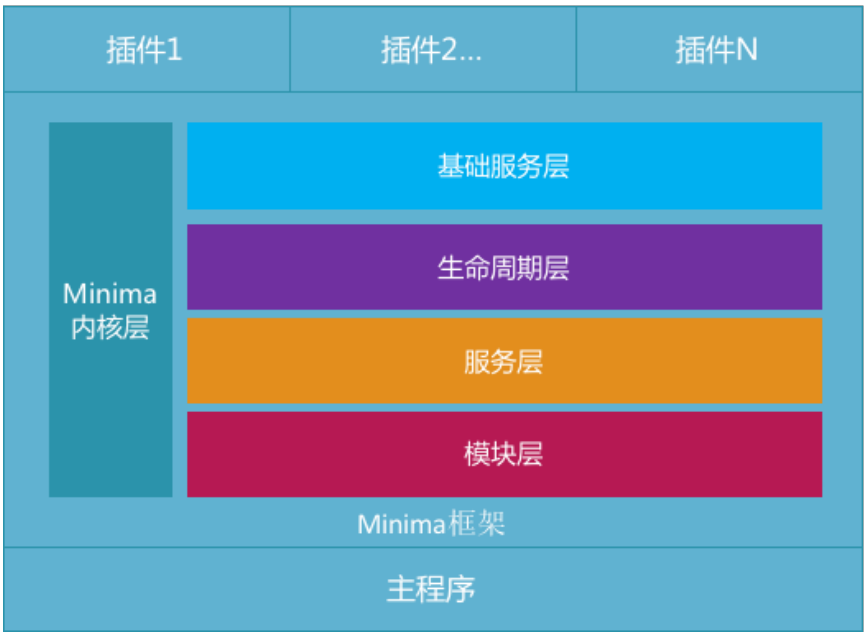
Minima 插件框架设计

设计者：陈贞宝

Table of Contents

1 总体设计

插件框架从总体上参考 OSGi 的设计，分为内核层、模块层、服务层和生命周期层，提供模块物理隔离、动态化特性。该框架基于 ES2017 开发，采用 Babel 进行转码，类加载机制基于 Node 的模块化规范。



主程序负责创建 Minima 框架实例，定义插件目录，实现特定应用的主程序。利用 Minima 加载业务插件进行组装，然后进入应用程序入口。

Minima 框架实现动态模块化、面向服务和模块扩展支持三大核心功能。模块层定义了插件的结构、插件的描述文件、插件依赖关系解析、插件类型空间与类加载机制、插件扩展机制，定义了插件的统一规范。服务层实现了服务的注册、卸载、绑定，实现了模块间基于服务的松耦合通讯。生命周期层实现模块各个生命周期状态定义及转换。基础服务层定义了通用的服务，包括 REST 服务、插件仓库访问服务、Web 模板引擎集成服务等。

最上层是应用插件，互相隔离，每一个插件由配置文件、js 文件、资源文件构成。

2 用户使用场景设计

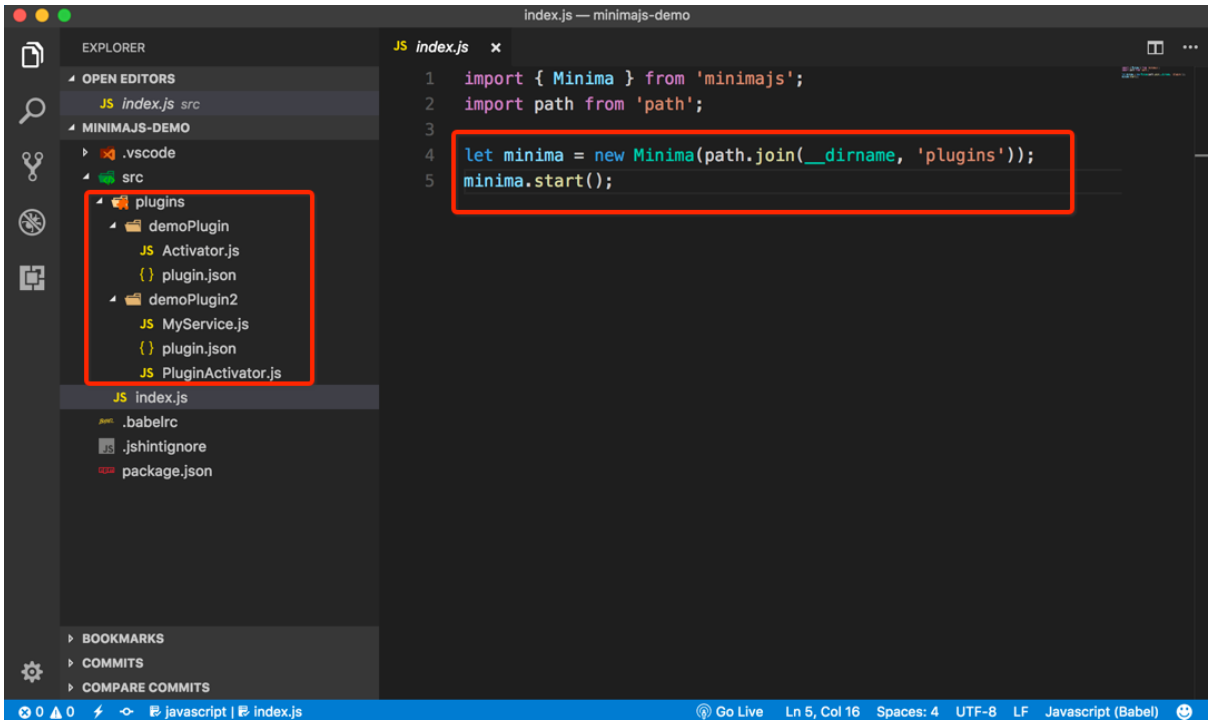
用户在使用框架时，基于 Minima 需要创建主程序和插件。主程序的用途是创建一个 Minima 框架实例，使用该实例加载插件、启动插件，获取插件程序的入口，并运行。

2.1 主程序

主程序的作用是注册全局服务，加载插件框架、启动，并进入程序入口，组合各个插件的功能。典型的主程序包括：index.js、plugins 目录。index.js 文件为主程序，plugins 为插件集合目录。以下小节介绍插件主程序。

2.1.1 典型目录结构

一个典型的插件应用程序目录如下所示。在 src 目录下，index.js 用于启动 Minima 框架，有一个 plugins 目录。每一个插件位于 plugins 目录下，至少包含一个 plugin.json 文件，此外，还会包含 Activator.js 激活器文件，定义插件的入口和出口。



2.1.2 主程序类

主程序实现了插件框架的创建和启动，此外，还要做一些全局的处理，比如 express 框架的创建、全局路由设置等。下图是一个典型的主程序类。主框架调用了 minima 框架，用来加载创建，并获取扩展点。

```
import { Minima, Extension, log } from 'minimajs';

import path from 'path';

import express from 'express';

import favicon from 'serve-favicon';

import cookieParser from 'cookie-parser';
```

```
import bodyParser from 'body-parser';


const app = express();
app.engine('html', require('express-art-template'));
app.engine('art', require('express-art-template'));

app.set('view engine', 'art');
app.set('views', path.join(__dirname, 'views'));
app.set('view options', {
    debug: true
});

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use('/static', express.static(path.join(__dirname, 'static')));

// 创建插件框架，添加全局服务，启动框架
let minima = new Minima(path.join(__dirname, 'test/plugins'));
minima.addService('app', app);
minima.start();

// 获取扩展，并进行处理，保存到 menus 数组
let extensions = minima.getExtensions('minima.menus');

let menus = [];
let id = 0;
for (let extension of extensions) {
    for (let menu of extension.data) {
        menu.id = id;
        menu.url = path.join(extension.owner.pluginDirectory, menu.url);
        menus.push(menu);
        id++;
```

```
    }
}

app.get('/pluginView', function(req, res) {
    let menuId = parseInt(req.query.id);
    let html = '';
    for (let menu of menus) {
        if (menu.id == menuId) {
            html = require(menu.url);
        }
    }

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.write(html);
    res.end();
});

module.exports = app;

app.listen(3000);
```

2.1.3 全局服务

全局服务由主程序在启动插件框架之前，将服务实例添加到框架，这样，各个加载的插件在启动时均可以获取该服务实例。以下为典型的全局服务使用方法。

```
const app = express();
// …… 初始化 app 代码
let minima = new Minima(path.join(__dirname, 'test/plugins'));
// 将 app 作为全局服务注册
minima.addService('app', app);
minima.start();
```

插件可以像使用普通服务一样来获取该服务。

```
/**
```

```
 * 插件入口
 *
 * @param {PluginContext} context 插件上下文
 * @memberof Activator
 */
start(context) {
    let app = context.getDefaultService('app');
    app.use(`/static/${context.plugin.id}`,
        express.static(path.join(context.plugin.pluginDirectory, 'views')));

    log.logger.info(`INFO: The plugin ${context.plugin.id} is started.`);
}
```

2.1.4 全局 Node 库

插件可能会使用到一些第三方 npm 的 node 库。插件如果作为单独的项目，并且最终通过 npm 发布是可以直接运行，如果插件在插件目录下，那么插件依赖的库需要在主程序进行定义，目前这是插件框架的一个限制，对于现有项目是不存在问题，未来可以进一步扩展。目前在插件的 plugin.json 文件有一个保留的配置，后续再考虑进一步支撑。

2.2 插件

2.2.1 插件配置文件

以下示例定义了一个插件配置文件 plugin.json，该文件必须有。该文件声明插件的基本信息、依赖信息、服务、扩展等信息。

```
{
    "id": "demoPlugin",
    "startLevel": 3,
    "version": "1.0.0",
    "extensions": [{
        "id": "minima.menus",
        "data": [{
            "text": "演示菜单",
```

```
            "order": 5,
            "icon": "fa fa-dashboard",
            "url": "renders/DemoViewRender.js"
        }, {
            "text": "系统管理",
            "order": "6",
            "icon": "fa fa-dashboard",
            "menus": [{
                "text": "插件浏览",
                "order": 1,
                "url": "renders/PluginsViewRender.js"
            }, {
                "text": "日志",
                "order": 2,
                "url": "renders/LogsViewRender.js"
            }]
        }]
    }]
}
```

2.2.2 插件激活器

插件激活器为插件的入口和出口。插件入口为 start 方法，参数为插件上下文。在 start 方法中一般用于获取/注册服务、获取扩展、监听变更事件，也可以访问插件框架遍历插件。在激活器会定义一些静态变量，比如将插件、服务、插件上下文作为静态变量定义，这样该插件的其它文件可以直接访问。

```
import path from 'path';
import { Express } from 'express';
import { Extension, ExtensionAction, PluginContext } from 'minimajs';
import DemoRestService from './services/DemoRestService';

export default class Activator {
    /**
     * 插件上下文缓存
```

```
     *
     * @type {PluginContext}
     * @static
     * @memberof Activator
     */
    static context = null;
    /**
     * Express 实例
     *
     * @type {Express}
     * @static
     * @memberof Activator
     */
    static app = null;
    constructor() {
        this.start = this.start.bind(this);
        this.stop = this.stop.bind(this);
        this.extensionChangedListener = this.extensionChangedListener.bind(this);
    }

    /**
     * 插件入口
     *
     * @param {PluginContext} context 插件上下文
     * @memberof Activator
     */
    start(context) {
        Activator.context = context;
        Activator.context.addExtensionChangedListener(
            this.extensionChangedListener);
        Activator.app = context.getDefaultService('app');

        this.demoRestService = new DemoRestService(context.plugin, Activator.app);
```

```
        this.demoRestService.register();
    }

    /**
     * 扩展变更监听器
     *
     * @param {Extension} extension 扩展对象
     * @param {ExtensionAction} action 扩展对象变化活动
     * @memberof Activator
     */
    extensionChangedListener(extension, action) {
        if (action === ExtensionAction.ADDED) {
            // 处理扩展增加
        }
        if (action === ExtensionAction.REMOVED) {
            // 处理扩展删除
        }
    }

    /**
     * 插件出口
     *
     * @param {PluginContext} context 插件上下文
     * @memberof Activator
     */
    stop(context) {
        this.demoRestService.unregister();
        Activator.context = null;
        Activator.app = null;
    }
}
```

2.3 插件开发相关 API

开发插件需要使用的 API 比较少，一般为激活器、插件上下文、插件。

2.3.1 插件激活器

插件激活器可以不定义。插件激活器默认为 Activator.js，在 plugin.json 中进行定义，用于定义插件的入口和出口。以下为一个空的激活器定义。一般情况，start 用于注册/获取服务、获取扩展、监听事件，获取插件的信息等。

```
import { PluginContext } from 'minimajs';


export default class Activator {
    constructor() {
        this.start = this.start.bind(this);
        this.stop = this.stop.bind(this);
    }


    /**
     * 插件入口
     *
     * @param {PluginContext} context 插件上下文
     * @memberof Activator
     */
    start(context) {


    }


    /**
     * 插件出口
     *
     * @param {PluginContext} context 插件上下文
     * @memberof Activator
     */
    stop(context) {
```

```
    }
}
```

2.3.2 PluginContxt

插件上下文提供了如下方法：

（1）服务：注册、获取服务，addService, getServices, getDefaultService；

（2）扩展：获取扩展点，getExtension；

（3）安装插件：动态安装一个插件，installPlugin；

（4）事件监听：框架状态变更、服务变化、扩展变化、插件状态变化；

（5）插件访问：获取当前插件，获取框架安装的插件 getPlugin, getPlugins。

2.3.3 Plugin

插件实例提供了对当前插件的访问，获取插件基本信息，包括 id、目录、状态、版本等，还可以从当前插件加载类型。

2.4 智能提示支持

智能提示支持基于 JSDoc 规范进行定义，需要严格按照该规范进行声明才能够使 IDE 对类具备智能提示支持。

（1）在注释时，指定具体的类或者类数组。这里指定具体类型、类型数组、Set.<类>、Map.<key 类型，value 类型>方式来执行，注意，在当前文件需要 import 相关的类型。

```
/**
 * 获取匹配的所有服务实例，注意，返回的服务没有了Properties，不合适。
 *
 * @param {string} name 服务名称
 * @param {Object} properties 服务过滤属性
 * @returns {ServiceRegistry[]} 服务实例的集合
 * @memberof Minima
 */
getServices(name, properties) {
    return this._framework.serviceManager.findServices(name, properties);
}
```

（2）通过 type 来声明变量。

```
import { ServiceAction, Extension, ExtensionAction, PluginContext, log } from 'minimajs';
import path from 'path';
import express from 'express';

export default class Activator {
    /**
     * 插件上下文缓存
     *
     * @type {PluginContext}
     * @static
     * @memberof Activator
     */
    static context = null;
    constructor() {
        this.start = this.start.bind(this);
        this.stop = this.stop.bind(this);
        this.serviceChangedListener = this.serviceChangedListener.bind(this);
        this.extensionChangedListener = this.extensionChangedListener.bind(this);
    }
```

```
/**
 * 创建一个扩展管理器
 *
 * @ignore
 * @param {Framework} framework 框架实例
 * @memberof ExtensionManager
 */
constructor(framework) {
    Assert.notNull('framework', framework);
    this.framework = framework;
    /**
     * @type {Map.<string, Extension>}
     */
    this.extensions = new Map();

    this.dispose = this.dispose.bind(this);

    this.add = this.add.bind(this);
    this.remove = this.remove.bind(this);
    this.removeByOwner = this.removeByOwner.bind(this);
    this.find = this.find.bind(this);
}
```

3 内核层设计

内核负责插件的加载、插件依赖解析、插件启动，提供了统一的事件管理器。

3.1 插件加载

内核层访问指定的插件目录，从插件集合目录按以下规则加载插件：
（1）插件集合目录下，遍历所有的子目录，如果子目录包含 plugin.json，就认为子目录是一个插件，这个子目录也称为插件目录；
（2）插件框架从子目录加载插件的任务是加载 plugin.json 并进行校验，生成上层要使用的 PluginConfiguration 实例，会自动添加默认值；完成后，插件加载即成功。

3.2 插件解析

内核层加载完成插件后，会采用标记法对所有加载的插件进行依赖解析，并支持插件增加、删除的动态解析。插件依赖关系用于解决插件启动时，需要另一个插件的支持情况。依赖关系用于，在内核启动插件时，会根据启动级别、依赖关系进行启动。

4 模块层设计

4.1 插件结构

每一个插件由以下文件构成：
（1）plugin.json：这是插件必须提供的文件，定义了插件的详细信息、插件依赖关系、插件服务、插件扩展等信息。
（2）**Activator.js：插件激活器，这是可选的文件，默认名称为 Activator.js，定义了插件的入口和出口，用于实现资源申请、服务注册与引用等。

4.2 插件清单文件

插件清单文件为 plugin.json 文件，它包含了基本信息定义、依赖关系定义、服务定义、扩展点定义。
基本信息定义包括如下：
（1）插件标识——id，该属性为必填项，不可以重复。
（2）插件名称——name，该属性为选填，如果不填写，默认与 id 相同。

（3）插件描述——description，该属性为选填，如果不填写，默认与 id 相同。
（4）插件版本——version，该属性为选填，默认为 0.0.0。
（5）启动级别——startLevel，表示插件启动顺序，该值越小，越早启动。默认为 50。
（6）初始状态——initializedState，表示插件被框架加载后，默认进入的状态，默认为 active。
（7）插件激活器——activator，该属性为选填，定义插件的入口和出口。如果没有指定，则会尝试去加载 Activator.js，如果没有找到激活器，则直接过度到启动状态。如果加载到，则启动时调用 start 方法、停止时调用 stop。
（8）是否允许停止——stoppable，表示插件被框架加载后，是否允许调用 stop 方法停止插件，默认为 true。

插件依赖关系定义通过 dependencies 来定义，该属性的值是一个数组，每一个数组元素即定义插件的依赖关系，由 id、version 组成，表示依赖插件的标识和版本，版本默认为 0.0.0。依赖解析时只要发现插件版本大于指定版本，则满足依赖关系。

服务定义了插件启动时自动注册的服务，也可以通过激活器自动定义。服务是一个数组，每一个服务的定义包含 name、service、properties 属性，分别表示服务名称、服务类型所在文件（注意，服务类使用 default 进行导出）、properties 表示服务属性，服务属性用于服务的过滤查询。默认会包含服务的插件信息。

扩展信息由 extensions 定义，它是一个数组，每一个元素由 id 和 data 构成，id 表示扩展的标识，data 表示扩展数据，data 为 json 对象。

```
{
    "id": "demoPlugin2",
    "name": "demoPlugin2Test",
    "description": "The demo plugin2.",
    "version": "1.0.1",
    "startLevel": 5,
    "initializedState": "active",
    "activator": "PluginActivator.js",
    "stoppable": true,
    "dependencies": [{
```

```
        "id": "demoPlugin",
        "version": "1.0.0"
    }],
    "services": [{
        "name": "myService",
        "service": "MyService.js",
        "properties": {
            "vendor": "lorry"
        }
    }],
    "extensions": [{
        "id": "myExtension",
        "data": {
            "extensionData": "lorry"
        }
    }, {
        "id": "myExtension2",
        "data": {
            "extensionData": "lorry2"
        }
    }, {
        "id": "minima.menus",
        "data": [{
            "url": "view2.js",
            "text": "view2"
        }]
    }]
}
```

4.3 插件激活器

插件激活器定义插件启动、停止的行为，有 start、stop 方法。插件可以没有激活器，或者默认为 Activator.js 文件作为激活器。下面是激活器的典型定义方法。

```
import { ServiceAction, Extension, ExtensionAction, PluginContext, log } from
'minimajs';
import path from 'path';
import express from 'express';

export default class Activator {
    /**
     * 插件上下文缓存
     *
     * @type {PluginContext}
     * @static
     * @memberof Activator
     */
    static context = null;
    constructor() {
        this.start = this.start.bind(this);
        this.stop = this.stop.bind(this);
        this.serviceChangedListener = this.serviceChangedListener.bind(this);
        this.extensionChangedListener = this.extensionChangedListener.bind(this);
    }

    /**
     * 插件入口
     *
     * @param {PluginContext} context 插件上下文
     * @memberof Activator
     */
    start(context) {
        Activator.context = context;
        Activator.context.addServiceChangedListener(this.serviceChangedListener);
        Activator.context.addExtensionChangedListener(
            this.extensionChangedListener);
```

```
        let app = context.getDefaultService('app');
        app.use(`/static/${context.plugin.id}`,
            express.static(path.join(context.plugin.pluginDirectory, 'views')));

        log.logger.info(`INFO: The plugin ${context.plugin.id} is started.`);
    }

    /**
     * 服务监听器
     *
     * @param {string} name 服务名称
     * @param {ServiceAction} action 服务变化活动
     * @memberof Activator
     */
    serviceChangedListener(name, action) {
        if (name === 'myService' && action === ServiceAction.ADDED) {
            let myService = Activator.context.getDefaultService(name);
            if (myService) {
                log.logger.info(`Get the myService instance successfully.`);
            }
        } else if (action === ServiceAction.REMOVED) {
            log.logger.info(`The service ${name} is removed.`);
        }
    }

    /**
     * 扩展变更监听器
     *
     * @param {Extension} extension 扩展对象
     * @param {ExtensionAction} action 扩展对象变化活动
     * @memberof Activator
     */
    extensionChangedListener(extension, action) {
```

```
        if (action === ExtensionAction.ADDED) {
            log.logger.info(`The extension ${extension.id} is added.`);
            let extensions = Activator.context.getExtensions('myExtension');
            log.logger.info(`The extension count is ${extensions.size}.`);
        }

        if (action === ExtensionAction.REMOVED) {
            log.logger.info(`The extension ${extension.id} is removed.`);
        }
    }

    /**
     * 插件出口
     *
     * @param {PluginContext} context 插件上下文
     * @memberof Activator
     */
    stop(context) {
        Activator.context = null;
        log.logger.info(`INFO: The plugin ${context.plugin.id} is stopped.`);
    }
}
```

4.4 插件依赖关系与解析

插件依赖关系影响插件是否可以启动、插件启动顺序。插件框架在启动时，会加载、解析依赖、启动那些初始状态为 active 的插件。依赖关系，即一个插件在运行时会使用到另一个插件定于的类型。框架在启动阶段，会将依赖关系对应的插件先启动，这意味着，插件启动首先依据启动级别，其次，根据依赖关系。

依赖解析算法采用标记法，即假设所有插件均可以解析，然后开始逐遍扫描插件，如果发现插件依赖缺失或者依赖的插件已经解析失败，则标记该插件解析失败，并且需要再次下一轮扫描直到没有发现解析失败的插件。

依赖解析算法描述如下：
（1）插件元数据描述为：pluginMetadata {id, version, resolvable（可以解析）或者 resolveSuccess（如果依赖关系为空，则解析成功）, dependencies = [{id, version}]}，即每一个插件依赖解析元数据为标识、版本、依赖列表；
（2）建立一个插件元数据队列，比如[a, b, c, d, e, f]；
（3）定义插件解析失败：依赖插件不存在或者依赖插件解析失败；
（4）从头到位，检查插件，如果插件 resolvable，分析插件是否解析失败，如果失败则修改 resolveFailed；
（5）重复步骤（4），如果没有再发现 resolvable 插件解析失败，则转到（6）；
（6）将所有剩下的 resolvable 插件的状态修改为 resolveSuccess。

增加一个元素：
（1）元素初始化为：resolvable；
（2）将 resolveFailed 的元素修改为 resolvable；
（3）重新解析。

删除一个元素，暂时不考虑，因此插件卸载时，不会直接删除，因为直接删除的话，需要停止掉那些依赖它的插件，过于复杂。
（1）如果该元素为 resolveFailed，则直接返回；
（2）如果该元素为 resolveSuccess，则需要重新标记并进行全部解析。

插件解析由 PluginMetadata、DependencyContraint、PluginResolver 三个类进行协作。PluginMetadata 称为解析元数据，包括插件 id、version、dependencyContraints（依赖约束的集合）、dependencyChain（依赖的插件链）、state（解析状态，解析成功、解析失败、可解析）。

4.5 插件类型空间插件类加载机制

插件可以从本地加载插件，也可以通过其它插件加载插件。在该框架，插件类空间为当前插件。从插件加载类型可以使用 Plugin 类，该类通过激活器 context.plugin 获得。
context.plugin.loadClass(‘index.js’)

我们可以通过 context 获取其它插件，加载类型。

4.6 扩展机制

插件的扩展机制基于 extension 模型。每一个插件都可以定义 extensions。

4.6.1 扩展

以下定义演示了扩展定义，这里定义了一个 minima.menus 的扩展，其数据格式是一个数组，数组里面的元素包含了 url、text 属性。

```
{
    "id": "demoPlugin",
    "startLevel": 3,
    "version": "1.0.0",
    "extensions": [{
        "id": "minima.menus",
        "data": [{
            "url": "view1render.js",
            "text": "view1"
        }]
    }]
}
```

处理该扩展的插件，可以通过以下方式来获取扩展信息，并处理。

```
let minima = new Minima(path.join(__dirname, 'test/plugins'));
minima.addService('app', app);
minima.start();

let extensions = minima.getExtensions('minima.menus');

let menus = [];
let id = 0;
for (let extension of extensions) {
    for (let menu of extension.data) {
        menu.id = id;
```

```
        menu.url = path.join(extension.owner.pluginDirectory, menu.url);
        menus.push(menu);
        id++;
    }
}

app.get('/pluginView', function(req, res) {
    let menuId = parseInt(req.query.id);
    let html = '';
    for (let menu of menus) {
        if (menu.id == menuId) {
            html = require(menu.url);
        }
    }

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.write(html);
    res.end();
});
```

context 也提供了扩展处理的支持，即插件也可以处理扩展。

4.6.2 扩展事件

我们可以通过以下方式来监听扩展事件，并进行扩展处理。

```
context.addExtensionChangedListener(this.extensionChangedListener);
```

处理函数如下所示。

```
    /**
    * 扩展变更监听器
    *
    * @param {Extension} extension 扩展对象
    * @param {ExtensionAction} action 扩展对象变化活动
    * @memberof Activator
    */
```

```
    extensionChangedListener(extension, action) {
        if (action === ExtensionAction.ADDED) {
            log.logger.info(`The extension ${extension.id} is added.`);
            let extensions = Activator.context.getExtensions('myExtension');
            log.logger.info(`The extension count is ${extensions.size}.`);
        }

        if (action === ExtensionAction.REMOVED) {
            log.logger.info(`The extension ${extension.id} is removed.`);
        }
    }
```

5 服务层设计

框架提供了 Minima、PluginContext 两个类用于注册服务。前者用于注册全局服务、后者由插件注册。全局服务一般在插件框架启动前，这样所有插件均可以直接访问；后者由插件注册特定服务，由于插件启动有先后顺序，因此可能服务获取不到。

这两个类提供了如下服务 API：
（1）注册：addService(name, service, properties)，注册指定名称的服务，包含服务属性，服务属性框架会默认将插件 id、插件版本等信息添加到属性，用于服务过滤使用；
（2）服务获取：getDefaultService(name)获取一个默认服务；getServices(filter)获取所有服务，此时返回服务过滤的服务，filter 用于匹配服务的属性，此时范围 ServiceRegistry 数组，用于遍历多个服务，进行特定匹配；
（3）服务卸载：removeService(name)卸载一个服务。

这两个类提供了服务变更事件的处理，其 API 如下：
（1）注册事件：addServiceChangedListener(listener)，listener 是一个函数，有两个参数，为服务名称和服务变化活动；
（2）卸载事件：removeServiceChangedListener(listener)。

6 生命周期层设计

6.1 框架插件启动顺序

插件框架启动插件以如下规则进行执行：
（1）在 plugin.json 中，所有 initializeState=’active’或者不指定（默认值）的插件，会被框架在启动时进行插件启动；
（2）框架启动插件以 startLevel 排序、插件字母排序，启动级别越小，越早启动；
（3）在启动插件时，会先启动插件的依赖关系，即使插件依赖关系链的插件启动级别大于当前插件。位于依赖链的末尾，最先执行启动。

6.2 插件状态与类加载

插件解析成功后，我们默认就可以从插件加载类型。加载类型，目前不会去触发插件启动。如果加载的类型依赖于服务，服务所在插件没有启动安装，那么类型加载依然成功，加载类型需要处理服务的动态性。

6.3 安装

插件框架会在启动时，从插件目录自动安装插件，安装成功后，会执行解析。安装插件时，会校验 plugin.json 文件的必填项、补充默认值。
PluginContext 也提供了动态安装插件的 API，即 installPlugin(pluginDirectory)，该 API 安装步骤同框架，校验、补充默认值、解析插件依赖关系。

6.4 启动

插件启动时机有两个，一个是通过插件框架，在 6.1 描述。另一个可以通过 Plugin 的 API 进行启动。Plugin 提供了 start 方法，用于执行启动，单独插件启动和停止，不会对依赖关系进行操作。启动顺序如下：
（1）如果已经启动，则返回；
（2）如果已经卸载，则抛出异常；
（3）如果插件启动级别大于框架启动级别，则抛出异常；
（4）解析插件，如果解析失败，则抛出异常；
（5）将状态更改为 starting；
（6）创建插件上下文；

（7）尝试加载激活器。首先，如果 plugin.json 指定了激活器文件，则从指定位置加载；如果没有指定，则判断插件是否存在 Activator.js 的默认激活器文件，如果存在则加载；如果没有指定也没有存在默认，则忽略激活器的创建和启动；创建和启动激活器时，如果出现异常，则启动失败，抛出异常；
（8）启动插件上下文，注册服务、注册扩展；
（9）更改状态为 active。

6.5 停止

插件框架停止时，会按照启动相反的顺序停止插件。此外，Plugin 的 stop 方法提供了插件停止的方法。插件停止步骤如下：
（1）判断是否卸载，如果卸载，则抛出异常；
（2）判断状态是否为 active，如果不是，则返回；
（3）更改状态为 stopping；
（4）如果有激活器，则调用激活器的 stop 方法，如果激活器抛出异常，则记录异常信息，但插件停止继续运行；
（5）停止插件上下文，卸载服务、卸载扩展、注销监听器（这三个步骤如果在激活器 stop 方法没有执行，也会自动注销插件的所有资源）；
（6）状态更改为 resolved。

6.6 卸载

插件卸载，首先判断是否卸载，如果卸载则抛出异常，然后，先执行停止，紧接着，状态修改为 uninstalled。

6.7 生命周期事件

框架 Minima 和 PluginContext 都提供了插件的状态更改监听，他们提供了如下的两个 API：
（1）addPluginStateChangedListener(listener)：注册监听器，监听器格式为 function(id, previous, current)；
（2）removePluginStateChangedListener：注销监听器。

7 插件仓库与自动升级设计

插件以 zip 文件存储在插件仓库。插件仓库使用 JSON 来描述存储的插件、插件各个版本、插件依赖关系。

自动升级基于插件仓库暴露的插件仓库 API，可以查询相应的插件最新版本并下载安装。