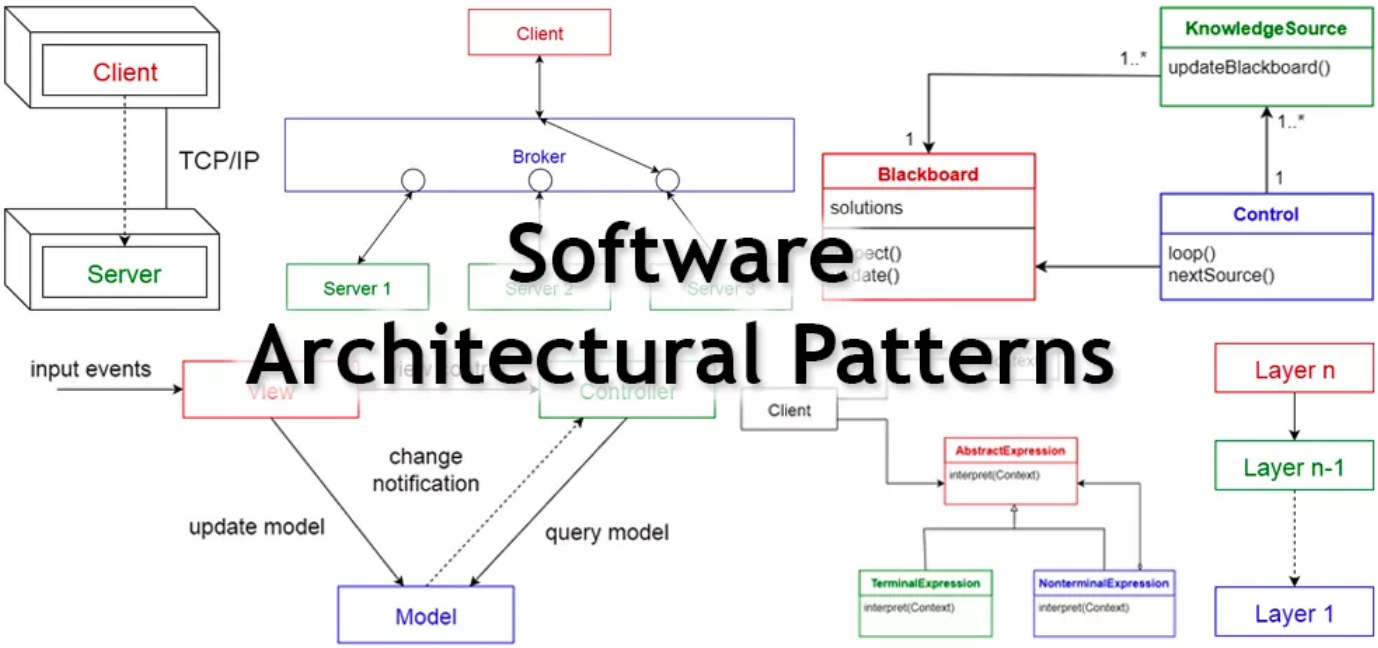
最经典的两种软件架构模式

知识小集　今天

以下文章来源于有赞coder ，作者有赞技术



作者：kk

团队：业务中台前端团队

什么是架构模式?

根据维基百科中的定义:

> An architecture pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context.

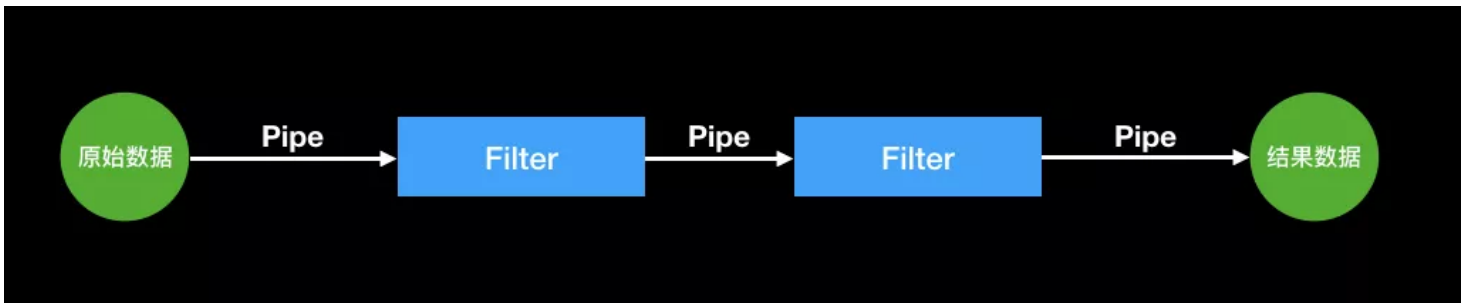
那么，在软件研发领域，最经典的两种架构设计模式，即微内核架构模式和 Pipe-Filter 架构模式，下面我们就来聊一聊这两种架构模式。

一、Pipe-Filter 架构模式

Pipe-Filter 模式，即管道过滤器模式，这是一种非常经典的架构模式，这种模式与工业制造生产流水线非常类似，就像薯片的生产过程，从土豆的清洗、去皮、切片、

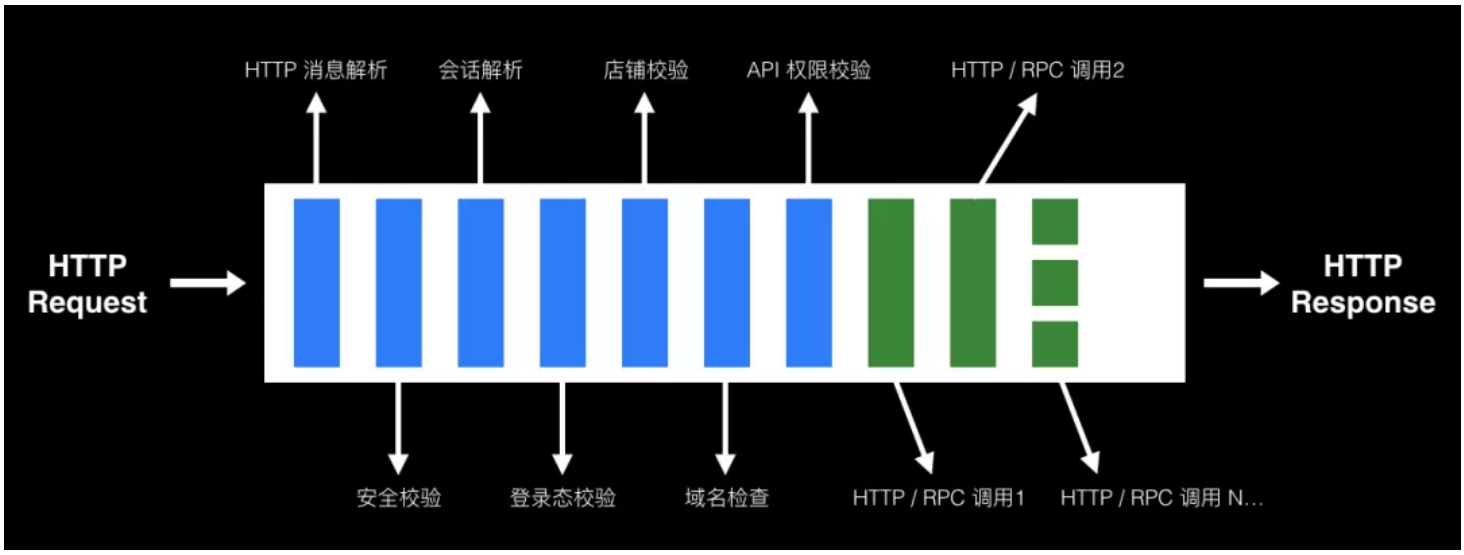
烘干、油炸，到最后打包完成，整个生产过程被拆分成了多个环节，每个环节处理完成之后，通过传送带传送到下一个环节的机器。整个生产过程每个环节都是独立的，但又环环相扣，只要有一个环节出问题了，生产出来的薯片就会有质量问题。

那么，对应 Pipe-Filter 管道过滤器模式，管道就像生产流水线上的传送带，过滤器就像每一道工序上的机器。管道负责数据的传递，原始数据通过管道传送给第一个过滤器，第一个过滤器处理完成之后，再通过管道把处理结果传送给下一个过滤器，重复这个过程直到处理结束，最终得到需要的结果数据，用一幅图来形象描述这个过程，如下图所示：



1.1 适用的场景

那么，Pipe-Filter 管道过滤器模式有哪些适用的场景呢？第一个就是 Web 框架，像 Koa 洋葱模型其实就是参照 Pipe-Filter 架构模式设计的。在软件研发领域，Web 系统可以说是最常见的一类系统了，生活中我们每天都在使用这样的系统，从社交工具微信、QQ，到购物网站淘宝、京东，生活中我们无时无刻不在使用这样的系统。那么，设计这样的一个系统，Pipe-Filter 模式又有哪些应用案例呢？以有赞商家管理后台为例（流程已精简），当一个用户访问我们的商家后台时，会经过如下的处理环节。



首先，一个 Web 系统肯定是基于 HTTP 协议做的，我们都知道，HTTP 协议是通过文本来传递信息的，所以第

一步我们要做的就是解析 HTTP 消息，包括 HTTP 消息头及消息体。接着，我们需要对接收的数据做一次安全校验，常见的安全校验包括 XSS 过滤、CSRF Token 校验等。再接着，我们需要识别当前访问的用户是谁，一般会拆分成会话解析及登录态校验，会话解析只负责解析会话数据，而会话数据一般都是存放在缓存系统，像 redis、kv 等，登录态校验则会从会话数据中取出用户信息，如果用户信息不存在，说明用户未登录，那么，服务端会返回一个重定向响应，浏览器会根据响应 header 头里面的 location 字段，跳转到 location 指定的登录页地址，之后的流程就不再详细解释了。

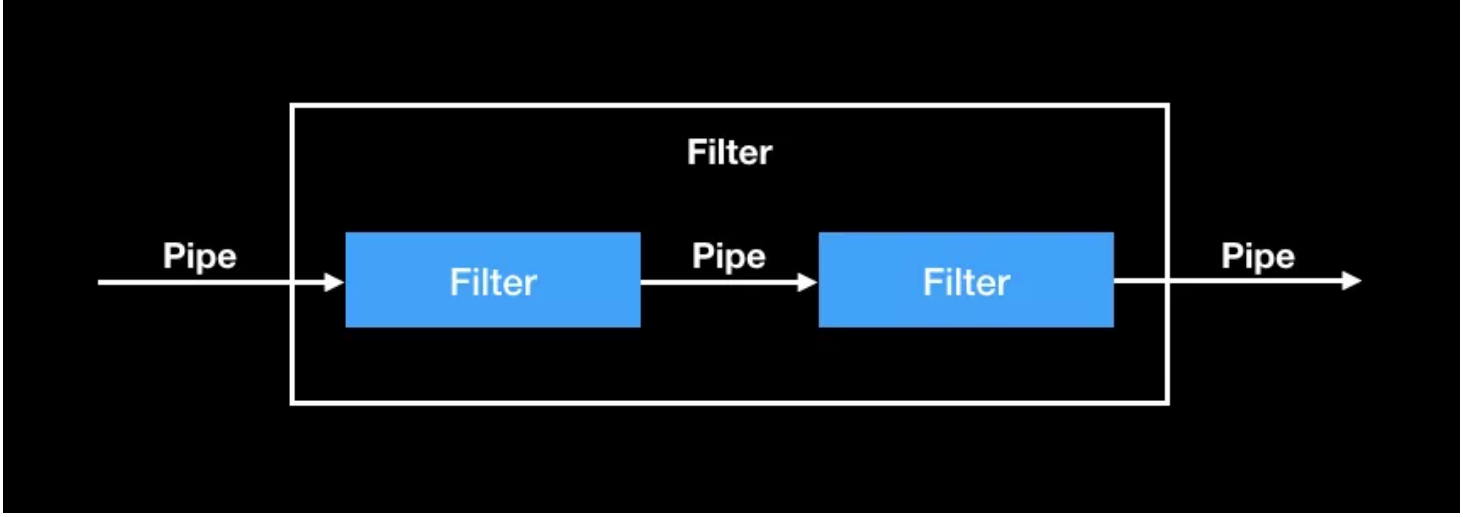
在实际的系统中，并不会把上面说的这些功能模块都整合到一个系统里面，一般都会拆分成多个系统，每个系统都只负责其中一个或几个功能模块，例如上面说的安全校验，一般稍微大一点的公司都会有专门的安全团队，安全团队会搭建一整套 Web 安全防火墙，俗称 WAF，所有的请求都会先经过这一道防火墙，如果发现请求存在安全风险，则会直接拒绝请求或对请求的内容做一次过滤。例

如，请求中如果带有 XSS 攻击脚本，则会将请求的脚本做一次过滤。

除了 Web 系统，其他像大数据数据处理与分析系统、编译器、Linux 管道命令等等，也大量使用了 Pipe-Filter 架构模式，这里就不再详细解析了，有兴趣的同学可以自行研究。

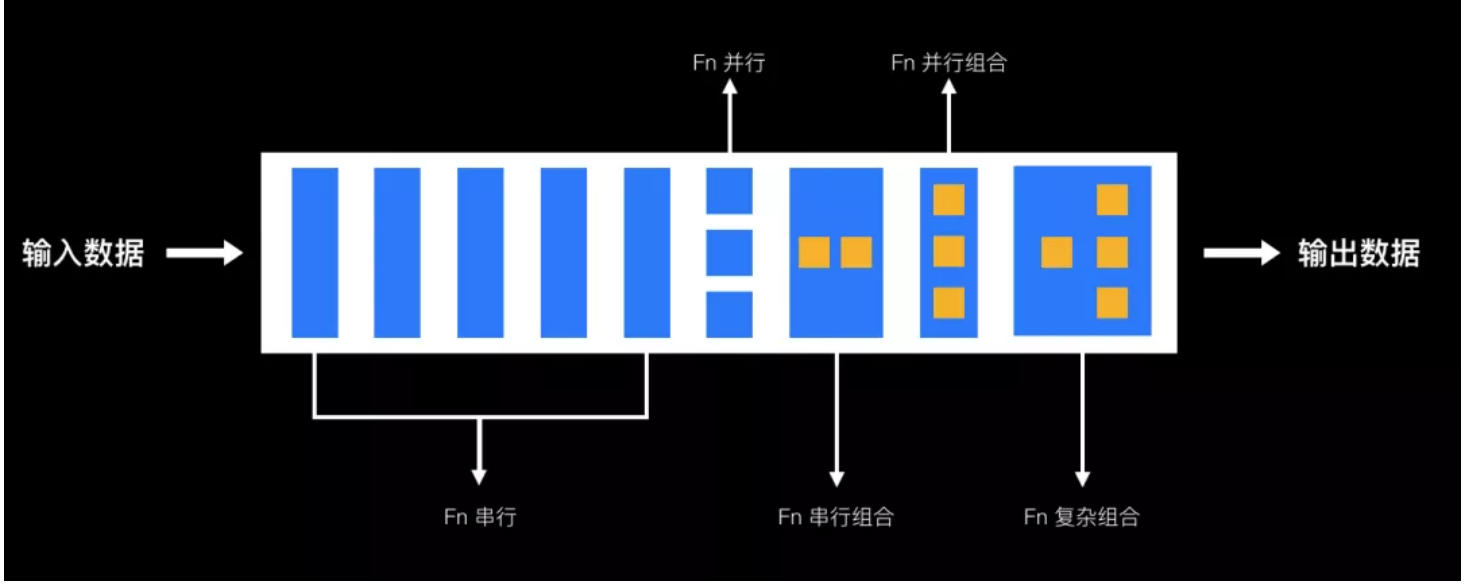
1.2 Filter 和组合模式

23 个经典设计模式里面有一个设计模式叫组合模式，当 Pipe-Filter 遇上组合模式时，多个 Filter 又可以再组合成一个新的 Filter，如下图所示，组合出来的 Filter 接收的数据与第一个 Filter 保持一致，返回的数据与最后一个 Filter 保持一致。通过组合，就可以将多个简单的 Filter 可以组合成一个更复杂的 Filter。应用这一套理论去实践，我们会发现，Filter 既可以做的很轻便，也可以做得很强大。



1.3 Serverless 和 FaaS

Filter 和组合模式又有哪些应用案例呢？在当下，有个非常火的系统架构叫 serverless 架构（即无服务架构），serveless 无服务架构可以显著降低企业中中长尾应用的成本，中长尾应用指那些每天大部分时间都没有流量或者有很少流量的应用。要实现这样的无服务架构，FaaS 便是其中一个非常核心的组件，FaaS 是 Function as a Service 的简称，FaaS 中有一个核心组件叫 Fn Actuator，负责 Fn 函数的加载（可以是热加载，也可以是冷启动）、调度、执行。如下图所示：



Fn 执行方式有哪些?

- Fn 串行：多个 Fn 可以串行执行，每个 Fn 的执行结果会传递给下一个 Fn，作为下一个 Fn 的输入数据；
- Fn 并行：多个 Fn 可并行执行，当所有的 Fn 都执行完成之后，将这多个 Fn 的执行结果封装成一个数组传递给下一个 Fn；
- Fn 串行组合：多个 Fn 串行执行，并对外封装成一个新的 Fn，新的 Fn 入参与第一个 Fn 保持一致，返回值与最后一个 Fn 保持一致。这样封装有个好处，新的 Fn - 对使用的人来说，就是一个 Fn，只需关注入参和返回值，而无需关注内部的实现；

- Fn 并行组合：多个 Fn 并行执行，并对外封装成一个新的 Fn，新的 Fn 可重新定义入参和返回值；
- Fn 复杂组合：上述方式的自由组合；

Fn 可以是哪些类型?

- 纯函数：就是一个最简单的函数
- 远程服务调用：可以将一个 HTTP 或 RPC 调用封装成一个函数
- 脚本：可以将一个 SQL 或 shell 语句封装成一个函数

总之，在封装一个 Fn 的时候，需要注意以下几点：

- 每个 Fn 只处理一件事情；
- 每个 Fn 的入参和返回值都必须显示声明；
- 每个 Fn 内部不能直接读取或修改外部的全局变量；

二、微内核架构模式

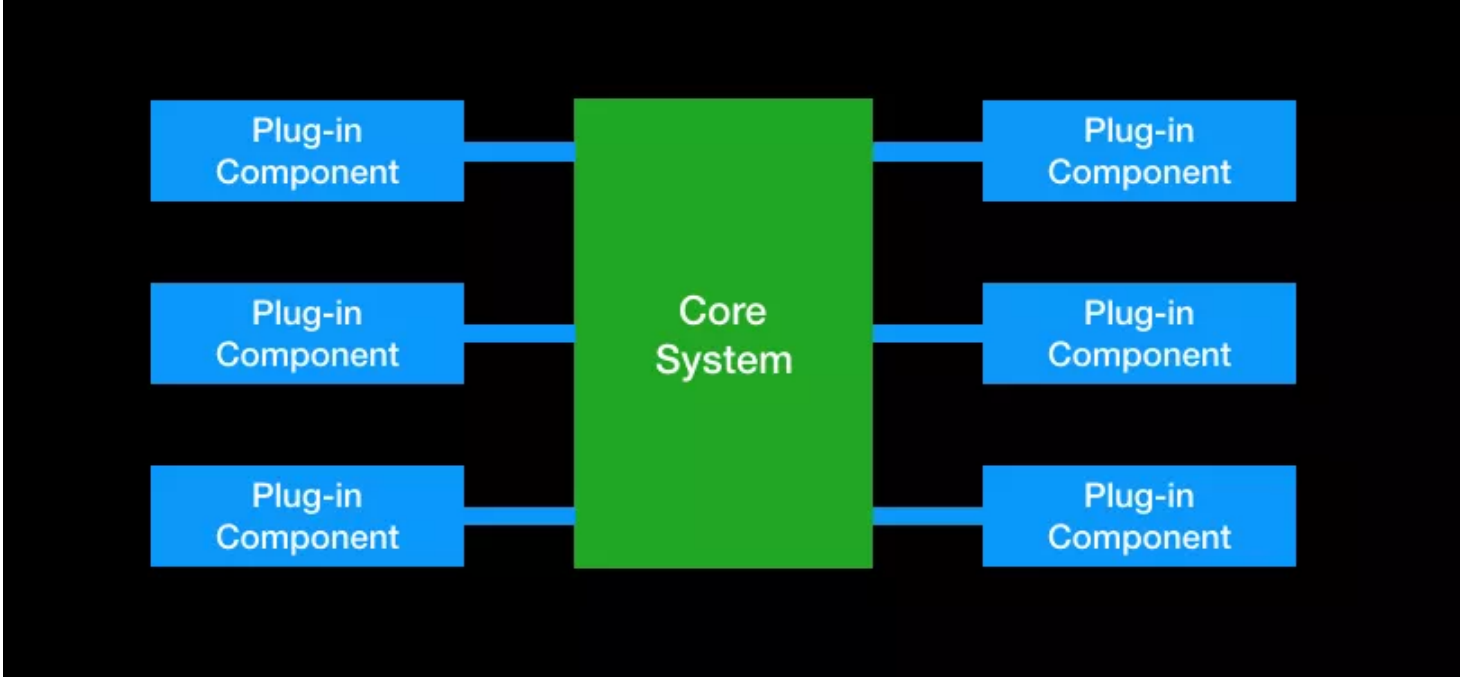
2.1 什么是微内核架构?

微内核架构（Microkernel Architecture），也被称为插件化架构（Plugin-in Architecture），是一种面向功能进行拆分的可扩展架构。例如 VS Code、Eclipse 这一类 IDE 软件、UNIX 操作系统等等，都是参照微内核架构设计实现的。

2.2 微内核架构的两个核心组件

微内核架构包含两类核心的组件：核心系统（Core System）和插件模块（Plug-in modules）。核心系统负责与具体功能无关的通用功能，例如应用生命周期的管理、插件模块的管理（包括：插件模块的注册、载入、卸载等等）；插件模块负责实现具体的功能，例如一个 Web 框架基本上会按照功能模块拆分成如下的插件模块：路由模块、安全模块、HTTP 编解码模块等等，每个模块都通过插件实现，每一个插件都只做一件事情。

微内核基本架构示意图如下所示：



核心系统功能尽量保持稳定，不要因为插件模块的扩展而不断修改，插件模块可以根据功能需求进行不断扩展。

2.3 核心系统设计的几个关键点

插件管理

- 插件安装：各种编程语言的生态都有提供类似 maven、npm 这样的包管理工具，所以一般无需自己再造轮子，例如 Node 通过 npm install 命令即可，例如：npm install [<@scope>/]@;
- 启用插件：安装了一个插件仅代表这个系统依赖了这个插件的功能，但要想插件真正生效，还需要启用插件，启用插件一般是通过配置文件来进行声

明。注意，在实现该功能模块的时候，需要考虑多环境（一般分：开发环境、测试环境、预发环境、生产环境等），插件可动态调整的配置一般也放到插件的配置文件里面；

- 插件通信：插件通信指的是指插件之间的通信，虽然设计插件的时候插件之间是完全解耦的，但实际运行的过程中，必然会出现多个插件需要协作完成某个功能，这时候就需要支持插件间的通信了，因此核心系统需要提供一套插件通信机制。有一个需要注意的地方，插件与插件之间最好不要直接通信，插件通信都通过核心系统来完成；
- 禁用插件：禁用插件就很简单了，一般通过插件的配置文件，将插件 enable 状态设置成 false 即可；
- 卸载插件：卸载插件也很简单，一般各个包管理工具都有提供类似命令，例如：npm uninstall [<@scope>/][@]

应用管理

不管你开发的是一个客户端 APP 软件，还是一个服务端的 Web 系统，在应用启动的时候，肯定会有一个主进

程，主进程设计的时候一般都比较轻量，这样更不容易出bug，插件一般是通过子进程来实现，这样即使子进程挂了，也不会影响主进程。像早期浏览器都是单进程的，浏览器的各种插件诸如 Web 播放器都是运行在浏览器同一个主进程之上的，所以经常会碰到一个插件奔溃了导致整个浏览器奔溃。现代的浏览器早已改变了这种实现方式，最新的 Chrome 浏览器就会将进程拆分成 Browser Process 、 Render Process 、 GPU Process 、 Network Process、Plugin Process 五类进程，其中 Browser Process 浏览器进程就是负责各个子进程的管理。

总结下，核心系统在实现应用管理模块功能的时候，其实就主要实现下面两块功能：

- 应用本身的生命周期管理
- 插件生命周期的管理

2.4 插件模块设计的几个关键点

插件元数据定义

包括插件的名称、描述、版本号等等，插件的命名从一开始就要制定好规范，这样方便后期管理。像 Spring Boot 很多插件都是以 spring-boot-starter- 开头命名的。

插件要可插拔、可配置

插件应该是灵活可插拔的，当不需要一个插件的时候，可随时将这个插件卸载掉。插件中可配置的属性需要暴露到外界使用环境，实现业务可定制。

每个插件都应该保持职责单一性

系统复杂度上升很多时候是因为模块拆分不合理导致的，一个插件其实就是实现了某一个功能模块，所以每个插件都要尽量保持职责单一性。

2.5 微内核架构应用案例

- VS Code：VS code 内核是非常轻量的，只提供最基础的能力，其他功能都是通过一个个插件来扩展；
- 规则引擎：规则引擎也属于微内核架构的一种实现，其中执行引擎可以看做是微内核，执行引擎通过解析、执行配置规则，来达到业务的灵活多变
- Koa Web 框架：框架本身只提供最基础的能力，其他都是通过一个个中间件来扩展，中间件既是一个插件，也是一个 Filter，可以说是微内核架构与 Pipe-Filter 架构的完美融合。

其他还有很多应用案例，因篇幅所限，就不再详细介绍，有兴趣的小伙伴可自行研究。

最后，这两种架构模式到这里就讲完了，其实不管是什么类型的架构模式，都是围绕着下面的这些问题来解决的：

- 如何降低系统的复杂度
- 如何提高系统的可维护性
- 如何提高系统的可扩展性
- 如何提高系统的可配置性

但最终还是为了解决现实中的问题，把一个大的问题拆分成一个个小的问题，再通过某种方式把这些功能聚合在一起，进而达到解决这个大问题的目的。就像 MapReduce 算法模型，其实也是这样一个思路，当一个大的计算任务没办法通过一台计算机计算得到结果，那就先把这个大的计算任务拆分成一个个小的计算任务，然后把这些小的计算任务交给一个个独立的计算节点，最终再把每个子任务计算得到的结果汇总起来，然后得到一个最终的结果，正所谓“分久必合合久必分”。

扩展阅读

1. 使用 Puppeteer 搭建统一海报渲染服务
2. Vant 2.0 发布：持之以恒，不乱节奏
3. React 中 getDerivedStateFromProps 的三个场景
4. 用函数式的方式思考——递归
5. Vant Weapp 1.0 正式版发布

6. 如何搭建一个高可用的服务端渲染工程

7. 有赞美业店铺装修前端解决方案

阅读原文