

软件体系结构复习

一、软件体系结构概述

- 什么是软件体系结构，你能用一句话描述出来吗？（软件体系结构的宏观定义）

① 架构是一个主观的东西，是一个项目中的专家开发者对系统设计的共同理解。通常这种共同的理解是以系统的主要组成部分的形式出现的，以及它们如何相互作用。它也是关于决定的，因为它是开发人员希望他们能在早期就做出正确的决定，也被认为是很难改变的。

② 一个程序或计算系统的软件体系结构是系统的结构，它由软件元素、这些元素的外部可见属性以及它们之间的关系组成。

③ 一组具有特定形式的架构元。——Perry 和 Wolf, 1992

④ 一个软件系统架构包括 一组组件、连接和约束 一组系统利益相关者的需求说明 一个证明这些组件、连接和约束能满足利益相关者需求说明的理由。——Boehm 等人, 1995

二、基于构件的软件架构：组件和 Spring IoC

- 软件构件的定义

一个软件组件，是一个具有合同规定的接口和明确的上下文依赖关系的组成单位。一个软件组件可以独立部署，并受制于第三方的组成。

- 什么是基于组件的开发 (CBSE)

基于组件的软件工程 Component-based software engineering (CBSE)，即基于组件的开发 components-based development (CBD)，是一种基于重复使用的方法，用于定义、实现和将松散耦合的独立组件组成系统。

- CBD 的目标

1. 用现有的（第三方）构件组装成一个复杂的系统
2. 通过添加 / 替换部件来更新一个系统，这些组件是
 - 部署的单位
 - 按原样处理（黑匣子）

- 什么是反转控制 Inversion of Control (IoC) / 反转控制的概念

1. 提供服务的组件是被“注入”的而不是被“直接写入”到请求者的代码中。
2. 在组件方面：应用程序独立地定义了一组组件和它们的依赖关系，组件框架（称为容器）使用这些信息在运行时将组件连在一起，在生命周期的特定时间调用其代码。
3. 反转控制的几种实现模式

- (1) 依赖性注入 (Dependency Injection)

汇编器将具体的实现实例化，并将它们“注入”需要它们的组件中。

- (2) 服务定位器 (Service Locator)

4. 目标：松散耦合的组件

- 一个组件不应该创建（实例化）它的依赖关系
- 该解决方案被称为“反转控制”，一个组件不创建（实例化）它的依赖关系，而是由其他人创建。

三、软件体系结构：分层架构

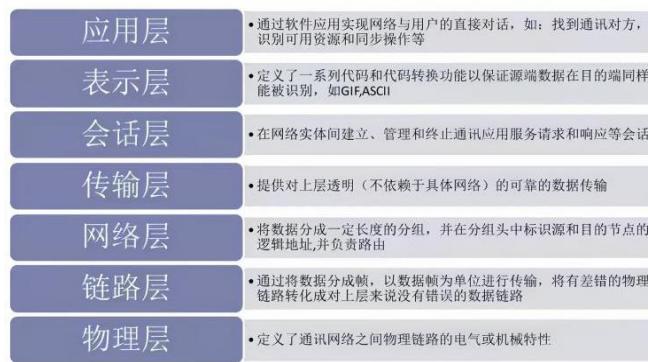
· 典型的分层设计思想

分层架构根据职能的差异，划分为多个层次。分层架构设计思想，有很多成功的例子，诸如：

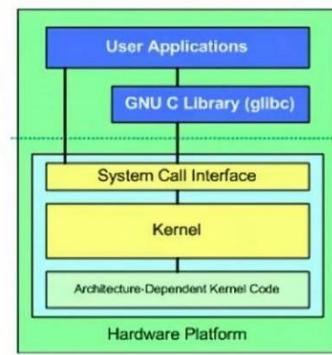
· **OSI 七层网络模型**：通过七个层次化的结构模型使不同的系统不同的网络之间实现可靠的通讯，因此其最主要的功能就是帮助不同类型的主机实现数据传输。OSI 网络模型是理论模型，工业实践中使用的 TCP/IP 协议，也遵循 OSI 七层网络模型，只是将 OSI 的应用层，表示层和会话层归并到应用层而已。

· **Linux 分层架构**：Linux 分层分为设备驱动层、Linux 内核层、系统服务层和应用层。Linux 分层基于以下规则：子系统之间自顶向下依赖；上层子系统依赖下层子系统；下层子系统不依赖上层子系统。Linux 内核层的核心功能是：抽象化和虚拟化硬件模块，并仲裁多个任务在并行使用计算机硬件资源。

OSI七层协议



Linux Architecture

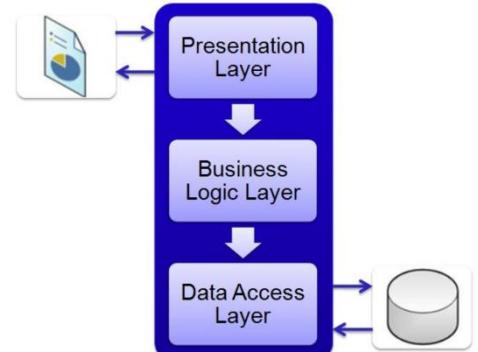


· 三层体系架构

1. **数据访问层**：基本上是存储所有应用程序数据的服务器
(处理数据库之间的交互，不应对数据做任何业务上的加工)

2. **业务逻辑层**：主要是作为数据访问层和表现层之间的桥
(接受从表示层传过来的数据，做业务上的数据校验，并实现业务逻辑，最后把加工后的数据传给数据访问层)

3. **表示层**：表现层是用户与应用程序互动的层
(从用户控件中/业务逻辑层取得数据，不进行任何加工将数据表现出来)



· 分层体系结构的优缺点

· **优点**：各层之间是独立的；灵活性好；结构上可分割开；易于实现和维护；能促进标准化工作。

· **缺点**：分层的层数次难以确定；有些功能会在不同的层次中重复出现，而产生了额外开销。

1.发展历程

为控制软件复杂度提取软件的共性成份而沉淀下来的一层软件，屏蔽系统低层的复杂度，在高层保持复杂度的相对稳定。

2.核心思想

- (1) 实现了对复杂问题的分步求解
- 自顶向下看：不断的假设过程
- 自底向上看：不断的抽象过程

(2) 各层语义良好

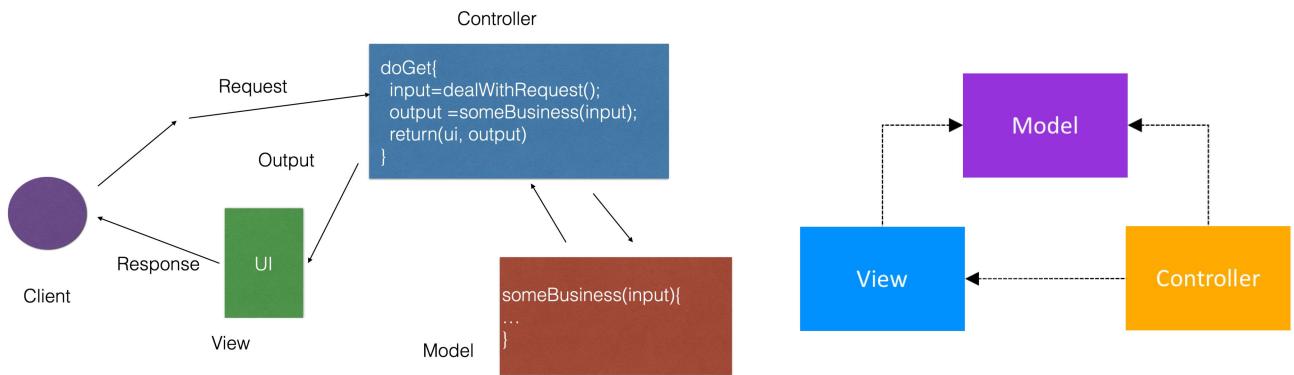
- 具有明确的使用场景
- 提供良好的复用条件

四、“模型-视图-控制器”架构 (MVC)

· MVC 具体含义

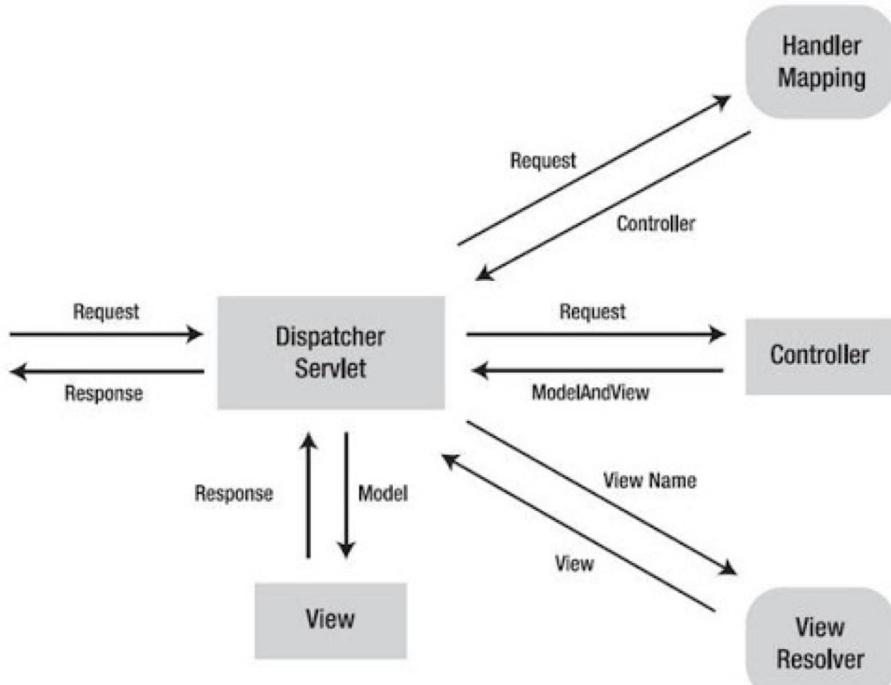
系统分割为 processing, output 和 input 三个部分:

- (1) 模型层 Model : processing, 核心数据功能
- (2) 视图层 View : output, 从 Model 获得数据显示给用户 (网页版/手机版)
- (3) 控制器层 Controller : input, 处理事件操作模型
- 过程: 当用户发出请求时, 交给控制层“Controller” → “Controller”和模型“Model”进行交互 → “Model”把自己要输出的数据交给模版渲染引擎“View”
- 好处: 可做到取得的数据一样, 但展示的形式不一样。



· 识别 Spring MVC

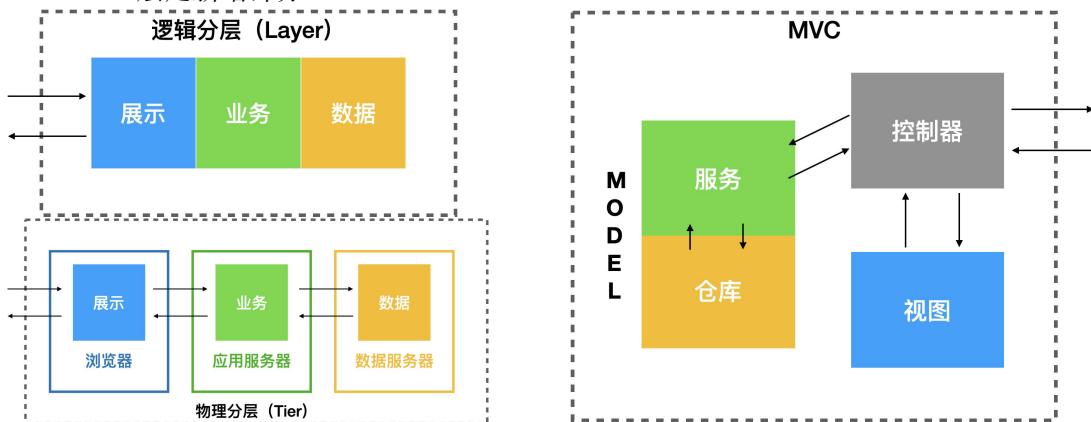
识别方法: Model + View + Controller



· MVC VS 三层体系架构

Model 层包含哪些功能：数据访问层+业务逻辑层的功能

Controller 层是新增部分



五、可扩展架构

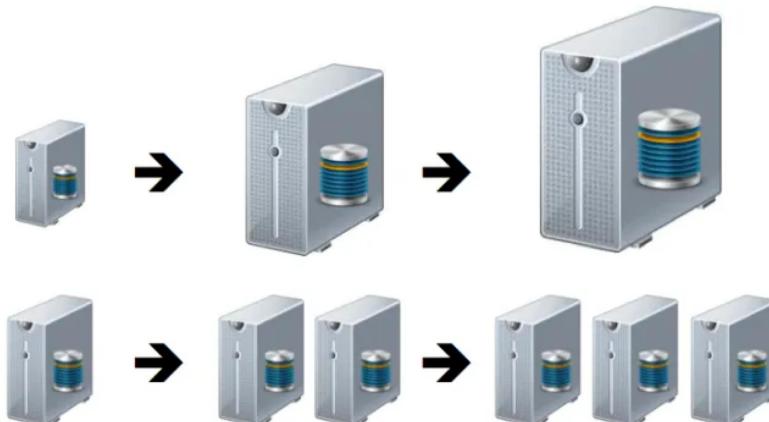
· 垂直扩展 (买更大的机器)

通过增强硬件设备的性能，来提升应用本身的性能。通过更新设备，找到更强的设备来提升系统性能。（缺点：①计算机本身性能存在上限

②很不经济，随着设备性能提升，价格差别远高于提升的性能差别。）

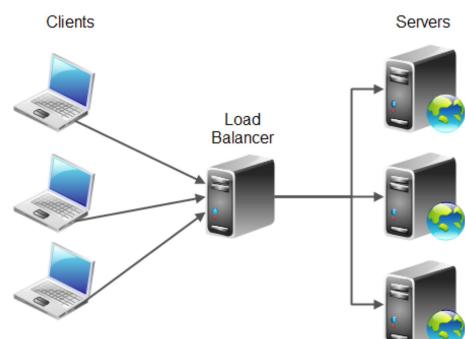
· 水平扩展 (买更多的机器)

用多台机器，把他们的资源集中起来。然后用多台机器同时去服务用户的请求，从而使得用户的性能指标能满足要求。多采用“负载均衡”的技术。



· 负载平衡

是一种计算机技术，用来在多个计算机、网络连接、CPU、磁盘驱动器或其他资源中分配负载，以达到最优化资源使用、最大化吞吐率、最小化响应时间、同时避免过载的目的。使用带有负载平衡的多个服务器组件，取代单一的组件，可以通过冗余提高可靠性。负载平衡服务通常是由专用软件和硬件来完成。



六、微服务架构

· 什么是微服务架构? (概念 / 定义)

微服务架构风格是一种将单个应用程序开发成一套小型服务的方法, 每个服务都在自己的进程中运行, 并与轻量级机制 (通常是 HTTP 资源 API) 进行通信。这些服务是围绕业务能力建立的, 并且可以通过完全自动化的部署机制进行独立部署。这些服务有最低限度的集中管理, 它们可以用不同的编程语言编写, 并使用不同的数据存储技术。

· 微服务 VS 单体应用程序 (进行水平扩展)

(1) 单体应用程序 / 聚系统:

必须把各个应用程序、各个进程在多个机器上复制。当一个系统在用户量大的情况下进行水平扩展, 每个实例都要将完整的副本系统独立地运行一份, 会造成资源的浪费。

(2) 微服务:

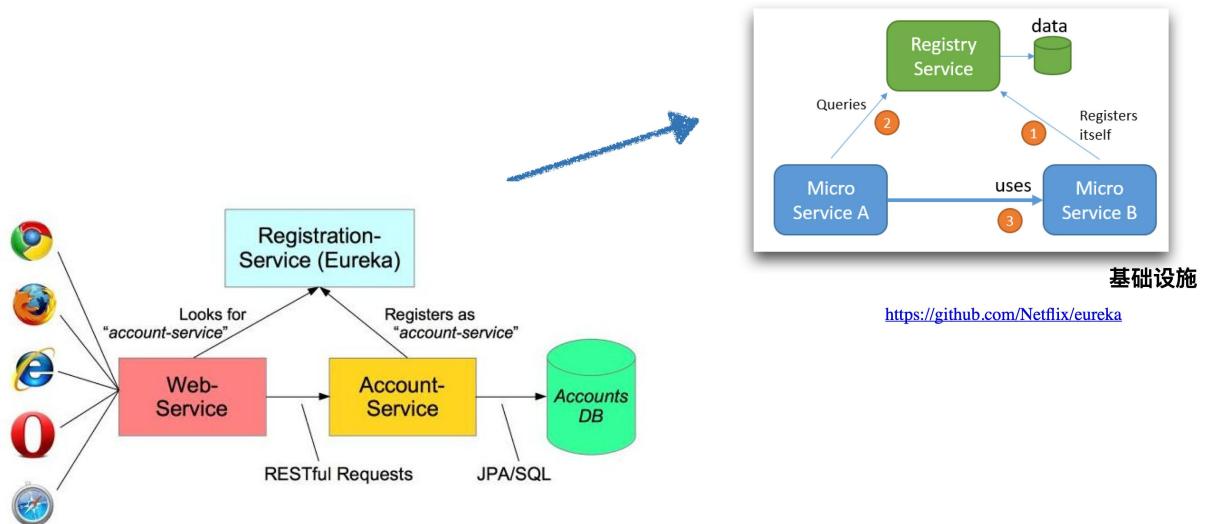
粒度会小一点, 每个服务都是一个小进程, 可以单独对一个小进程进行扩展。每个功能部件是一个独立的系统, 在负载均衡下可以将每个模块独立地进行水平扩展。可以根据每个模块的复杂度和用户访问请求量, 可有效地节省系统资源, 使资源利用更高效。

· 微服务基础设施: **RegistrationService** (名服务) — Eureka

(1) WebService 和 AccountService 相互独立, 并且 WebService 要能调用 AccountService。RegistrationService 作为名服务可以提供内部 DNF, 通过固定名称去访问, 避免手工查看, Eureka 则作为查找中介。AccountService 将名称注册到 Eureka, WebService 通过名称去 Eureka 里进行查找对应的 AccountService 是哪个 URL。

(2) 微服务之间有相互依赖, 子系统之间有相互调用。有了名服务作为基础设施之后。使得一个服务可以通过名称找到另外一个服务访问的入口地址。实现了服务与服务之间的相互发现、查找和调用。

(3) 服务注册 + 负载均衡



七、REST 架构

- 什么是 REST 架构 (概念 / 定义)

资源表现层状态转化 Resource Representational State Transfer

1.REST：指的是一组架构约束条件和原则

- 为设计一个功能强、性能好、适宜通信的 Web 应用
- 如果一个架构符合 REST 的约束条件和原则，我们就称它为 RESTful 架构

2.核心概念

(1) 资源 (Resources)

- 网络上的一个实体，或者说是网络上的一个具体信息，任何事物，只要有被引用到的必要，它就是一个资源。

· 统一资源接口：HTTP 方法

RESTful 架构应该使用相同的接口进行资源的访问。接口应该使用标准的 HTTP 方法如 GET, PUT 和 POST。如果按照 HTTP 方法的语义来暴露资源，那么接口将会拥有安全性和幂等性的特性。

(2) 表现层 (Representation)

资源具体呈现出来的形式，叫做它的表现层：

- 文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式
- 图片可以用 JPG 格式表现，也可以用 PNG 格式表现

(3) 状态转化 (State Transfer)

客户端应用状态，在服务端提供的超媒体/超链接的指引下发生变迁。服务端通过超媒体告诉客户端当前状态有哪些后续状态可以进入。

八、无服务器架构

- 什么是无服务器体系结构 / 无服务器体系结构的特点

1. 无服务器架构是结合了第三方 "后台即服务" (BaaS) 服务的应用设计，和包括在 "功能即服务" (FaaS) 平台上的受管理、短暂的容器中运行的自定义代码。

2. 通过使用这些理念以及单页应用等相关理念，这种架构消除了对传统的永远在线的服务器组件的大部分需求。

3. 无服务器架构可能会受益于大幅降低的运营成本、复杂性和工程准备时间，但代价是对供应商的依赖性和相对不成熟的支持服务增加。

· 无服务器架构包含哪两个主要方面？

无服务器架构是指应用程序使用第三方管理的 Function 和服务，因此不需要管理服务器。

无服务器架构主要包含了两个方面：

1.BaaS (Backend as a Service, 后端即服务) :

使用第三方服务（如 Firebase、Auth0）来达成目的。使用 BaaS 的应用程序通常是富客户端应用程序，如 SPA 或移动 App。客户端负责处理大部分的业务逻辑，其他部分则依赖外部服务，如认证、数据库、用户管理，等等。

2.FaaS (Function as a Service, 函数即服务) :

包含服务器端业务逻辑的无状态函数。这些函数运行在独立的容器里，基于事件驱动，并由

第三方厂商托管，如 AWS Lambda 或者 Azure Functions。

- 无服务器架构的优缺点

1.优点

- 不需要管理服务器
- 自动伸缩(Scale to Zero)
- 没有运营成本
- 成本由事件驱动
- 具有较高的安全性

2.缺点

- 在上面运行的应用无法被监控
(服务器不属于自己，企业机构无法完整地看到上面所运行的一切，也就更加难以衡量一个应用的性能。同时，他们也无法轻易看到性能问题的发展趋势，或主动预防问题。)
- 资源限制
(虽然无服务器架构具有弹性，可以通过加速和减速来为应用分配资源，但它是有限制的。)

九、“管道-过滤器”架构

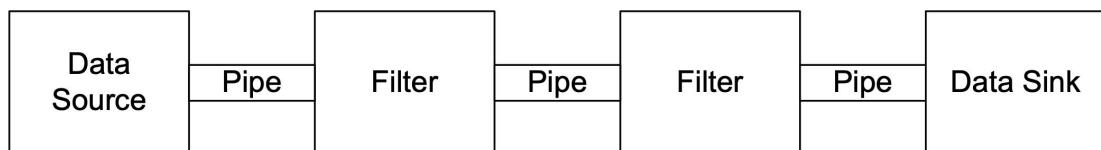
- 构成管道-过滤器架构的主要元素

组成：管道 Pipes / 过滤器 Filters / 数据源 Data Source / 数据汇点 Data Sink

1.Filter 完成单步数据处理功能

2.Data Source/Data Sink/Filter 以 Pipe 连接

3.Pipe 连接相邻元素，前一元素的输出为后一元素的输入



- 管道-过滤器架构的优缺点

1.优点

- 过滤器可以重用/重组合/可替换
- 不需保存中间结果
- 高效的并行处理 (多 active 部件)

2.缺点

- 数据传输开销较大
- 数据转换开销较大
- 错误处理较为复杂

十、事件驱动架构

- 什么是事件驱动架构？(Event-driven Architecture)

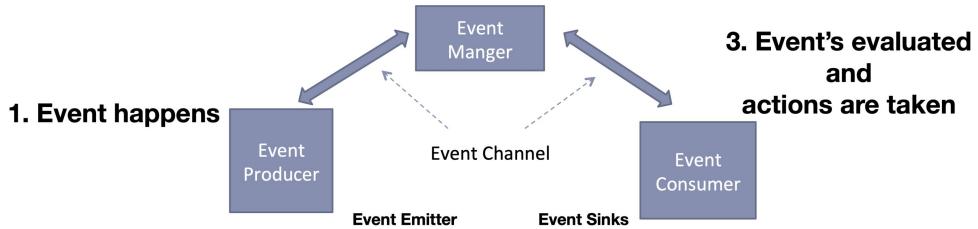
1.事件驱动架构 (EDA) 是一种软件架构范式，促进事件的生产、检测、消费和反应。一个事件可以被定义为 "状态的重大变化"。

2.整个事件的运行，是靠事件发生来驱动的。

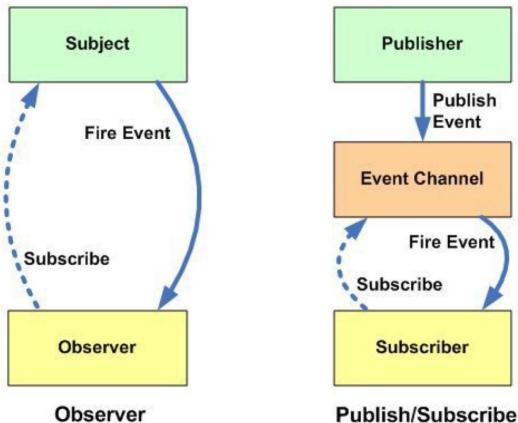
事件的生产方会产生一个事件，这个事件会由事件管理器散布出去，告诉若干个对该事件已经进行过注册的事件的消费方。然后由事件的消费方，对该事件做出评估，决定要不要发生

响应。如果要，就执行响应动作，如果不要，就不做处理。

2. Event disseminates

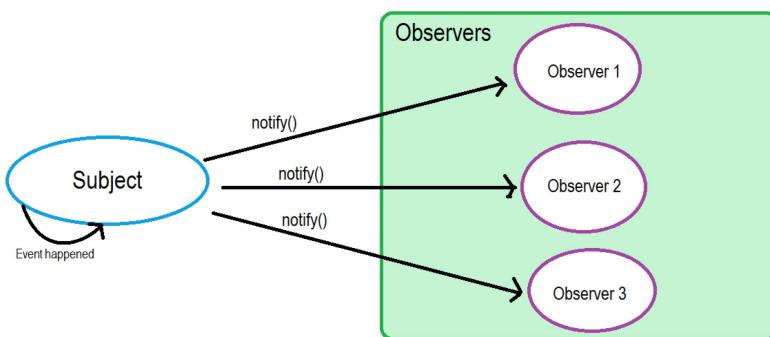


- 实现事件驱动架构的两种手段：观察者 vs 发布/订阅



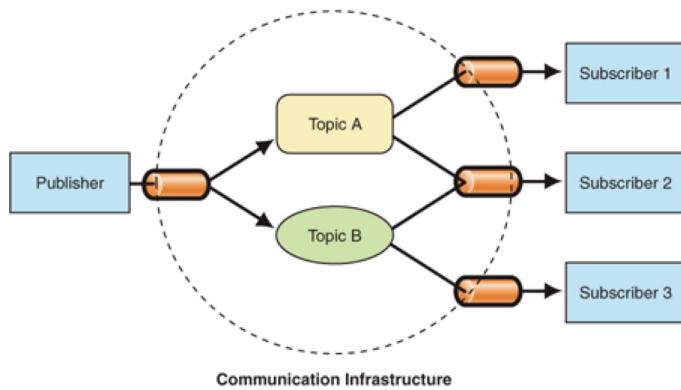
1. 观察者模式

事件源本身，既充当事件源，又充当事件管理器。由于事件观察者对事件进行了注册，事件源在发生时逐个通知事件观察者，简化了的由三部分组成的事件驱动架构。（发布者和观察者耦合度会高一些）



2. 发布/订阅模式

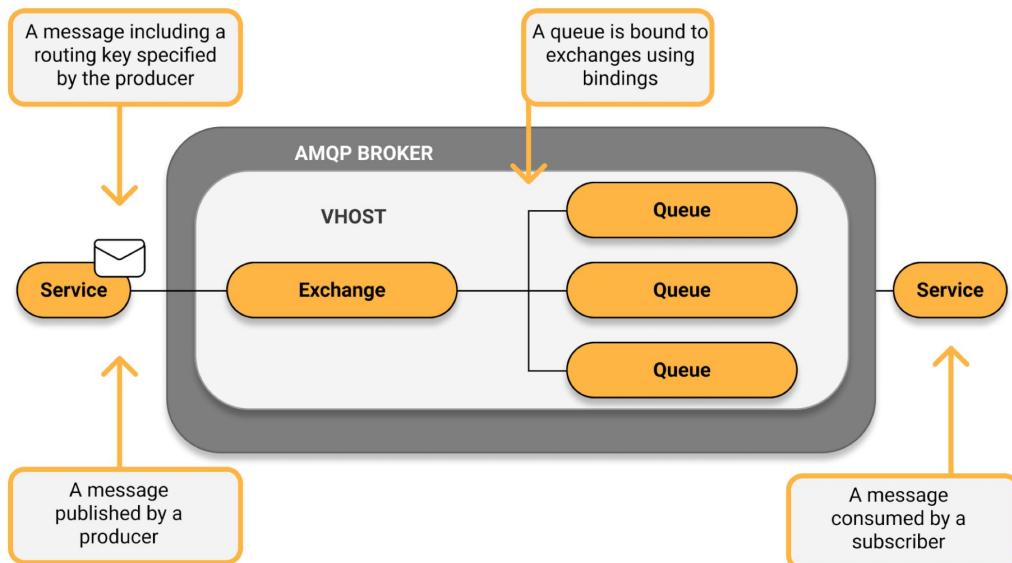
有独立的通信机制，实现事件的分发过程。



· AMQP 的定义

高级消息队列协议即 Advanced Message Queuing Protocol 是面向消息中间件提供的开放的应用层协议，其设计目标是对于消息的排序、路由、保持可靠性、保证安全性。

(Exchange 规定什么样的消息，应该交给什么样的队列；队列应该保存消息，直到应用去取这个消息。可类比：信箱收信)



· 实现 AMQP 的系统

1.RabbitMQ

- (1) 生产者向交易所发布一条信息
- (2) 交易所收到该消息并负责消息的路由
- (3) 队列和交易所之间必须建立绑定，这里我们有两个不同的队列的绑定，交易所将消息路由到这两个队列
- (4) 消息停留在队列中，直到它们被消费者处理
- (5) 消费者处理消息。

2.Spring AMQP

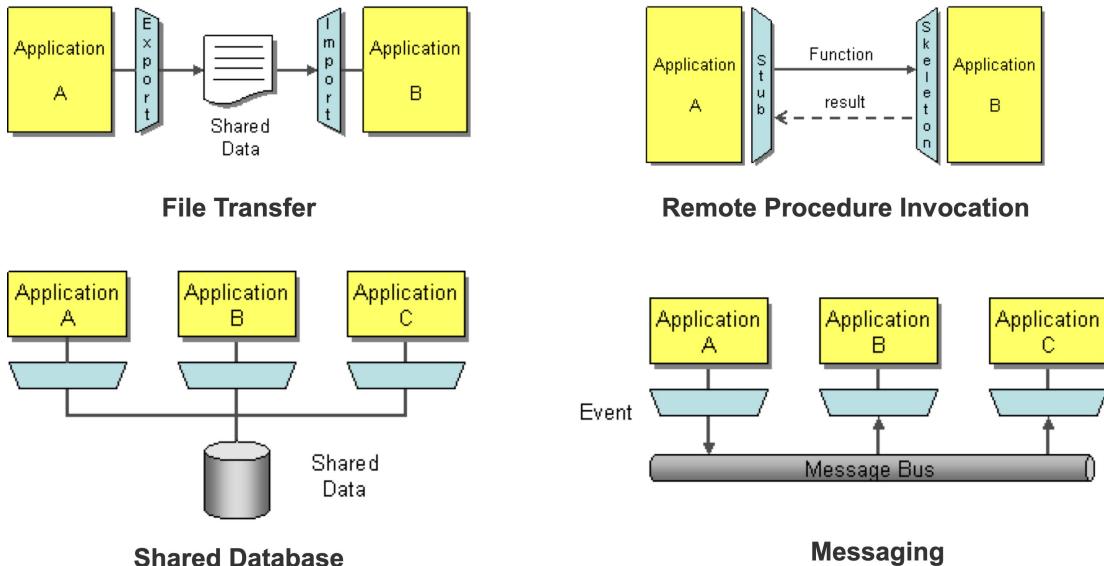
- **AMQP 实体** - 我们用消息、队列和交换类创建实体、绑定和交换类来创建实体
- **连接管理** - 通过使用一个 `CachingConnectionFactory` 连接到我们的 RabbitMQ 中间件
- **消息发布** - 我们使用 `RabbitTemplate` 来发送消息
- **消息消费** - 我们使用 `@RabbitListener` 来从队列中读取消息

十一、企业应用集成

· 四大集成模式

文件传输 / 远程接口调用 / 共享数据库 / 消息传递

(两个系统独立开发、独立运行，如何实现信息的互通)



1. 文件传输

A 和 B 之间没有交互的接口，但在运行过程中会产生一系列文件，代表了 A 实际运行的一些结果和一些状态，与 B 集成的方式，就是让 B 去读取这些文件来获得产生的结果和改变的状态。A 将状态写入文件，B 来读取文件获得状态，实现 A 与 B 之间信息的互通。

2. 远程方法调用

A 之间存在接口 B，B 有功能接口让 A 调用。则 A 可以通过调用 B 提供的接口，来获得 B 的响应，由此 AB 之间可以实现信息的共享。

3. 共享数据库

若干系统之间没有输出文件，也没有相互调用接口。但 ABC 都可以访问同一个共享数据源（比如：数据库）。内部状态通过外部存储，被持久化下来，供别人读取所需的信息，

4. 消息传递

(1) 使用消息通信机制，该模式以消息为中心—离散的数据有效载荷，通过预定的渠道从一个源系统或流程移动到一个或多个系统或流程。

(2) 核心：使用消息通道连接应用程序，其中一个应用程序向通道写入信息，另一个则从通道读取该信息。

(3) 该模式是以一种最灵活的方式来整合多个不同的系统：

- 几乎完全解耦参与集成的系统
- 允许参与集成的系统对彼此的底层协议、格式化或其他实现细节完全不了解
- 鼓励参与集成的组件的开发和重复使用

十二、响应式架构

· 什么是响应式系统？

作为响应式系统构建的系统更加灵活，松散耦合和可扩展。这使得它们更容易开发，更容易改变。它们对失败的容忍度要高得多，当失败发生时，它们会以优雅的方式来应对，而不是灾难。反应式系统是高度响应的，给予用户有效的互动反馈。

一、软件体系结构概述

- 介绍软件体系结构的基本概念
- 当前流行的软件设计开发方法
- 基于 Spring 框架的软件设计和开发



1. 软件危机(Software Crisis)

(1) 1968 年, 联邦德国, NATO 科技委员会:

成本不断提高、开发进度难以控制、质量低下、维护困难、用户难以满意、软件生产率赶不上硬件发展和用户需求增长

(2) 软件是有缺陷的、不可靠的、永远变化的、不应该的

(3) 软件的不适用性: 美国陆军对联邦项目的研究 47% 已交付, 但未使用 29% 已付款, 但未交付, 19% 放弃或返工, 3% 修改后使用, 2% 按交付使用

2. 软件工程: 软件危机的解决之道

软件工程一门研究如何用系统化、规范化、数量化等工程原则和方法, 去进行软件的开发和维护的学科。

(1) 软件开发技术

开发方法学 / 软件工具 / 软件工程环境

(2) 软件项目管理

项目估算 / 进度控制 / 人员组织 / 项目计划

3. 软件体系结构: 研究动机

(1) 随着软件系统规模越来越大、越来越复杂, 整个系统的结构和规格说明显得越来越重要

(2) 对于大规模的复杂软件系统来说, 对总体的系统结构设计和规格说明比起对计算的算法和数据结构的选择已变得明显重要得多

(3) 对于软件体系结构的系统、深入的研究将会成为提高软件生产率和解决软件维护问题

的新的最有希望的途径

- (4) 对于软件体系结构的研究成为了最新的研究方向和独立学科分支

4. 软件体系结构: 定义

一个程序或计算系统的软件体系结构是系统的结构，它由软件元素、这些元素的外部可见属性以及它们之间的关系组成。

其他定义: ①②③④⑤⑥⑦⑧⑨⑩⑪⑫⑬⑭⑮⑯⑰

①一组具有特定形式的架构元。——Perry 和 Wolf, 1992

②一个软件系统架构包括 一组组件、连接和约束 一组系统利益相关者的需求说明 一个证明这些组件、连接和约束能满足利益相关者需求说明的理由。——Boehm 等人, 1995

* 架构是一个主观的东西，是一个项目中的专家开发者对系统设计的共识。通常这种共同的理解是以系统的主要组成部分的形式出现的，以及它们如何相互作用。它也是关于决定的，因为它是开发人员希望他们能在早期就做出正确的决定，因为他们被认为是很难改变的。

5. 架构层设计: 关乎设计

在计算的算法和数据结构之外，设计并确定系统整体结构：

- (1) 总体组织结构和全局控制结构
- (2) 通信、同步和数据访问的协议
- (3) 设计元素的功能分配
- (4) 物理分布与性能
- (5) 备选设计的选择

6. 为什么软件架构很重要？

(1) 利益相关者之间的沟通 (2) 早期的设计决策 (3) 系统作为可传递的财富
软件开发组织的一种核心财富。

7. 利益相关者

软件系统的每个利益相关者都关注受架构影响的不同系统特性。

用户 -- 系统是否可靠，需要时是否可用？

客户 -- 架构能否按计划和预算实施？

经理 -- 架构能否允许团队基本独立工作，以有纪律和有控制的方式进行互动？

架构师 -- 策略能否实现所有这些目标？

8. 软件架构：早期的设计决策

- (1) 定义实施的约束条件
- (2) 决定组织结构
- (3) 抑制或促成系统的质量属性（性能/可修改性/安全性/可扩展性/相互耦合）
- (4) 通过研究架构来预测系统质量
- (5) 使得推理和管理变化更加容易
- (6) 使得成本和进度估算更加准确

9. 来自瓦萨的教训：贪心不可取

- (1) 为满足苛刻的要求而精心设计的成功架构的案例研究，以帮助确定当今技术竞争环境。
- (2) 在任何系统建立之前对架构进行评估的方法，以减少与启动前所未有的设计有关的风险。
- (3) 基于架构的增量开发技术，以便在纠正之前发现设计缺陷，为时已晚。

10. 系统可传递的财富

架构设计将从某个领域的经验中收集到的具体知识编成法典，好的设计被称为模式。

- (1) 在开发软件系统时要遵循
- (2) 为了实现特定的属性

(3) 有效而优雅地解决设计问题

11. 体系结构风格：模式

本质上反映了一些特定元素按照特定的方式组成一个特定的结构，该结构应有利于上下文环境中特定问题的解决：

- Pipe/filter, c/s, oo, bb, layered, p2p……
- 特定领域相关的模型参考
- 黑板、客户端-服务器（2-层、3-层、n-层）、基于组件的事件驱动（隐式调用）、分层（多层架构）、单一的应用程序、对等的（P2P）、管道和过滤器、插件、代表性状态传输（REST）、基于规则的面向服务的架构和微服务、共享的无架构、基于空间的架构……

Eg. 模型-视图-控制器（Model- View-Controller）

背景介绍：人机交互界面设计

解决的问题：功能的变化要求用户界面的改变

平台的改变要求用户界面随之改变

界面的改变需足够方便，且不应影响系统功能本身

12. 分类

- (1) 架构模式 -- 有助于将软件系统结构化为子系统
- (2) 设计模式 -- 支持子系统和组件的细化
- (3) 成语 -- 帮助在特定的编程语言中实现特定的设计内容

13. 架构模式

- (1) 表示软件系统的基本结构组织模式。
 - (2) 它提供了一套预定义的子系统，规定了它们的职责，并包括组织它们之间的关系：
 - 规则和指南
 - 全系统的基本设计决定
- 例如，MVC 为交互式软件系统提供了一个结构

14. 研究内容

体系结构分析

体系结构设计

软件体系结构评价方法

体系结构发现

体系结构演化

特定领域的体系结构框架

软件体系结构支持工具

15. Spring

Spring 框架是 Java 平台的一个开源的全栈（Full-stack）应用程序框架和控制反转容器实现，一般被直接称为 Spring。该框架的一些核心功能理论上可用于任何 Java 应用，但 Spring 还为基于 Java 企业版平台构建的 Web 应用提供了大量的拓展支持。Spring 没有直接实现任何的编程模型，但它已经在 Java 社区中广为流行，基本上完全代替了企业级 JavaBeans（EJB）模型。

16. Spring Projects Covered

Spring Core Containers - Components and IoC

Spring Boot & Spring Web & Spring Mobile - MVC

Spring Cloud - Microservices

Spring Cloud Function - Serverless

Spring Batch - Pipes-and-filters

Spring Integration - Message-driven
Spring Event & Spring AMQP - Event-driven
Spring HATEOAS - RESTful
Spring Webflux and Project Reactor - Responsive

二、基于构件的软件架构：组件和 **Spring IoC**

1. 基于构件

软件工程强调软件开发过程应该采用工程化开发方法和工业化生产技术，从传统工业借鉴方法和技术是一种有效手段。

- 基于标准化构件的生产技术

汽车生产：汽车零件

建筑工程：建筑材料

- 第三方提供 **vs.** 自身积累

汽车轮胎、变速箱总成

2. 软件构件

一个软件组件，是一个具有合同规定的接口和明确的上下文依赖关系的组成单位。一个软件组件可以独立部署，并受制于第三方的组成。

类比：装机

一个组合单元、合同规定的接口、明确的上下文依赖关系、由第三方组成。

3. CBSE

基于组件的软件工程 (CBSE)，即基于组件的开发 (CBD)，是一种基于重复使用的方法，用于定义、实现和将松散耦合的独立组件组成系统。

4. CBD 的目标

- (1) 用现有的（第三方）构件组装成一个复杂的系统
- (2) 通过添加/替换部件来更新一个系统，这些组件是

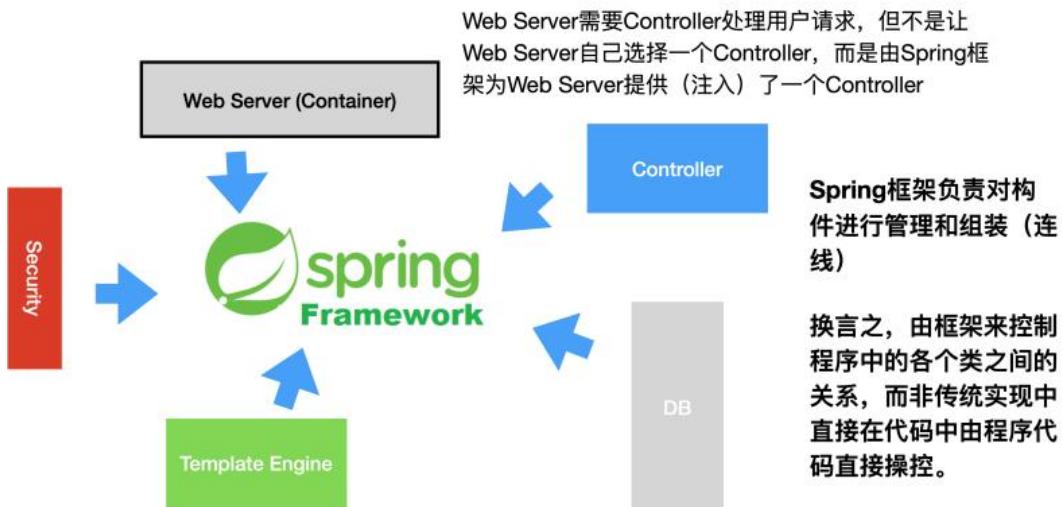
- 部署的单位

- 按原样处理（黑匣子）

5. Spring

Spring 框架是一个模块化的框架，提供了大量的“工具”来支持开发者编写基于组件的现代应用程序。

6. 合成/组装/布线



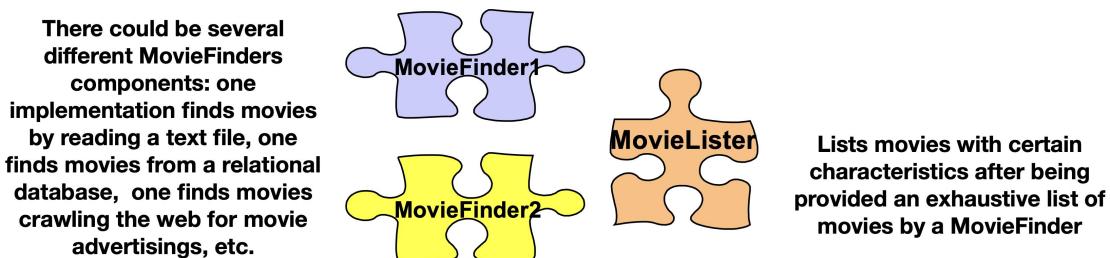
7. 目标：松散耦合的组件

MovieLister 应该与任何 MovieFinderImplementation 一起工作，MovieLister 不需要知道它所使用的特定类型的查找器实现。

- 好的解决方案：消除所有代码行，如：MovieFinder f = new MyParticularMovieFinderImpl(); 一个组件不应该创建（实例化）它的依赖关系
- 该解决方案被称为“反转控制”，一个组件不创建（实例化）它的依赖关系，而是由其他人为它创建。

8. 反转控制的概念

- 提供服务的组件是被“注入”的而不是被“直接写入”到请求者的代码中。
- 在组件方面：应用程序独立地定义了一组组件和它们的依赖关系，组件框架（称为容器）使用这些信息在运行时将组件连在一起，在生命周期的特定时间调用其代码。



9. 反转控制的几种实现模式

(1) 依赖性注入 (Dependency Injection)

汇编器将具体的实现实例化，并将它们“注入”需要它们的组件中。

(2) 服务定位器 (Service Locator)

10. 依赖性注入的形式

(1) 构造器注入

MovieLister 有一个构造器，它将获得 MovieFinderImplementation。

(2) Setter 方法注入

MovieLister 有一个 setter 方法，将获得 MovieFinderImplementation。

(3) 接口注入

一个接口 InjectFinder，有方法 injectFinder，由 MovieFinder 接口的提供者定义，MovieLister（以及任何想使用 MovieFinder 的类）需要实现这个接口。

11. 组件容器

DI 模式与组件框架有什么关系？

- (1) DI 模式的汇编器组件被称为组件容器，是组件框架的一部分。
- (2) 装配器（容器）是通用的（适用于任何应用程序），因此它：
 - 要求组件遵循一定的惯例（构造器、设定器、注入器接口）
 - 需要被告知（通过代码或配置文件）哪个实现与哪个接口相关联

12.例子：使用 Spring 框架进行 Setter 方法注入

- (1) 定义 Setter

```
class MovieLister...
```

```
public void setFinder(MovieFinder finder) {  
    this.finder = finder;  
}
```

Each class defines
setters that include
everything it needs
injected

```
class ColonMovieFinder...
```

```
public void setFilename(String filename) {  
    this.filename = filename;  
}
```

- (2) 配置文件中描述 (spring.xml)

bean 与 component 都是 Spring 管理的软件构件，我们在 demo 中看到的 controller 是一种特殊的 component。

```
<beans>  
    <bean id="MovieLister" class="spring.MovieLister">  
        <property name="finder">  
            <ref local="MovieFinder"/>  
        </property>  
    </bean>  
    <bean id="MovieFinder" class="spring.ColonMovieFinder">  
        <property name="filename">  
            <value>movies1.txt</value>  
        </property>  
    </bean>  
</beans>
```

- (3) 启动容器

```
public void testWithSpring() throws Exception {  
    ApplicationContext ctx = new FileSystemXmlApplicationContext("spring.xml");  
    MovieLister lister = (MovieLister) ctx.getBean("MovieLister");  
    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");  
    assertEquals("Once Upon a Time in the West", movies[0].getTitle());  
}
```

13.小结

- 软件构件与构件化开发
- Spring 构件化开发演示
- 构件框架核心：反转控制 / 依赖注入
- Spring bean / component

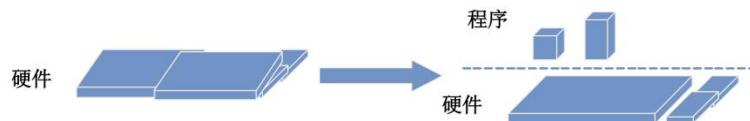
三、软件体系结构：分层架构

1.软件设计发展

初始状态: 硬件

-> 如何提高算法适应性?

-> 分离出了程序(汇编)



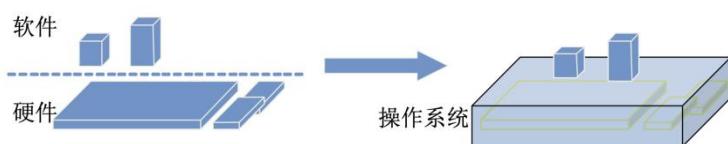
2.发展-操作系统软件

初始状态: 硬件 + 程序

程序的共性 (稳定) 成分: 计算资源管理

产生: 操作系统

分离出了: 应用程序



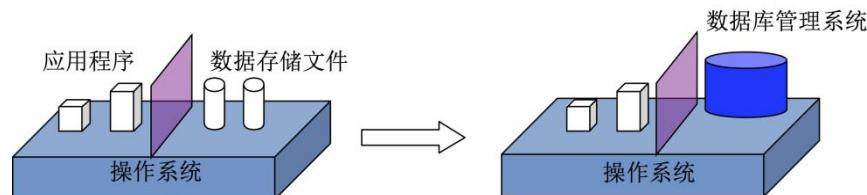
3.发展-数据库软件

初始状态: 硬件 + 操作系统 + 应用程序

程序的共性 (稳定) 成分: 数据管理

产生了: 数据库管理系统

分离出了: 应用软件



4.发展-中间件软件

初始状态: 硬件 + 操作系统 + 数据库管理系统 + 应用软件

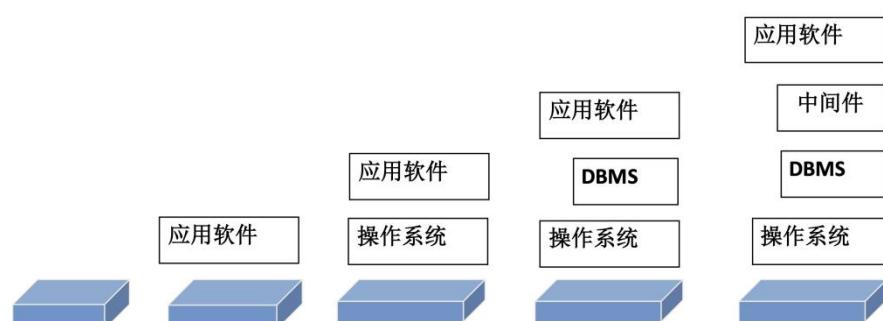
程序的共性 (稳定) 成分: 资源管理和服务

产生了: 中间件 (应用服务器)

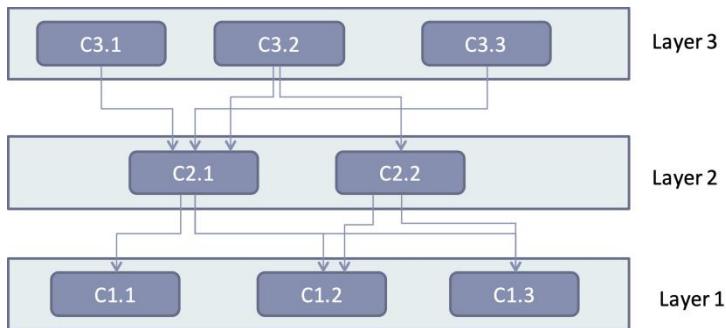
分离出了: 业务逻辑

5.发展历程

为控制软件复杂度提取软件的共性成份而沉淀下来的一层软件，屏蔽系统低层的复杂度，在高层保持复杂度的相对稳定。



6.分层结构



7.核心思想

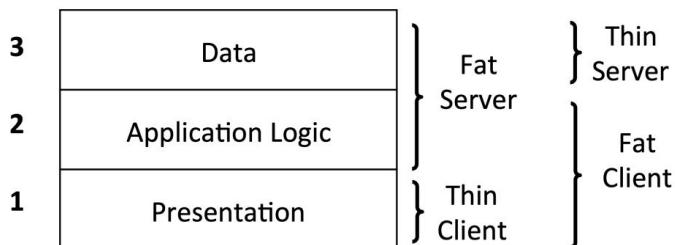
- (1) 实现了对复杂问题的分步求解
 - 自顶向下看：不断的假设过程
 - 自底向上看：不断的抽象过程
- (2) 各层语义良好
 - 具有明确的使用场景
 - 提供良好的复用条件

8.三层体系架构

- (1) 数据访问层：基本上是存储所有应用程序数据的服务器。
- (2) 业务逻辑层：主要是作为数据访问层和表现层之间的桥梁。
- (3) 表现层：表现层是用户与应用程序互动的层。

四、MVC “模型-视图-控制器” 架构

1.三层的分割



(1) 胖客户机

C/S 模式的传统形式，一般用户个人软件系统

- 应用系统在 Client 端运行
- Client 知道 Server 上的数据、文件等如何组织和存储
- 为用户端程序的设计和开发提供较大的灵活性和便利性

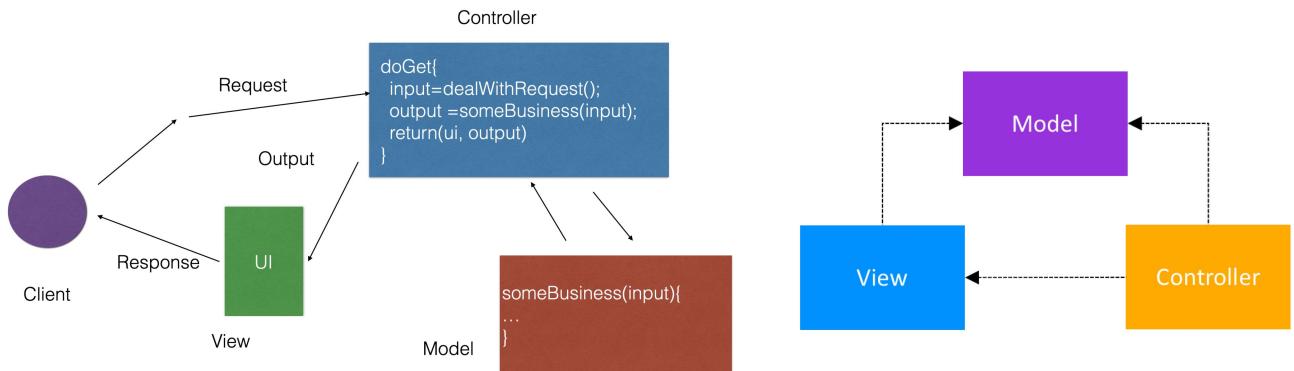
(2) 胖服务器

- Server 通过一组确定的过程提供资源访问，而非提供对资源直接操作
- Client 提供 GUI 界面供用户进行操作，并通过远程方法调用与 Server 通信，获得服务
- 应用代码集中于 Server 端，便于部署和管理，减少网络通信开销

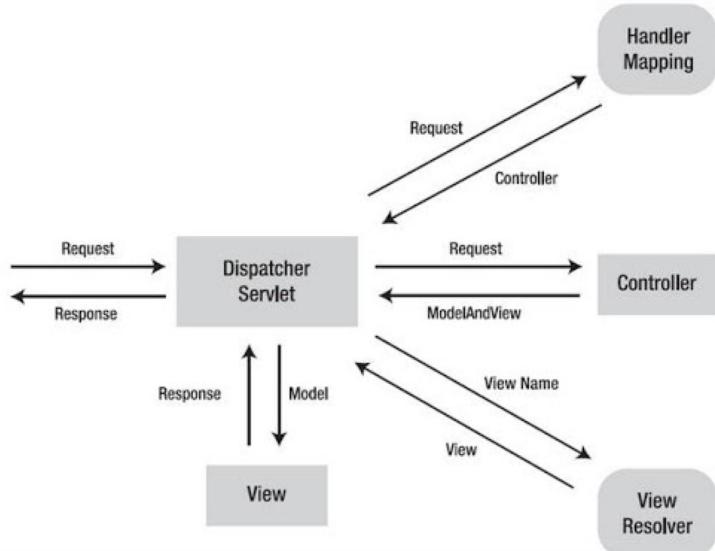
2.MVC

系统分割为 processing, output 和 input 三个部分：

- (1) “Model” : processing, 核心数据功能
- (2) “View” : output, 从 Model 获得数据显示给用户 (网页版/手机版)
- (3) “Controller” : input, 处理事件操作模型 (URL 找对应控制器)
- 过程: 当用户发出请求时, 交给控制层“Controller” → “Controller”和模型“Model”进行交互 → “Model”把自己要输出的数据交给模版渲染引擎“View”
- 好处: 可做到取得的数据一样, 但展示的形式不一样。

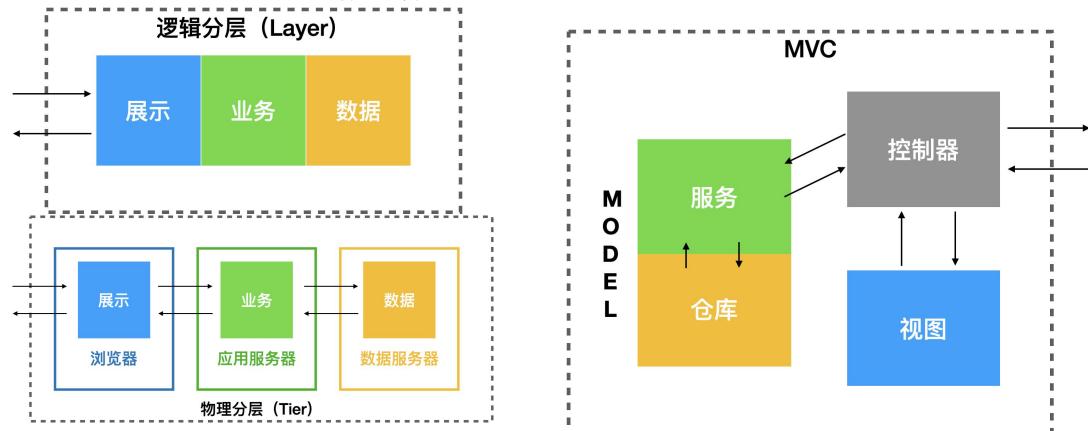


3. 识别 Spring MVC



4. MVC VS 三层体系架构

MODEL 层包含哪些功能: 数据访问层+业务逻辑层的功能



五、可扩展架构

1. 垂直扩展 (买更大的机器)

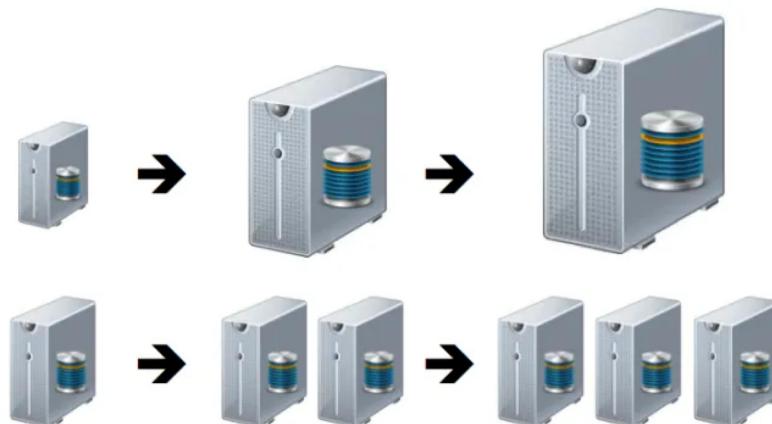
通过增强硬件设备的性能，来提升应用本身的性能。通过更新设备，找到更强的设备来提升系统性能。

(缺点：①计算机本身性能存在上限

②很不经济，随着设备性能提升，价格差别远高于提升的性能差别。)

2. 水平扩展 (买更多的机器)

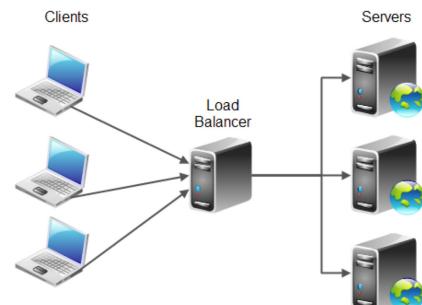
用多台机器，把他们的资源集中起来。然后用多台机器同时去服务用户的请求，从而使得用户的性能指标能满足要求。多采用“负载均衡”的技术。



3. 负载平衡

是一种计算机技术，用来在多个计算机、网络连接、CPU、磁盘驱动器或其他资源中分配负载，以达到最优化资源使用、最大化吞吐率、最小化响应时间、同时避免过载的目的。使用带有负载平衡的多个服务器组件，取代单一的组件，可以通过冗余提高可靠性。

负载平衡服务通常是由专用软件和硬件来完成。



4. L3/L4/L7 负载均衡

TCP/IP Model		OSI Model
Application Layer	HTTP, HTTPS, SMTP, IMAP, FTP, DNS, NNTP	Application Presentation Session
Transport	UDP, TCP, SCTP	Transport
Internet		Network
Network Access (Link)		Data Link Physical

Diagram illustrating the layers of load balancing:

- A blue arrow points from the Application layer to the text "Layer 7 负载均衡" (Layer 7 Load Balancing).
- A blue arrow points from the Transport layer to the text "Layer 4 负载均衡" (Layer 4 Load Balancing).
- A blue arrow points from the Network layer to the text "Layer 3 负载均衡" (Layer 3 Load Balancing).

六、微服务架构

1. 什么是微服务架构？（概念 / 定义）

微服务架构风格是一种将单个应用程序开发成一套小型服务的方法，每个服务都在自己的进程中运行，并与轻量级机制（通常是 HTTP 资源 API）进行通信。这些服务是围绕业务能力

建立的，并且可以通过完全自动化的部署机制进行独立部署。这些服务有最低限度的集中管理，它们可以用不同的编程语言编写，并使用不同的数据存储技术。

2. 微服务 VS 单例应用程序（进行水平扩展）

(1) 单例应用程序 / 聚系统：

必须把各个应用程序、各个进程在多个机器上复制。当一个系统在用户量大的情况下进行水平扩展，每个实例都要将完整的副本系统独立地运行一份，会造成资源的浪费。

(2) 微服务：

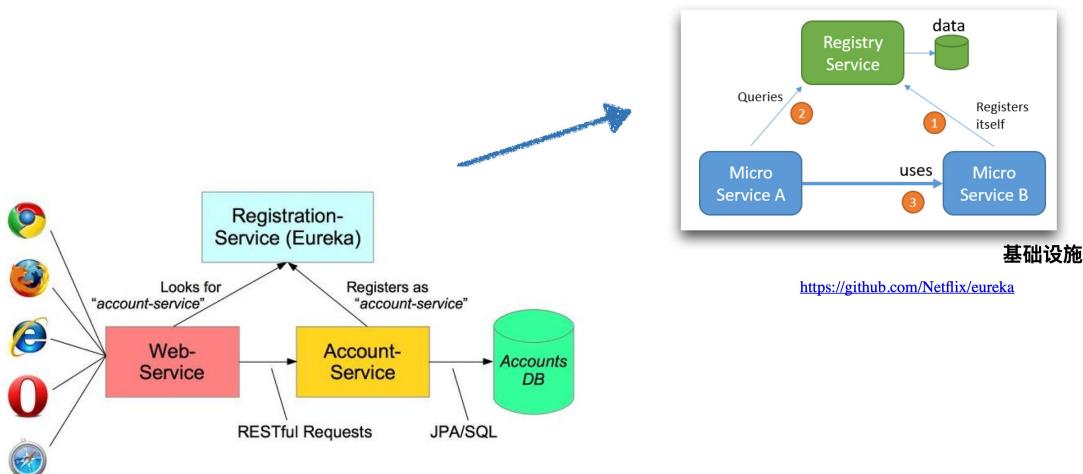
粒度会小一点，每个服务都是一个小进程，可以单独对一个小进程进行扩展。每个功能部件是一个独立的系统，在负载均衡下可以将每个模块独立地进行水平扩展。可以根据每个模块的复杂度和用户访问请求量，有效地节省系统资源，使资源利用更高效。

3. 微服务基础设施：RegistrationService（名服务）— Eureka

(1) WebService 和 AccountService 相互独立，并且 WebService 要能调用 AccountService。RegistrationService 作为名服务可以提供内部 DNF，通过固定名称去访问，避免手工查看，Eureka 则作为查找中介。AccountService 将名称服务注册到 Eureka，WebService 通过名称去 Eureka 里进行查找对应的 AccountService 是哪个 URL。

(2) 微服务之间有相互依赖，子系统之间有相互调用。有了名服务作为基础设施之后。使得一个服务可以通过名称找到另外一个服务访问的入口地址。实现了服务与服务之间的相互发现、查找和调用。

(3) 服务注册 + 负载均衡



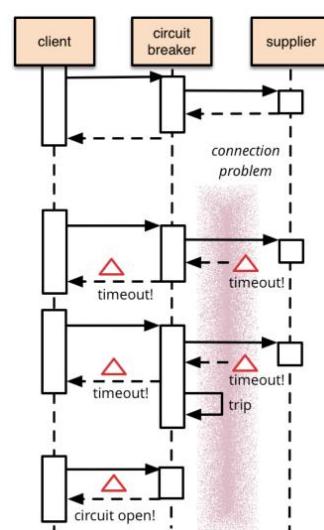
4. 断路器

(1) 成因

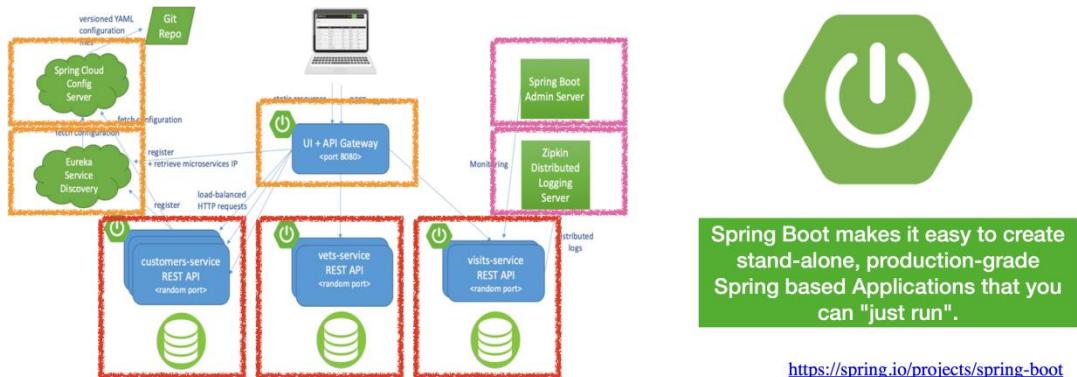
对于软件系统来说，对运行在不同进程中的软件进行远程调用是很常见的，可能是在网络上的不同机器上。内存调用和远程调用之间的一个很大的区别是，远程调用可能会失败，或者在没有响应的情况下挂起，直到达到某种超时限制。更糟糕的是，如果你在一个没有反应的供应商上有许多调用者，那么你可能会耗尽关键资源，导致跨多个系统的连带故障。

(2) 断路器的基本理念

你将一个受保护的函数调用包裹在一个断路器对象中，该对象监控故障。一旦故障达到一定的阈值，断路器就会跳闸，所有对断路器的进一步调用都会返回错误，而受保护的调用根本就没有进行过。通常情况下，你也希望在断路器跳闸时有某种监控警报。



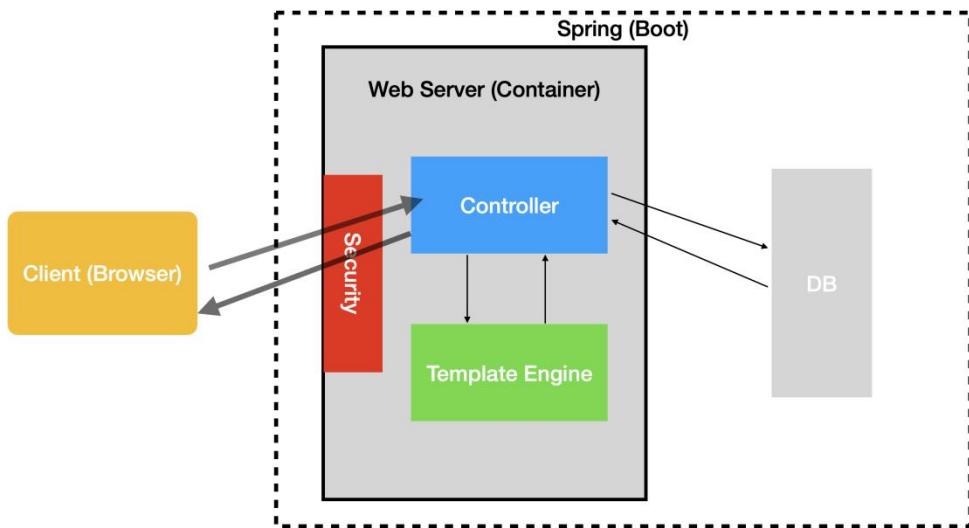
5. Spring Boot



6. Spring Starter 启动器

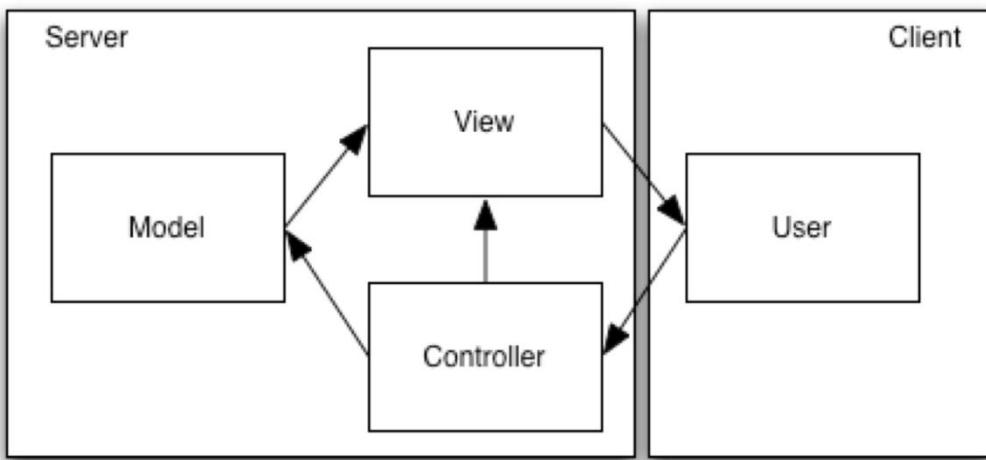
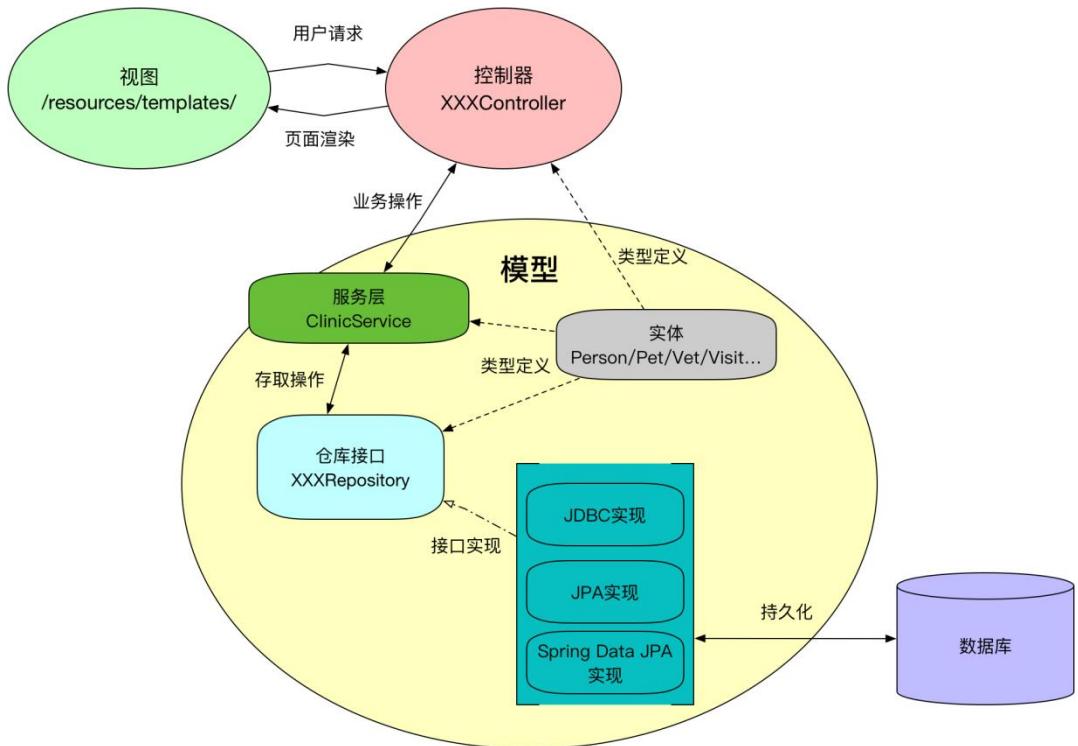
启动器是一组方便的依赖描述符，你可以在你的应用程序中包含它们。你可以获得所有你需要的 Spring 和相关技术的一站式服务，而不需要在样本代码中寻找和复制粘贴大量的依赖描述符。例如，如果你想开始使用 Spring 和 JPA 进行数据库访问，可以在你的项目中包含 `Spring-boot-starter-data-jpa` 依赖项。

7. 如何启动



七、REST 架构

1. Spring MVC 的架构



2. Spring MVC 依旧存在的问题

- (1) 展示层页面与计算逻辑混杂
- (2) 用户接口并不标准

3. REST 架构风格

架构风格和基于网络的软件架构设计。

4. 资源表现层状态转化 Resource Representational State Transfer (REST)

(1) REST 指的是一组架构约束条件和原则

- 为设计一个功能强、性能好、适宜通信的 Web 应用
- 如果一个架构符合 REST 的约束条件和原则，我们就称它为 RESTful 架构

(2) 核心概念

- 资源 (Resources)
- 表现层 (Representation)
- 状态转化 (State Transfer)

5. 资源

(1) 定义

网络上的一个实体，或者说是网络上的一个具体信息，任何事物，只要有被引用到的必要，它就是一个资源。

- 一段文本，一张图片，一首歌曲
- 数据库中的一行数据
- 一个手机号码，某用户的个人信息
- 一种服务

(2) 资源标识

要让一个资源可以被识别，需要有个唯一标识，在 Web 中这个唯一标识就是 **URI(Uniform Resource Identifier)**

<http://www.ex.com/software/releases/latest.tar.gz>

(3) **URI** 设计原则

- 易读

<http://www.oschina.net/news/38119/oschina-translate-reward-plan>

- / 表达资源的层级关系

<https://github.com/git/git/commit/e3ae056f87e1d675913d08/orders/2012/10>

- : 表示资源的同级关系

</git/block-sha1/sha1.h/compare/e3af72cda056f87e;bd63e61bdf38eb264>

- ? 表达资源的过滤

<https://github.com/git/git/pulls?state=closed>

(4) 统一资源接口 (GET / PUT / POST)

· RESTful 架构应该遵循统一接口原则，统一接口包含了一组受限的预定义的操作，不论什么样的资源，都是通过使用相同的接口进行资源的访问。接口应该使用标准的 **HTTP** 方法如 GET, PUT 和 POST，并遵循这些方法的语义。

· 如果按照 HTTP 方法的语义来暴露资源，那么接口将会拥有安全性和幂等性的特性

① GET 和 HEAD 请求是安全的，无论请求多少次，都不改变服务器状态

② GET、HEAD、PUT 和 DELETE 请求是幂等的，无论对资源操作多少次，结果总是一样的，后面的请求并不会产生比第一次更多的影响

GET

获取表示，变更时获取表示（缓存）。安全且幂等。

200 (OK) - 表示已在响应中发出
204 (无内容) - 资源有空表示
301 (Moved Permanently) - 资源的URI已被更新
303 (See Other) - 其他 (如，负载均衡)
304 (not modified) - 资源未更改 (缓存)
400 (bad request) - 指代坏请求 (如，参数错误)
404 (not found) - 资源不存在
406 (not acceptable) - 服务端不支持所需表示
500 (internal server error) - 通用错误响应
503 (Service Unavailable) - 服务端当前无法处理请求

DELETE

删除资源。不安全但幂等。

200 (OK) - 资源已被删除
301 (Moved Permanently) - 资源的URI已更改
303 (See Other) - 其他，如负载均衡
400 (bad request) - 指代坏请求
404 (not found) - 资源不存在
409 (conflict) - 通用冲突
500 (internal server error) - 通用错误响应
503 (Service Unavailable) - 服务端当前无法处理请求

POST

使用服务端管理的（自动产生）的实例号创建资源，或创建子资源，部分更新资源，如果没有被修改，则不过更新资源（乐观锁）。不安全且不幂等。

200	(OK) - 如果现有资源已被更改
201	(created) - 如果新资源被创建
202	(accepted) - 已接受处理请求但尚未完成（异步处理）
301	(Moved Permanently) - 资源的URI被更新
303	(See Other) - 其他（如，负载均衡）
400	(bad request) - 指代坏请求
404	(not found) - 资源不存在

POST

使用服务端管理的（自动产生）的实例号创建资源，或创建子资源，部分更新资源，如果没有被修改，则不过更新资源（乐观锁）。不安全且不幂等。

406	(not acceptable) - 服务端不支持所需表示
409	(conflict) - 通用冲突
412	(Precondition Failed) - 前置条件失败（如执行条件更新时的冲突）
415	(unsupported media type) - 接受到的表示不受支持
500	(internal server error) - 通用错误响应
503	(Service Unavailable) - 服务当前无法处理请

PUT

用客户端管理的实例号创建一个资源，通过替换的方式更新资源，如果未被修改，则更新资源（乐观锁）。不安全但幂等。

200	(OK) - 如果已存在资源被更改
201	(created) - 如果新资源被创建
301	(Moved Permanently) - 资源的URI已更改
303	(See Other) - 其他（如，负载均衡）
400	(bad request) - 指代坏请求
404	(not found) - 资源不存在

(5) 指导意义

统一资源接口要求使用标准的 HTTP 方法对资源进行操作，所以 URI 只应该来表示资源的名称，而不应该包括资源的操作。通俗来说，URI 不应该使用动作来描述。例如，

- POST /getUser?id=1 => GET /User/1
- GET /newUser => POST /User
- GET /updateUser => PUT /User/1
- GET /deleteUser?id=2 => DELETE /User/2

(6) 资源表现（Representation）/ 表述 / 表征

“资源”是一种信息实体，它可以有多种外在表现形式。我们把“资源”具体呈现出来的形式，叫做它的“表现层”（Representation）

- 文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以

采用二进制格式

- 图片可以用 JPG 格式表现，也可以用 PNG 格式表现

(7) 资源表述

- URI 只代表资源的实体，不代表它的形式。严格地说，有些网址最后的“.html”后缀名是不必要的，因为这个后缀名表示格式，属于“表现层”范畴，而 URI 应该只代表“资源”的位置。
- 资源的表述包括数据和描述数据的元数据，例如，HTTP 头“Content-Type”就是这样一个元数据属性
- 客户端可以通过 Accept 头请求一种特定格式的表述，服务端则通过 Content-Type 告诉客户端资源的表述形式

(8) 资源链接

- 当你浏览 Web 网页时，从一个连接跳到一个页面，再从另一个连接跳到另外一个页面，就是利用了超媒体的概念：把一个个资源链接起来
- 同样，我们在表述格式里边加入链接来引导客户端

6. 状态转移

- 状态应该区分应用状态和资源状态，客户端负责维护应用状态，而服务端维护资源状态。
- 客户端与服务端的交互必须是无状态的，并在每一次请求中包含处理该请求所需的一切信息。
- 服务端不需要在请求间保留应用状态，只有在接受到实际请求的时候，服务端才会关注应用状态。
- 这种无状态通信原则，使得服务端和中介能够理解独立的请求和响应。在多次请求中，同一客户端也不再需要依赖于同一服务器，方便实现高可扩展和高可用性的服务端。
- 客户端应用状态在服务端提供的超媒体的指引下发生变迁。服务端通过超媒体告诉客户端当前状态有哪些后续状态可以进入。

八、无服务器架构

1. 无服务器架构

- 无服务器架构是结合了第三方 “后台即服务” (BaaS) 服务的应用设计和包括在 “功能即服务” (FaaS) 平台上的受管理、短暂的容器中运行的自定义代码。
- 通过使用这些理念以及单页应用等相关理念，这种架构消除了对传统的永远在线的服务器组件的大部分需求。
- 无服务器架构可能会受益于大幅降低的运营成本、复杂性和工程准备时间，但代价是对供应商的依赖性和相对不成熟的支持服务增加。

2. JPetStore

传统上，架构里会包含一个部署了应用程序和前端的单体服务器。上述架构采用的是瘦客户端方式，所有的业务逻辑（如认证、会话管理、宠物管理等）都部署在服务器端。哪怕你将这个单体服务拆开成若干个微服务，也并没有改变这一点。



3. Serverless

- (1) 删除原用户认证代码，使用第三方认证服务（例如 Auth0.）
- (2) 使用第三方数据服务（例如 Google Firebase）管理产品数据
- (3) Client 端基于 Single Page Application 技术直接实现认证和数据访问展现
- (4) 将“查询”业务功能实现为一个“函数”，每次用户查询时启动函数运行将“购买”业务功能实现为另一个函数

4.无服务器架构主要包含的两个方面

无服务器架构是指应用程序使用第三方管理的 Function 和服务，因此不需要管理服务器。

无服务器架构主要包含了两个方面：

- BaaS (Backend as a Service, 后端即服务)：使用第三方服务（如 Firebase、Auth0）来达成目的。使用 BaaS 的应用程序通常是富客户端应用程序，如 SPA 或移动 App。客户端负责处理大部分的业务逻辑，其他部分则依赖外部服务，如认证、数据库、用户管理，等等。
- FaaS (Function as a Service, 函数即服务)：包含服务器端业务逻辑的无状态函数。这些函数运行在独立的容器里，基于事件驱动，并由第三方厂商托管，如 AWS Lambda 或者 Azure Functions.

5.优点

- 不需要管理服务器
- 自动伸缩(Scale to Zero)
- 没有运营成本
- 成本由事件驱动
- 具有较高的安全性

6.功能即服务 Function as a service (FaaS)

功能即服务是云计算服务的一个类别，它提供了一个平台，允许客户开发、运行和管理应用程序的功能，而不需要建立和维护通常与开发和启动应用程序有关的基础设施的复杂性。

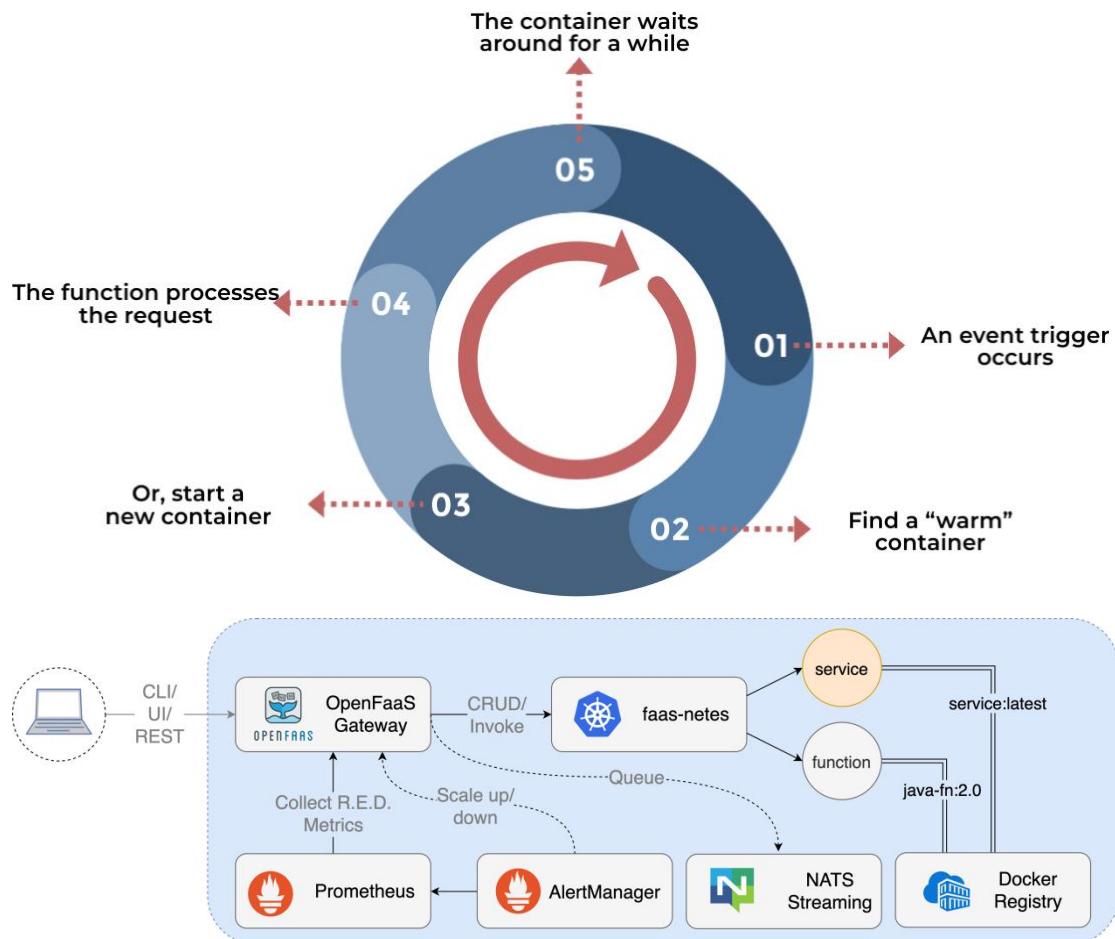
（AWS Lambda 是大型公共云供应商提供的第一个 FaaS 产品）

7.开源无服务器平台



8.OpenFaaS

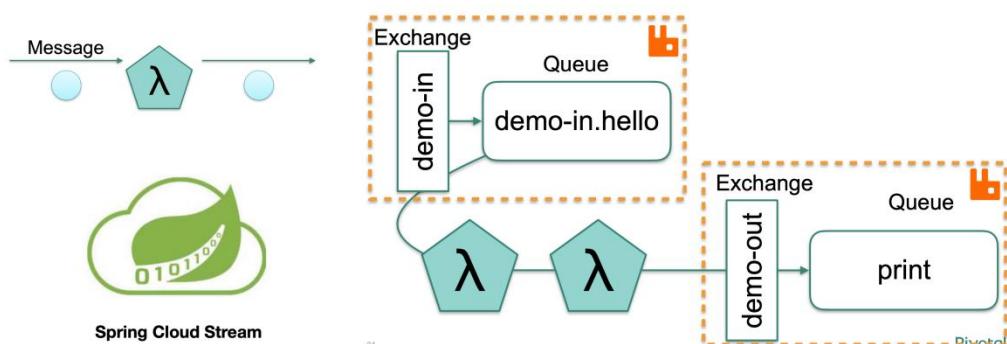
响应时间——FaaS 需要一些初始化时间。如果负载很小（比如一个小时只有一个事件），每个请求都会经历冷启动，导致整体响应变慢。



9. Spring Cloud Function

Spring Cloud Function 是来自 Pivotal 的 Spring 团队的新项目，它致力于促进函数作为主要的开发单元。该项目提供了一个通用的模型，用于在各种平台上部署基于函数的软件，包括像 Amazon AWS Lambda 这样的 FaaS (函数即服务, function as a service) 平台。

(1) 通过消息进行调用



(2) 特点

- 编程风格的选择 - 反应式、命令式或混合式。
- 函数组合和适应（例如，用反应式组合命令式函数）。
- 支持具有多个输入和输出的反应式函数，允许合并、连接等复杂的流操作由函数来处理。
- 输入和输出的透明的类型转换
- 针对目标平台，为部署打包函数

- 适配器，将函数作为 HTTP 端点暴露给外部世界等。
- 部署一个 JAR 文件，其中包含这样的应用上下文，有一个隔离的 classloader，这样就可以把它们打包在一个 JVM 中。

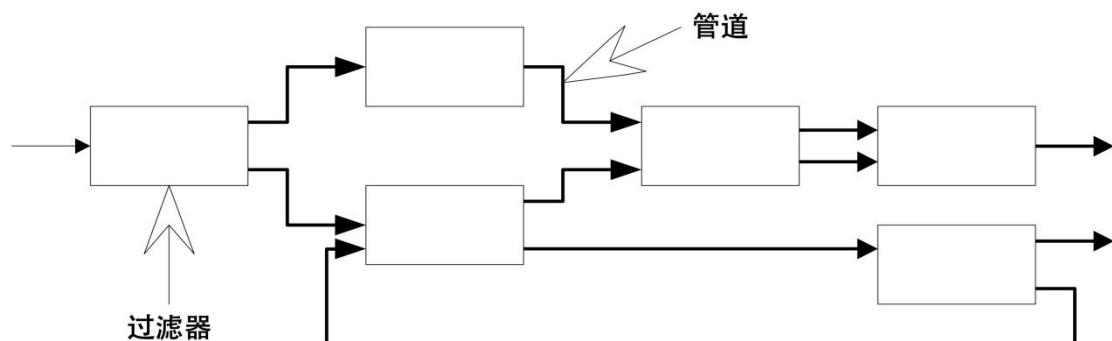
(3) 目标平台

AWS Lambda、Microsoft Azure、Apache OpenWhisk 以及可能的其他 "无服务器" 服务提供商的适配器。

九、“管道-过滤器”架构

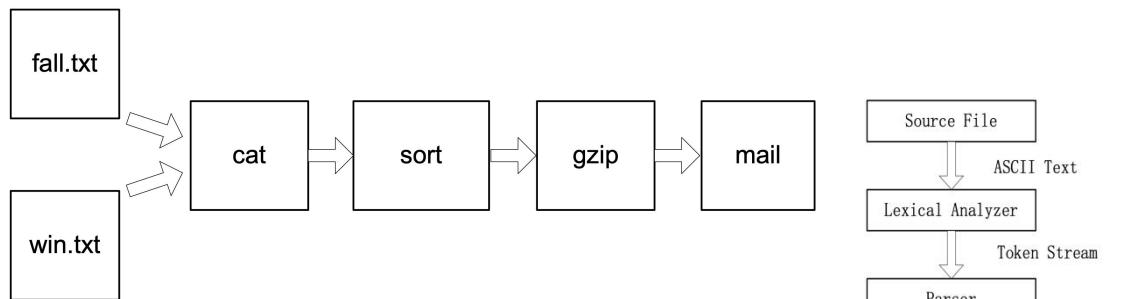
1. 数据处理场景

管道+过滤器，实现数据的多步转化

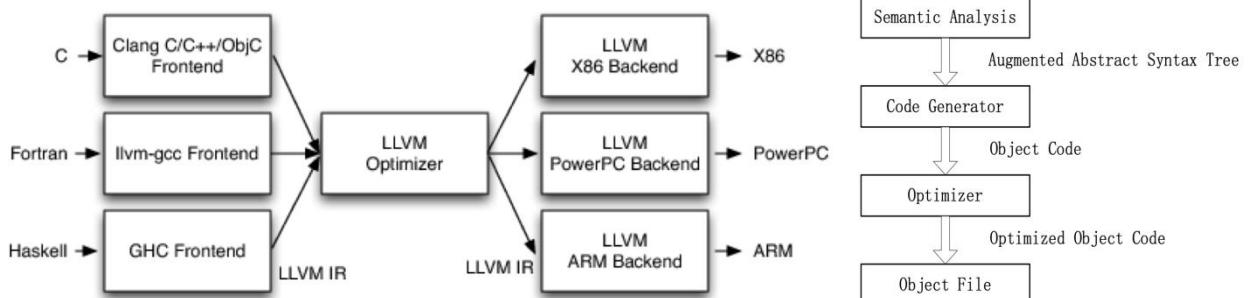


2. UNIX Command Pipelines

root@linux: cat fall.txt win.txt | sort | gzip | mail fred@byu.edu



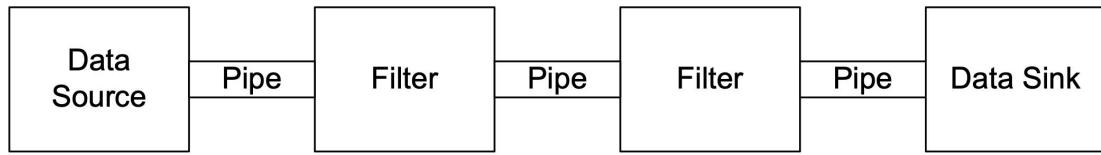
3. LLVM 的三阶段设计的实现



4. “管道-过滤器”架构模式

组成: Pipes / Filters / Data Source / Data Sink

- Filter 完成单步数据处理功能
- Data Source/Data Sink/Filter 以 Pipe 连接
- Pipe 连接相邻元素，前一元素的输出为后一元素的输入



5.过滤器 (Filters)

过滤器是流水线的处理单元，负责丰富、提炼或转换他的输入数据。它以下面的三种方式工作：

- 后继单元从过滤器中拉出数据（被动式）
- 前序单元把新的输入数据压入过滤器（被动式）
- 从前序单元拉出输入数据并且将其输出数据压入后继（主动式）

6.管道 (Pipes)

- 管道表示过滤器之间的连接；数据源和第一个过滤器之间的连接；以及最后的过滤器和数据宿之间的连接。
- 如果管道连接两个主动过滤器，那么管道需要进行缓冲和同步。

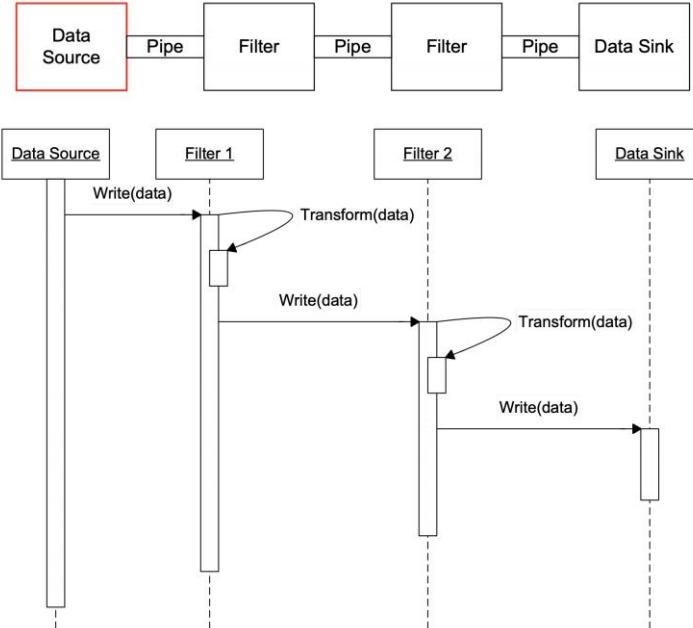
7.数据源/数据汇点 (Data Source/Sink)

- 数据源表示系统的输入，它提供一系列相同结构或者类型的数值
 - 数据源可以主动把数据值推入 (push) 第一个处理阶段，也可以由第一个处理阶段主动获取 (pull) 数据
- 数据汇点收集来自流水线终点的结果
 - 主动数据汇点可以获取数据
 - 被动数据汇点允许前面的过滤器把结果推进汇点

8.不同主体的主动/被动情况

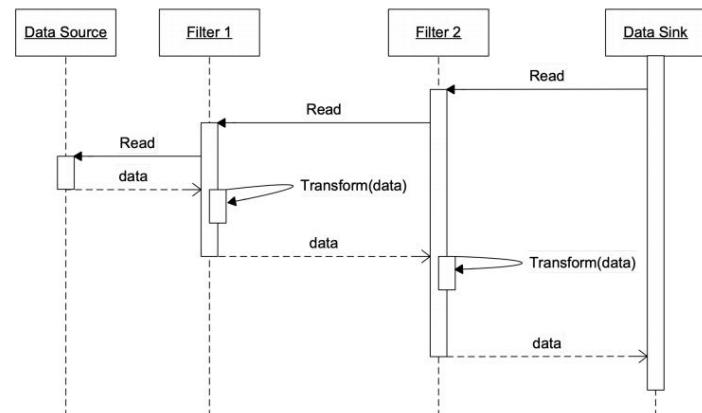
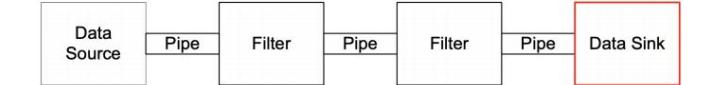
Case I

Active Source
/ Passive Filter
/ Passive Sink



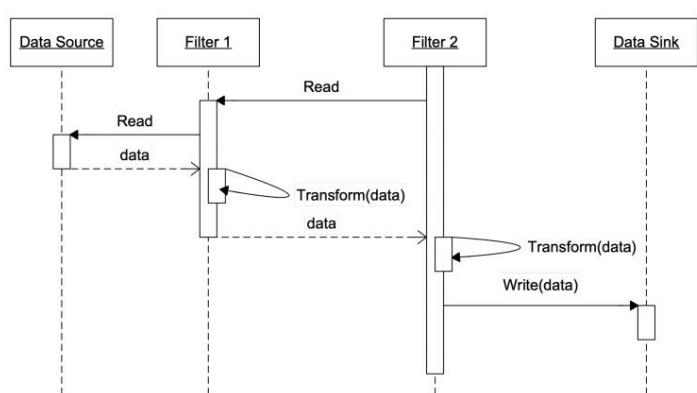
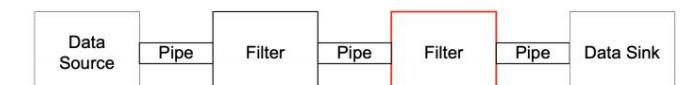
Case II

Passive Source
/ Passive Filter
/ Active Sink



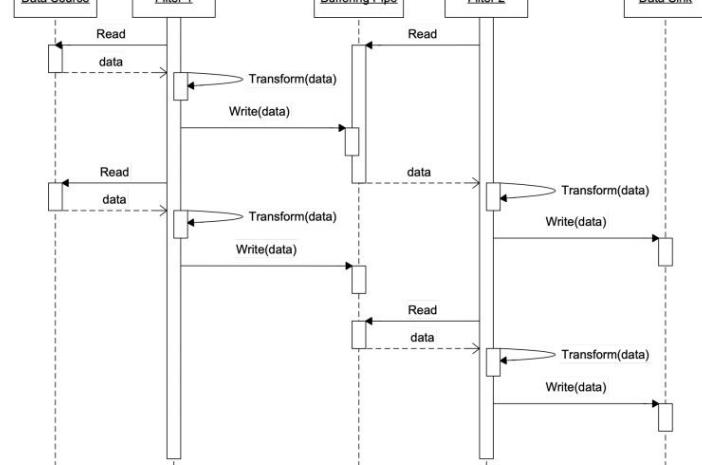
Case III

Passive Source
/ Active Filter
/ Passive Sink



Case IV

Passive Source
/ Multiple
Active Filters /
Passive Sink

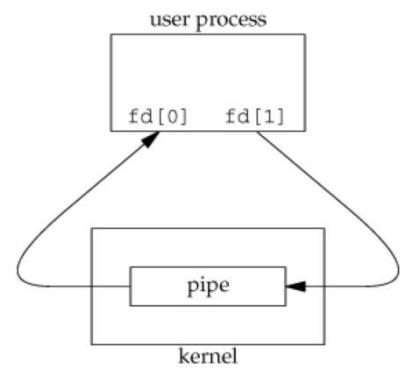
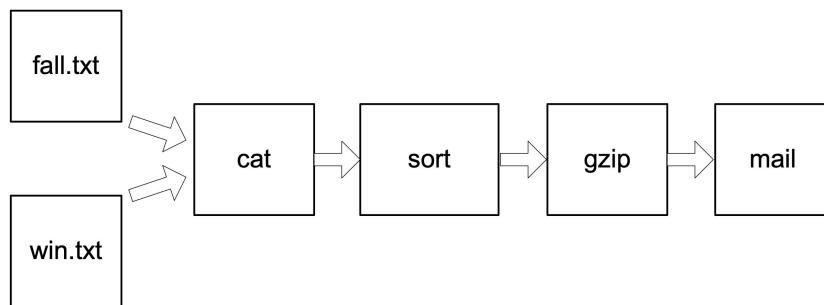


9.优缺点

- 优点
 - 过滤器可以重用/重组合/可替换
 - 不需保存中间结果
 - 高效的并行处理 (多 active 部件)
- 缺点
 - 数据传输开销较大
 - 数据转换开销较大
 - 错误处理较为复杂

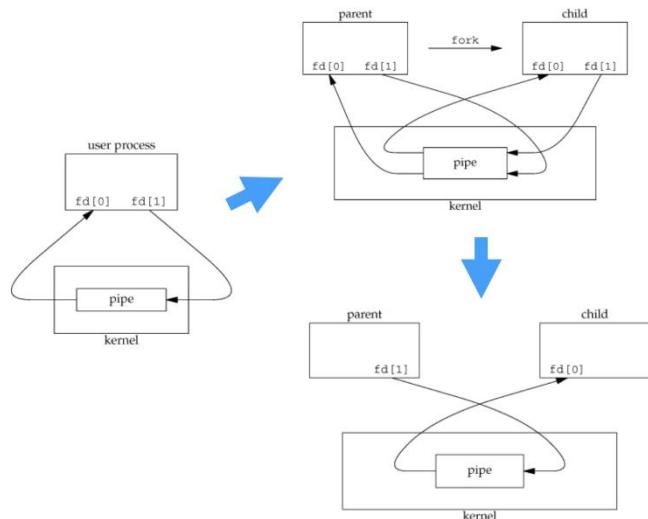
10.Unix Pipe

root@linux: cat fall.txt win.txt | sort | gzip | mail fred@byu.edu

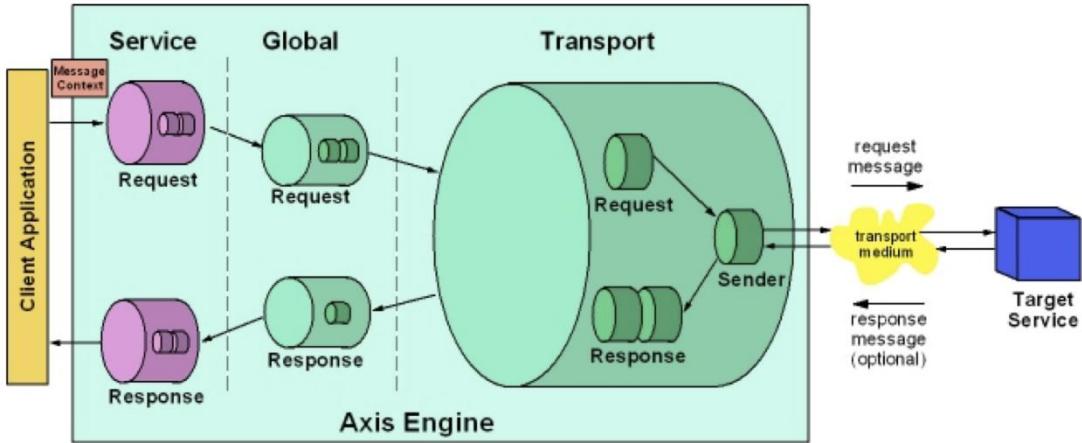


11.进程间通信

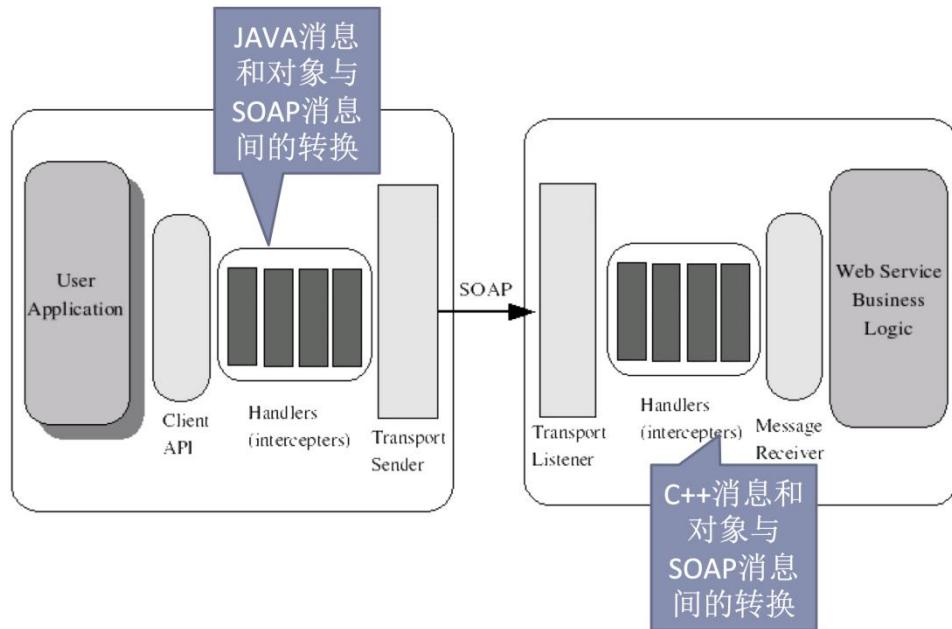
```
# include "apue.h"
int main(void){
    int n;
    int fd[2];
    pid_t pid;
    char line[MAXLINE];
    if (pipe(fd)<0)
        err_sys("pipe error");
    if ((pid = fork())<0)
        err_sys("fork error");
    else if (pid >0) {
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
    }
    exit(0);
}
```



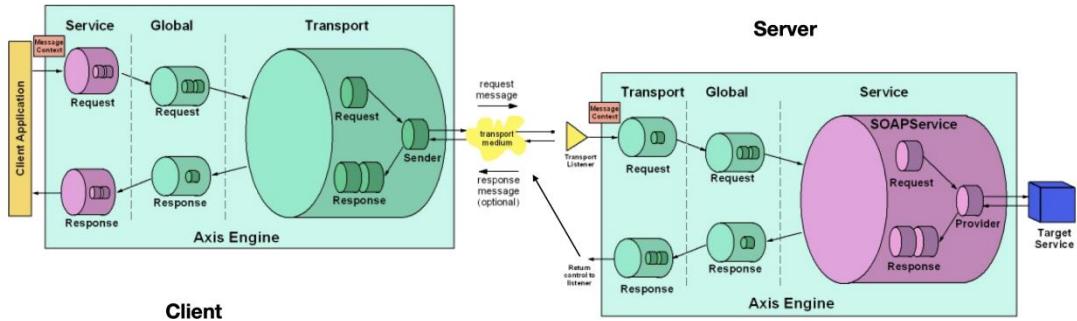
12.Axis



13. Axis Handlers (Filters)



14. Axis Architecture



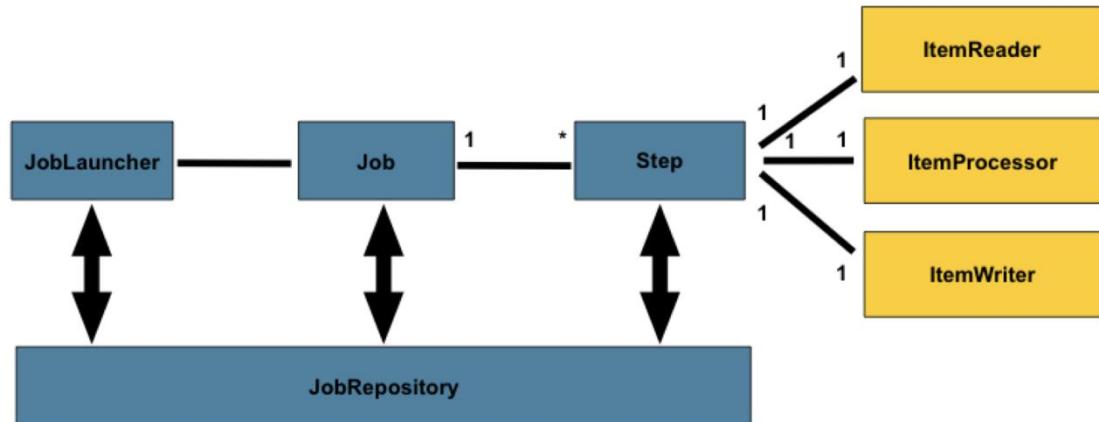
15. Spring 批处理

- 自动、复杂地处理大量的信息，这些信息在没有用户互动的情况下得到最有效的处理，包括基于时间的事件（如月末计算、通知或信件）
- 定期应用复杂的业务规则，在非常大的数据集上进行重复处理（例如：保险福利确定/费率调整）
- 整合从内部和外部系统收到的信息，这些信息通常需要格式化、验证，并以交易的方式处

理到记录系统中。

16. Batch Stereotypes

一个作业有一个到多个步骤，每个步骤正好有一个 ItemReader、一个 ItemProcessor 和一个 ItemWriter。一个作业需要被启动（用 JobLauncher），关于当前运行进程的元数据需要被存储（在 JobRepository）。

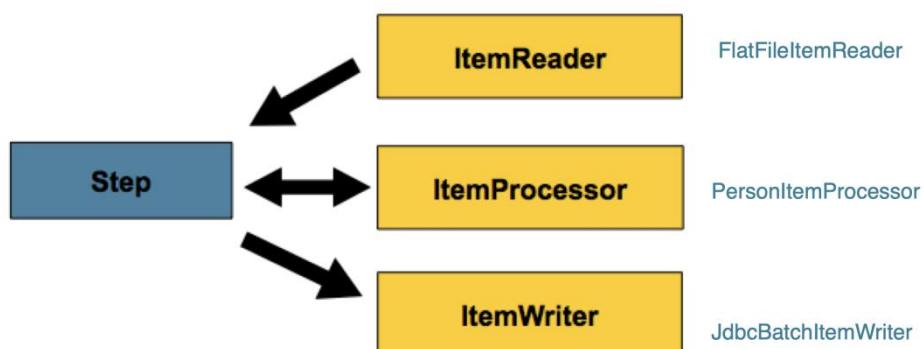


17. 流畅的接口

在软件工程中，流畅的接口是一种面向对象的 API，其设计广泛地依赖于方法链。它的目标是通过创造一种特定领域的语言来提高代码的可读性。这个术语是由 Eric Evans 和 Martin Fowler 在 2005 年创造的。

18. 批处理步骤

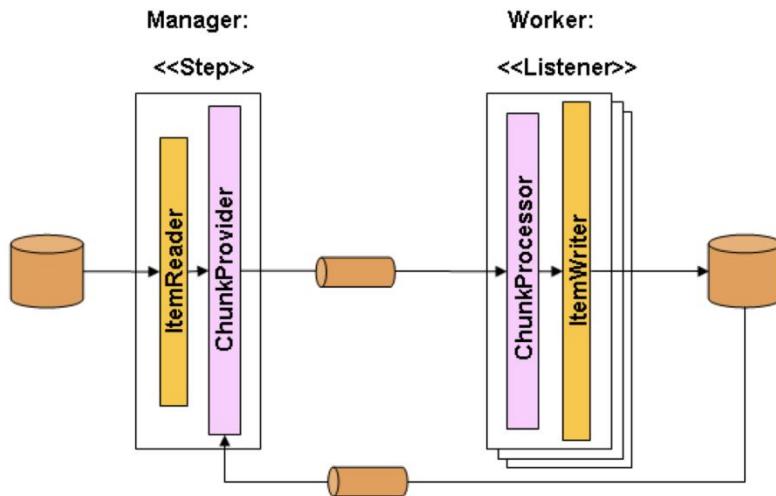
步骤是一个领域对象，它封装了批处理作业的一个独立的、连续的阶段，并包含定义和控制实际批处理的所有必要信息。



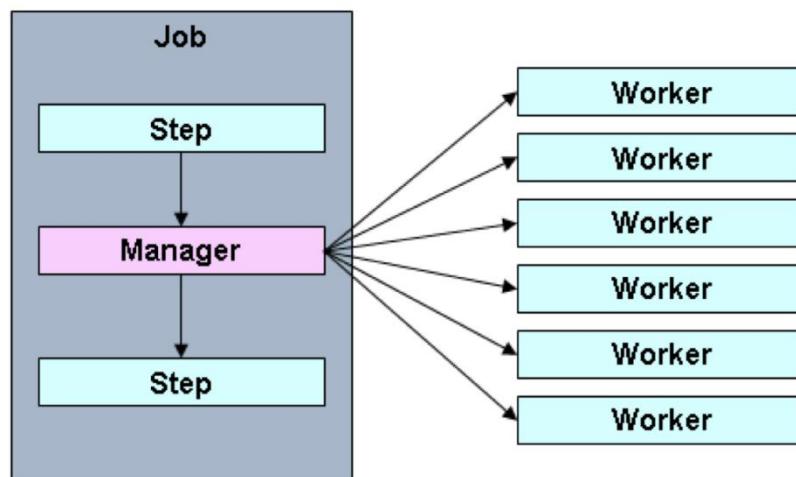
19. 缩放和并行处理

- 多线程的步骤 (单进程)
- 并行步骤(单进程)
- 步骤的远程分块 (多进程)
- 分割一个步骤 (单进程或多进程)

20. 远程分块 Remote Chunking



21.分区 Partitioning



十、事件驱动架构

1.事件驱动

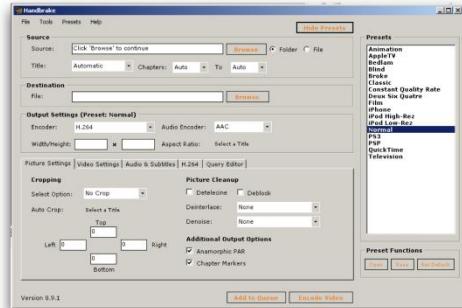
生活中的“事件驱动”：

- 银行存款：更新余额/用户等级提升/贷款广告推送
- 信用卡消费：信用额度检查/风险控制/消费通知短信发送
- 航班延误：广播通知乘客/后续行程更新/酒店订单更新

2.事件驱动架构 (Event-driven Architecture)

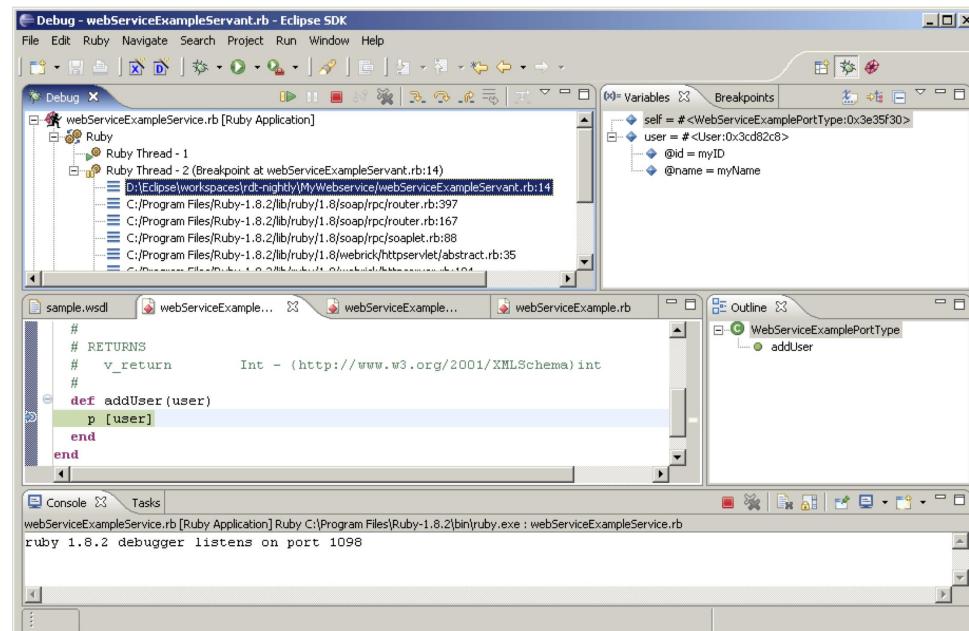
- 事件驱动架构 (EDA) 是一种软件架构范式，促进事件的生产、检测、消费和反应。
- 一个事件可以被定义为“状态的重大变化”。例如，当一个消费者购买一辆汽车时，汽车的状态从“待售”变为“已售”。一个汽车经销商的系统架构可以将这种状态变化视为一个事件，其发生可以让架构内的其他应用程序知道。

3.事件驱动的 GUI



4.事件驱动的调试器

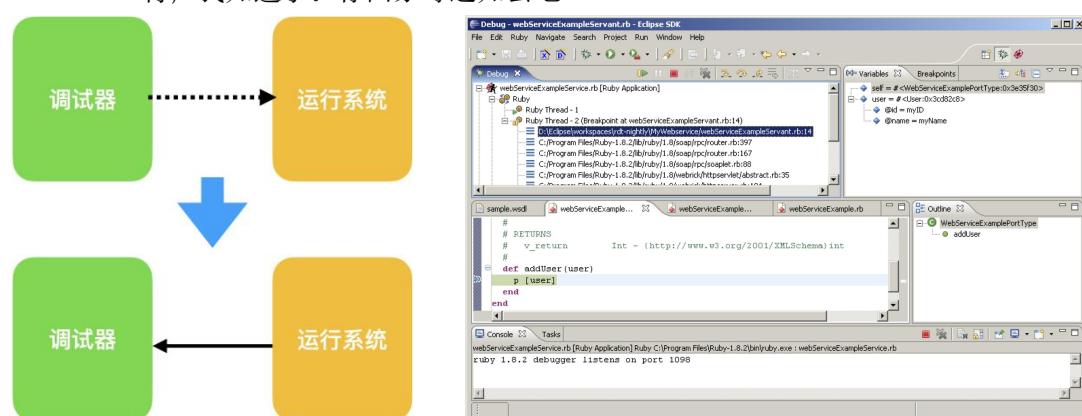
设置断点，程序运行到指定点后通知调试系统



5.反转控制关系

我主动设置断点后，由运行系统来主动通知你来观察这个事件。主观上认为是我主动去调试这个系统，实际上是由运行系统来通知你，推动了这个过程。

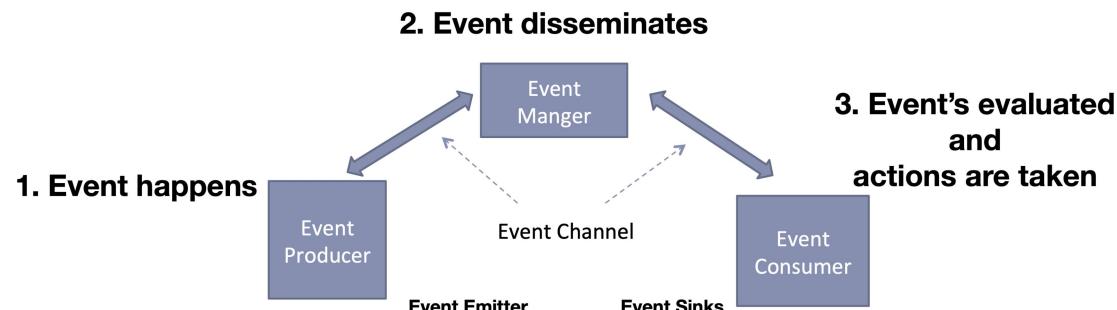
belike：“行，我知道了。你回家等通知去吧～”



6.事件驱动架构

整个事件的运行，是靠事件发生来驱动的。

事件的生产方会产生一个事件，这个事件会由事件管理器散布出去，告诉若干个对该事件已经进行过注册的事件的消费方。然后由事件的消费方，对该事件做出评估，决定要不要发生响应。如果要，就执行响应动作，如果不，就不做处理。



7.事件：“状态的显著变化”

- 标题：描述事件发生的内容
 - 事件类型
 - 事件名称
 - 事件时间戳
 - 事件创建者
- 主体：描述正在发生的事情
 - 事件被触发的原因
 - 为有关方面提供的事件信息

8.松耦合

事件的生产方、事件管理器、事件的消费方，三个部分职责明确，彼此之间没有预判和假设。

- 事件生产器不知道事件的消费者，它甚至不知道消费者是否存在，如果存在的话，它也不知道事件如何被使用或进一步处理。
- 事件管理器只有责任在事件发生时，立即应用反应。
- 关于事件正确分布的知识完全存在于事件通道中。事件通道的物理实现可以基于传统的组件，如面向消息的中间件或点对点通信。

9.Java AWT

- 事件来源
 - 按钮、滚动条等
- 事件
 - 鼠标点击/移动
 - 窗口调整大小/移动
 - 键盘按下/松开

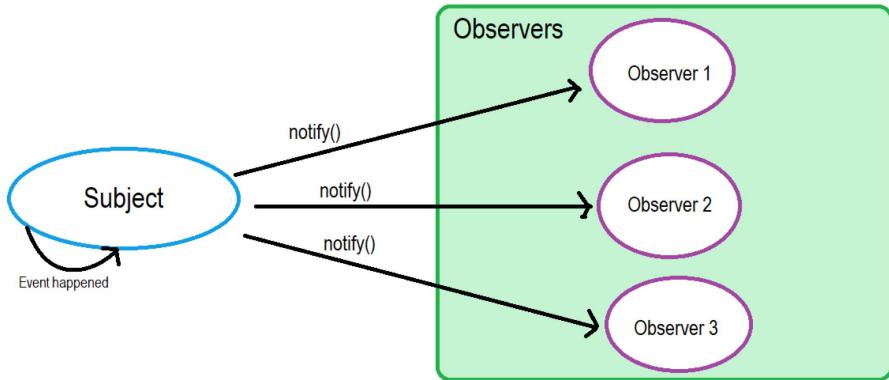
10.AWT 事件处理模型

- 标准 AWT Listener
 - ActionListener/AdjustmentListener/ComponentListener/...
- 事件源在特定事件发生时会触发特定 Listener 的特定方法
- 用户对于事件的处理包括两个步骤
 - 实现一个 Listener 或继承一个 Adapter
 - 利用事件源提供的 addXXXListener 来将自己实现的事件处理与事

- 件源可能产生的事件绑定起来

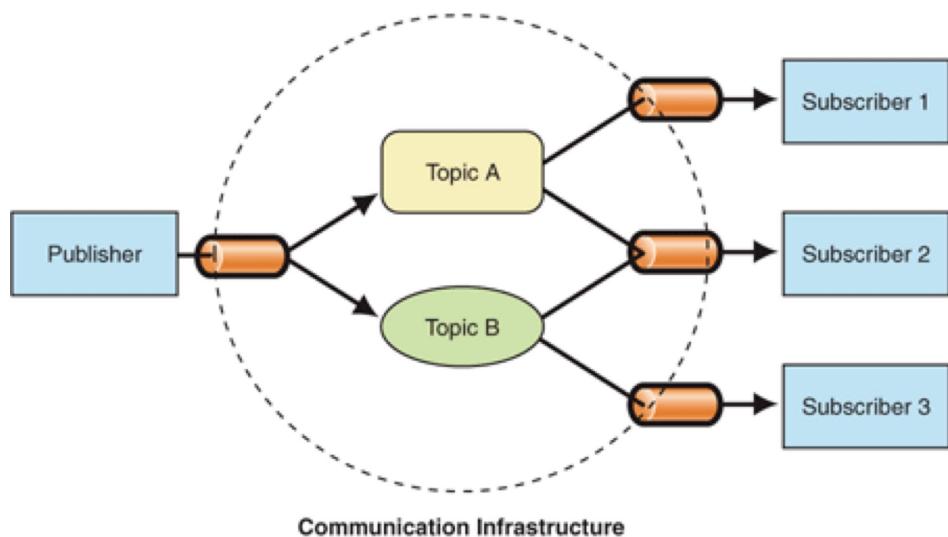
11. 观察者模式

事件源本身，既充当事件源，又充当事件管理器。由于事件观察者对事件进行了注册，事件源在发生时逐个通知事件观察者，简化了的由三部分组成事件驱动架构。（发布者和观察者耦合度会高一些）

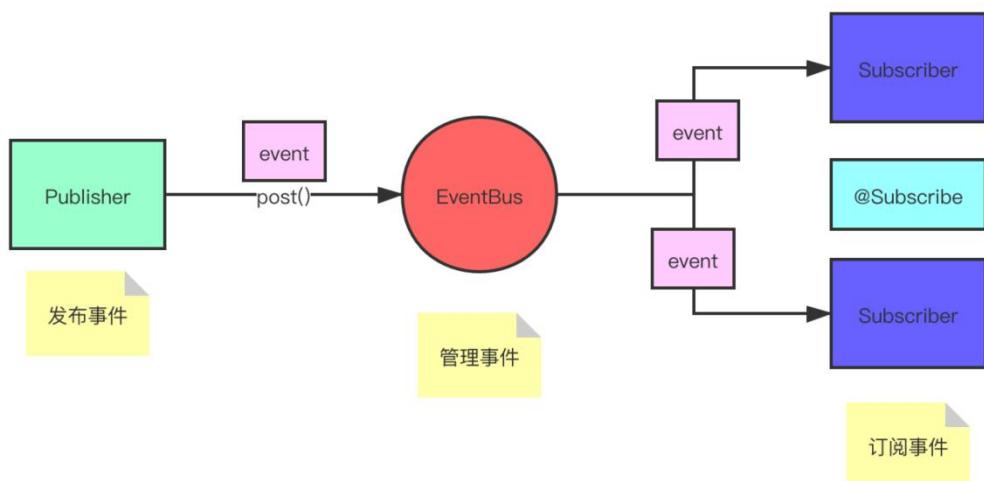


12. 发布/订阅模式

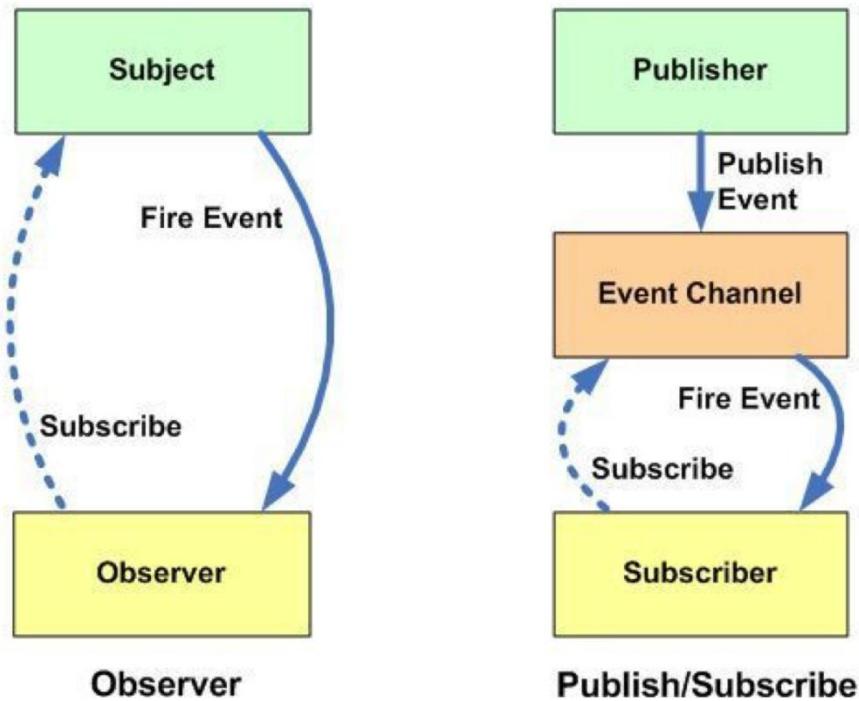
有独立的通信机制，实现事件的分发过程。



13. Guava EventBus



14. 观察者 vs 发布/订阅



15. Spring Events (典型的观察者模式的实现)

ApplicationContext 中的事件处理是通过 **ApplicationEvent** 类和 **ApplicationListener** 接口提供。如果一个实现 **ApplicationListener** 接口的 Bean 被部署到上下文中，每当一个 **ApplicationEvent** 被发布到 **ApplicationContext** 中，该 Bean 就会被通知。从本质上讲，这就是标准的观察者设计模式。

16. 标准事件

```

ContextRefreshedEvent
ContextStartedEvent
ContextStoppedEvent
ContextClosedEvent
RequestHandledEvent
ServletRequestHandledEvent

```

17.事件管理器/事件通道

- 事件通道是将事件从事件发射者传送给事件消费者的管道...
- 事件通道的物理实现可以基于传统组件，如面向消息的中间件...
- 面向消息的中间件（MOM）是支持分布式系统之间发送和接收消息的软件/硬件基础设施

18.消息中间件产品

Pivotal

 RabbitMQ


http://zero.mq/





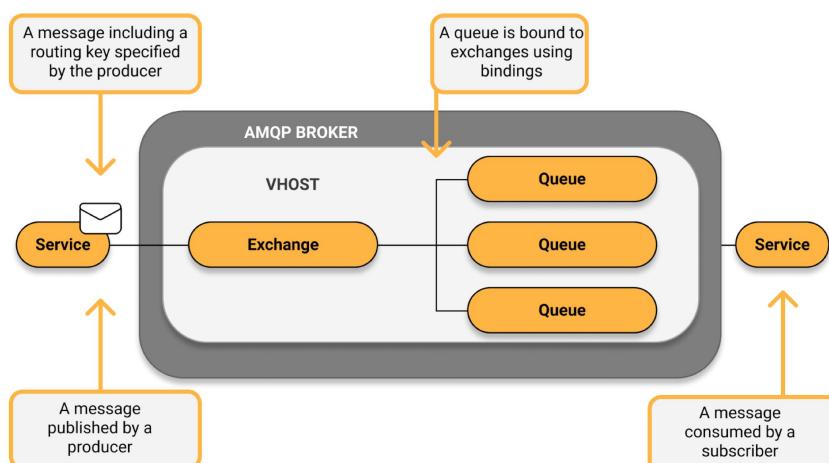
 APACHE
kafka®
A distributed streaming platform

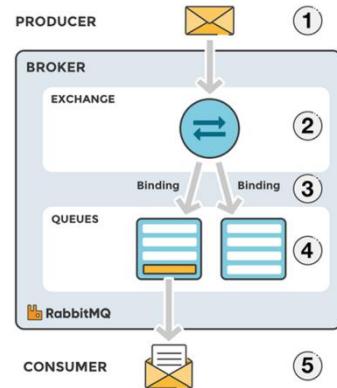


19.AMQP

高级消息队列协议即 Advanced Message Queuing Protocol 是面向消息中间件提供的开放的应用层协议，其设计目标是对于消息的排序、路由、保持可靠性、保证安全性。

（Exchange 规定什么样的消息，应该交给什么样的队列；队列应该保存消息，直到应用去取这个消息。可类比：信箱收信）





20.RabbitMQ

- (6) 生产者向交易所发布一条信息
- (7) 交易所收到该消息并负责消息的路由
- (8) 队列和交易所之间必须建立绑定，这里我们有两个不同的队列的绑定，交易所将消息路由到这两个队列
- (9) 消息停留在队列中，直到它们被消费者处理
- (10) 消费者处理消息。

21.Spring AMQP

- **AMQP 实体** - 我们用消息、队列和交换类创建实体、绑定和交换类来创建实体
- **连接管理** - 通过使用一个 CachingConnectionFactory 连接到我们的 RabbitMQ 中间件
- **消息发布** - 我们使用 RabbitTemplate 来发送消息
- **消息消费** - 我们使用 @RabbitListener 来从队列中读取消息

Event-driven systems

Event-driven systems reflect how modern businesses actually work—thousands of small changes happening all day, every day.

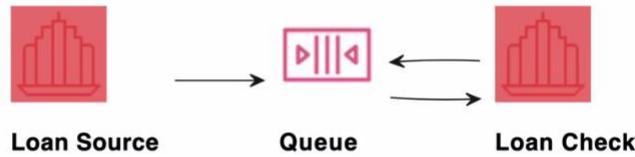
Spring's ability to handle events and enable developers to build applications around them, means your apps will stay in sync with your business.

Event-driven Spring Applications

Spring has a number of options to choose from, from integration and streaming all the way to cloud functions and data flows.

- Event-driven microservices
- Streaming data
- Integration

Event-driven microservices



<https://spring.io/blog/2019/10/15/simple-event-driven-microservices-with-spring-cloud-stream>

Spring Cloud Stream

Spring Cloud Stream is a framework for building highly scalable event-driven microservices connected with shared messaging systems.

The framework provides a flexible programming model built on already established and familiar Spring idioms and best practices, including support for persistent pub/sub semantics, consumer groups, and stateful partitions.

<https://spring.io/projects/spring-cloud-stream>

Binder

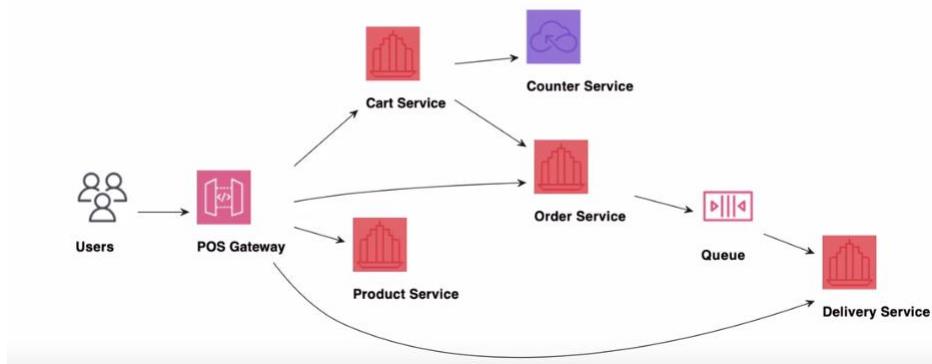
Spring Cloud Stream supports a variety of binder implementations.

- RabbitMQ
- Apache Kafka
- Google PubSub
- ...



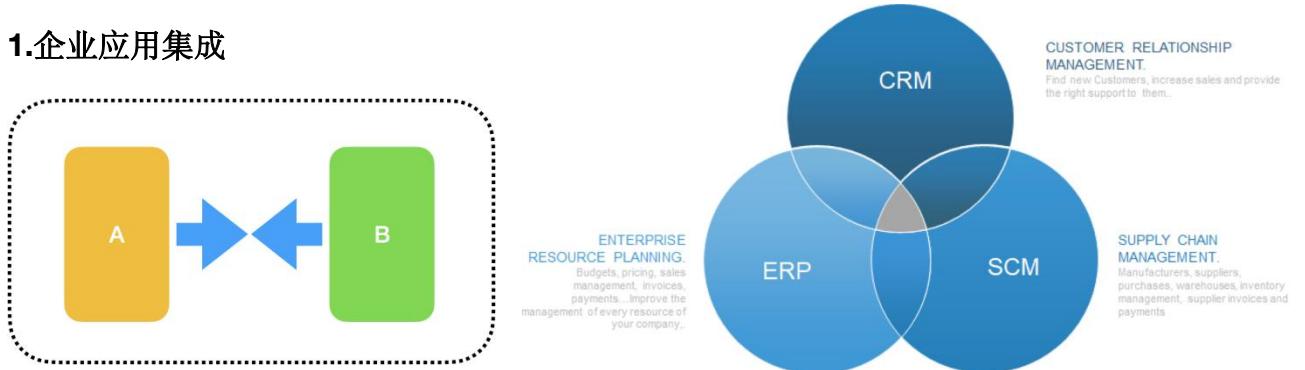
sa-spring/stream-loan

Event-driven MicroPoS



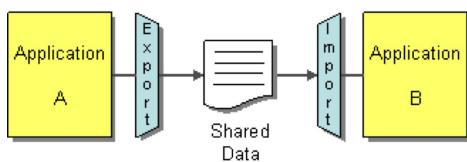
十一、企业应用集成

1.企业应用集成

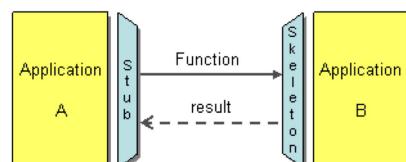


2.四大集成模式

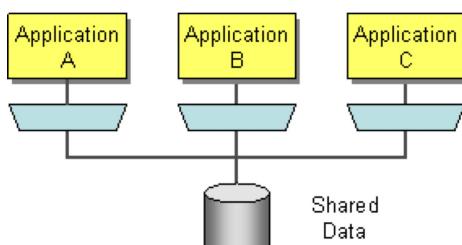
文件传输 / 远程接口调用 / 共享数据库 / 消息通信



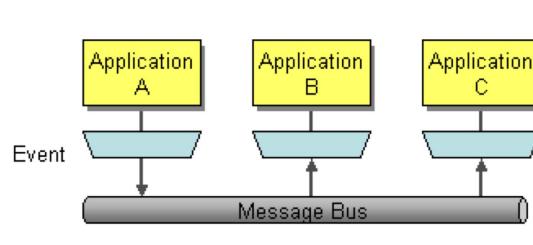
File Transfer



Remote Procedure Invocation



Shared Database



Messaging

3.消息传递模式

该模式以消息为中心 — 离散的数据有效载荷, 通过预定的渠道从一个源系统或流程移动到

一个或多个系统或流程。

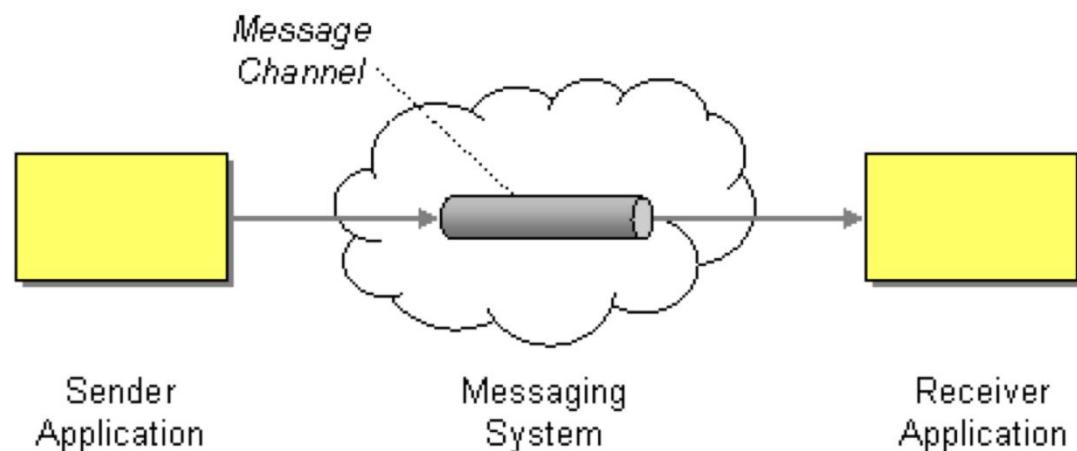
4. 消息传递模式的优点

该模式是以一种**最灵活**的方式来整合多个不同的系统：

- 几乎完全解耦参与集成的系统
- 允许参与集成的系统对彼此的底层协议、格式化或其他实现细节**完全不了解**
- 鼓励参与集成的组件的开发和重复使用

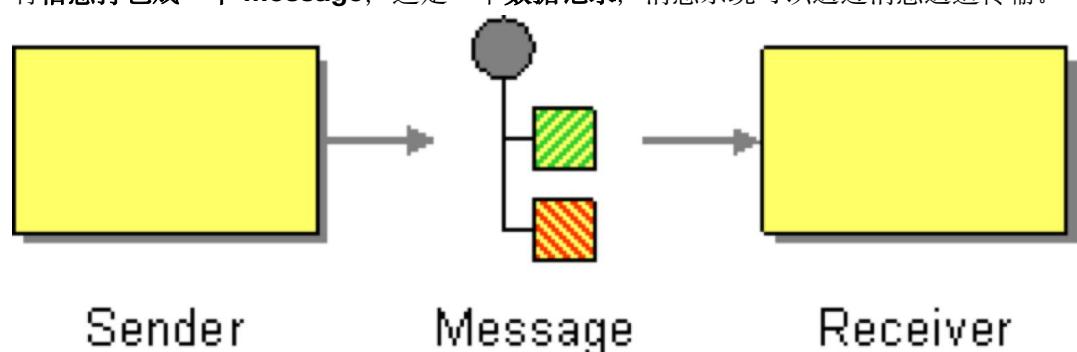
5. 核心：消息通道（Message Channel）

使用消息通道连接应用程序，其中一个应用程序向通道写入信息，另一个则从通道读取该信息。



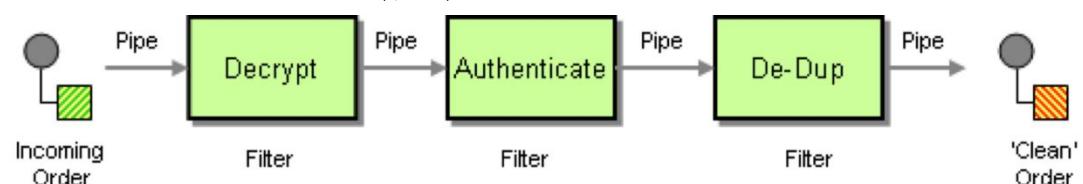
6. 信息封装：Message

将信息打包成一个**Message**，这是一个数据记录，消息系统可以通过消息通道传输。



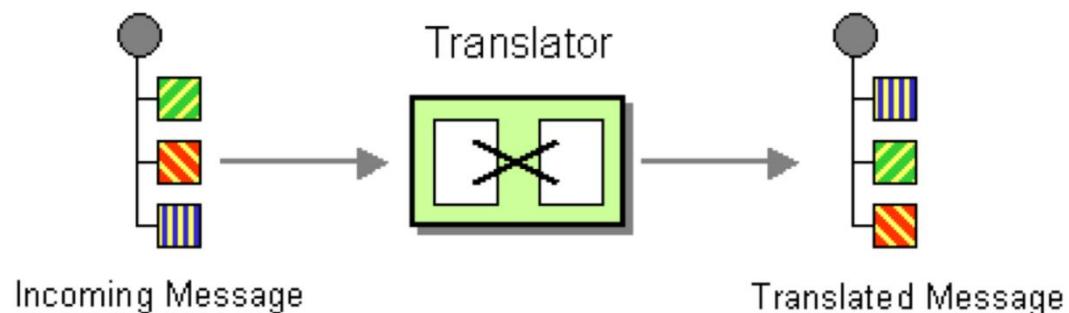
7. 处理过程：Pipes and Filters

使用管道和过滤器架构风格，将一个较大的处理任务划分为一系列较小的、独立的处理步骤（过滤器），这些步骤由通道（管道）连接。



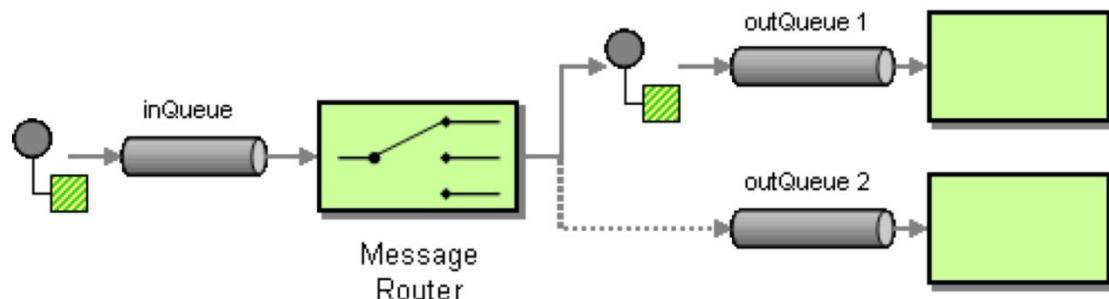
8.转换逻辑: Message Translator

在其他过滤器或应用程序之间使用一个特殊的过滤器，即信息翻译器，将一种数据格式翻译成另一种。



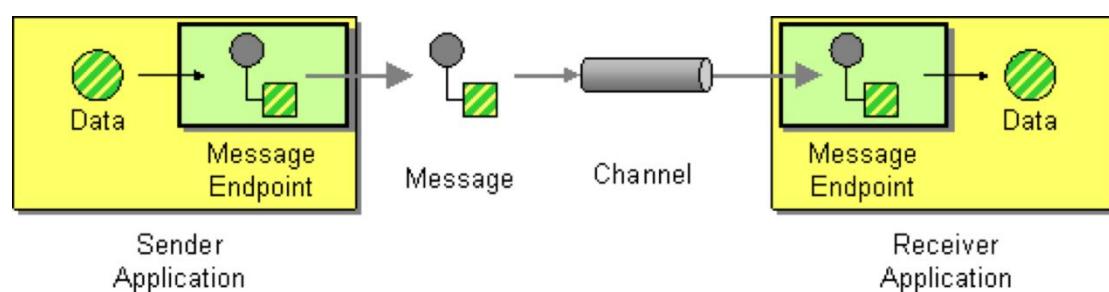
9.处理条件: Message Router

插入一个特殊的过滤器，即消息路由器，它从一个消息通道消费一个消息，并根据一组条件将其重新发布到不同的消息通道。

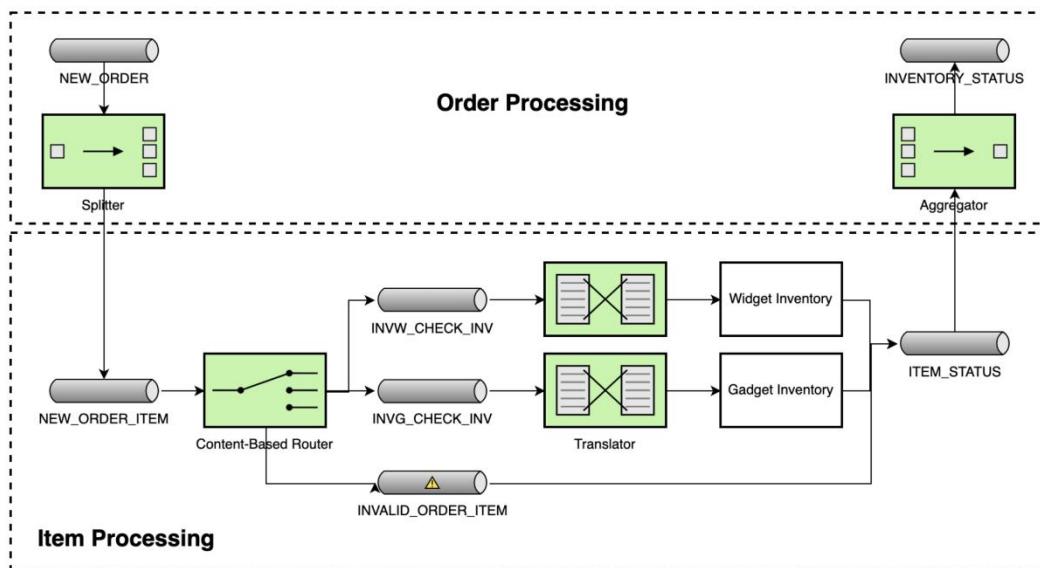
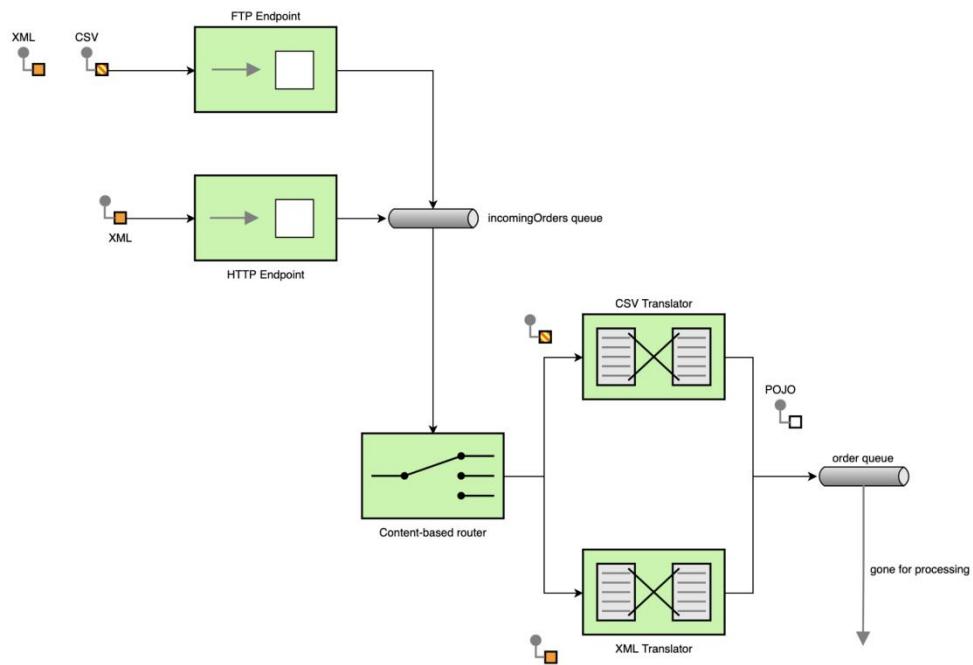


10.源/汇: Message Endpoint

使用消息端点将应用程序连接到消息通道，消息端点是消息系统的一个客户端，然后应用程序可以使用它来发送或接收消息。



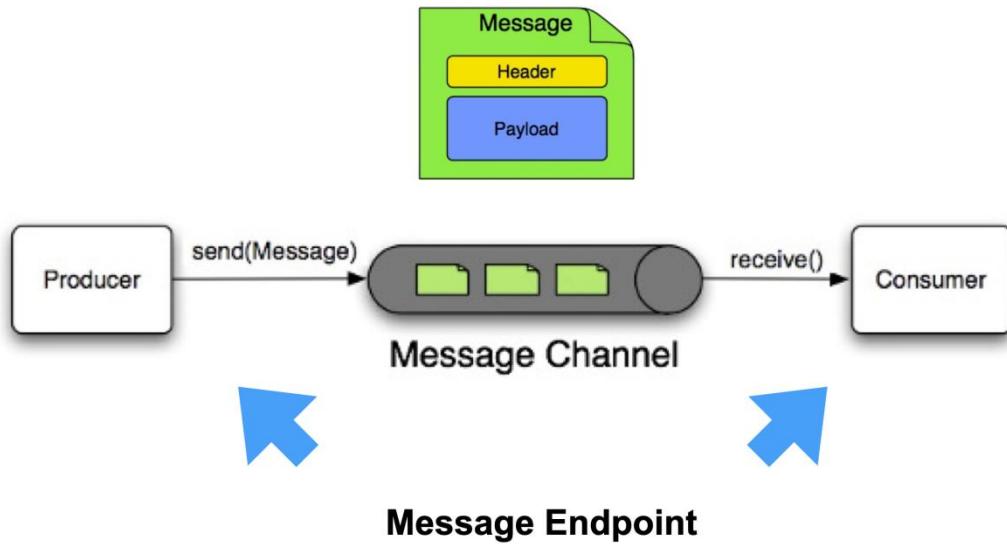
11.例子



12. Spring 集成

- Spring Integration 在基于 Spring 的应用程序中实现了轻量级的消息传递，并支持通过声明式适配器与外部系统集成。这些适配器在 Spring 对远程、消息传递和调度的支持上提供了更高层次的抽象性。
- Spring 对远程、消息传递和调度的支持。Spring Integration 的主要目标是为构建企业集成解决方案提供一个简单的模型，同时保持关注点的分离，这对产生可维护、可测试的代码至关重要。

13. 主要组件

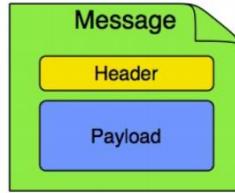


14. 功能介绍

- 实现大部分的企业集成模式 (EIP) :
 - 端点
 - 通道 (点对点和发布/订阅)
 - 过滤器
 - 转化器
 - 聚合器
 - ...
- 与外部系统集成:
 - ReST/HTTP
 - FTP/SFTP
 - Twitter
 - WebServices (SOAP and ReST)
 - TCP/UDP
 - JMS
 - RabbitMQ
 - Email ...

15. Spring Integration 消息系统的设计

(1) 消息类型 (Message Type)



```

public interface Message<T> {
    T getPayload();
    MessageHeaders getHeaders();
}

Message<String> message1 = MessageBuilder.withPayload("test").setHeader("foo", "bar").build();
Message<String> message2 = MessageBuilder.fromMessage(message1).build();

```

(2) 信息通道类型 (Message Channel Type)

```

public interface MessageChannel {
    boolean send(Message message);
    boolean send(Message message, long timeout);
}

public interface PollableChannel extends MessageChannel {
    Message<?> receive();
    Message<?> receive(long timeout);
}

public interface SubscribableChannel extends MessageChannel {
    boolean subscribe(MessageHandler handler);
    boolean unsubscribe(MessageHandler handler);
}

```

(3) 消息通道的实现 (Message Channel Implementations)

发布订阅信道 / 队列信道 / 优先级信道 / 会合信道 / 直接信道 / 执行者信道 / 通量信息信道

(4) 消息端点/过滤器 (Message Endpoints /Filters)

消息转换器 / 消息过滤器 / 消息路由器 / 分离器 / 聚合器 / 服务激活器 (EIP 中的端点) / 通道适配器 (EIP 中的端点)

(5) 消息转换器 (Message Transformer)

消息转换器负责转换消息的内容或结构并返回修改后的消息。最常见的转化器类型可能是将消息的有效载荷从一种格式转换成另一种格式（例如从 XML 转换成 java.lang.String）。同样地，转化器可以添加、删除或修改消息的标题值。

(6) 消息转化器的实现 (Message Transformer Implementations)

对象到字符串的转化器 / 对象到地图和地图到对象的转化器 / 流转化器 / JSON 转化器

(7) 消息过滤器 (Message Filter)

消息过滤器是用来决定一个消息是否应该根据某些标准，如消息头值或消息内容本身被传递或放弃。因此，消息过滤器类似于路由器，只是对于从过滤器的输入通道收到的每个消息，同一消息可能会或不会被发送到过滤器的输出通道。

(8) 消息路由器 (Message Router)

消息路由器负责决定下一个接收消息的通道或通道（如果有的话）。消息路由器经常被用作服务激活器或其他能够发送回复消息的端点上静态配置的输出通道的动态替代品。

有效载荷类型路由器 / 头值路由器 / 收件人列表路由器

(9) 分割器 (Splitter)

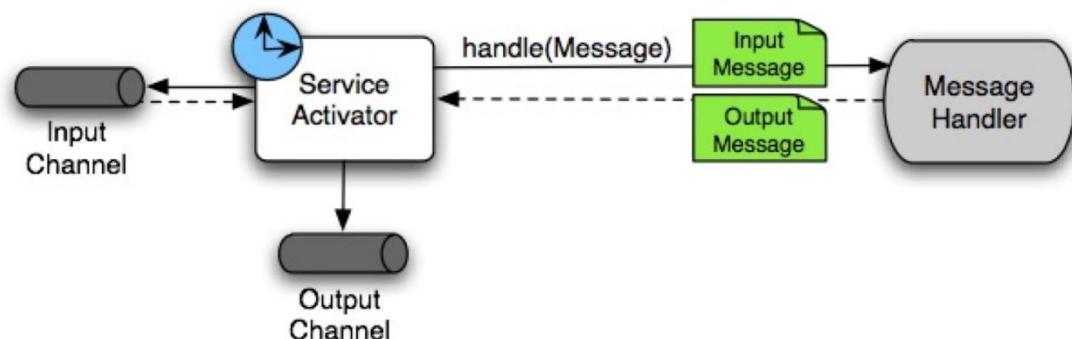
分割器是另一种类型的消息端点，其职责是接受来自其输入通道的消息，将该消息分割成多个消息，并将其中每个消息发送到其输出通道。这通常用于将一个“复合”有效载荷对象分割成一组包含细分有效载荷的消息。

(10) 聚合器 (Aggregator)

聚合器是一种消息端点，它接收多个消息并将其合并为一个消息。从技术上讲，聚合器比分割器更复杂，因为它需要维护状态（要聚合的消息），决定完整的消息组何时可用，并在必要时进行超时处理。此外，在超时的情况下，聚合器需要知道是否要发送部分结果，丢弃它们，或者将它们发送到一个单独的通道。Spring Integration 提供了一个 CorrelationStrategy、一个 ReleaseStrategy，以及关于超时、超时后是否发送部分结果和丢弃通道的可配置设置。

(11) 服务激活器 (Service Activator)

服务激活器是一个通用端点，用于将服务实例连接到消息传递系统。必须配置输入消息通道，如果要调用的服务方法能够返回一个值，也可以提供一个输出消息通道。

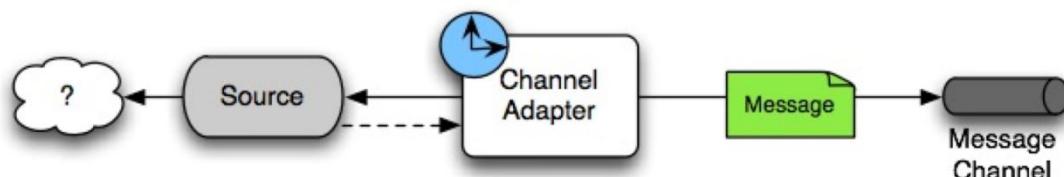


(12) 通道适配器 (Channel Adapter)

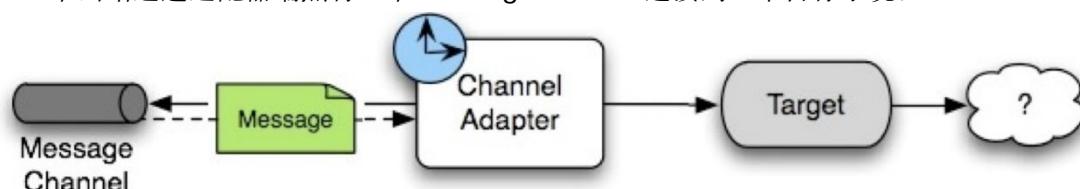
通道适配器是一个端点，它将一个消息通道连接到其他系统或传输。通道适配器可以是入站的，也可以是出站的。通常，通道适配器在消息和从其他系统接收或发送到其他系统的任何对象或资源（文件、HTTP 请求、JMS 消息和其他）之间做一些映射。根据传输方式的不同，通道适配器也可以填充或提取消息头的值。Spring Integration 提供了许多通道适配器，这些适配器将在接下来的章节中描述。

(13) 入站和出站 (Inbound & Outbound)

- 一个人站通道适配器端点将一个源系统连接到一个 MessageChannel 上。



- 一个出站通道适配器端点将一个 MessageChannel 连接到一个目标系统。



16.例子:

(1) Hello World

- inputChannel: MessageChannel

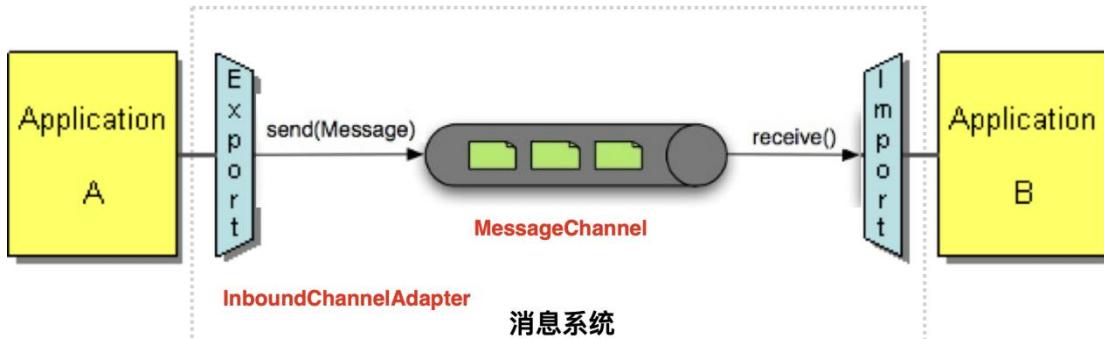
- outputChannel: MessageChannel
- service-activator: ServiceActivator
- **Hello World 组件:**

```
MessageChannel inputChannel = context.getBean("inputChannel", MessageChannel.class);
PollableChannel outputChannel = context.getBean("outputChannel", PollableChannel.class);
inputChannel.send(new GenericMessage<String>("World"));
logger.info("==> HelloWorldDemo: " + outputChannel.receive(0).getPayload());
```

```
<channel id="inputChannel" />
<channel id="outputChannel">
    <queue capacity="10" />
</channel>
<service-activator input-channel="inputChannel" output-channel="outputChannel" ref="helloService" method="sayHello" />
<beans:bean id="helloService" class="com.example.spring.integration.helloworld.HelloService" />
```

(2) File-Copying Integration

- fileReadingMessageSource: InboundChannelAdapter
- fileWritingMessageHandler: ServiceActivator
- fileChannel: MessageChannel



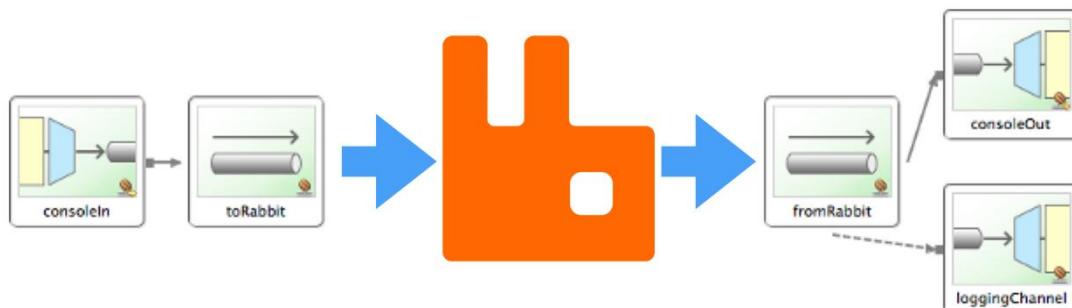
- **File-Copying Integration 组件:**

```
@Bean
@InboundChannelAdapter(value = "fileChannel", poller = @Poller(fixedDelay = "5000"))
public MessageSource<File> fileReadingMessageSource() {
    FileReadingMessageSource sourceReader = new FileReadingMessageSource();
    sourceReader.setDirectory(new File(INPUT_DIR));
    sourceReader.setFilter(new SimplePatternFileListFilter(FILE_PATTERN));
    return sourceReader;
}
@Bean
public MessageChannel fileChannel() {
    return new DirectChannel();
}
@Bean
@ServiceActivator(inputChannel = "fileChannel")
public MessageHandler fileWritingMessageHandler() {
    FileWritingMessageHandler handler = new FileWritingMessageHandler(new File(OUTPUT_DIR));
    handler.setFileExistsMode(FileExistsMode.REPLACE);
    handler.setExpectReply(false);
    return handler;
}
```

17. 集成端点 (Integration Endpoints)

Module	Inbound Adapter	Outbound Adapter	Inbound Gateway	Outbound Gateway
AMQP	Inbound Channel Adapter	Outbound Channel Adapter	Inbound Gateway	Outbound Gateway
Events	Receiving Spring Application Events	Sending Spring Application Events	N	N
Feed	Feed Inbound Channel Adapter	N	N	N
File	Reading Files and Tailing Files	Writing files	N	Writing files
FTP(S)	FTP Inbound Channel Adapter	FTP Outbound Channel Adapter	N	FTP Outbound Gateway
Gemfire	Inbound Channel Adapter and Continuous Query	Outbound Channel Adapter	N	N
HTTP	HTTP Namespace Support	HTTP Namespace Support	Http Inbound Components	HTTP Outbound Components
JDBC	Inbound Channel Adapter and Stored Procedure	Outbound Channel Adapter and Stored Procedure	N	Outbound Gateway and Stored Procedure Outbound
JMS	Inbound Channel Adapter and Message-driven Channel	Outbound Channel Adapter	Inbound Gateway	Outbound Gateway
JMX	Notification-listening Channel Adapter and Attribute-based	Notification-publishing Channel Adapter and Operation-invoking	N	Operation-invoking Outbound Gateway
JPA	Inbound Channel Adapter	Outbound Channel Adapter	N	Updating Outbound Gateway and Retrieving Outbound
Mail	Mail-receiving Channel Adapter	Mail-sending Channel Adapter	N	N
MongoDB	MongoDB Inbound Channel Adapter	MongoDB Outbound Channel Adapter	N	N
MQTT	Inbound (Message-driven) Channel Adapter	Outbound Channel Adapter	N	N
Redis	Redis Inbound Channel Adapter and Redis Queue	Redis Outbound Channel Adapter and Redis Queue	Redis Queue Inbound Gateway	Redis Outbound Command Gateway and Redis Queue
Resource	Resource Inbound Channel Adapter	N	N	N
RMI	N	N	Inbound RMI	Outbound RMI
RSocket	N	N	RSocket Inbound Gateway	RSocket Outbound Gateway
SFTP	SFTP Inbound Channel Adapter	SFTP Outbound Channel Adapter	N	SFTP Outbound Gateway
STOMP	STOMP Inbound Channel Adapter	STOMP Outbound Channel Adapter	N	N
Stream	Reading from Streams	Writing to Streams	N	N
Syslog	Syslog Inbound Channel Adapter	N	N	N
TCP	TCP Adapters	TCP Adapters	TCP Gateways	TCP Gateways
UDP	UDP Adapters	UDP Adapters	N	N
WebFlux	WebFlux Inbound Channel Adapter	WebFlux Outbound Channel Adapter	Inbound WebFlux Gateway	Outbound WebFlux Gateway
Web Services	N	N	Inbound Web Service Gateways	Outbound Web Service Gateways
Web Sockets	WebSocket Inbound Channel Adapter	WebSocket Outbound Channel Adapter	N	N
XMPP	XMPP Messages and XMPP Presence	XMPP Messages and XMPP Presence	N	N

18.AMQP/RabbitMQ 集成



19.小结: 消息传递模式 (Messaging Pattern)

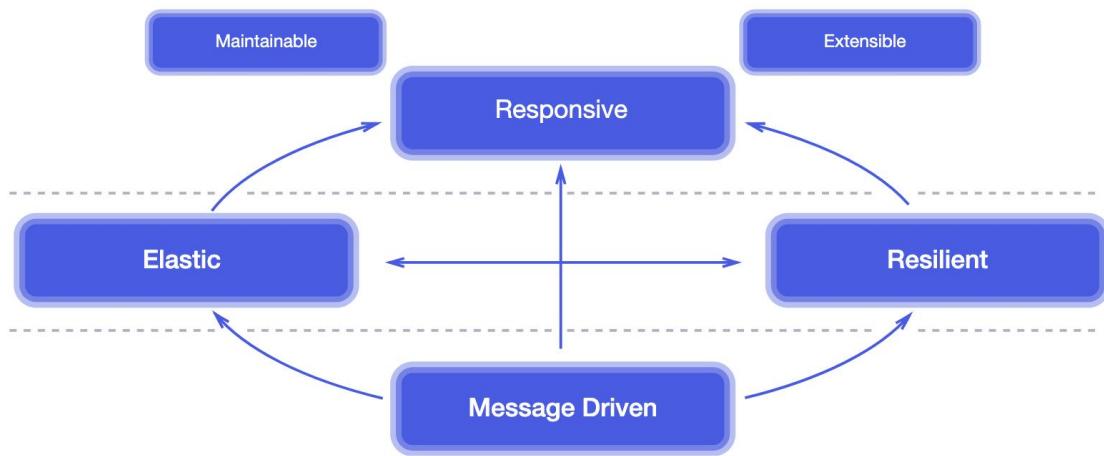
该模式是以一种最灵活的方式来整合多个不同的系统:

- 几乎完全解耦参与集成的系统
- 允许参与集成的系统对彼此的底层协议、格式化或其他实现细节完全不了解
- 鼓励参与集成的组件的开发和重复使用

十二、响应式架构

1.响应式系统

作为响应式系统构建的系统更加灵活，松散耦合和可扩展。这使得它们更容易开发，更容易改变。它们对失败的容忍度要高得多，当失败发生时，它们会以优雅的方式来应对，而不是灾难。反应式系统是高度响应的，给予用户有效的互动反馈。



2.弹性

- 系统在不同的工作量下保持响应。反应性系统可以通过增加或减少分配给这些输入的资源来对输入率的变化做出反应。这意味着设计中没有争论点或中心瓶颈，从而有能力分片或复制组件并在它们之间分配输入。反应式系统通过提供相关的实时性能测量，支持预测性以及反应式的扩展算法。他们在商品硬件和软件平台上以经济有效的方式实现弹性。
- 系统在面对故障时能保持响应。这不仅适用于高度可用的关键任务系统--任何没有弹性的系统在发生故障后都会没有反应。弹性是通过复制、遏制、隔离和授权实现的。故障被包含在每个组件中，使组件之间相互隔离，从而确保系统的一部分可以在不影响整个系统的情况下发生故障和恢复。每个组件的恢复被委托给另一个（外部）组件，必要时通过复制来确保高可用性。一个组件的客户端没有处理其故障的负担。

3.消息驱动

反应式系统依靠异步消息传递来建立组件之间的边界，确保松散耦合、隔离和位置透明。这个边界也提供了将故障委托给消息的方法。采用明确的消息传递，可以通过塑造和监测系统中的消息队列，并在必要时应用反向压力，实现负载管理、弹性和流量控制。将透明的消息传递作为一种通信手段，使得故障管理有可能在整个集群或单个主机内使用相同的结构和语义。非阻塞式通信允许接收者只在活动时消耗资源，从而减少系统开销。

4.响应性

如果可能的话，系统会及时作出反应。响应性是可用性和实用性的基石，但不仅如此，响应性意味着问题可能被迅速发现并得到有效处理。响应性系统专注于提供快速和一致的响应时间，建立可靠的上限，因此它们提供一致的服务质量。这种一致的行为反过来又简化了错误处理，建立了终端用户的信心，并鼓励进一步的互动。

十三、云计算架构

1.计算机应用：分 → 合

分



早期计算技术以合为特征
曲高和寡



个人电脑的发展使分成为了主流
计算机飞入寻常百姓家

合



网络技术的发展使云计算成为了合的模式，计算和存储通过网络隐形于云端

Cloud computing was popularized with Amazon.com releasing its Elastic Compute Cloud product in 2006.

2.互联网的发展，使得我们可以很便捷得获得远程的计算/存储资源

云计算架构



云计算的基本原理是通过使计算分布在大量的分布式计算机上，而非本地计算机或远程服务器中。这使得用户可以根据需求访问计算机和存储系统。

所以，云计算架构指的是软件与硬件资源间的设计决策（但也影响软件本身的设计）

3.云计算

- 云计算是指按需提供计算机系统资源，特别是数据存储（云存储）和计算能力，而无需用

户直接主动管理。 — 维基百科

- 云计算是一个通用术语，指任何涉及通过互联网提供托管服务的事物。
- 云计算是一种模式，用于实现方便、按需的网络访问可配置的计算资源共享池。

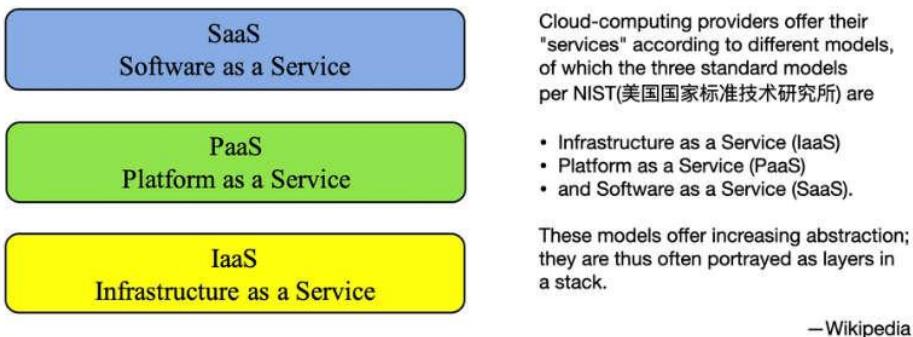
4. 服务模式

云计算供应商根据不同的模式提供他们的"服务"，其中 NIST (美国国家标准技术研究所) 规定的三种标准模式是：

- SaaS, 软件即服务
- PaaS, 平台即服务
- IaaS, 基础设施即服务

这些模式提供了越来越多的抽象性，因此它们经常被描绘成一个堆栈中的层。

Service Models



(1) 基础设施即服务 Infrastructure as a Service (IaaS)

- 计算机基础设施交付模式
 - 提供给消费者部署和运行任意软件的能力，其中可以包括操作系统和应用程序。
 - 消费者不管理或控制底层云基础设施，但可以控制操作系统、存储和部署的应用程序；可能还可以有限地控制选定的网络组件（如主机防火墙）。
- ① 管理程序作为客人运行虚拟机。云操作系统中的管理程序池可以支持大量的虚拟机，并能够根据客户的不同要求扩大和减少服务。

典型IaaS

学生特惠

The diagram illustrates the transition from traditional deployment to virtualized deployment. On the left, 'Traditional Deployment' shows three separate application stacks ('App', 'Bin/ Library', 'Operating System') running directly on a single 'Hardware' layer. An arrow points to the right, labeled 'Virtualized Deployment', which shows four application stacks ('App', 'Bin/ Library', 'Operating System', 'Virtual Machine') running on top of a 'Hypervisor', which in turn runs on a single 'Hardware' layer. This visualizes how virtualization abstracts the hardware from the application layers.

A hypervisor runs the virtual machines as guests. Pools of hypervisors within the cloud operational system can support large numbers of virtual machines and the ability to scale services up and down according to customers' varying requirements.

云服务器 ECS

CPU 1核 内存 1G 带宽 15G 硬盘 40G / 20G (Windows / Linux)

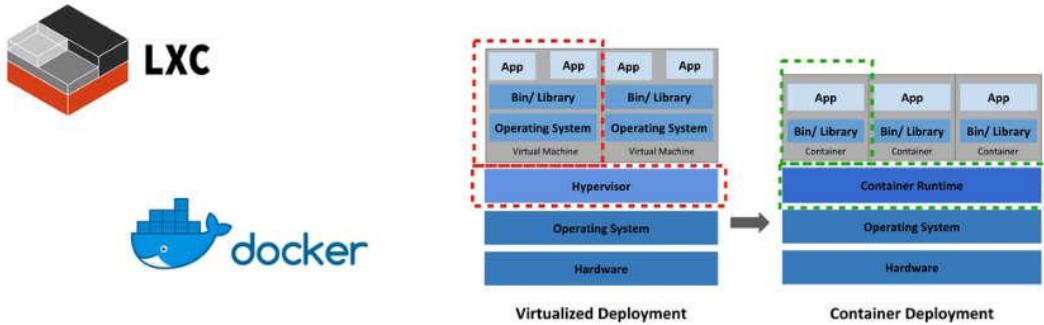
大学生专享价：9.92元/月 (原价：14.92元/月) (41元实例费+10.8元流量代扣费)

立即购买 详细检测

购买流程：① 注册 → ② 实名认证 → ③ 学生认证 → ④ 9.92元扣机。

②Linux 容器运行在直接运行在物理硬件上的单一 Linux 内核的隔离分区中。Linux 和命名空间是用于隔离、保护和管理容器的基础 Linux 内核技术。容器化提供了比虚拟化更高的性能，因为没有管理程序的开销。此外，容器容量随着计算负载的变化而自动扩展，这消除了过度供应的问题，并实现了基于使用的计费。

容器化技术



Linux containers run in isolated partitions of a single Linux kernel running directly on the physical hardware. Linux cgroups and namespaces are the underlying Linux kernel technologies used to isolate, secure and manage the containers. Containerisation offers higher performance than virtualization because there is no hypervisor overhead. Also, container capacity auto-scales dynamically with computing load, which eliminates the problem of over-provisioning and enables usage-based billing.

③Kubernetes (常简称为 K8s) 是用于自动部署、扩展和管理“容器化 (containerized) 应用程序”的开源系统。该系统由 Google 设计并捐赠给 Cloud Native Computing Foundation (今属 Linux 基金会) 来使用。Kubernetes (在希腊语意为“舵手”或“驾驶员”) 由 Joe Beda、Brendan Burns 和 Craig McLuckie 创立，并由其他谷歌工程师，包括 Brian Grant 和 Tim Hockin 等进行加盟创作，并由谷歌在 2014 年首次对外宣布。

(2) 平台即服务 Platform as-a-Service (PaaS)

· 平台交付模式



kubernetes

- 提供给消费者的能力是在云上部署基础设施上部署由消费者创建或获得的应用程序，这些应用程序使用供应商支持的编程语言、库、服务和工具。
- 消费者不管理或控制底层云基础设施，包括网络、服务器、操作系统或存储，但可以控制已部署的应用程序和可能的应用程序托管环境的配置设置。

PaaS Examples



Heroku是一个支持多种编程语言的云平台即服务。在2010年被Salesforce.com收购。
Heroku作为最元祖的云平台之一，从2007年6月起开发，当时它仅支持Ruby，但后来增加了对Java、Node.js、Scala、Clojure、Python以及（未记录在正式文件上）PHP和Perl的支持。
基础操作系统是Debian，在最新的技术堆栈则是基于Debian的Ubuntu。

PaaS Examples



(3) 软件即服务 Software as a Service (SaaS)

- 软件交付模式
 - 提供给消费者的能力是使用供应商的
 - 提供给消费者的能力是使用供应商在云基础设施上运行的应用程序。应用程序可通过瘦客户机界面，如网络浏览器（如基于网络的电子邮件），或程序界面，从各种客户设备上访问。
 - 消费者不管理或控制底层的云基础设施，包括网络、服务器、操作系统、存储，甚至个别应用程序的能力，可能的例外是有限的用户特定的应用程序配置设置。

SaaS Example



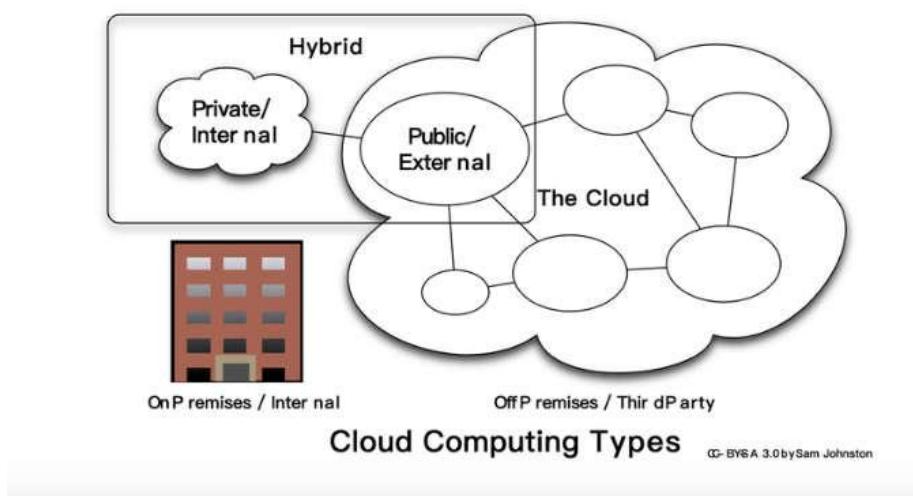
Salesforce公司于1999年由当时37岁的甲骨文(Oracle)高级副总裁，俄罗斯裔美国人马克·贝尼奥夫(Marc Benioff, 1964/09/25-)创办。

贝尼奥夫长期推广 SaaS (Software As A Service, 软件即服务) 的观念。Salesforce 公司提供按需定制的软件服务，用户每个月需要支付类似租金的费用来使用网站上的各种服务，这些服务涉及客户关系管理的各个方面，从普通的联系人管理、产品目录到订单管理、机会管理、销售管理等。他提供一个平台，使得客户无需拥有自己的软件，也无需花费大量资金和人力用于记录的维护、储存和管理，所有的记录和数据都储存在 salesforce.com 上面。同时和普通的自己购买的软件不一样，用户随时可以根据需要去增加新的功能或者去除一些不必要的功能，真正地实现了按需使用。

—Wikipedia

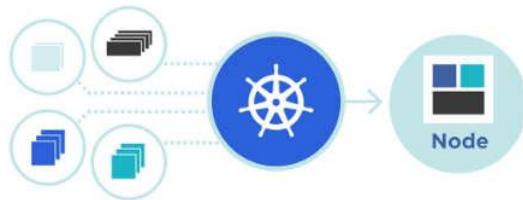


Deployment models



Kubernetes

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.



<https://kubernetes.io/>

Minikube

1 Installation

The image shows the Minikube installation guide. It features the Kubernetes logo at the top left, followed by the "minikube" logo. To the right is a screenshot of a Windows desktop with a "Windows" tab selected. The window contains instructions for installing minikube via Chocolatey Package Manager or downloading the Windows installer. Below the guide is a URL: <https://kubernetes.io/zh/docs/tasks/tools/install-minikube/>. To the right of the URL is the command `minikube start --image-mirror-country=cn`, which is highlighted with a red dashed box.

Demo

`sa-spring/spring-boot-echo`

```
# 启动minikube
minikube start

# 将当前shell的docker环境指向minikube内
eval $(minikube docker-env)

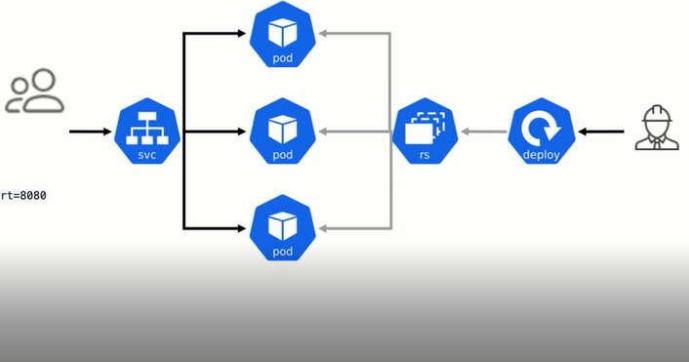
# 编译打包 (docker镜像保存在minikube内)
mvn compile jib:dockerBuild

# 创建一个deployment
kubectl apply -f echo-deployment.yaml

# 创建一个service (暴露出deployment的端口)
kubectl expose deployment echo-boot --type=NodePort --port=8080

# 获得service的url
minikube service echo-boot --url
curl http://.....

# 横向扩展
kubectl scale deployment echo-boot --replicas=2
curl http://.....
```



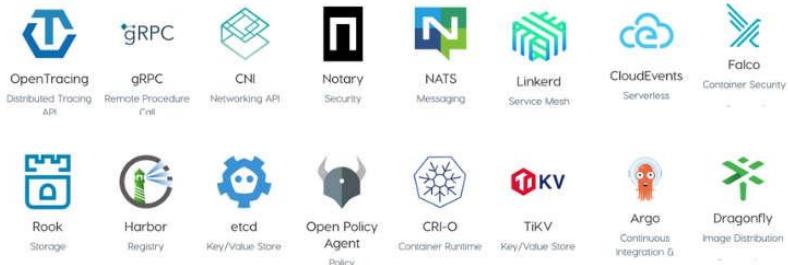
5. 云原生计算 (Cloud Native Computing)

云原生计算是软件开发中的一种方法，它利用云计算 "在现代动态环境中构建和运行可扩展的应用程序，如公共云、私有云和混合云"。 —维基百科

6. 云原生 (Cloud Native)

- 云原生技术使企业能够在现代动态环境中构建和运行可扩展的应用程序，如公共云、私有云和混合云。容器、服务网、微服务、不可变的基础设施和声明式接口都是这种方法的典范。
- 这些技术使松散耦合的系统具有弹性、可管理和可观测性。与强大的自动化相结合，它们使工程师能够以最小的工作量频繁地、可预测地进行高影响的改变。我们希望系统是有反应的、有弹性的、有弹性的和消息驱动的。我们称这些为反应型系统。

7. 云原生计算基金会 CNCF



The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.

8. Spring Cloud Kubernetes

- Spring Cloud Kubernetes 提供了 Spring Cloud 的通用接口实现，可以消费 Kubernetes 本地服务。这些项目的主要目标是促进在 Kubernetes 内运行的 Spring Cloud 和 Spring Boot 应用程序的整合。
- Kubernetes 意识
- DiscoveryClient 实现
- 通过 ConfigMaps 配置的 PropertySource 对象
- 通过 Netflix Ribbon 实现客户端负载均衡

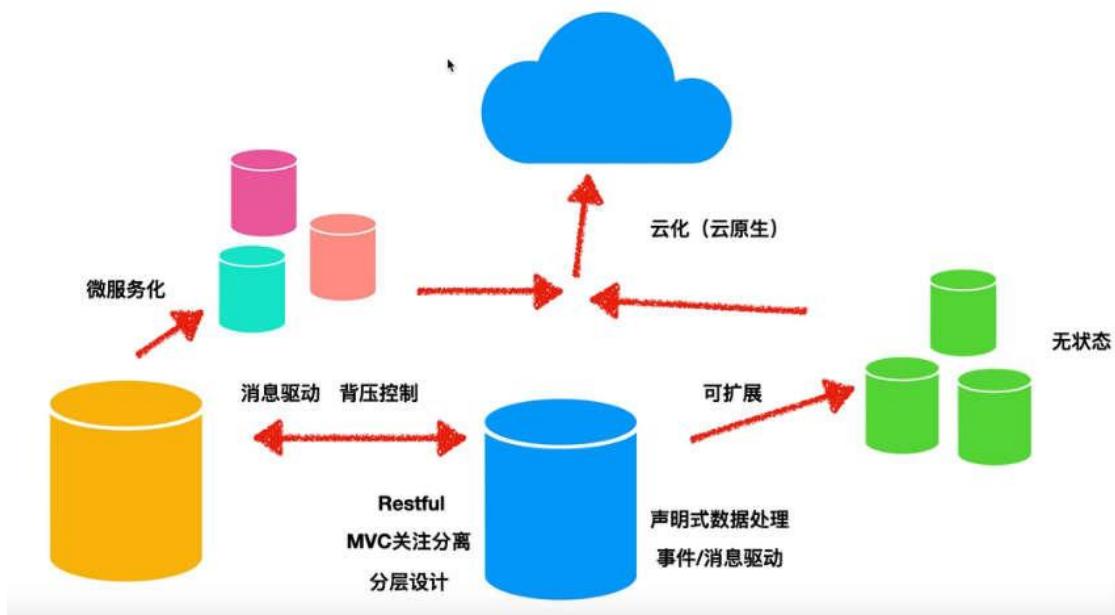
Spring Cloud

Spring Cloud

Spring Cloud Azure	Spring Cloud Schema Registry
Spring Cloud Alibaba	Spring Cloud Security
Spring Cloud for Amazon Web Services	Spring Cloud Skipper
Spring Cloud Bus	Spring Cloud Sleuth
Spring Cloud CLI	Spring Cloud Stream
Spring Cloud for Cloud Foundry	Spring Cloud Stream Applications
Spring Cloud - Cloud Foundry Service Broker	Spring Cloud Stream App Starters
Spring Cloud Cluster	Spring Cloud Task
Spring Cloud Commons	Spring Cloud Task App Starters
Spring Cloud Config	Spring Cloud Vault
Spring Cloud Connectors	Spring Cloud Zookeeper
Spring Cloud Consul	Spring Cloud App Broker
Spring Cloud Contract	Spring Cloud Circuit Breaker
Spring Cloud Function	Spring Cloud Kubernetes
Spring Cloud Gateway	Spring Cloud OpenFeign
Spring Cloud GCP	
Spring Cloud Netflix	
Spring Cloud Open Service Broker	
Spring Cloud Pipelines	

**主要是微服务架构应用相关技术与机制
&
与各个云计算平台的适配机制**

<https://spring.io/projects/spring-cloud>



十四、性能与监控

1.

Petclinic



[spring-petclinic/spring-petclinic-microservices](https://github.com/spring-petclinic/spring-petclinic-microservices)

2.

Behind the Scenes

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

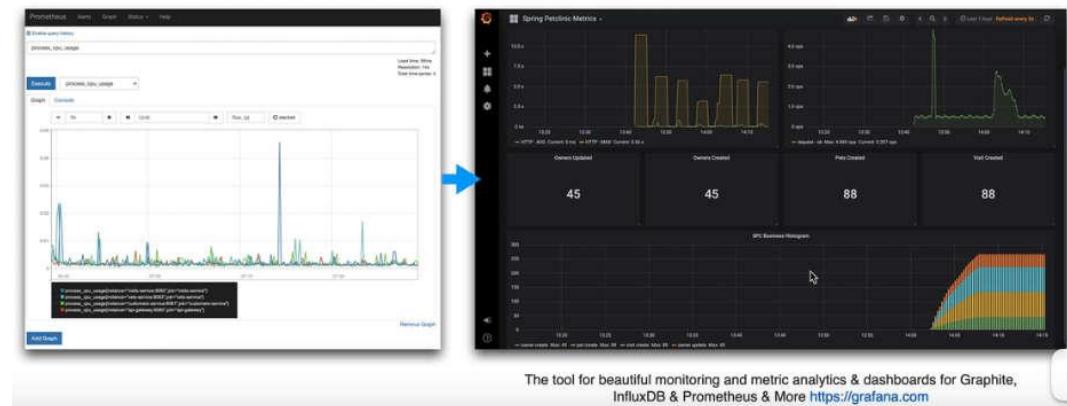


```
grafana-server:
  build: ./docker/grafana
  container_name: grafana-server
  mem_limit: 256M
  ports:
    - 3000:3000

prometheus-server:
  build: ./docker/prometheus
  container_name: prometheus-server
  mem_limit: 256M
  ports:
    - 9091:9090
```

3.

Dashboard



4.

How it works



5.

Why?

