WORK LOG 8

Shandong University

April 26, 2020

高德琛

Contents

1 软件架构风格原理

1.1 定义

软件体系结构风格（Architectural Styles）是描述特定系统组织方式的惯用范例，强调了软件系统中通用的组织结构。

一个软件体系结构风格定义了构件和连接件类型的符号集，及规定了它们怎样组合起来的约束集合。

架构风格是一组原则。你可以把它看成是一组为系统家族提供抽象框架的粗粒度模式。架构风格能改进分块，还能为频繁出现的问题提供解决方案，以此促进设计重用。

1.2 涉及问题

- 设计词汇表是什么？或者构件和连接器的类型是什么？
- 可容许的结构模式是什么？
- 基本的计算模型是什么？
- 风格的基本不变性是什么？
- 其使用的常见例子是什么？
- 使用此风格的优缺点是什么？
- 其常见特例是什么？

1.3 意义

软件体系结构设计的一个中心问题是能否重用软件体系结构模式，或者采用某种软件体系结构风格。有原则地使用软件体系结构风格具有如下意义：

- 它促进了设计的复用，使得一些经过实践证实的解决方案能够可靠地解决新问题。
- 它能够带来显著的代码复用，使得体系结构风格中的不变部分可共享同一个解决方案。
- 便于设计者之间的交流与理解。
- 通过对标准风格的使用支持了互操作性，以便于相关工具的集成。

- 在限定了设计空间的情况下，能够对相关风格作出分析。
- 能够对特定的风格提供可视化支持。

1.4 待明确问题

与此同时，人们目前尚不能准确回答的问题是：

- 系统设计的哪个要点可以用风格来描述；
- 能否用系统的特性来比较不同的风格，如何确定用不同的风格设计系统之间的互操作；
- 能否开发出通用的工具来扩展风格；
- 如何为一个给定的问题选择恰当的体系结构风格，或者如何通过组合现有的若干风格来产生一个新的风格。

2 架构风格实例

2.1 系统层次划分

M.Shaw 等人根据此框架给出了管道与过滤器、数据抽象和面向对象组织、基于事件的隐式调用、分层系统、仓库系统及知识库和表格驱动的解释器等一些常见的软件体系结构风格。

- 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。

2.2 按领域划分

- 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

3 典型风格

3.1 管道/过滤器风格

3.1.1 定义

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。

3.1.2 优势

- 使得软构件具有良好的隐蔽性和高内聚、低耦合的特点；
- 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；
- 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；
- 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；
- 允许对一些如吞吐量、死锁等属性的分析；
- 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。

3.1.3 劣势

- 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。
- 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。
- 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

3.2 数据抽象与面向对象风格

3.2.1 定义

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。

3.2.2 优势

面向对象的系统有许多的优点，并早已为人所知：

- 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。
- 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。

3.2.3 劣势

- 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。
- 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。

3.3 基于事件的隐式调用风格

3.3.1 定义

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。

3.3.2 优势

- 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。
- 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。

3.3.3 劣势

- 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。
- 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。
- 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。

3.4 C2 风格

3.4.1 规则

C2 体系结构风格可以概括为：通过连接件绑定在一起的按照一组规则运作的并行构件网络。C2 风格中的系统组织规则如下：

- 系统中的构件和连接件都有一个顶部和一个底部；
- 构件的顶部应连接到某连接件的底部，构件的底部则应连接到某连接件的顶部，而构件与构件之间的直接连接是不允许的；
- 一个连接件可以和任意数目的其它构件和连接件连接；
- 当两个连接件进行直接连接时，必须由其中一个的底部到另一个的顶部。

3.4.2 特点

- 系统中的构件可实现应用需求，并能将任意复杂度的功能封装在一起；
- 所有构件之间的通讯是通过以连接件为中介的异步消息交换机制来实现的；
- 构件相对独立，构件之间依赖性较少。系统中不存在某些构件将在同一地址空间内执行，或某些构件共享特定控制线程之类的相关性假设。

3.5 层次系统风格

3.5.1 定义

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。

3.5.2 优势

- 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；
- 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；
- 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。

3.5.3 劣势

- 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；
- 很难找到一个合适的、正确的层次抽象方法。

3.6 仓库风格

3.6.1 定义

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。

3.6.2 组成

- 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。
- 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。
- 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。