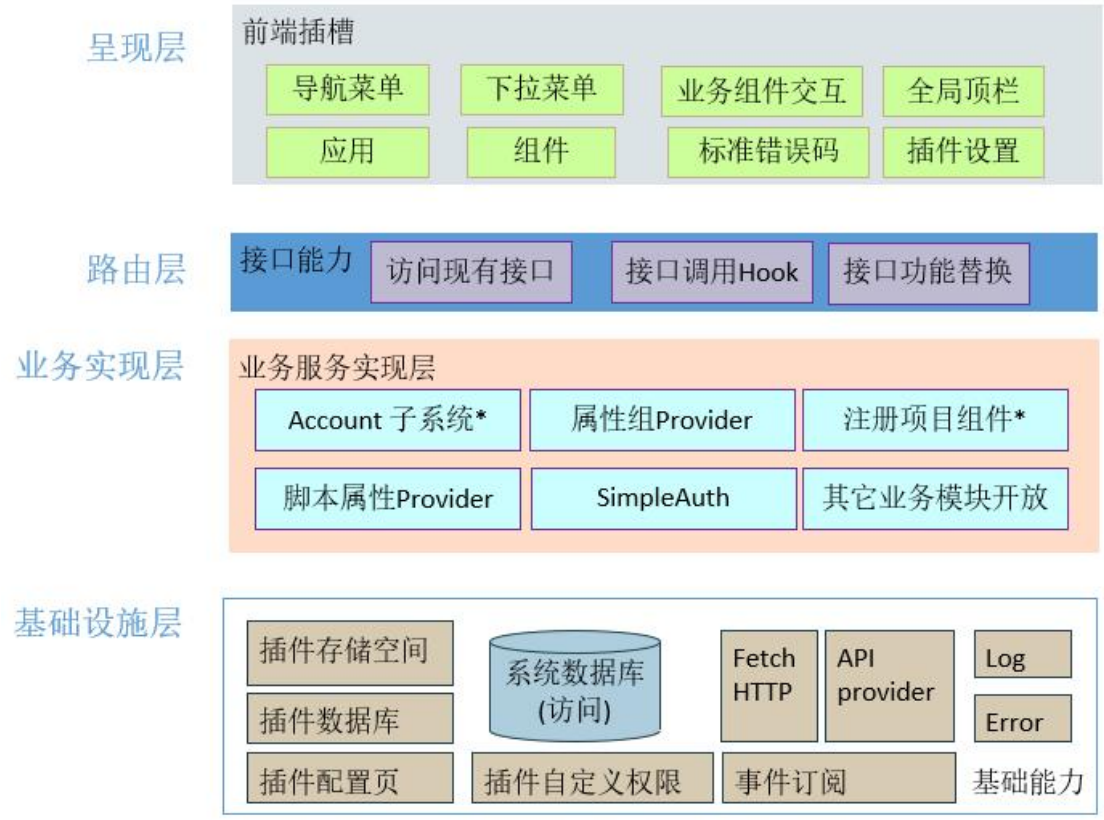

图 4.1 开放能力类型

4.2.1.1. 基础能力

插件可以申请和使用一些系统资源，并调用一些基础方法。
包括：插件记日志、存储结构化/非结构化数据、自定义账号/权限点/功能配置页面等等。

4.2.1.2. 接口能力

针对插件对接口层面的一些访问封装、劫持、注册等，我们也提供了一些相关的能力。

4.2.1.3. 插槽模块

插件可以使用标准系统前端的开放能力，我们称为 前端插槽模块。

前端插槽模块即标准系统前端界面上的一些组件、页面等的注入点。

4.2.1.4. 业务能力

插件可以使用标准系统的业务层面的开放能力；

标准系统从业务功能层面，对插件开放了一些功能置换、补充的功能点，我们称为 业务能力。

例如，插件可以注册一个新的登录方式，集成到标准系统中，具体登录校验过程则由插件实现；

关于这些能力的详细内容，请参考对应的能力说明文档。

4.3. 业务开放能力标准化

4.3.1. 概述

从使用者的角度观察业务开放能力，我们可以认为，这些能力主要解决的是：“插件将自己注册为某个能力的承载者，或者说某个系统模块功能的新实现者”这样的场景。例如，一个插件需要将自己注册为某个接口的劫持者（接口劫持）、某个登录方式的提供者（SimpleAuth）、或者某个 layout 卡片的提供者（LayoutCard）。

这样的能力能够有一些标准化的声明和实现流程。通过这个标准化流程，我们能够简化这些能力的实现。主要目标是，这些能力在实现过程中，无需对现有的平台、通讯协议和 Common 库有任何调整；只需要在能力实现端（OPS 中）以及能力使用端（SDK 封装）进行一些适配即可。

一个能力在插件描述文档中的描述一般如下表所示：

字段名	字段值	用途
id	自定义的能力在这个插件中的 ID	标注这个能力
name	自定义的能力名字	描述这个能力的目的
abilityType	固定可选的能力类型	明确是什么能力
function	kv 结构体，能力设计的方法路由； key 是能力固有的方法， value 则是插件中的方法；	帮助进行方法路由
Config	能力对应的配置；	快速使用能力开发插件；

表 4.1 业务开放能力结构

这些方法会有固定的 pattern。

其中，function 中每个方法，也需要对应的格式定义。同时，function 的参数都是标准结构，其中可变的部分则是通过 json 来封装。这个封装对平台的通讯协议是透明的。

4.3.2. 接口与过程

这个能力在实际使用时，会有一些标准的过程过程。

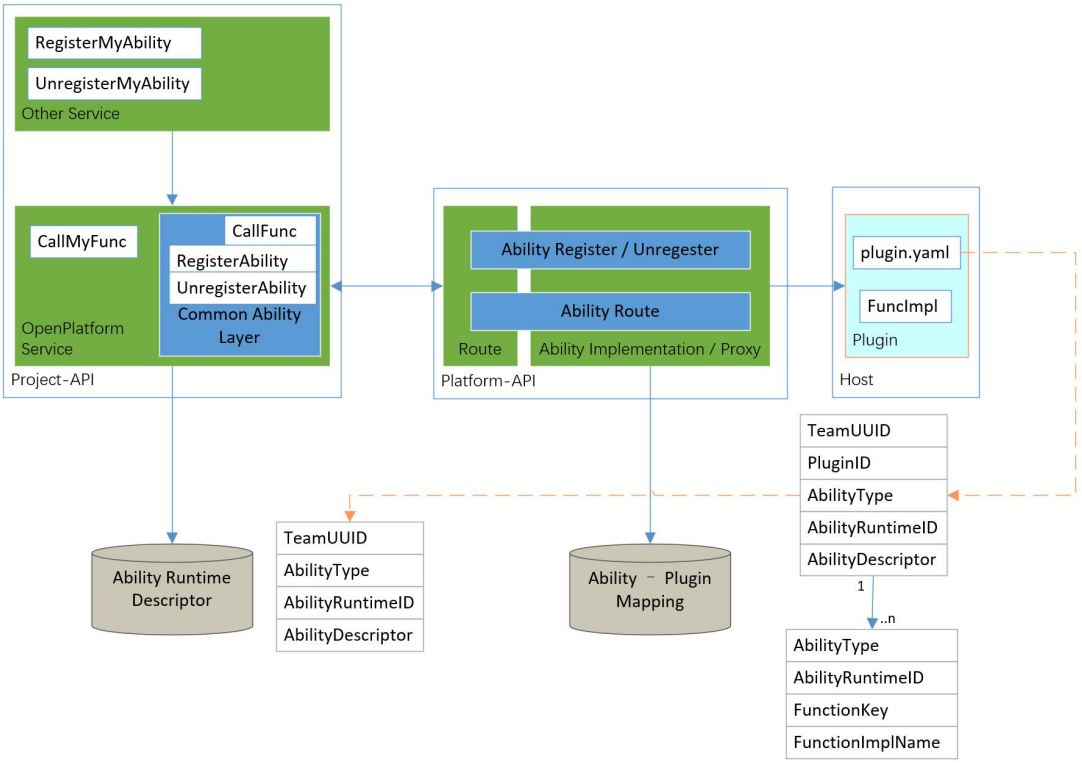

图 4.2 能力注册与使用过程

假设我们在 OPS 的底层（common ability layer），提供了一组标准的能力注册/注销接口，以及对应的调用接口。其中包括：

为平台准备的方法：

RegisterAbilityByTypeAndRuntimeID(abilityType integer, abilityRuntimeID UUID, abilityDescriptor Descriptor)

UnregisterAbility(abilityType integer, abilityRuntimeID UUID)

为能力开发者准备的方法：

CallAbilityFunctionByTypeAndIDAndFunctionPreDefinedName(abilityType integer, abilityRuntimeID UUID, abilityDescriptor Descriptor, parameter JSON_string)

而对于使用能力的开发者，只需要定义好对应能力的接口中的方法，并按照能力的规范（方法签名及方法业务意义）来实现它们，即可完成系统功能层面的注入，达到定制的目的。

5. 开发者工具与支持

5.1. 插件开发与调试环境

插件开发与调试的过程如图所示。

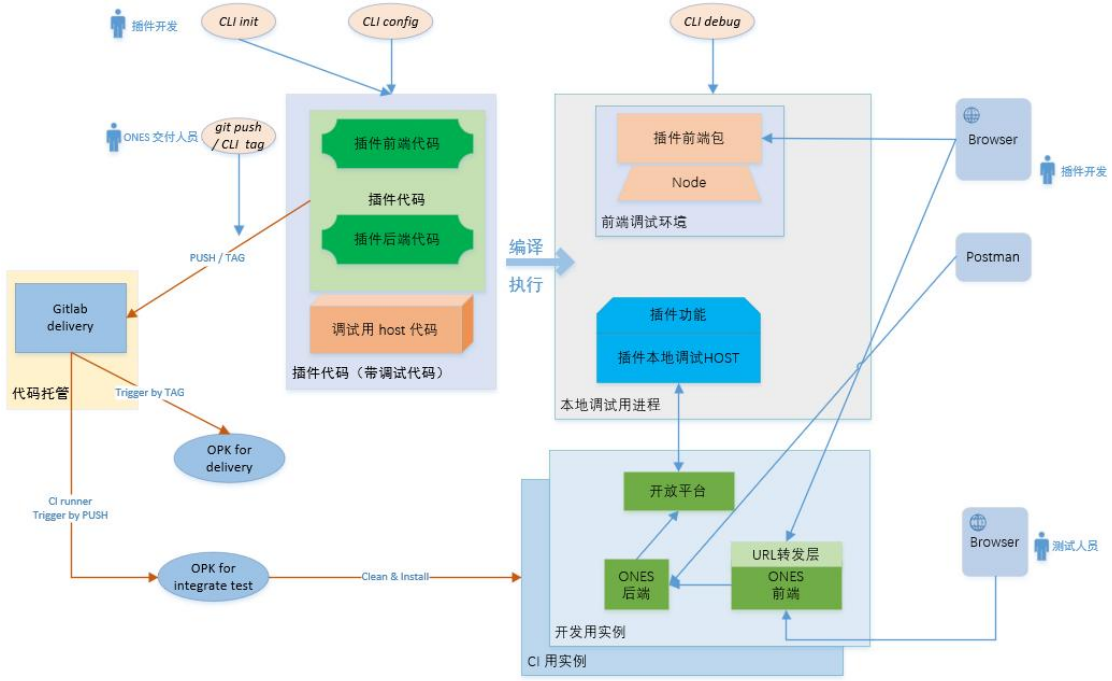

图 5.1 开放平台开发与本地调试

借助我们提供的插件开发环境与工具，开发者可以实现两个目标：

- 本地调试插件代码，并且在调试过程中可以所见即所得，直接在调试实例上看到插件生效的结果。
- 插件持续集成，一旦提交插件代码，即可触发流水线，将插件自动打包、上传、安装到指定的 CI 环境上去。

5.2. 开发者工具与调试流程

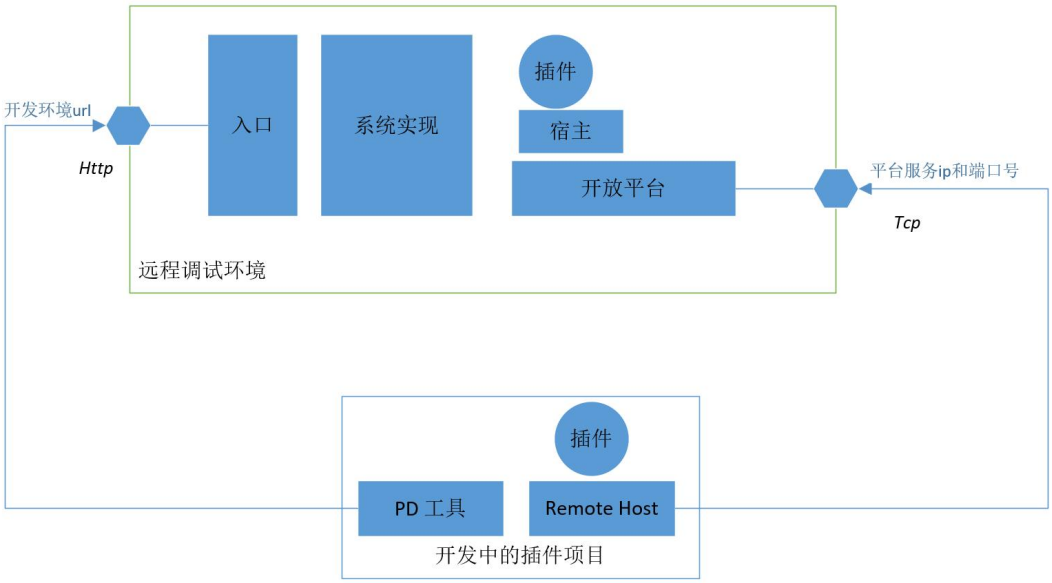

图 5.2 插件调试资源使用

如图，插件的本地调试，首先需要工具具备远程调试环境的权限，能过获取数据、操作团队等，这些可以通过登录（login 命令）到远端的调试环境来获取。我们的远程实时调试，实际上是在本地执行一个"远程宿主机"。

从前面的插件生效过程可以看到，我们的插件在运行的时候，后端都是通过通讯来完成其功能的。这就为我们的实时调试提供了基础。我们可以在本地执行一个调试用的插件宿主进程，并加载插件代码，完成插件功能。

插件前端的实时生效方式也是一样。开发人员用本地的浏览器访问调试环境页面的时候，如果前端判断插件有前端需要渲染的内容，将会直接重定向到本机的 npm 托管的插件前端工程中去。

5.3. 插件持续集成

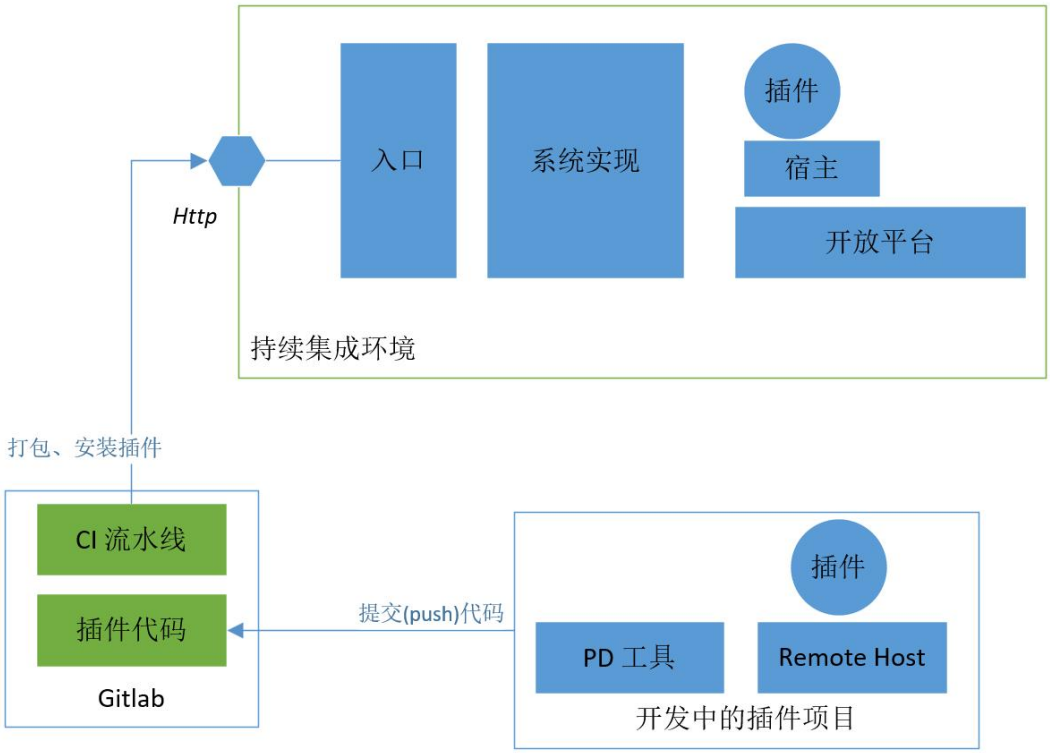

图 5.3 插件持续集成资源使用

如图所示，当插件代码托管在 gitlab 上，且进行了相关的配置时，本地代码一旦提交，即会驱动 gitlab 的流水线，打包插件并安装到配置的持续集成环境中。这样做的前置条件是：

- gitlab 上配置了流水线；如果你使用的是 https://gitlab.plugins.myones.net/ 来托管插件代码，则无需配置；
- 正确配置了项目文件夹中的 ci-deploy.yaml 文件；

5.4. 开发者文档

我们为开发者提供了一系列的培训视频，以及关于工具使用、开放能力的一系列文档手册。包括：

- 开始一个插件开发
- 向插件代码中增加开放能力；
- 在本地调试插件；
- 插件持续集成测试；
- 基于 gitlab 的插件交付流程；
- 开放能力列表与详细说明；

这些内容请查阅我们提供的对应文档，这里不再赘述。