# 定义

软件体系结构是具有一定形式的结构化元素，即构件的集合，包括处理构件、数据构件和连接构件。处理构件负责对数据进行加工，数据构件是被加工的信息，连接构件把体系结构的不同部分组组合连接起来。这一定义注重区分处理构件、数据构件和连接构件，这一方法在其他的定义和方法中基本上得到保持。

通俗理解，就是把一个系统更加有调理的结构化起来，通过运用5个面向对象设计原则、分包原则及23种设计模式，让系统的结构清晰可见。类比建造房屋，就是为软件提供一个好的建筑结构，而不是想到一砖就添一砖，未来想要移动这块儿砖头，就要牵扯到整个房屋。好的设计模式可以有效的将每一块砖合理分配到应该的位置，高内聚，低耦合，让每一块砖之间的影响达到最小，但整个房屋却变得更加坚固。

# 产生

软件体系结构的产生与软件行业的发展有关，随着软件变得越来越大、越来越复杂，一个良好的结构设计，往往成为了这个软件系统成功的重要因素之一。体系结构设计在数据结构与程序算法中变得越来越重要。软件体系结构很好的把软件的各个元素组合起来，函数、结构、类、层、子系统等等。

体系结构会受到系统相关联系人（用户、客户、项目经理、维护者、开发者）、开发部门、技术环境、架构师经验等影响。

# 作用

良好的设计结构，可以提高软件的效率，减少冗余的软件结构设计，方便进行功能扩展，易于修改，可维护，可复用，灵活性好。

体系结构可以很好的把整个系统架构抽象出来，以一个整体的形式让人们直观的了解，隐藏了具体的细节。这样，体系结构可以作为相关联系人之间沟通的工具。

# 应用场景

首先需要承认的是，在软件设计的不同阶段需要不同的设计方法。面对不同粒度，不同复杂度的系统，你需要使用不同的思考框架。

体系结构设计与算法设计就需要不同的思考框架。

在算法设计中，动态规划、贪心算法等这些算法思想是非常重要的思考方式思考框架，他们在一定程度上能够保证快速得到计算的结果。它们面临的场景更多类似于“快速在某个字符串中快速寻找某个子串”，这样一些小问题。

算法虽然能够攻克粒度较小的问题，但是在构建系统的时候，往往面临的问题是：软件系统规模大且复杂性高。这时对系统的全局结构设计和规划更加重要。比如说，给你一个需求，设计一个图书管理系统，这时候动态规划这些思考框架就显得有些力不从心了。这时候，体系结构设计就应运而生了，体系结构设计是一种构建复杂系统时的思考框架，它服务的对象是粗粒度的复杂系统。

# 组成

软件体系结构组成部分：

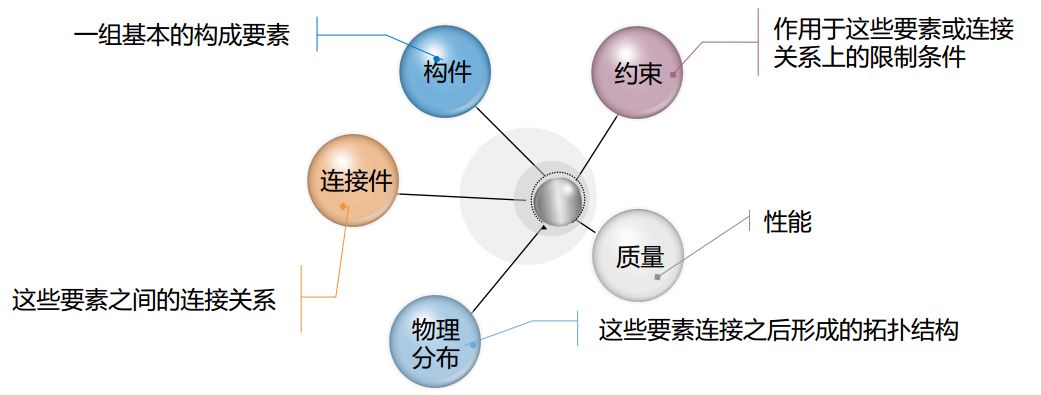
1、构件（Component）：一组代码，或程序块或一个独立的程序。（如SQL服务器）

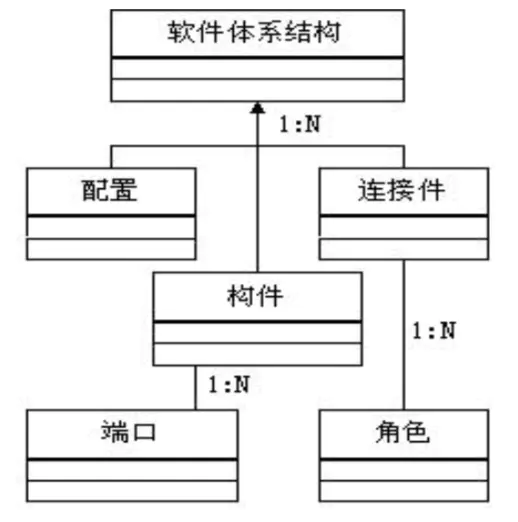
2、连接件（Connector）：关系的抽象，用以表示构件之间的相互作用。（如过程调用、管道等）

3、配置（Configuration）：用于对构件与连接件的语义说明。

4、端口：表示构件与外部环境的交互点。

5、角色：连接件的接口。



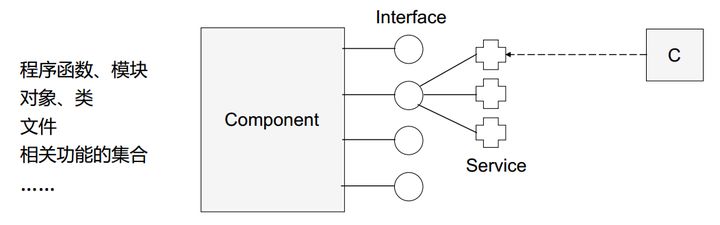


软件体系结构核心模型

## 构件

构件是具有某种功能的可复用的软件结构单元，表示系统中主要的计算元素和数据存储。

构件是一个抽象的概念，在程序中可以指程序函数、模块、对象、类等。

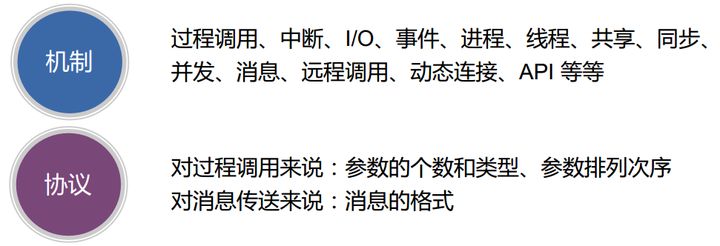


## 连接件

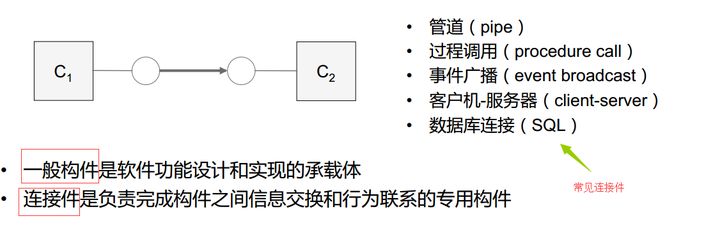
连接是构件间建立和维护行为关联与信息传递的途径。连接包含下面两种要素：

其中，机制指的实际中的消息传递方式。

而协议则决定了消息的语义理解。



连接件表示构件之间的交互并实现构件之间的连接。



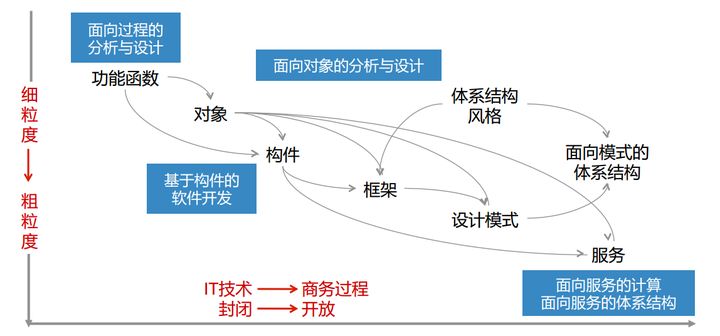
# 软件体系结构目标

所有的设计原则等理论基本上都可以映射到下面一个或几个目标上。



# 体系结构的发展

现在软件的复杂性及多变性，导致了软件粒度越来越粗，越来越开放。



# 软件设计的原则

设计原则是系统分解和模块设计的基本标准，应用这些原则可以使代码更加灵活、易于维护和扩展。这里提到的设计原则包括以下几点：



需要指出的是，这些原则有一些交叉的部分，并非完全独立没有交集的，理解这一点，将能更清楚细微的区别。比如：

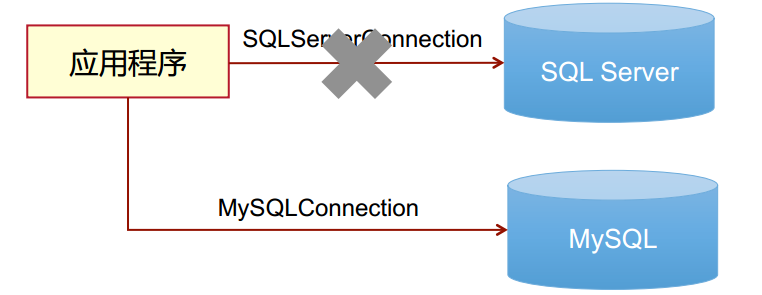
1. 通过封装来实现模块化
2. 通过将抽象程度相同的模块放在同一层次，形成层次化的结构。
3. 另外，层次化可以理解为模块化的特例

接下来分别介绍上述原则：

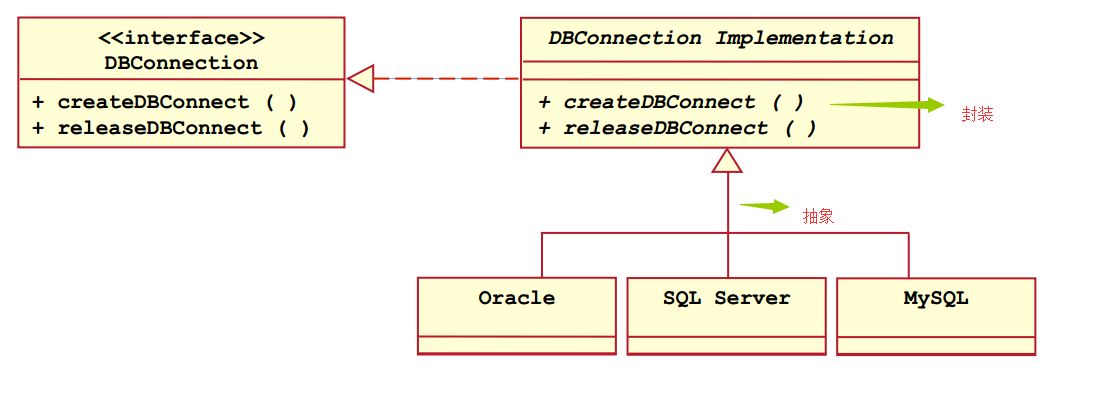
## 抽象

抽象是关注事物中与问题相关部分而忽略其他无关部分的一种思考方法。

待抽像的例子：



抽象后：

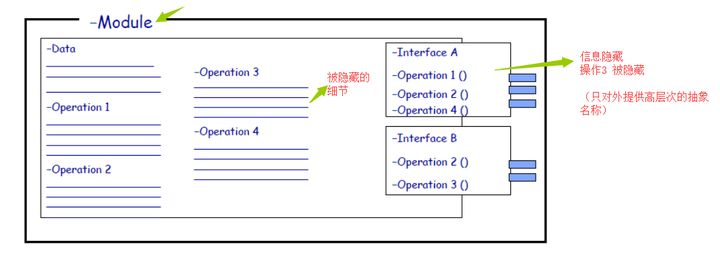


## 封装

封装和信息隐藏是指每个软件单元对其他所有单元都隐藏自己的设计决策，各个单元的特性通过其外部可见的接口来描述。

【要求】：应将单元接口设计得尽可能简单，并将单元对于环境的假设和要求降至最低。

举例：



## 模块化

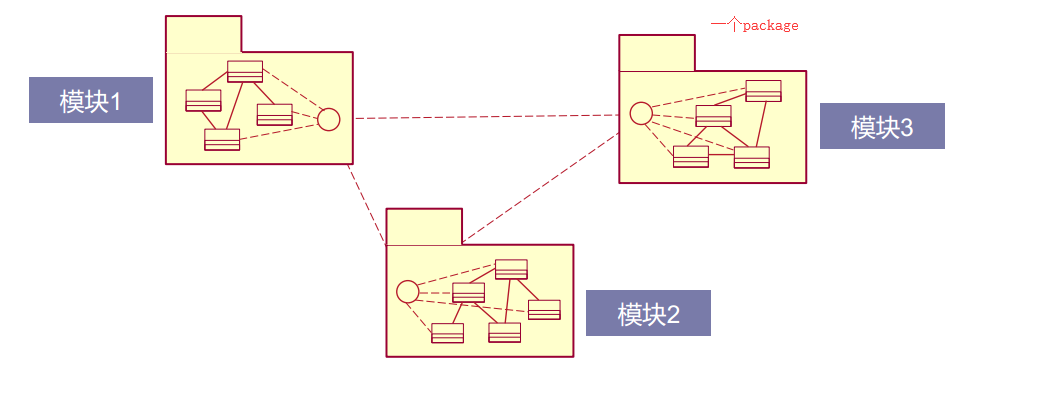
模块化是在逻辑和物理上将整个系统分解成多个更小的部分，其实质是“分而治之” ，即将一个复杂问题分解成若干个简单问题，然后逐个解决。

（一般是先进行逻辑模块化，随后进行物理分配）

举例：

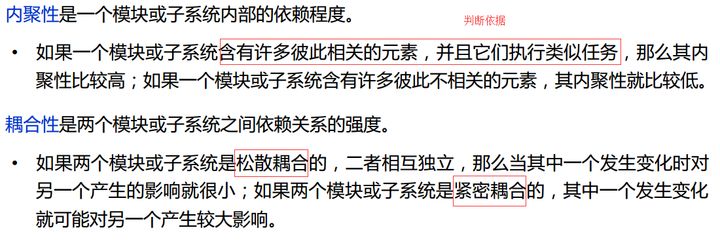
下面的模块化，主要是指逻辑上的模块化，并为涉及到如何将这些模块如何部署到物理机上的过程。

（在物理模块化的时候，可能将模块1和模块3放在同一台机器上，而模块2放在另一台机器上）



## 系统分解

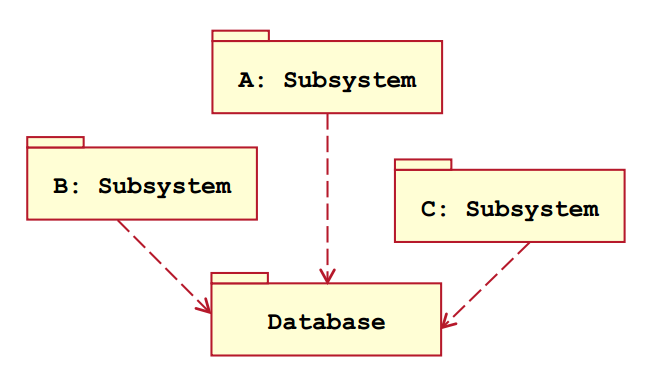
模块化，不可避免的一个问题是，对于系统如何进行分解。在大量实践中，人们总结出了两个衡量的原则：高内聚、 低耦合。



举个简单的例子：

在一个软件系统中，有三个子系统A、 B、 C都要访问一个关系数据库。

设计方案1：（高耦合方案）



设计方案2：（增加隔离层）

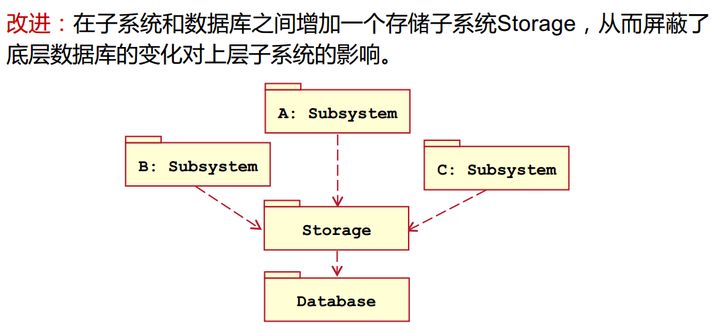
说说这种方案的好处。

设计1中，如果数据库结构发生变化，那么你需要分别在A，B，C中分别进行更改，这样分散的修改，特别容易造成错误。而在设计2中，只需要修改Storage层就可以了。

另外，在A，B，C系统中可能有一些共同的读数据库操作，比如三个系统中都有ReadAnimal（）这个操作，设计1中需要分别在A,B，C中进行修改，而设计2中将该共用方法提到Storage模块中，从而修改时，只需要进行一次修改。

PS:

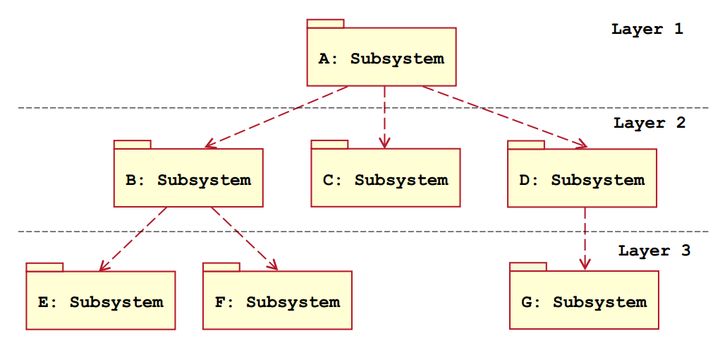
这里加入Storage，是因为假设3个系统中有一些共同的部分。但是如果子系统没有任何共享的数据库数据（每个表被一个系统所独占），那么去掉Storage层也没有关系，因为数据库某个表的变化只会影响到一个子系统。



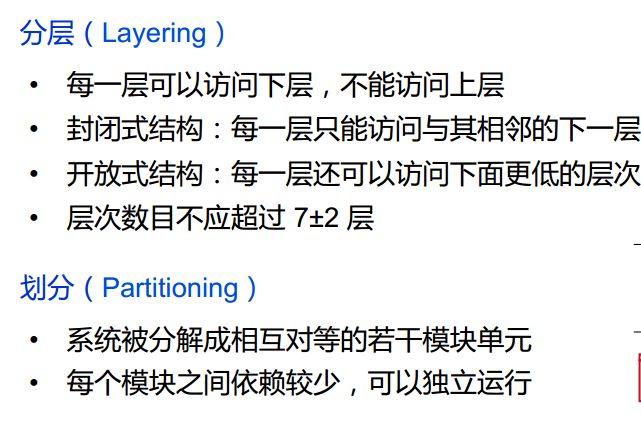
## 层次化

先给出层次化的表现形式：

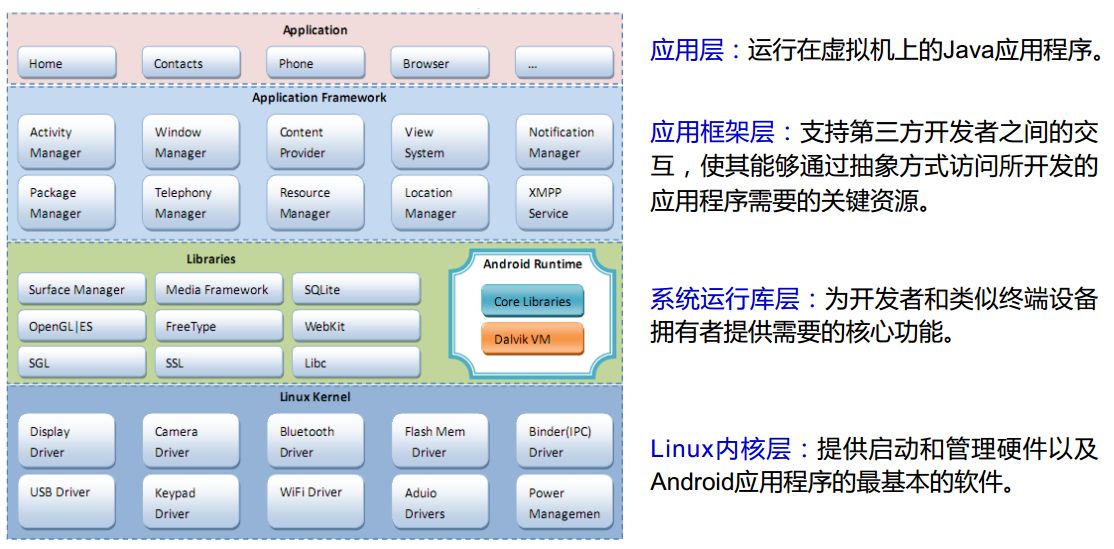
层次化是模块化的一种特例。在模块化的基础上，加入了一些限制。在层次化结构中，layer可以作为一种特殊的模块（包含特定的子系统集合），在这类模块（layer）的联系中，规定了layer模块只能被设计成类似于双向链表的顺序结构，而不能出现树状结构或者图状结构。



层次化的一些说明：



举例：Android操作系统



## 复用

复用（Reuse）是利用某些已开发的、 对建立新系统有用的软件元素来生成新的软件系统，其好处在于提高生产效率，提高软件质量。

源代码复用：对构件库中的源代码构件进行复用

• 软件体系结构复用：对已有的软件体系结构进行复用

• 框架复用：对特定领域中存在的一个公共体系结构及其构件进行复用

• 设计模式：通过为对象协作提供思想和范例来强调方法的复用

# 10种常见的软件体系架构模式

架构模式是一个通用的、可重用的解决方案，用于在给定上下文中的软件体系结构中经常出现的问题。架构模式与软件设计模式类似，但具有更广泛的范围。

## 一. 分层模式

这种模式也称为多层体系架构模式。它可以用来构造可以分解为子任务组的程序，每个子任务都处于一个特定的抽象级别。每个层都为下一个提供更高层次服务。

一般信息系统中最常见的是如下所列的4层。

• 表示层(也称为UI层)

• 应用层(也称为服务层)

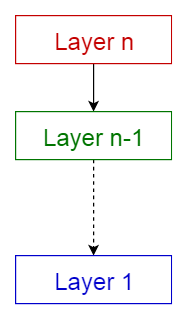
• 业务逻辑层(也称为领域层)

• 数据访问层(也称为持久化层)

使用场景：

• 一般的桌面应用程序

• 电子商务Web应用程序

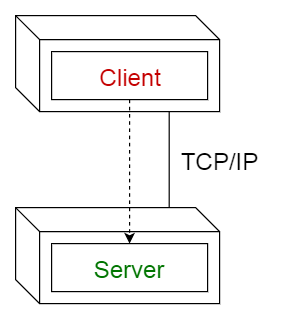


## 二. 客户端-服务器模式

这种模式由两部分组成：一个服务器和多个客户端。服务器组件将为多个客户端组件提供服务。客户端从服务器请求服务，服务器为这些客户端提供相关服务。此外，服务器持续侦听客户机请求。

使用场景：

• 电子邮件，文件共享和银行等在线应用程序



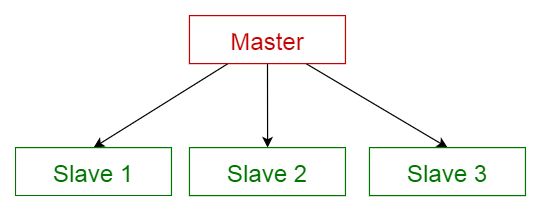
## 三. 主从设备模式

这种模式由两方组成;主设备和从设备。主设备组件在相同的从设备组件中分配工作，并计算最终结果，这些结果是由从设备返回的结果。

使用场景：

• 在数据库复制中，主数据库被认为是权威的来源，并且要与之同步

• 在计算机系统中与总线连接的外围设备(主和从驱动器)



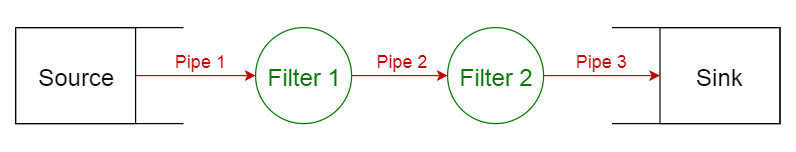
## 四. 管道-过滤器模式

此模式可用于构造生成和处理数据流的系统。每个处理步骤都封装在一个过滤器组件内。要处理的数据是通过管道传递的。这些管道可以用于缓冲或用于同步。

使用场景：

• 编译器。连续的过滤器执行词法分析、解析、语义分析和代码生成

• 生物信息学的工作流



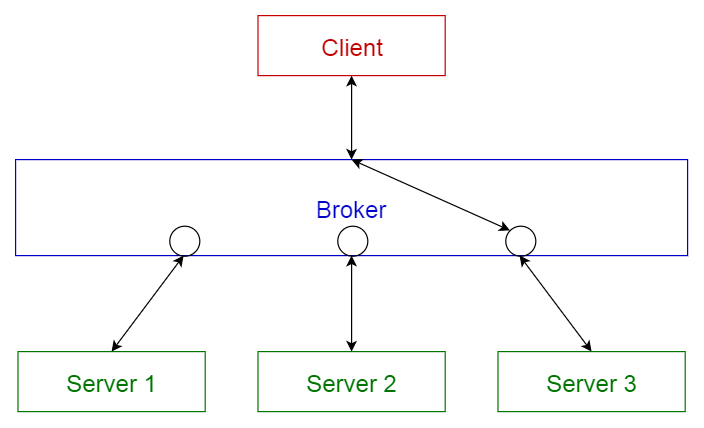
## 五. 代理模式

此模式用于构造具有解耦组件的分布式系统。这些组件可以通过远程服务调用彼此交互。代理组件负责组件之间的通信协调。

服务器将其功能(服务和特征)发布给代理。客户端从代理请求服务，然后代理将客户端重定向到其注册中心的适当服务。

使用场景：

• 消息代理软件，如Apache ActiveMQ，Apache Kafka，RabbitMQ和JBoss Messaging



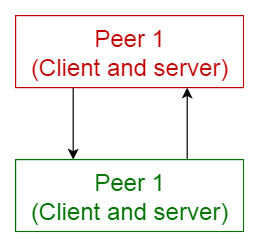
## 六. 点对点模式

在这种模式中，单个组件被称为对等点。对等点可以作为客户端，从其他对等点请求服务，作为服务器，为其他对等点提供服务。对等点可以充当客户端或服务器或两者的角色，并且可以随时间动态地更改其角色。

使用场景：

• 像Gnutella和G2这样的文件共享网络

• 多媒体协议，如P2PTV和PDTP• 像Spotify这样的专有多媒体应用程序



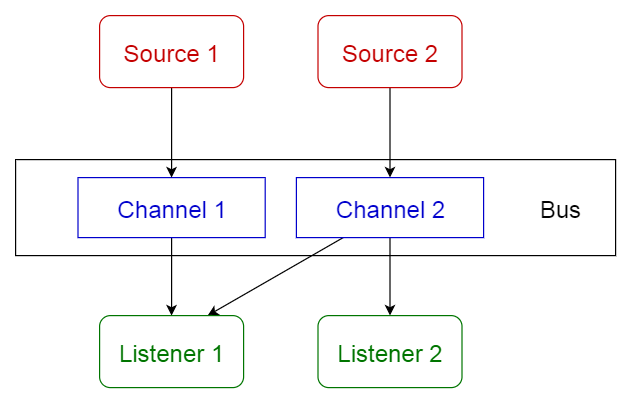
## 七. 事件总线模式

这种模式主要是处理事件，包括4个主要组件：事件源、事件监听器、通道和事件总线。消息源将消息发布到事件总线上的特定通道上。侦听器订阅特定的通道。侦听器会被通知消息，这些消息被发布到它们之前订阅的一个通道上。

使用场景：

• 安卓开发

• 通知服务



## 八. 模型-视图-控制器模式

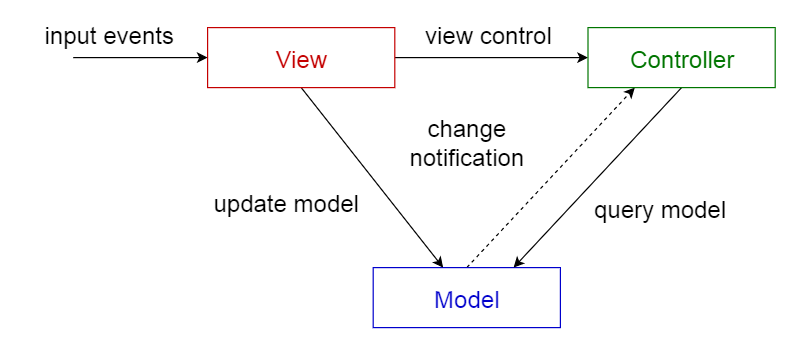
这种模式，也称为MVC模式，把一个交互式应用程序划分为3个部分，• 模型：包含核心功能和数据• 视图：将信息显示给用户(可以定义多个视图)• 控制器：处理用户输入的信息

这样做是为了将信息的内部表示与信息的呈现方式分离开来，并接受用户的请求。它分离了组件，并允许有效的代码重用。

使用场景：

• 在主要编程语言中互联网应用程序的体系架构

• 像Django和Rails这样的Web框架



## 九. 黑板模式

这种模式对于没有确定解决方案策略的问题是有用的。黑板模式由3个主要组成部分组成。

• 黑板——包含来自解决方案空间的对象的结构化全局内存

• 知识源——专门的模块和它们自己的表示

• 控制组件——选择、配置和执行模块所有的组件都可以访问黑板。组件可以生成添加到黑板上的新数据对象。组件在黑板上查找特定类型的数据，并通过与现有知识源的模式匹配来查找这些数据。

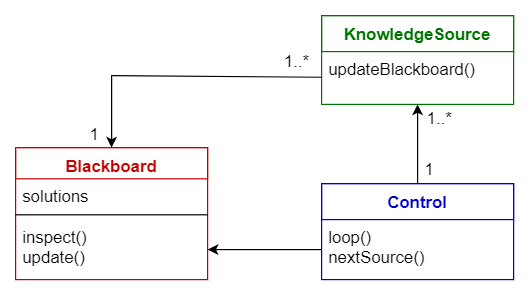
使用场景：

• 语音识别

• 车辆识别和跟踪

• 蛋白质结构识别

• 声纳信号的解释



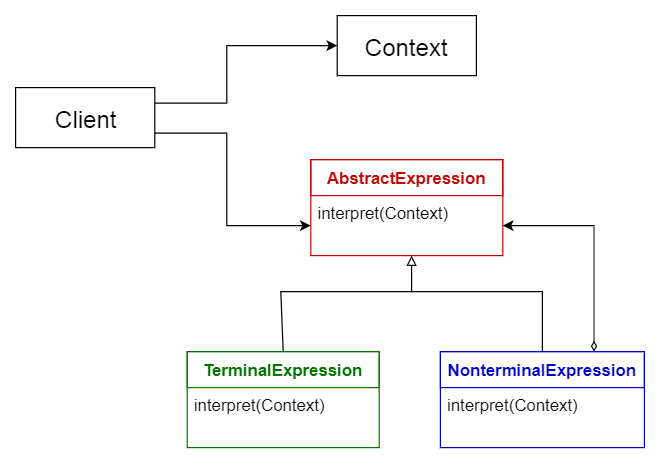
## 十. 解释器模式

这个模式用于设计一个解释用专用语言编写的程序的组件。它主要指定如何评估程序的行数，即以特定的语言编写的句子或表达式。其基本思想是为每种语言的符号都有一个分类。

使用场景：

• 数据库查询语言，比如SQL

• 用于描述通信协议的语言



## 体系架构模式的比较

下面给出的表格总结了每种体系架构模式的优缺点。

