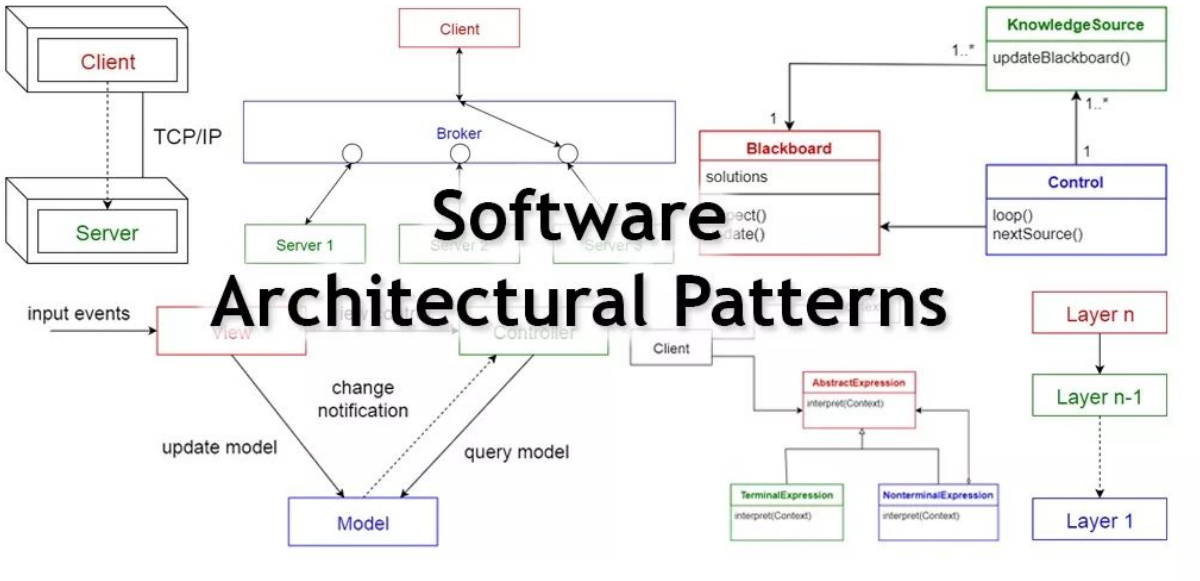
10 个常用的软件架构模式

原创：伯乐在线 Linux爱好者 5月13日

> 编译：伯乐在线/Alick

你是否曾经思考过如何设计大型的企业级系统？在决定启动软件开发之前，首要的是选择恰当的架构来指引系统的功能及质量属性设计。因此在将软件架构应用于设计之前，必需要了解常用的架构模式。



什么是架构模式?

Wikipedia 的解释：

> 在软件架构中，架构模式是对特定环境下常见问题的通用且可重用的解决方案。架构模式与软件设计模式很相似，但架构模式的层次更高，且外延更大。

这篇文章将简述常见的 10 种架构模式的概念、用法以及其优缺点。

1. 分层模式(Layered pattern)

2. 客户端 / 服务器模式(Client-server pattern)
3. 主 / 从模式(Master-slave pattern)
4. 管道 / 过滤器模式(Pipe-filter pattern)
5. 代理模式(Broker pattern)
6. 对等模式(Peer-to-peer pattern)
7. 事件总线模式(Event-bus pattern)
8. 模型 / 视图 / 控制器(MVC)模式(Model-view-controller pattern)
9. 黑板模式(Blackboard pattern)
10. 解析器模式(Interpreter pattern)

1. 分层模式(Layered pattern)

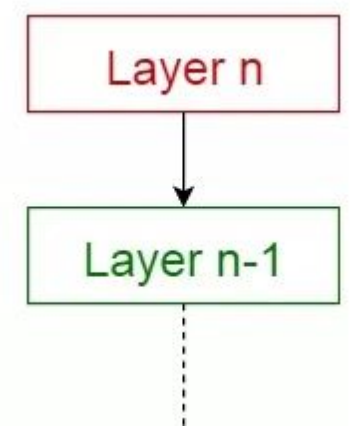
分层模式用于对结构化设计的软件进行层次拆解，每个层次为独立的抽象，为其上层抽象提供服务。

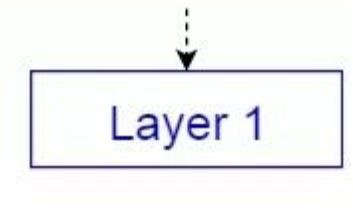
系统通常被拆分为以下四个层次：

- 表示层（也称为 UI 层）
- 应用层（也称为服务层）
- 业务逻辑层（也称为领域层）
- 数据访问层（也称为持久化层）

使用场景

- 通用桌面应用程序
- 电子商务 Web 应用



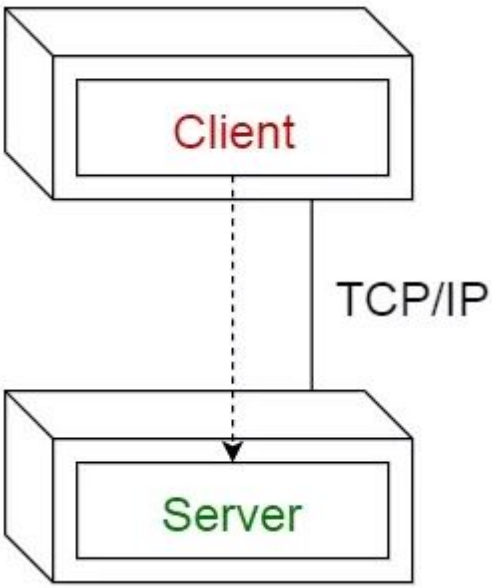


2. 客户端 / 服务器模式(Client-server pattern)

客户端 / 服务器模式由两个部分构成：一个服务器与多个客户端。服务器组件同时为多个客户端组件提供服务。客户端向服务器发启服务请求，服务器将相应服务信息回应给客户端。此外，服务器持续监听来自客户端的请求。

使用场景

- 电子邮件、文件共享及银行业务等在线应用

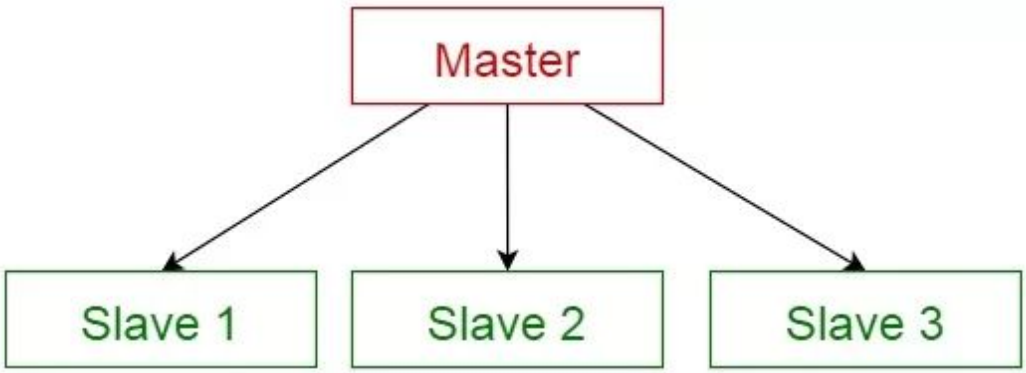


3. 主 / 从模式(Master-slave pattern)

主 / 从模式由两个部分构成：主设备与从设备。主服务组件将作业分发给多个从设备组件，并根据这些从设备反馈的结果，计算生成最终结果。

使用场景

- 数据库复制，主数据库被认定为权威数据源，各从数据库与主数据保持同步
- 在计算机系统中通过总线互连的各设备（包括主设备与从设备）

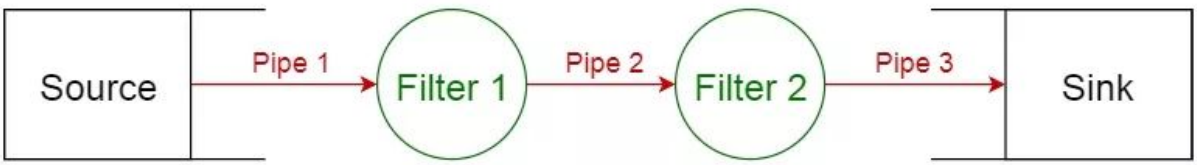


4. 管道 / 过滤器模式(Pipe-filter pattern)

管道 / 过滤器模式用于构造用于生成及处理数据流的系统。每个处理过程都封装在过滤器（filter）组件之中，要处理的数据通过 管道(pips) 进行投递。管道同时用于作为 过滤器（filter） 间的缓冲及同步。

使用场景

- 编译器，一系列的过滤器用于词法分析、语法分析、语义分析及代码生成
- 生物信息学的工作流



5. 代理模式（Broker pattern）

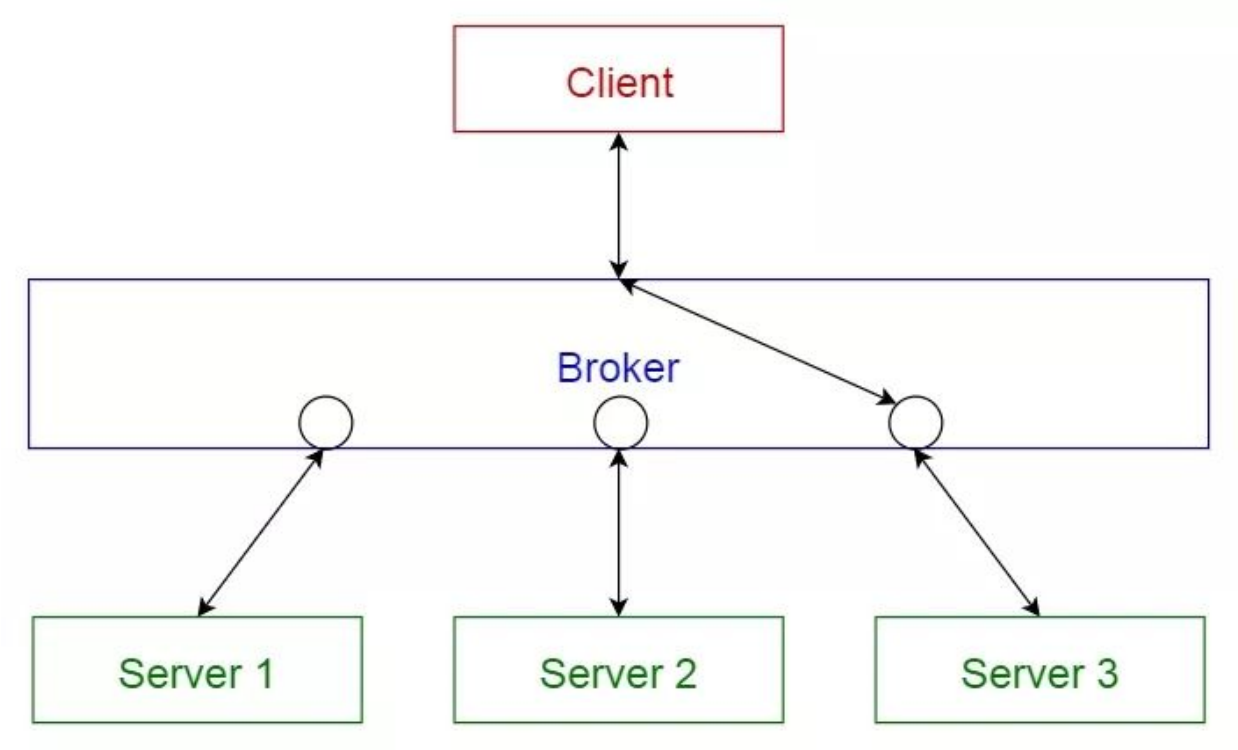
代理模式用于在结构化系统中对组件解耦。系统内各组件间采用远过程调用（remote service invocations）的方式交互。代理（Broker）组件充当组件间通讯的协调角色。

提供服务的组件将其能力（服务以及特性）发布给代理，客户端均向代理请求服务，由代理

将请求重定向到先前已发布过对应服务的组件进行处理。

使用场景

- 消息中间件软件：Apache ActiveMQ，Apache Kafka，RabbitMQ 与 JBoss 等等

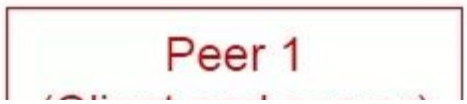


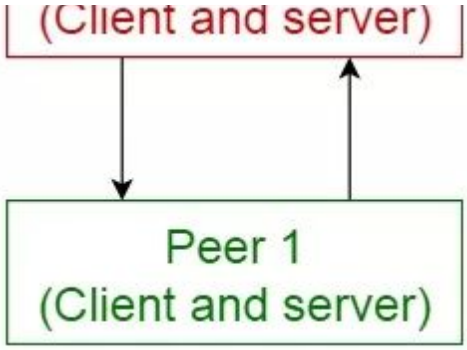
6. 对等模式(Peer-to-peer pattern)

对等模式中的组件称之为对等体（peer），对等体既作为向其他对等体请求服务的客户端，同时也做为响应其他对等体请求的服务端。对等体可以在运行过程中动态地改变其角色，即，既可以单独做为客户端或服务端运行，又可同时作为客户端与服务端运行。

使用场景

- 网络文件共享：Gnutella 与 G2)
- 流媒体协议：P2PTV 与 PDTP.
- 流媒体应用： Spotify.



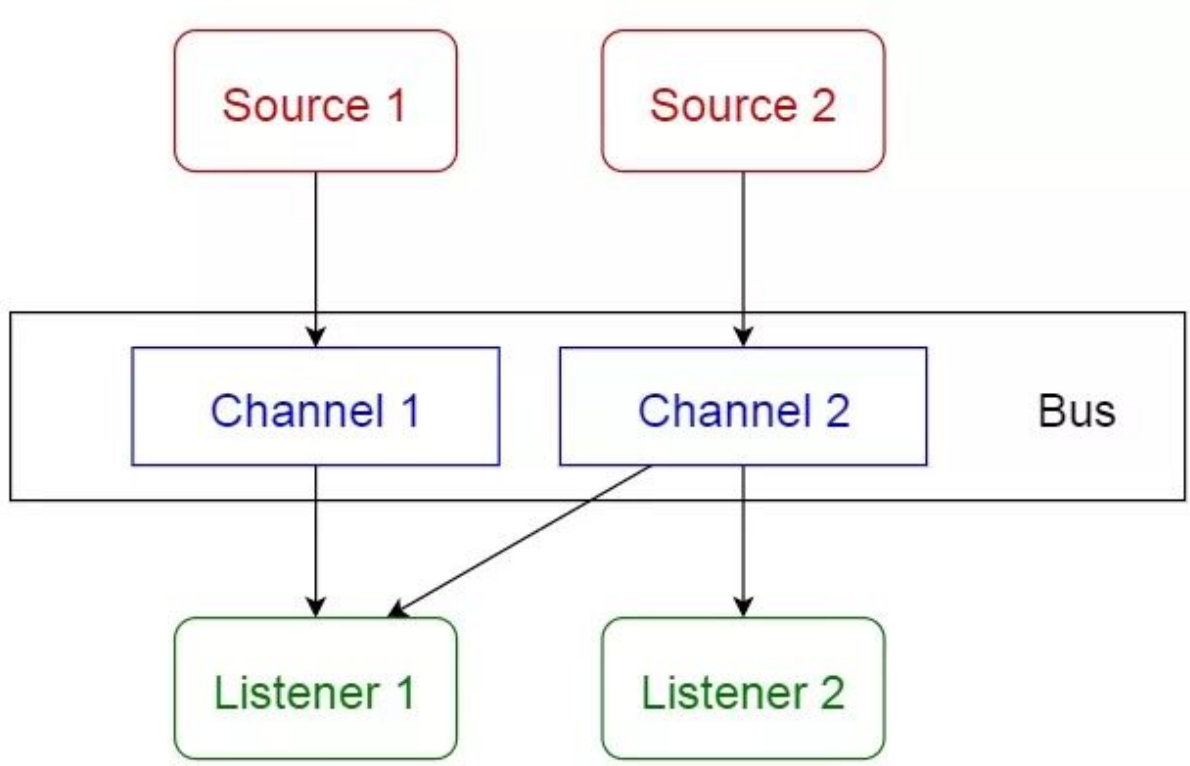


7. 事件总线模式（Event-bus pattern）

事件总线模式应用于事件处理，主要由四个组件构成：事件源（event source），事件侦听者（event listener），通道（Channel）以及总线（event bus）。 事件源将消息发布到总线的特定通道，侦听者订阅相应的通道，事件源所发布的消息经通道通告给订阅通道的侦听者。

使用场景

- Android 开发
- 通告（Notification）服务



8. 模型 / 视图 / 控制器(MVC)模式（Model-view-controller pattern）

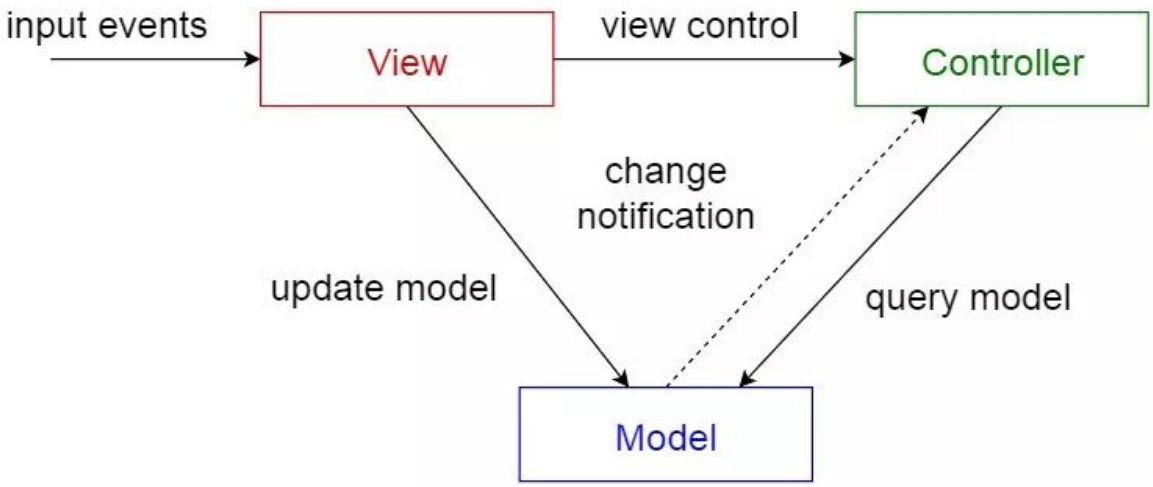
模型 / 视图 / 控制器模式（简称 MVC 模式）将交互式应用程序拆分为三个部分：

1. 模型（model） – 包含核心功能及数据
2. 视图（view） – 呈现信息给用户（通过有多个视图）
3. 控制器（controller） – 处理用户的输入操作

MVC 模式通过将内部信息表示、用户信息呈现以及用户操作接收分开的方式解耦组件，实现高效代码重用。

使用场景

- 主流开发语言所构建的互联网网页应用架构
- Django 与 Rails 等网页应用开发框架



9. 黑板模式（Blackboard pattern）

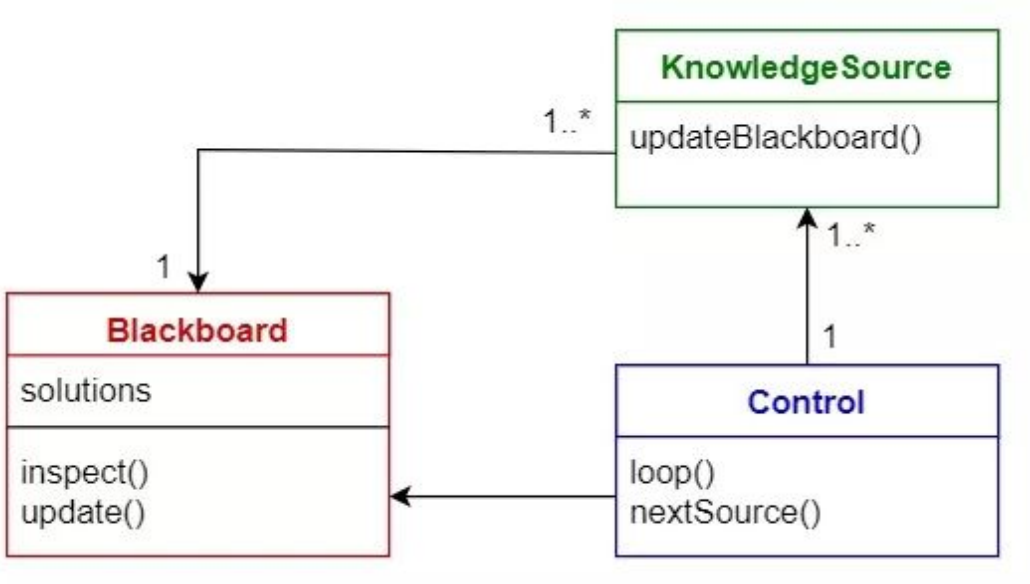
黑板模式适用于 无预知确定解决策略 的问题，主要由三个组件构成：

- 黑板（blackboard） – 用于存储解空间对象的结构化全局内存
- 知识（knowledge）源 – 能自表意的专用模块
- 控制（control）组件 – 选择、配置与执行的模块

所有的组件均能访问黑板，组件可将新生成的数据对象写入黑板，也可以通过模式匹配从黑板中获取知识源所生成的特定数据。

使用场景

- 语音识别
- 车辆识别和追踪
- 蛋白质的结构鉴定
- 声纳信号解析

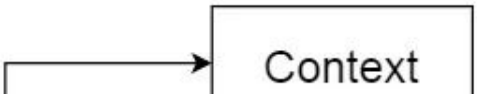


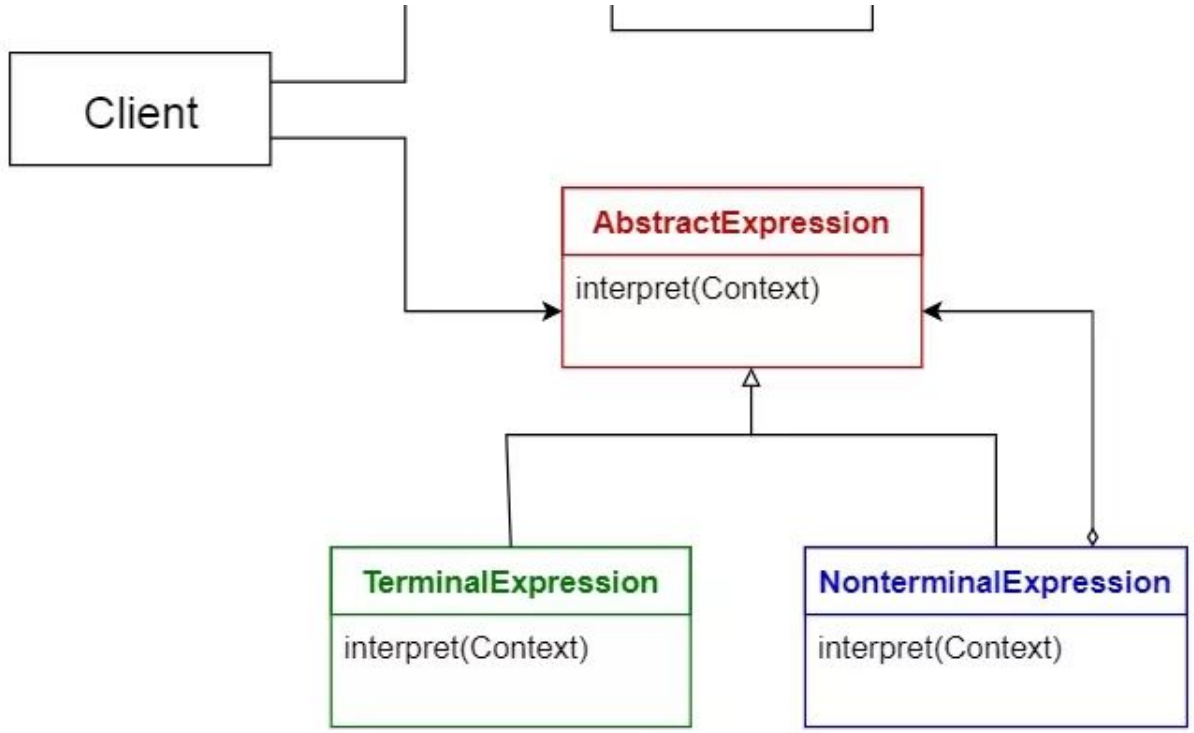
10. 解析器模式（Interpreter pattern）

解析器模式用于设计语言的解析程序，主要用于指定评估程序代码行，即解析出特定语言的语句与表达式，其核心思想是为语言的每个符号定义相应的类。

使用场景

- SQL 等数据库查询语言
- 通讯协议描述语言





上述架构模式的对比

下表格总结了各架构模式的优缺点

模式	优点	缺点
分层模式	低层的服务可以被不同的高层所使用。 通过清晰的层次定义，得标准化更加容易。 单个层次内部的变化不影响其他层次。	没有普适的分层方法。 在某些情况下，某些层次可能被跳过。
客户端 / 服务器模式	可以很好地定义一组被多客户端请求的服务	服务器端通常在单独的线程中处理请求。 由于不同客户端有不同的表示法，会导致服务器的进程间通讯过载
主 / 从模式	精确性 - 将服务的执行委托给不同的从组件处理可以有不同的实现方法	从组件间是相互独立的，无法共享状态。 在实时系统中，主与从组件间的通讯时延可能成为问题。 这种模式只能适用于处理可被分解的问题
管道 / 过滤器模式	呈现并发处理的过程。 对于由输入与输出构成的流，过滤器在收到流数据时进行计算。 易于增加过滤器，系统易于扩展，且过滤器可以重用。 通过对过滤器的重新组合，可以构造出不同的流水线。	整体的处理性能受限于瓶径的过滤器。 数据从一个过滤器传到另一个过滤器时，会存在过载。
代理模式	可以动态地添加、更新、删除以及重定向对象，对象的分发过程对开发者透明。	需要有标准化服务的描述
对等模式	支持去中心的计算方法，系统任一节点失效场景的鲁棒性更好，计算能力以及资源扩展性好	由于系统各节点是自发地参与计划，因此系统服务质量难以得到保障，同时系统安全性也难保证。 系统性能取决于多个节点，难以度量
事件总线模式	易于动态添加发布、订阅者以及建立二者间的交互。提高对分布式应用的效率。	所有消息都要经过同个总线，难以实现系统的高扩展性
MVC 模式	对于同个模型可以同时有多个视图，视图可以在运行过程中动态地与模型关联或解关联	实现复杂度高，可能会导致大量不必要的用户操作更新响应
黑板模式	易于增加新的应用程序及扩展数据的结构空间	更改空间的结构较为困难，会影响所有的应用程序。 需要增加数据同步及访问控制措施
解析器模式	可实现高度动态的行为，有益于终端用户的编程，易于替换单个解析器，灵活性好	解析式语言的执行效率通常比编译式语言低，性能可能将成为问题

希望这篇文章对你所帮助，同时我也想听听你的想法。😇

感谢阅读

阅读原文