“软件体系结构包括有关软件系统组织的一系列重要决策，包括对构成系统的结构要素及其接口的选择；这些要素之间协作规定的行为；将这些结构和行为要素组成更大的子系统；以及指导该组织的建筑风格。软件体系结构还涉及功能，可用性，弹性，性能，重用性，可理解性，经济和技术约束，折衷和美学方面的考虑。”

**软件体系结构风格：**

一种体系结构风格以结构组织模式定义了一个系统家族。

关于构件和连接件类型的术语；一组约束对它们组合方式的规定；一个或多个语义模型，规定了如何从各成分的特性决定系统整体特性

概括地说，一种软件体系结构风格刻划一个具有共享结构和语义的系统家族.

架构风格:

**客户端-服务器:**

将系统分为两个应用，其中客户端向服务器发送服务请求。

**基于组件的架构:**

把应用设计分解为可重用的功能、逻辑组件，这些组件的位置相互透明，只暴露明确定义的通信接口。

**分层架构:**

把应用的关注点分割为堆栈组（层）。

**消息总线:**

指接收、发送消息的软件系统，消息基于一组已知格式，以便系统无需知道实际接收者就能互相通信。

**N层/三层架构:**

用与分层风格差不多一样的方式将功能划分为独立的部分，每个部分是一个层，处于完全独立的计算机上。

**面向对象:**

该架构风格是将应用或系统任务分割成单独、可重用、可自给的对象，每个对象包含数据，以及与对象相关的行为。

**分离表现层:**

将处理用户界面的逻辑从用户界面（UI）视图和用户操作的数据中分离出来。

**面向服务架构(SOA):**

是指那些利用契约和消息将功能暴露为服务、消费功能服务的应用。

这些架构风格分别适用于特定领域:

**通信:**

SOA，消息总线，管道和过滤器

**部署:**

客户端/服务器，三层架构，N层架构

**领域:**

领域模型，网关

**交互:**

分离表现层

**结构:**

基于组件的架构，面向对象，分层架构

**下面介绍几种常见的架构风格:**

**管道和过滤器风格:**

在管道/过滤器风格的软件体系结构中，每个构件都有一组输入和输出，构件读输入的数据流，经过内部处理，然后产生输出数据流。这个过程通常通过对输入流的变换及增量计算来完成，所以在输入被完全消费之前，输出便产生了。因此，这里的构件被称为过滤器，这种风格的连接件就像是数据流传输的管道，将一个过滤器的输出传到另一过滤器的输入。此风格特别重要的 过滤器必须是独立的实体，它不能与其它的过滤器共享数据，而且一个过滤器不知道它上游和下游的标识。一个管道/过滤器网络输出的正确性并不依赖于过滤器进 行增量计算过程的顺序。

图1是管道/过滤器风格的示意图。一个典型的管道/过滤器体系结构的例子是以Unix shell编写的程序。Unix既提供一种符号，以连接各组成部分(Unix的进程)，又提供某种进程运行时机制以实现管道。另一个著名的例子是传统的编 译器。传统的编译器一直被认为是一种管道系统，在该系统中，一个阶段(包括词法分析、语法分析、语义分析和代码生成)的输出是另一个阶段的输入。

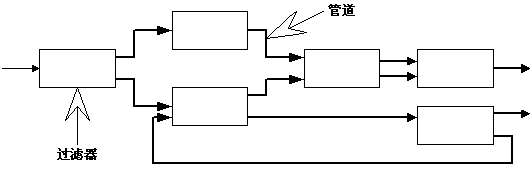
[](http://images2015.cnblogs.com/blog/15172/201510/15172-20151011151841518-1216718751.gif)

图 1管道/过滤器风格的体系结构

管道/过滤器风格的软件体系结构具有许多很好的特点：

(1)使得软构件具有良好的隐蔽性和高内聚、低耦合的特点；

(2)允许设计者将整个系统的输入/输出行为看成是多个过滤器的行为的简单合成；

(3)支持软件重用。重要提供适合在两个过滤器之间传送的数据，任何两个过滤器都可被连接起来；

(4)系统维护和增强系统性能简单。新的过滤器可以添加到现有系统中来；旧的可以被改进的过滤器替换掉；

(5)允许对一些如吞吐量、死锁等属性的分析；

(6)支持并行执行。每个过滤器是作为一个单独的任务完成，因此可与其它任务并行执行。

但是，这样的系统也存在着若干不利因素:

(1)通常导致进程成为批处理的结构。这是因为虽然过滤器可增量式地处理数据，但它们是独立的，所以设计者必须将每个过滤器看成一个完整的从输入到输出的转换。

(2)不适合处理交互的应用。当需要增量地显示改变时，这个问题尤为严重。

(3)因为在数据传输上没有通用的标准，每个过滤器都增加了解析和合成数据的工作，这样就导致了系统性能下降，并增加了编写过滤器的复杂性

**数据抽象与面向对象风格:**

抽象数据类型概念对软件系统有着重要作用，目前软件界已普遍转向使用面向对象系统。这种风格建立在数据抽象和面向对象的基础上，数据的表示方法和它们的相应操作封装在一个抽象数据类型或对象中。这种风格的构件是对象，或者说是抽象数据类型的实例。对象是一种被称作管理者的构件，因为它负责保持资源的完整性。对象是通过函数和过程的调用来交互的。

图2是数据抽象和面向对象风格的示意图。

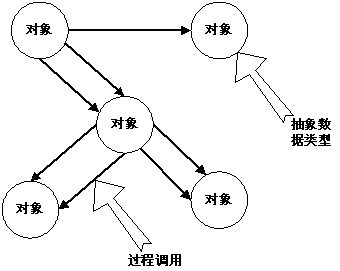
[](http://images2015.cnblogs.com/blog/15172/201510/15172-20151011151842409-117441877.gif)

图 2数据抽象和面向对象风格的体系结构

面向对象的系统有许多的优点，并早已为人所知：

(1) 因为对象对其它对象隐藏它的表示，所以可以改变一个对象的表示，而不影响其它的对象。

(2) 设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合。

但是，面向对象的系统也存在着某些问题：

(1)为了使一个对象和另一个对象通过过程调用等进行交互，必须知道对象的标识。只要一个对象的标识改变了，就必须修改所有其他明确调用它的对象。

(2)必须修改所有显式调用它的其它对象，并消除由此带来的一些副作用。例如，如果A使用了对象B，C也使用了对象B，那么，C对B的使用所造成的对A的影响可能是料想不到的。

**基于事件的隐式调用风格:**

基于事件的隐式调用风格的思想是构件不直接调用一个过程，而是触发或广播一个或多个事件。系统中的其它构件中的过程在一个或多个事件中注册，当一个事件被触发，系统自动调用在这个事件中注册的所有过程，这样，一个事件的触发就导致了另一模块中的过程的调用。

从体系结构上说，这种风格的构件是一些模块，这些模块既可以是一些过程，又可以是一些事件的集合。过程可以用通用的方式调用，也可以在系统事件中注册一些过程，当发生这些事件时，过程被调用。

基于事件的隐式调用风格的主要特点是事件的触发者并不知道哪些构件会被这些事件影响。这样不能假定构件的处理顺序，甚至不知道哪些过程会被调用，因此，许多隐式调用的系统也包含显式调用作为构件交互的补充形式。

支持基于事件的隐式调用的应用系统很多。例如，在编程环境中用于集成各种工具，在数据库管理系统中确保数据的一致性约束，在用户界面系统中管理数据，以及在编辑器中支持语法检查。例如在某系 统中，编辑器和变量监视器可以登记相应Debugger的断点事件。当Debugger在断点处停下时，它声明该事件，由系统自动调用处理程序，如编辑程 序可以卷屏到断点，变量监视器刷新变量数值。而Debugger本身只声明事件，并不关心哪些过程会启动，也不关心这些过程做什么处理。

隐式调用系统的主要优点有：

(1)为软件重用提供了强大的支持。当需要将一个构件加入现存系统中时，只需将它注册到系统的事件中。

(2)为改进系统带来了方便。当用一个构件代替另一个构件时，不会影响到其它构件的接口。

隐式调用系统的主要缺点有：

(1)构件放弃了对系统计算的控制。一个构件触发一个事件时，不能确定其它构件是否会响应它。而且即使它知道事件注册了哪些构件的构成，它也不能保证这些过程被调用的顺序。

(2)数据交换的问题。有时数据可被一个事件传递，但另一些情况下，基于事件的系统必须依靠一个共享的仓库进行交互。在这些情况下，全局性能和资源管理便成了问题。

(3)既然过程的语义必须依赖于被触发事件的上下文约束，关于正确性的推理存在问题。

**层次系统风格:**

层次系统组织成一个层次结构，每一层为上层服务，并作为下层客户。在一些层次系统中，除了一些精心挑选的输出函数外，内部的层只对相邻的层可见。这样的系统中构件在一些层实现了虚拟机(在另一些层次系统中层是部分不透明的)。连接件通过决定层间如何交互的协议来定义，拓扑约束包括对相邻层间交互的约束。

这种风格支持基于可增加抽象层的设计。这样，允许将一个复杂问题分解成一个增量步骤序列的实现。由于每一层最多只影响两层，同时只要给相邻层提供相同的接口，允许每层用不同的方法实现，同样为软件重用提供了强大的支持。

图3是层次系统风格的示意图。层次系统最广泛的应用是分层通信协议。在这一应用领域中，每一层提供一个抽象的功能，作为上层通信的基础。较低的层次定义低层的交互，最低层通常只定义硬件物理连接。

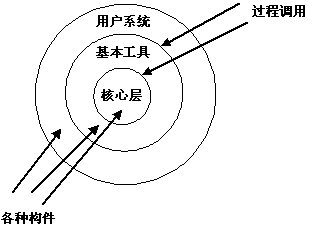
[](http://images2015.cnblogs.com/blog/15172/201510/15172-20151011151843128-242581666.gif)

图 3层次系统风格的体系结构

层次系统有许多可取的属性：

(1)支持基于抽象程度递增的系统设计，使设计者可以把一个复杂系统按递增的步骤进行分解；

(2)支持功能增强，因为每一层至多和相邻的上下层交互，因此功能的改变最多影响相邻的上下层；

(3)支持重用。只要提供的服务接口定义不变，同一层的不同实现可以交换使用。这样，就可以定义一组标准的接口，而允许各种不同的实现方法。

但是，层次系统也有其不足之处：

(1)并不是每个系统都可以很容易地划分为分层的模式，甚至即使一个系统的逻辑结构是层次化的，出于对系统性能的考虑，系统设计师不得不把一些低级或高级的功能综合起来；

(2)很难找到一个合适的、正确的层次抽象方法。

**仓库风格:**

在仓库风格中，有两种不同的构件：中央数据结构说明当前状态，独立构件在中央数据存贮上执行，仓库与外构件间的相互作用在系统中会有大的变化。

控制原则的选取产生两个主要的子类。若输入流中某类时间触发进程执行的选择，则仓库是一传统型数据库；另一方面，若中央数据结构的当前状态触发进程执行的选择，则仓库是一黑板系统。

图4是黑板系统的组成。黑板系统的传统应用是信号处理领域，如语音和模式识别。另一应用是松耦合代理数据共享存取。

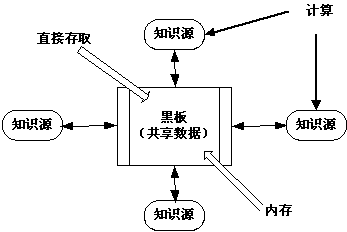
[](http://images2015.cnblogs.com/blog/15172/201510/15172-20151011151843862-418442312.gif)

图 4黑板系统的组成

我们从图4中可以看出，黑板系统主要由三部分组成：

(1)知识源。知识源中包含独立的、与应用程序相关的知识，知识源之间不直接进行通讯，它们之间的交互只通过黑板来完成。

(2)黑板数据结构。黑板数据是按照与应用程序相关的层次来组织的解决问题的数据，知识源通过不断地改变黑板数据来解决问题。

(3)控制。控制完全由黑板的状态驱动，黑板状态的改变决定使用的特定知识。