架构设计与软件开发

主题

- 架构设计
  - 架构设计基础
  - 常用模式及中间件
  - 表示层设计方法
  - 业务层设计方法
  - 数据访问层设计方法
  - 面向方面编程（AOP）
  - 通用架构服务设计
  - 创建软件架构
- 软件开发
  - JavaEE5新特性
  - JavaEE主流应用服务器
  - JavaEE应用调优

第一部分：架构设计

软件架构基础

软件架构设计

- 处于软件系统建设的上游

- 需要全面考虑多方面的因素
- 对于同一个问题，可以有多种设计结果
- 是在各种制约条件下取得的较好折衷方案
- 科学 + 经验 + 艺术
- “系统架构”往往被滥用

架构的概念

- 架构师的角色：
  - 系统的规模
  - 系统的分布
- 架构满足风险管理的需要
  - 高层规划的目标：
    - 部分失效时系统的强健性
    - 处理请求负载
    - 并发使用的扩展能力

- 架构的功能
  - 技术职责
    - 标识对架构重要的用例
    - 指导架构原型的开发
  - 管理职责
    - 成本管理
      - 技术和风险转移的方法
    - 沟通管理
      - 与项目干系人和团队成员的有效合作的沟通技巧

- 架构功能和设计功能

	架构	**设计**
抽象级别	高层的、广泛的，很少关注细节	底层的、特定的，关注更多的细节
提交物	系统和子系统规划，架构原型	组件设计，代码规范
关注点	非功能性需求，风险管理	功能性需求

- 面向对象的分析和设计职责
  - 基于组件设计的关键：
    - 抽象
    - 封装
    - 内聚
    - 耦合

- 系统架构
  - 可视化硬件和服务器软件的设计与实现
  - 有数据库设计、容量规划、服务器集群、负载平衡及容错策略等方面的经验
  - 提供支持RAS的部署环境
  - 通常称为系统架构师或基础平台架构师

- 应用架构师
  - 可视化应用软件和组件集成的设计和实现
  - 有典型的业务应用、集成应用和OO方法方面的经验
  - 提供实现端到端功能并支持非功能性需求的应用结构

架构的关键点

- 架构的关键点
  - 架构过程
  - J2EE技术
  - 风险管理
  - 模式使用
  - 原型开发

- 创建满足QoS需求的蓝图
  - 典型的架构文档
    - 愿景文档
    - 需求规范
    - 风险识别和转移计划
    - 应用的域模型
    - 上下文环境描述
    - 项目计划
    - 假设列表

- 评估J2EE技术
  - 考虑技术决策点
  - 确保团队正确地使用了所选技术

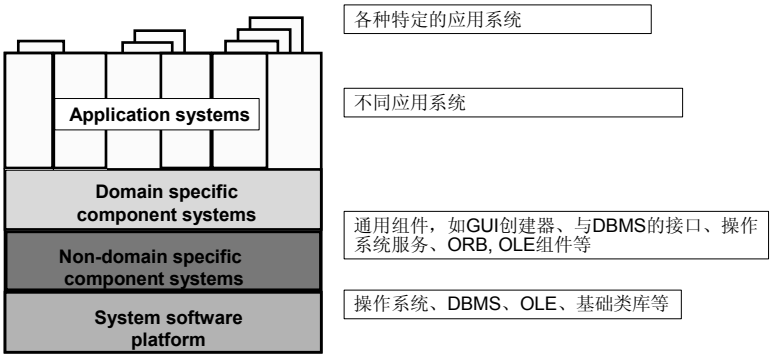
- 识别及控制风险
  - 非功能性需求
    - 业务规则
    - 约束
    - 系统质量
  - 风险评估
  - 成本分析

- 使用适当的模式
  - 设计模式
    - 支持功能性需求
  - 架构模式
    - 支持非功能性需求

- 开发原型
  - 架构原型描述系统并按照经验确定计划是否得到满足
  - 包括：
    - 域模型
    - 交互图

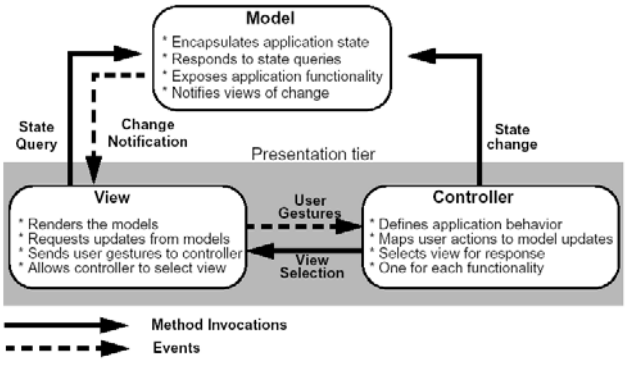
架构模式

- Layer模式
  - 确保抽象边界的定义和使用



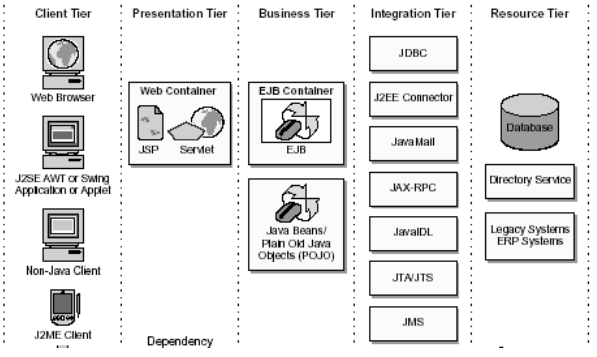
- 使用 Layer模式
  - 将大型任务划分为若干子任务组件，每个子任务是一个特定的抽象层
    - 系统功能的变化不会引起整个系统的波动
    - 组件之间的接口稳定
    - 调换系统的各部分不会影响其它系统组件
    - 系统的各部分可以复用
    - 每一层都只与其下层进行通讯

- MVC 模式
  - 将视图与控制器从模型中分离出来

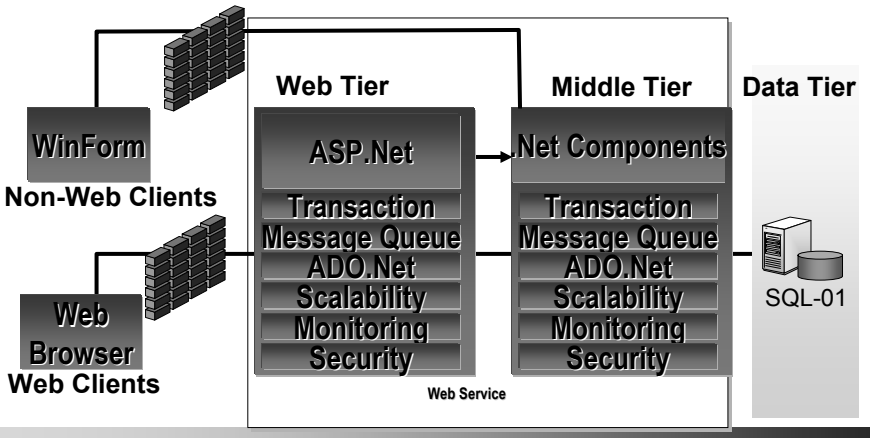


- Tier模式
  - 解决方案
    - 客户/服务器
    - 表现层/业务层/数据层
    - ……

- J2EE中各Tier技术：



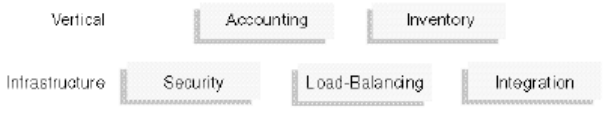
– .Net架构



- 使用可靠的框架
  - 框架“是将要实现的整体软件系统的一部分，定义了同类系统内的架构并提供基本的积木组件块创建系统。”
    - 通过装配适当的模式构建框架
    - 框架成为装配系统的模式
    - 框架适用于特定的问题域
  - 典型框架如：Struts、EJB等

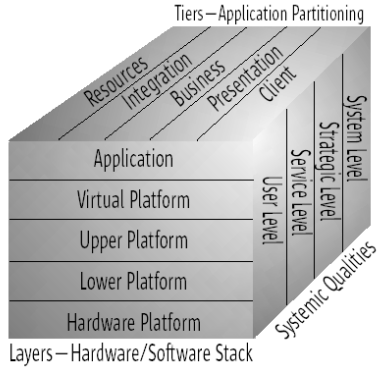
- 使用基于服务（Service-Based）的架构
  - 特点
    - 客户与实现松散耦合
    - 易于复用，提供扩展能力和可管理性

  - 服务类型
    - 垂直服务
      - 基于系统的内容
    - 水平服务
      - 基于系统基础设施平台



– 开发基于服务架构指南
  - 服务是类似于C/S架构中服务器的软件
  - 服务用来管理并发用户访问资源
    – 用户=》服务=》资源

- SunTone 架构框架



中间件

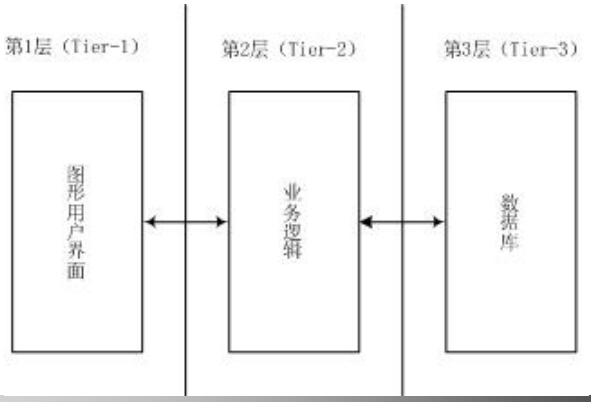
- 中间件的特点
  - 满足大量应用的需要
  - 运行于多种硬件和OS平台
  - 支持分布计算，提供跨网络、硬件和OS平台的透明性的应用或服务的交互
  - 支持标准的协议－保证互操作性
  - 支持标准的接口－保证可移植性

- 典型中间件
  - 消息中间件（MOM）
  - 对象请求代理（ORB）
  - 事务处理监控器（TPM）
  - 其它中间件
    - 安全中间件
    - 数据库中间件
    - 远程方法调用（RPC）

应用服务器

- 应用服务器（Application Server）
  - 企业级应用在Internet上迅速发展的条件下，出现的一种中间件技术
  - 可以处理客户和数据层之间的交互操作，并提供一组前面提到的中间件服务，包括事务管理、ORB、MOM、系统安全、负载均衡及资源管理等
  - 应用服务器还提供了一个称为容器的管理环境，可以对应用中的组件进行配置和管理
  - 通过应用服务器的采用可以将一个企业级应用安全、有效地部署到Internet上，实现电子商务
  - 采用应用服务器技术可以大大缩短开发周期、减小风险、降低成本

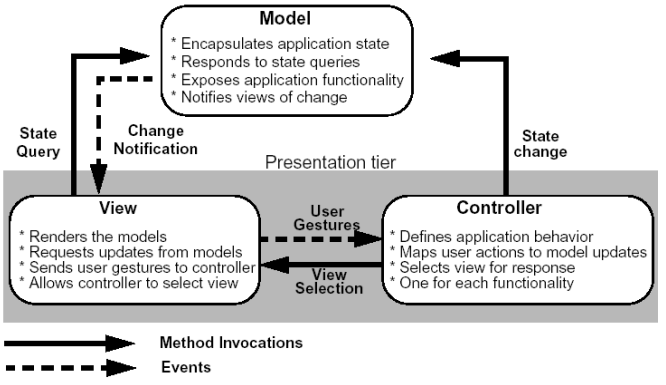
- 应用服务器三层架构



表现层设计

MVC 模式

- MVC模式



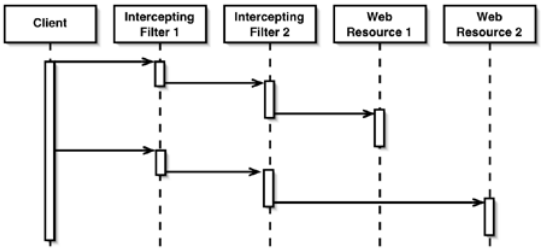
- Web控制器职责
  - 接收用户请求
  - 获取请求参数
  - 验证
  - 根据用户的不同请求，调用对应的模型组件来执行相应的业务逻辑
  - 获取业务逻辑执行结果
  - 根据当前的状态数据及业务逻辑的处理结果，选择适合的视图组件返回给客户

Web层设计

- 拦截过滤器
  - 问题：
    - 表现层的请求可能需要不同的处理
    - 某些请求可能需要预处理，而某些请求可能需要后续处理

– 解决方案
  - 创建可插入的过滤器以标准的方式处理通用服务，而不需要改变核心的请求处理代码
  - 过滤器拦截输入的请求和输出的响应，以进行预处理或后续处理

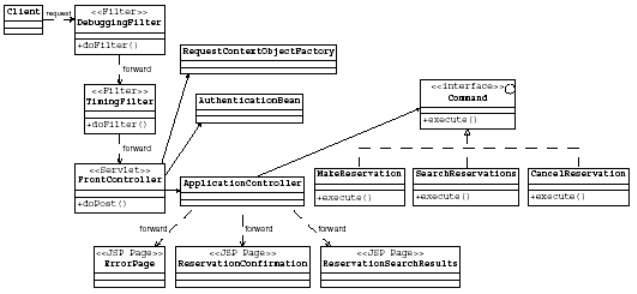
– 示例：



- 前端控制器
  - 问题：
    - 系统缺少一个集中处理请求的机制，会导致对每个请求都要完成的活动被随意地放在多个组件中
    - 通用的系统服务（如安全和审计）不应当在每个视图组件中都重复

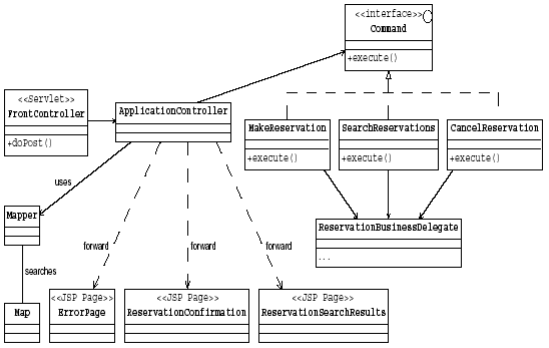
  - 解决方案
    - 提供一个集中处理请求的点
      - 调用安全服务，如认证和授权
      - 代理业务处理
      - 管理相应的视图选择
      - 处理错误
      - 管理内容的创建策略

– 示例：



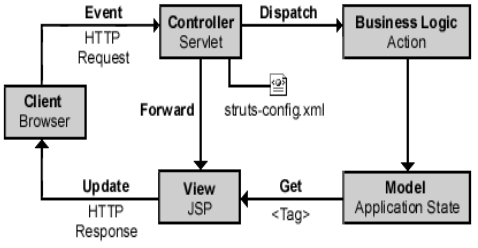
- 应用程序控制器：
  - 动作管理
    - 控制器决定要调用哪个动作
    - 该动作接下来会调用业务处理过程
  - 视图管理
    - 控制器决定将请求转发到哪个视图

– 示例一：

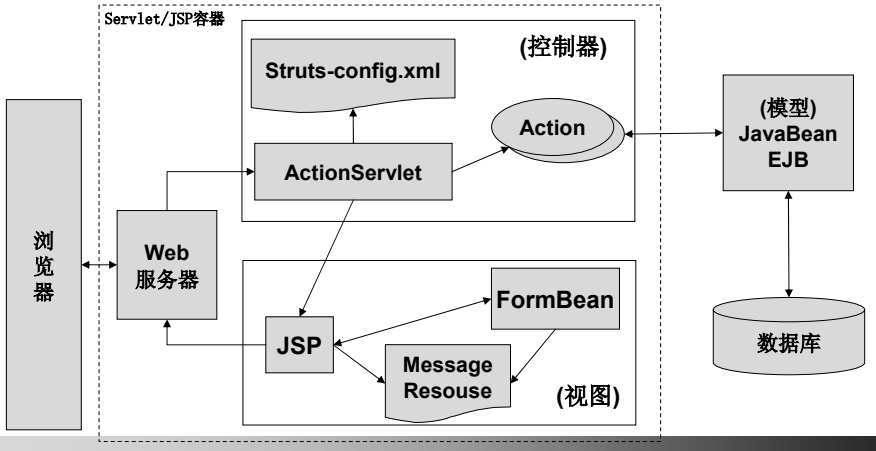


– 示例二：

- Struts



Struts框架



表现层的其它考虑

- 页面Cache
  - 页面的Cache依赖于Http1.1协议
    - 需要客户端支持
  - Cache保存的位置
    - 客户端、服务器、代理服务器
  - Cache的有效条件
    - 过期时间
  - 页面的Cache的版本
    - 可以根据Request中的请求参数或Http Header的不同保存不同版本的Cache
  - 页面Cache也可以通过程序实现

- 状态保存
  - 需要保存的状态是客户端、还是服务器端
  - 如果在服务器端，状态全局的，还是局部的
  - 状态是否有效期
  - 关闭浏览器以后，状态是否还需要保存
  - 状态数据有没有可能在网络上传输，传输过程中是否需要加密、是否防篡改

- 用户界面（UI）组件
  - 类型：
    - Console、Web、Plug-in
  - 搭建UI的框架
    - 自定义标签
    - AJAX
    - ……

- 动态页面生成：
  - XML
  - 模板技术

```
<screen name="main">
    <parameter key="banner" value="/banner.jsp" />
    <parameter key="sidebar" value="/sidebar.jsp" />
    <parameter key="body" value="/main.jsp" />
    <parameter key="footer" value="/footer.jsp" />
</screen>
```


- UI流程（UIP）组件：
  - 作用：
    - 隔离了UI与业务逻辑层
    - 对流程中的UI进行了管理
    - 提供了状态保存和传递的机制
  - 定制导航流程
    - 在配置文件中定制的流程

业务层设计

业务服务

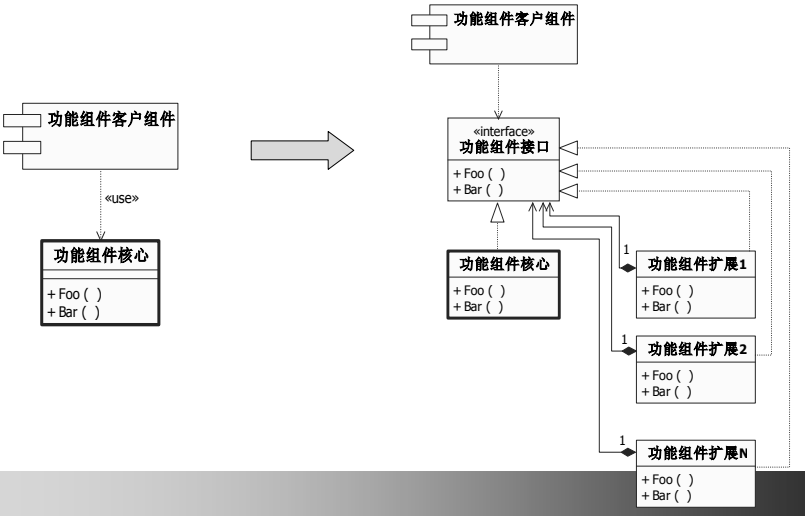
- 实现业务规则及执行业务工作的组件
  - 实现业务功能，是对特定业务逻辑和内部业务流程的封装
  - 负责发起事务，是根事务发起者，支持事务与补偿交易
  - 通过封装已存在的业务能够获得更高等级的操作和业务逻辑

- 业务服务与事务
  - 为了保证业务处理的完整性，业务服务必须提供事务的支持
    - 是事务的发起者，必须参与事务的投票
    - 能发起或参与异构系统的分布式事务，设置组件事务属性
    - 为业务处理提供补偿交易处理

  - 实现事务方式
    - 数据库事务与应用事务
    - 手工事务与自动事务

- 设计要点
  - 对于大型的系统，在保证性能的前提下，保证组件结构的可扩展性
  - 尽量保持组件之间的松耦合，允许并行、渐进及独立的开发与测试
  - 尽可能采用基于消息的通讯
  - 确定透过服务接口所暴露的处理流程是能处理多次重复信息的情况。
  - 选择事务边界要仔细，设置合适的事务隔离度
  - 选择和保持用一致的数据格式作为输入和返回参数

- 基于接口的可扩展性设计



业务流程

- 业务流程（ Business Workflow ）
  - 具有各种不同功能的活动相连的一组有相互关系的任务
  - 有起点和终点，而且都是可重复的，由多个业务过程（Business Process）组成
  - 业务过程包含多个业务步骤，且具有一定顺序
  - 定义及协调长期执行业务过程，支持长事务

- 业务流程的实现
  - 流程引擎
    - 实现业务流程同时管理活动的启用和终止或商业功能
  - 资源管理器
    - 资源管理器使实现商业功能或活动所必须的资源具有可用性
  - 调度程序
  - 审计管理器
  - 安全管理器

服务接口

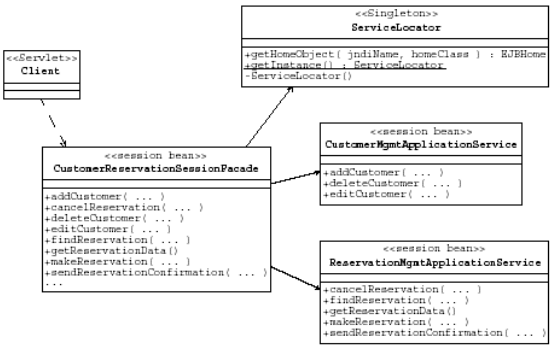
- 服务接口是为服务提供的进入点：
  - 是一个软件实体，将其实现为处理映射和转换服务的外观（ Façade ）组件
  - 与服务进行通讯，并强制执行通讯的处理流程及原则
  - 服务接口暴露方法，这些方法可被个别调用或以特定的顺序被调用

- 服务接口的设计：
  - 将服务接口视为应用程序的边界
  - 同一功能发布多种服务接口，不同的服务接口可以执行不同的SLA,以及不同事务处理能力
  - 实现外观组件，帮助隔离业务的变化
  - 事务性传输 或非事务性传输
  - 尽可能提高与其它平台和服务的交互操作性

- 服务接口的实现
  - 交换信息，负责实现通信时的所有细节：
    - 网络协议
    - 数据格式
    - 安全性
    - 服务级别协议

- 服务接口设计模式
  - 外观（Façade）模式
    - 目标：
      - 子系统提供单一接口给用户端
    - 问题：
      - 子系统内的类分别提供部分功能，用户端必须调用每个类，导致两层间联系复杂，不易维护，违反封装原则
    - 效果：
      - 简化设计，易于维护

– 示例：

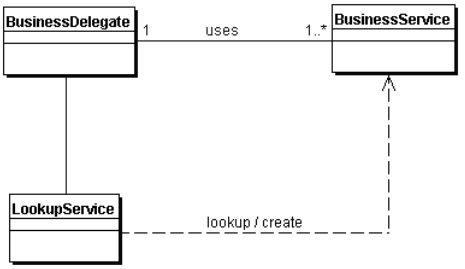


- 实现服务接口的方式
  - XML Web 服务
    - 使用 SOAP 和 HTTP
  - 消息队列方式
    - 一般采用系统服务中的队列组件

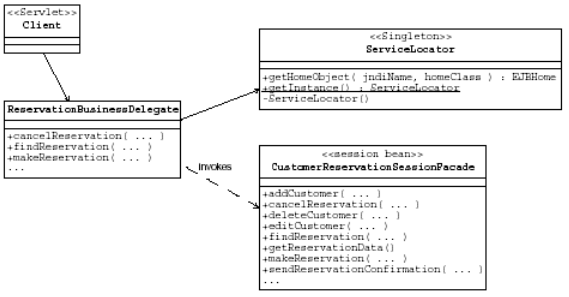
业务代理

- 问题
  - 将把服务 API 的底层实现细节暴露给客户
  - 业务服务的改变会对客户造成影响
  - 表现层组件可能会产生过多的远程调用
  - 表现层需要知道如何调用业务服务组件

- 解决方案
  - 使用业务代理减少表现层客户与业务服务之间的耦合
  - 业务代理隐藏底层业务服务的实现细节


BusinessDelegate
1
uses
1..*
BusinessService
LookupService
lookup / create

- 示例：

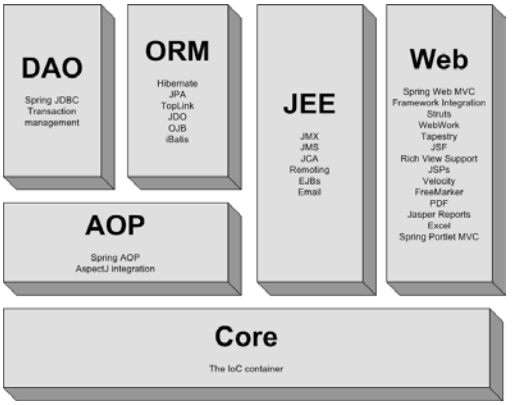

<<Servlet>>
Client
ReservationBusinessDelegate
+cancelReservation(...)
+findReservation(...)
+makeReservation(...)
...
Invokes
<<Singleton>>
ServiceLocator
+getHomeObject(jndiName, homeClass) : EJBHome
+getInstance() : ServiceLocator
-ServiceLocator()
<<session bean>>
CustomerReservationSessionFacade
+addCustomer(...)
+cancelReservation(...)
+deleteCustomer(...)
+editCustomer(...)
+findReservation(...)
+getReservationData()
+makeReservation(...)
+sendReservationConfirmation(...)
...

- 应用业务代理模式：
  - 优点：
    - 减少了表现层和业务层之间的耦合
    - 加入了简单的、统一的接口，以便客户访问业务服务
    - 缓存请求，可提高性能
  - 缺点：
    - 加入了额外的层次
    - 隐藏远程方法的调用，容易导致开发人员忽略方法调用的成本

Spring 框架

- 用于构造应用程序的轻量级框架
  - IoC（Inversion of Control）容器、非侵入性的框架
  - AOP（Aspect-oriented programming）
  - 提供对持久层、事务的支持
  - 提供MVC Web 框架的实现
  - 对常用的企业服务API提供一致的模型封装
  - 对于现存的各种框架（Struts、JSF、Hibernate 等）提供了整合方案

- Spring框架



- 依赖注入（ Dependency Injection，DI）的基本原则
  - 应用对象不应该负责查找资源或者其它依赖的协作对象
  - 配置对象的工作应该由IoC容器完成
  - “查找资源”的逻辑应该从应用代码中抽取出来，交给容器负责

– **Setter注入**

```
<bean id="exampleBean" class="examples.ExampleBean">

  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne">
          <ref bean="anotherExampleBean"/>
  </property>

  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```


业务实体

- 业务实体
  - 提供了业务数据的封装
  - 将显示数据与实际的存储隔离，保证数据的独立性，提高可重用性

- 业务服务与业务实体的关系
  - 管理业务实体的生命周期
  - 根据业务需求选取业务实体集合
  - 管理业务实体间的关系
  - 并非所有的情况都必须有业务实体，业务服务可以直接调用数据访问层

- 业务实体的形式
  - XML：
    - 可用于执行查询直接传回，或由DataSet/ResultSet做数据转换
  - DataSet/ResultSet ：
    - 数据访问结果
  - 自定义业务实体 ：
    - 符合面向对象概念，程序逻辑简单，数据转换的额外开销最多

XML业务实体

- 利用XML来表现业务实体，注意：
  - 确定XML文档包含一个业务实体还是多个业务实体的集合
  - 用命名空间唯一确认表现业务实体的XML文档
  - 为属性和元素选择合适的名称，表现业务实体数据

- DTD定义例

```
<? xml version="1.0" encoding="GB2312"? >

<!ELEMENT 联系人列表(联系人)*>
<!ELEMENT 联系人(姓名，ID，公司，EMAIL，电话，地址)>
<!ELEMENT 地址(街道，城市，省份)>
<!ELEMENT 姓名(#PCDATA)>
<!ELEMENT ID(#PCDATA)>
<!ELEMENT 公司(#PCDATA)>
<!ELEMENT EMAIL(#PCDATA)>
<!ELEMENT 电话(#PCDATA)>
<!ELEMENT 街道(#PCDATA)>
<!ELEMENT 城市(#PCDATA)>
<!ELEMENT 省份(#PCDATA)>
```

– XML Schema定义例：

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="loan">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="initialBalance" type="balance" />
        <xs:element name="currentBalance" type="balance" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="balance">
    <xs:restriction base="xs:decimal">
      <xs:totalDigits value="10" />
      <xs:fractionDigits value="2" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```


自定义业务实体

- 自定义业务实体
  - 实体组件包含数据的快照
    - 其数据只能保证与发起读取事务时上下文环境一致
  - 组件储存数据值，并透过其属性加以显露
  - 业务实体不直接存取数据库
  - 业务实体不初始化任何事务交易
  - 考虑是否将业务实体设计为序列化

- 设计建议
  - 自定义业务实体实现要简单
  - 从封装通用方法的基类继承
  - 业务实体实现一组公用的接口
  - 利用内部数据集或 XML文件保存复杂数据
  - 将验证规则分离为元数据
  - 设计保证实体之间的关系
  - 通过数据存取逻辑组件操作数据库数据

- 优点
  - 自行控制序列化实现，有利于性能调优
  - 隔离内部格式和应用程序所使用的数据结构描述
  - 代码可读性
  - 封装
    - 给数据增加自己的操作
  - 对复杂系统建模
    - 用自定义实体类定义一个良好的接口，将复杂的问题隐藏在类中
  - 本地验证
  - 私有成员
    - 隐藏不想暴露给调用者的信息

- 缺点
  - 业务实体集合
    - 一个业务实体对象表现单个的业务实体，而不是业务实体的集合
  - 序列化
    - 在业务实体中必须实现自己的序列化机制
  - 业务实体对象中表现复杂的数据关系和层次数据
  - 数据查询和排序
  - 部署
  - 对系统服务客户端的支持
  - 可扩展性的问题

数据访问层设计

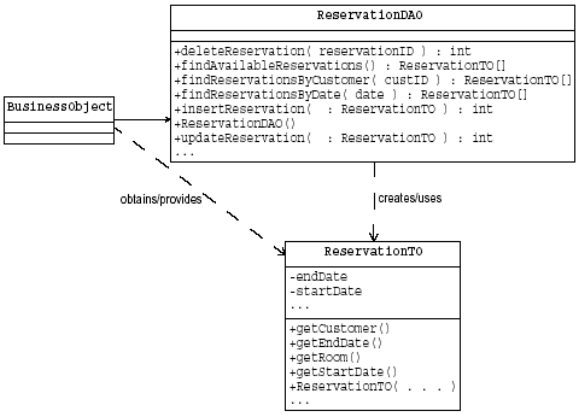
数据访问层

- 数据源
  - 可以是关系数据库、文件系统等
- 数据访问层
  - 隔离业务逻辑层和数据存储
  - 好处
    - 增加代码重用性
    - 尽可能消除业务逻辑层对数据源的依赖
      - 可以通过配置文件进行改变
    - 隐藏了数据操作的细节，可以实现各种优化

数据访问对象（DAO）

- 数据访问对象
  - 使用DAO抽象并封装对数据源的所有访问
  - DAO管理数据源的连接以获取和存储数据
  - 使用工厂管理DAO

– 示例：



- DAO设计考虑
  - 提供无状态的方法来实现对数据源的操作
  - 实现涉及到三方面问题：
    - 无状态
    - 如何划分组件
    - 参数如何定义

- DAO组件的接口定义和划分
  - 先定义好DAO组件的接口
    - 可以通过不同工厂类来建立不同DAO实现类
  - DAO组件的划分
    - 可以考虑按照业务实体进行划分
    - 某一组件最好只访问一种数据源

- DAO组件与事务
  - 保证事务完整性
    - 如何传递事务上下文？
    - 事务是否跨越不同的数据源(分布式事务)？

- 组件的输入输出参数
  - 输入参数
    - 一般读操作所要传递的参数
      - » 查询条件、分页信息
    - 一般写操作需要转递的参数
      - » 更新数据
  - 返回值
    - 普通类型
    - 结果集
    - 自定义的数据结构

- DAO组件和存储过程
  - 利用存储过程
    - 有助于优化和提高性能
    - 有助于将某些二维表的结果集转换为另外一种二维表
  - 不适合使用存储过程的操作
    - 查询/修改的条件或参数是变化的
    - 存储过程尽量避免实现业务逻辑
    - 有些操作用SQL语句很难写

- 设计通用DAO
  - 为数据访问层提供通用的数据访问接口
  - 减少数据访问操作的代码
    - 简化执行SQL语句和调用存储过程的代码
  - 进行数据库连接的管理
    - 包括数据库名称、认证信息等
  - 在不同的数据源之间，可以提供统一的接口

相关数据层设计

- 考虑：
  - 数据架构规划与数据库设计
  - 数据库设计与类的设计
  - 数据库设计与XML设计
  - 数据库性能规划
  - 封装数据库设计

业务层与数据层间传送数据

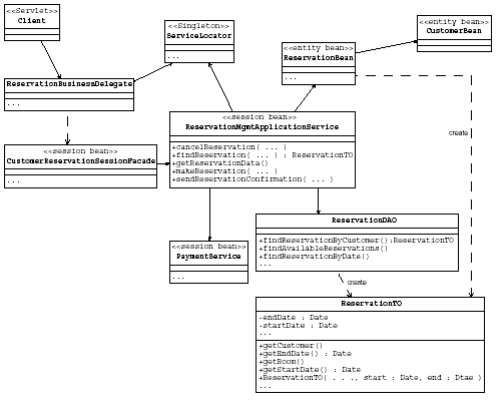
- 带CRUD操作的自定义业务实体
  - 优点：
    - 封装
      - 自定义业务实体封装了下游数据存取逻辑组件的操作
    - 接口调用
      - 调用者只需要通过一个接口与业务实体数据打交道
    - 私有成员
      - 可以隐藏不希望暴露给调用者的信息
  - 缺点：
    - 处理业务实体类的集合
    - 增加开发时间

数据传输对象（ DTO）

- 数据传输对象（ DTO）
  - 为业务实体对象与实际的关系数据存储提供松散耦合关系
  - 一个数据传输对象是一个没有业务逻辑但可序列化的数据实体

– 数据传输对象的设计
  - 数据传输对象在实现上可以是一个只有构造器和简单类型字段成员（或属性）的类或结构体
  - 内部唯一标识与外部唯一标识
  - 数据实体成员
  - 数据版本信息
    – 用于支持并发检测

– 示例：



对象关系映射（ORM）

- 对象关系映射（ORM）
  - 作用
    - 通过结构化的映射使程序更强健
    - 减少错误代码
    - 始终对性能进行优化
    - 数据源独立性
  - 目标：
    - 利用关系型数据库的优点
    - 保持面向对象的语言对象方式
    - 做更少的工作，减少DBA压力

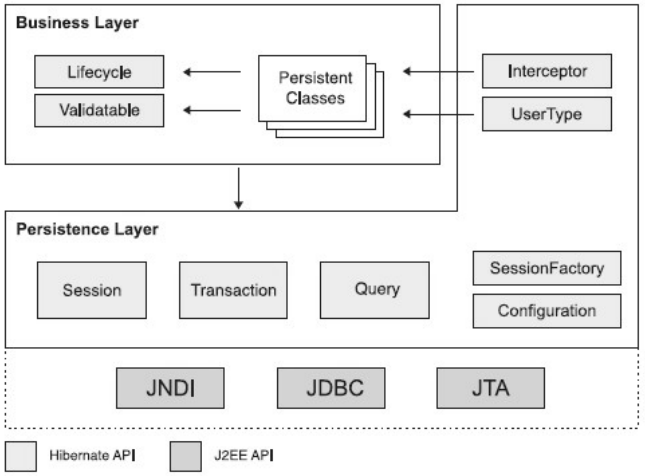
- 对象－关系数据库的匹配

对象	关系数据库
类的属性（基本类型）	表的列
类	表
1:n/n:1	外键
n:m	关联表
继承	。。。

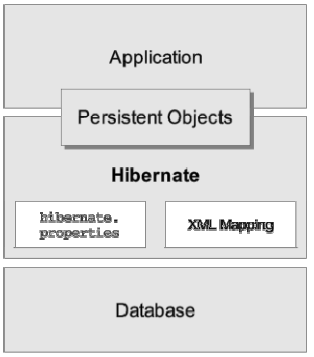
- ORM 实现：
  - 透明的持久化 ：
    - 用面向对象的语言直接操作关系型数据的能力
  - 自动进行数据同步问题的检查
  - 灵活的获取与缓存
    - 实现Lazy Loading， Outer Join Fetching，Runtime SQL Generation功能
  - 继承的映射策略
  - 开发工具

- 具体实现方式
  - 利用反射机制，在运行时自动产生SQL 语句，执行ORM操作
  - 通过ORM工具，生成代码，通过生成的代码来实现ORM

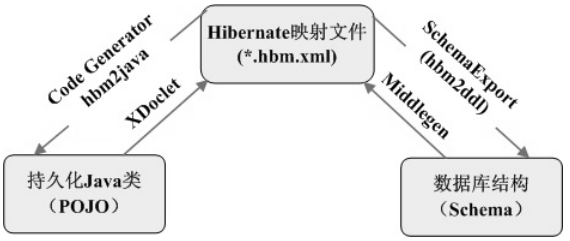
Hibernate架构



- Hibernate开发
  - 持久化类设计
  - 持久化类和关系数据库的映射
  - 应用开发



– 持久化类和关系数据库的映射



– 映射文件－User.hbm.xml

```
<hibernate-mapping>
<class name=" com.test.hibernate.User " table="TBL_USER">
  <id name="id" column="ID">
     <generator class="native"/>
  </id>
  <property name="name" column="NAME"/>
  <property name="birthday" column="BIRTHDAY"/>
  <property name="email" column="EMAIL"/>
</class>
</hibernate-mapping>
```

– 映射一对多关联
  • User-Address的映射

```
<class name="com.test.hibernate.User" table="T_USER">
        <id name="id" column="userId">
        <generator class="native"/></id>
        <set name="addresses" lazy="true" cascade="all">
                <key column="addressId"/>
                <one-to-many class="com.test.hibernate.Address"/>
        </set>
  </class>

   <class name="com.test.hibernate.Address" table="T_ADDRESS">
     <id name="id" column="addressId"> <generator class="native"/></id>
  </class>
```


面向方面的编程（ AOP）

- 面向方面的编程（ AOP）
  – 是一种编程技术
  – 允许程序员对横切关注点或横切典型的职责分界线的行为（例如日志和事务管理）进行模块化
  – 将那些影响多个类的行为封装到可重用的模块中

– 创建自己的Interceptor

```
public class MyInterceptor implements MethodInterceptor {
    private final Log logger = LogFactory.getLog(getClass());

    public Object invoke(MethodInvocation methodInvocation) throws
  Throwable {
        logger.info("Beginning method (1): " +
        methodInvocation.getMethod().getDeclaringClass() + "." +
        methodInvocation.getMethod().getName() + "()");
        long startTime = System.currentTimeMillis();
        try{
                Object result = methodInvocation.proceed();
                return result;
        }finally{
                logger.info("Ending method (1): " +
                methodInvocation.getMethod().getDeclaringClass() + "." +
                methodInvocation.getMethod().getName() + "()");
                logger.info("Method invocation time (1): " +
                (System.currentTimeMillis() - startTime) + " ms.");
            }
    }
  }
```


– 编写业务对象及其接口

```
public interface BusinessInterface {
        public void hello();
        }
public class BusinessInterfaceImpl implements BusinessInterface{
        public void hello() {
        System.out.println("hello Spring AOP.");
        }
        }
```

– 使用Interceptor

```
<bean id="businessTarget"
   class="com.spring.testaop.BusinessInterfaceImpl"/>
<bean id="myInterceptor" class="com.spring.testaop.MyInterceptor"/>

<bean id="businessBean"
   class="org.springframework.aop.framework.ProxyFactoryBean">
     <property name="proxyInterfaces">
          <value>com.spring.testaop.BusinessInterface</value>
     </property>
     <property name="interceptorNames">
        <list>
        <value>myInterceptor</value>
        <value>businessTarget</value>
        </list>
     </property>
</bean>
```


– 测试类

```
ClassPathResource resource = new
   ClassPathResource("com/spring/testaop/aop_bean.xml");
XmlBeanFactory beanFactory = new XmlBeanFactory(resource);

BusinessInterface businessBean = (BusinessInterface)
   beanFactory.getBean("businessBean");
businessBean.hello();
```

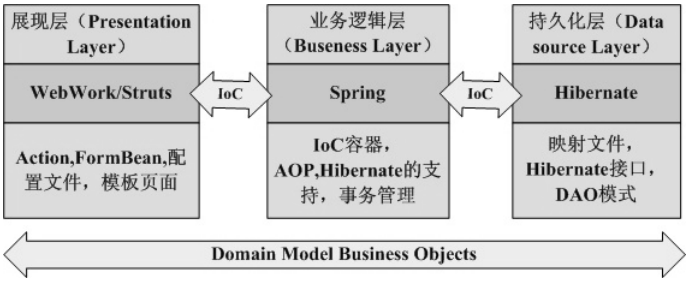
- Spring声明式事务处理
  - 使用TransactionInterceptor拦截器和常用的代理类TransactionProxyFactoryBean
  - 定义数据源，事务管理类
  - 定义事务拦截器

```
<bean id = "transactionInterceptor"
  class="org.springframework.transaction.interceptor.TransactionInterceptor"
  >
  <property name="transactionManager">
        <ref bean="transactionManager"/>
  </property>
  <property name="transactionAttributeSource">
        <value>com.test.UserManager.*r=PROPAGATION_REQUIRED<
  /value>
  </property>
</bean>
```


- 为组件声明一个代理类：ProxyFactoryBean

```
<bean id="userManager"
  class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
        <value>com.test.UserManager</value>
  </property>
  <property name="interceptorNames">
        <list>
              <idref local="transactionInterceptor"/>
        </list>
  </property>
</bean>
```

基于B/S的典型三层架构



通用架构服务设计

通用架构机制

- 通用架构机制：
  - 并发
  - 事务
  - 安全
  - 通信
  - 数据格式
  - 日志
  - 配置管理
  - 异常处理

系统安全设计

- 设计流程
  - 分析潜在的威胁
  - 评估威胁并设定优先级
  - 针对威胁进行分类
  - 根据威胁选择安全技术
  - 设计对应的安全服务

- 设计基本原则
  - 使用成熟的安全系统
  - 以小人之心度输入数据
  - 外部系统是不安全的
  - 最小授权
  - 减少外部接口
  - 缺省使用安全模式
  - 安全不是似是而非
  - 从STRIDE思考
  - 在入口处检查
  - 从管理上保护好你的系统

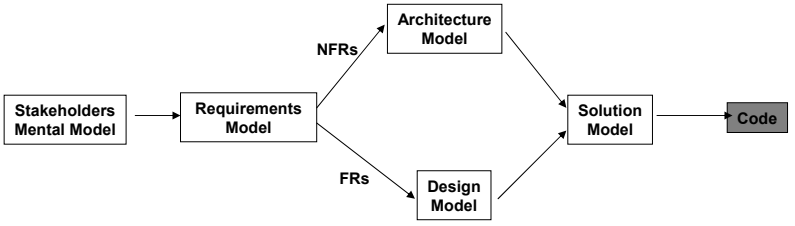
- 常用安全技术
  - 认证
  - 授权
  - 安全通信
  - 审计

通讯设计

- 通讯策略
  - 三要素：
    - 同步/异步、格式和协议
  - 模式：
    - 无连接：于消息
    - 有连接：分布式组件
  - 考虑因素：
    - 扩展性
    - 可用性
    - 可管理性

创建软件的架构

软件开发过程模型



架构模型

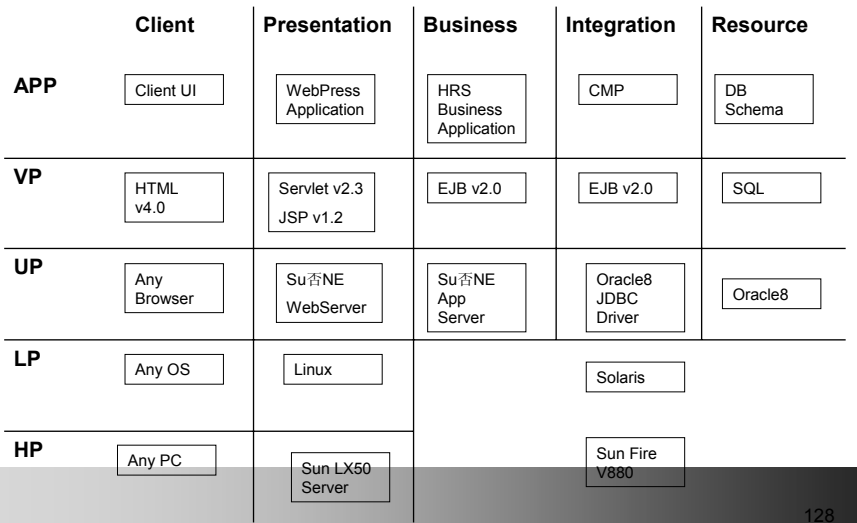
- 架构模型包括：
  - 高层部署图：
    - 模型化系统的网络拓扑结构及每个硬件节点中的基本应用
  - 详细的部署图：
    - 模型化高层部署图中每个硬件节点中的底层组件的组织情况
  - Tier与Layer包图：
    - 记录架构模型中每个Tier与Layer矩阵的技术选择

- UML图：
  - 组件图
    - 表示可部署的系统软件组件及其之间的关系
  - 部署图
    - 表示系统的硬件节点及其之间的关系
  - 包图
    - 体现系统模块划分及其关系

将系统拆分为组件

- 系统组件
  - 定义为可部署的系统单元
  - 可部署的系统单元类型：
    - 应用程序组件
    - 服务器组件
    - 基础设施组件

- 架构的形成过程示例
  1. 选择组件技术
  2. 确定组件需要的资源及如何成为其它组件的资源
  3. 确定系统的高级部署图
  4. 确定系统的详细部署图
  5. 确定系统的Tier与Layer包图



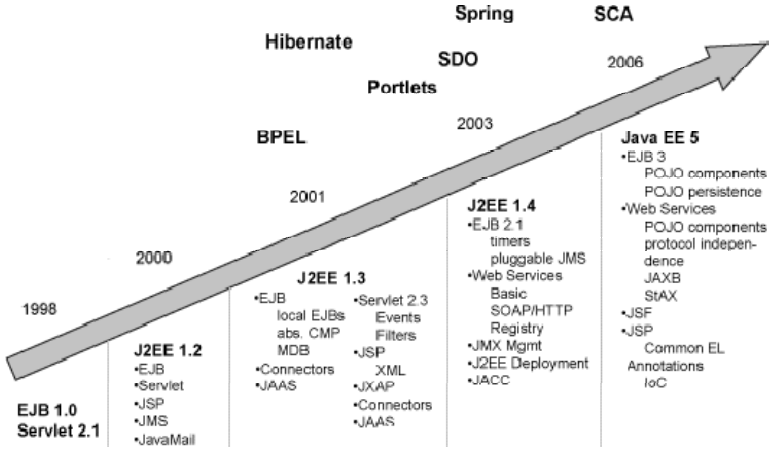
架构的迭代过程

- 建立初步的架构模型
- 调整架构模型使之符合系统的QoS
  - 调整分解出来的基础设施组件，以满足各QoS需求
  - 评估各种设计方案
  - 创建部署配置的原型，并进行性能测试以确认QoS
- 对上述过程进行迭代，产生最终的架构模型

第二部分：软件开发

Java EE 新特性

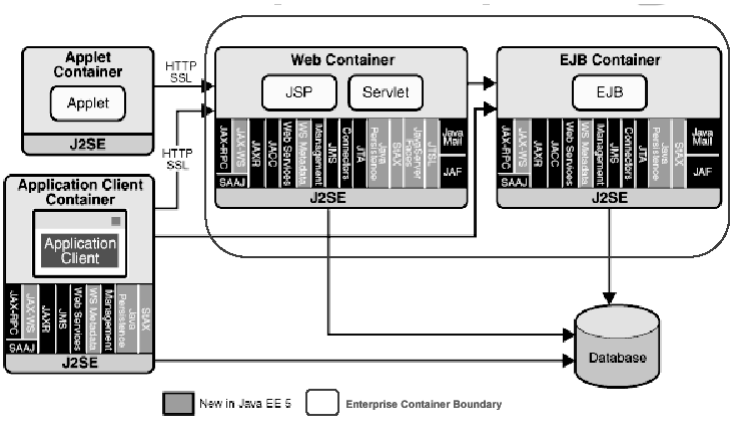
Java EE 的历史



- Java EE 5规范的以前版本的“难点”：
  - 业务逻辑编程的复杂性
  - 持久性编程模型的复杂性和性能
  - 表示层/逻辑混合
  - Web 服务的类型、复杂性、文档模型、扩展和性能
  - 团队协作开发
  - 编辑-编译-调试周期

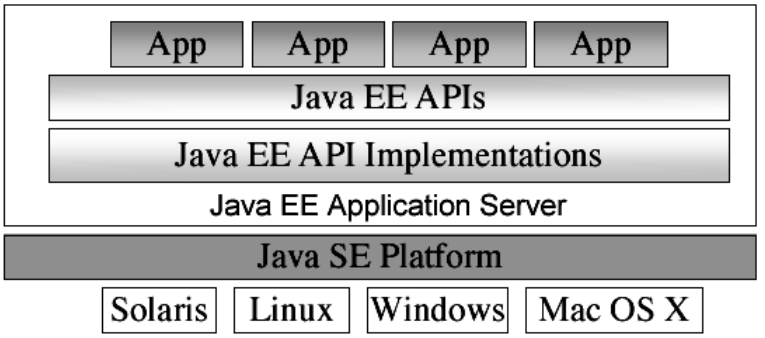
- Java EE 5 规范的主题是简化：
  - 简化业务逻辑开发
  - 简化测试和依赖关系管理
  - 简化 O/R 持久性
  - 增强 Web 服务编程模型

Java EE 技术



- Java EE 5 主要技术规范
  - Java Persistence API (JSR-220)
  - EJB 3.0 (JSR-220)
  - JavaServer Faces 1.2 (JSR-252)
  - JSP 2.1 (JSR-245)
  - JAX-WS 2.0 (JSR-224)
  - JAXB 2.0(JSR-222)
  - Common Annotations (JSR-250)
  - StAX (JSR-173)

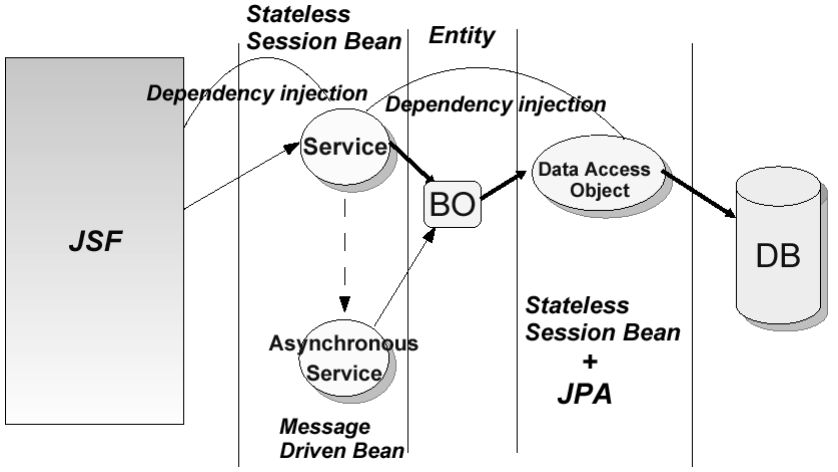
Java EE 应用平台



JavaEE主流应用服务器

- JavaEE主流应用服务器
  - IBM WebSphere AS
  - Oracle/Bea WebLogic AS
  - GlassFish
  - JBoss
  - Tomcat
  - ……

Java EE 5 架构



Java EE 5新特性

- Java EE 5新特性
  - 标注取代部署描述符
  - 简化EJB 开发
  - 使用依赖注入资源
  - Java 持久化 API
  - Web 服务

标注的使用

- Java EE 5 不需要任何部署描述符
  - （Servlet 规范所需的部署描述符 web.xml 文件除外）
- Java EE 5的标注：
  - 使用标注的情况：
    - 定义和使用 Web 服务
    - 开发 EJB 软件应用程序
    - 将 Java 技术类映射到 XML
    - 将 Java 技术类映射到数据库
    - 将方法映射到操作
    - 指定外部依赖关系
    - 指定部署信息，其中包括安全属性

- 示例：

```
package test;
import javax.ejb.*;

@Stateless
public class HelloWorldSessionBean implements
   mypackage.HelloWorldSessionLocal {

}
```

EJB 的开发

- EJB 3.0 规范已拆分为三个子规范：
  - EJB 3.0 简化 API：
    - 定义用于EJB 组件（特别是会话 Bean 和消息驱动的 Bean）编码的 API
  - 核心契约和要求：
    - 定义 Bean 和 EJB 容器之间的 EJB 契约
  - Java 持久化 API：
    - 为持久化定义新实体 Bean 模型

- EJB 3.0 的主要改变：
  - EJB 组件不再要求主接口
  - Java SE 5.0 标注是实现 EJB 3.0 组件的重要基础
  - EJB 3.0 引入了业务接口概念，而非单独的远程和本地接口
  - 使用JPA简化了对象关系映射

- 代码示例：

```
@Stateless
public class StockBean{
        @BusinessMethod
         public double getQuote(String symbol){
                return 45.9;
        }
}
```


– EJB 3.0 客户示例：

```
@EJB StockBean myStock;
...
double quote = myStock. getQuote(345) ;
```

- EJB 3.0 的其它改变：
  - 容器服务
  - 回调
  - 拦截器
    - 执行面向方面编程(AOP)
  - 依赖注入
    - EJB 没有使用 JNDI 查找，而是通过注入代码定义资源引用

- 容器服务

```
@Stateless
public class StockBean{
   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public double getQuote(String symbol){
     return 45.9;
   }
}
```

– 回调

```
@Stateless public class StockBean {
  public double getQuote(String symbol)
  {
        return 45.9;
  }

  @PostConstruct initializeCache()
  {
  }
}
```


– 使用回调监听器：

```
public class MgmtCallbackListener
{
  @PrePassivate public clearCache(Object obj)
  {
        Stock stock = (Stock) obj;
        //perform logic
  }
}

@CallbackListener(MgmtCallbackListener)
@Stateless public class StockBean {
  public double getQuote(String symbol)
  {
        return 45.9;
  }
}
```

– 使用拦截器：

```
public class StockRequestAudit {
   @AroundInvoke
   public Object auditStockOperation(InvocationContext inv) throws Exception {
            try {
                     Object result = inv.proceed();
                     Auditor.audit(inv.getMethod().getName(),
   inv.getParameters[0]);
                     return result;
            } catch (Exception ex) {
                     Auditor.auditFailure(ex);
                     throw ex;
            }
   }
}
```



```
@Stateless
@Interceptors({StockRequestAudit})
public class StockBean implements Stock{
   public double getQuote(String symbol){
            return 45.9;
   }
}
```

– 依赖注入：

```
@Stateless
public class StockBean
{
        @EJB(name="MarketBean", businessInterface="Market")
        Market market;

        @Resource(name="StockDB, resourceType="javax.sql.DataSource")
        DataSource stockDS;

        public double getQuote(String symbol)
        {
            Connection con = stockDS.getConnection();
            //DO JDBC work
            return market.getCurrentPrice(symbol);
        }
}
```


使用依赖注入资源

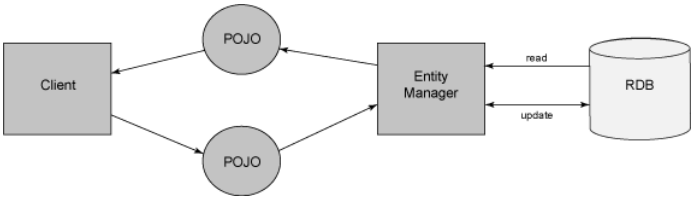
- 通过依赖注入，对象可以使用标注直接请求外部资源
  - 可以注入的资源：
    - SessionContext 对象
    - DataSource 对象
    - EntityManager 接口
    - 其他 Enterprise Beans
    - Web 服务
    - 消息队列和主题
    - 资源适配器的连接工厂

Java 持久化 API （JPA）

- JPA 可以用于：
  - EJB 组件
  - Web 应用程序
  - 应用程序客户端
  - Java SE 应用程序
- Java 持久化 API
  - @Entity
  - EntityManager API
  - Query API
  - EJB QL

- JPA 主要特点：
  - 实体是 POJO
  - 标准化的对象关系映射
    - 使用标注来指定对象关系映射信息，仍支持 XML 描述符
  - 命名查询
  - 简单的打包规则
    - 几乎可以在 Java EE 应用程序中的任意位置将其打包
  - 分离的实体
    - 可以对其序列化，通过网络其发送到其他地址空间
  - EntityManager API
    - 可以使用标准 EntityManager API 来执行涉及实体的CRUD操作

- JPA结构：



– 类型：实体和表

```
@Entity
@Table(name="CUSTOMER")
public class Customer implements Serializable {
...
```

– 使用 EJB 容器中的依赖注入将EM注入到应用程序
  • 在 Java SE 中也可以通过EntityManagerFactory 查找

```
@PersistenceContext (unitName="db2")
private EntityManager em;
```


– 应用程序使用EM API：

```
Customer customer =
  (Customer)em.find(Customer.class,customerId);

CustomerOrder newOrder = new CustomerOrder();
newOrder.setStatus("OPEN");
…
em.persist(newOrder);
```

- 主键：

```
@Entity @Table(name="CUSTOMER")
public class Customer implements Serializable {

   private Integer id;
   …

   @Id
   @Column(name="CUST_ID")
   public Integer getId() {
      return id;
   }
   public void setId(Integer id) {
      this.id = id;
   }
...
```


- 关系：

```
@Entity @Table(name="CUSTOMER")
public class Customer implements Serializable {
   …
   @OneToOne(cascade=CascadeType.ALL ,
   fetch=FetchType.EAGER )
   @JoinColumn(name="OPEN_ORDER_ID",referencedColumnName
   ="ORDER_ID")
   public CustomerOrder getCurrentOrder() {
         return currentOrder;
   }
   public void setCurrentOrder(CustomerOrder currentOrder) {
         this.currentOrder = currentOrder;
   }

}
```

- 继承：

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE,
                discriminatorType=STRING,
                discriminatorValue="CUST")
public class Customer { ... }

@Entity
@Inheritance(discriminatorValue="VCUST")
public class ValuedCustomer extends Customer { ... }
```


Web 服务

- Java EE 5 通过使用标注改进和简化了对 Web 服务支持
- 相关规范：
  - JSR 224
    - Java API for XML-Based Web Services (JAX-WS) 2.0
  - JSR 222
    - Java Architecture for XML Binding (JAXB) 2.0
  - JSR 181
    - Web Services Metadata for the Java Platform

- JAX-WS 2.0
  - Java EE 5 为 Web 服务引入的新编程模型
  - 与JAX-RPC 1.1 相比
    - JAX-WS 仍然支持 SOAP 1.1 over HTTP 1.1
    - JAX-WS 仍然支持 WSDL 1.1
    - 与 JAX-RPC 1.1 的主要区别在于，其将所有数据绑定都委托给了 JAXB 2.0

- JAX-WS 还可以与 EJB 3.0 一起使用来简化编程模型

```
@WebService
public interface StockQuote {
    public float getQuote(String sym);
}

@Stateless
public class QuoteBean implements StockQuote {
   public float getQuote(String sym) { ... }
}
```

Java EE Web 技术

- Java EE Web技术包括：
  - Servlets
  - Filters
  - Java Server Pages (JSP)
  - JSP Standard Tag Library (JSTL)
  - Java Server Faces (JSF)

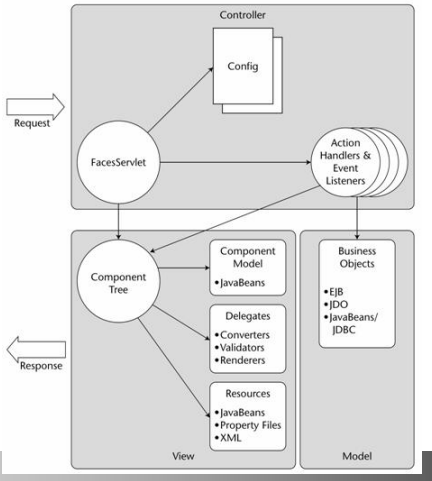
JavaServer Faces (JSF)

- JSF 是 Java EE 5 规范的一部分
- JSF 的优点
  - 用户界面组件
  - 事件驱动模型
  - 通过使用 Renderer 灵活地表示 UI
  - 很容易与各种工具集成
  - 通过IDE工具支持拖放式开发

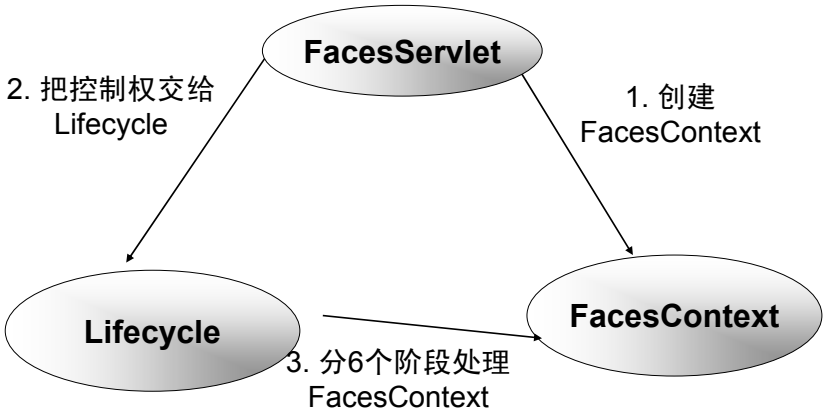
- JSF 与 JSP 的比较

	JSP	JSF
开发	更底层	易开发
事件	Http 请求	应用事件
UI	标签	UI 组件
应用	页面	应用程序

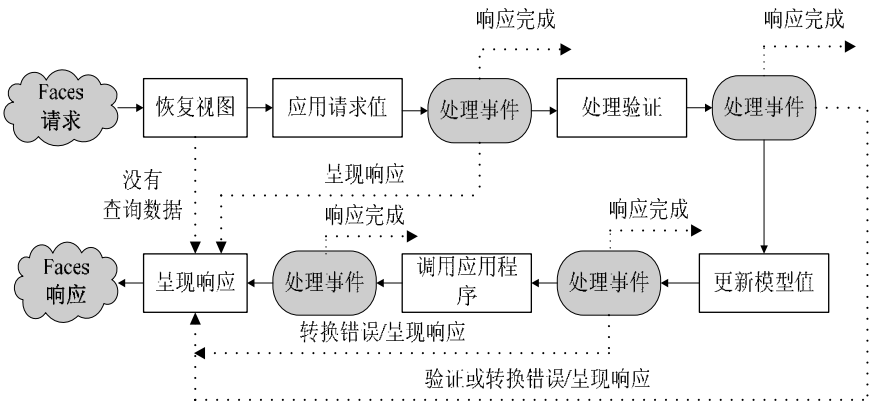
- JSF基于组件的Model-2 MVC策略



- JSF的工作方式



- JSF请求处理生命周期



- 事件监听器

```
//实现 ActionListener
class CountryListener implements ActionListener {
    public void processAction(ActionEvent event)
        throws AbortProcessingException {
      //处理 ActionEvent 的代码
  }
}

<h:commandLink id=“selectCountry" action="bookstore">
   <f:actionListener type="CountryListener" />
</h:commandLink>
```


- 托管的Bean：

```
<managed-bean>
<managed-bean-name> Userbean</managed-bean-name>
<managed-bean-class>messages.User</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>


<h:inputText id="email" value="#{Userbean.email}"/>
```

Java EE优化

Java EE应用的性能问题

- 性能问题的最明显表现是网页的响应时间变慢
- 相关方面：
  - 应用服务器资源的使用情况
  - JVM堆的使用情况
  - 系统资源的使用情况
  - 数据库资源的使用情况
  - 网络活动

常用的性能优化配置项

- 使用应用服务器与Java EE有关的配置项提高系统性能
  - JVM堆和垃圾回收的配置
  - 线程池的配置
  - EJB的配置
  - 数据库的配置

- JVM堆和垃圾回收的配置
  - 是任何Java应用的性能优化的基础
  - 可配置初始堆大小和最大堆大小
    - 考虑物理内存的大小
  - 选择不同的垃圾回收算法

- 线程池的配置
  - 通过调整线程池的大小满足应用负载的需要
  - 线程池的大小应该能充分利用CPU，同时又不使其过载

- EJB的配置
  - 不同类型的EJB以资源池的方式实现
  - 池大小和初始Bean的数量会明显影响应用的性能

- 数据库的配置
  - 增加数据库连接池的大小会提高性能
  - 不同类型的SQL操作（例如事务型和批处理型）应使用不同的连接池
  - 充分利用PreparedStatement的缓存功能

Java EE 应用代码的性能优化

- Java 编程
  - 使用StringBuffer代替String
    - 如果字符串有连接的操作，替代的做法是用StringBuffer类的append方法
  - 创建对象时，分配合理的空间和大小
    - 将对象初始化为适当的大小，如集合类

– 优化循环体

```
for(int i=0;i =list.size();i++)

len = list.size();
for(int i=0;i =len;i++)
```


– 对象的创建

- new操作符是很消耗系统资源
- 尽量在使用时再创建该对象
- 应该尽量重复使用一个对象，而不是声明新的同类对象

– 变量的注意事项
  - 尽量使用局部变量
  - 使用静态变量共享

– 方法调用
  - 减少方法的调用
  - 不影响可读性等情况下，可以把几个小的方法合成一个大的方法
  - 在方法前加上final，private关键字有利于编译器的优化

– 慎用异常处理
  - 流程尽量用while，if等处理
  - 在不影响代码强健性的前提下，可以把几个try/catch块合成一个

– 同步
  - 应用多线程也是为了获得性能的提升，应该尽可能减少同步。
  - 需要同步的地方，尽量减少同步的代码段

- 使用Java系统API
  - Java的API一般都做了性能的考虑，如果完成相同的功能，优先使用API而不是自己写的代码

- Web组件代码
  - 尽量不使用同步
    - 太多的同步会失去多线程的优势
  - HttpSession的使用
    - 尽量减少session的使用
    - 不用保存太多的信息在HttpSession中
    - 手工清除Session

- JSP代码
  - include指令及include动作
  - 无需跟踪会话状态的JSP，关闭自动创建的会话
- Servlet代码
  - 在Servlet之间跳转时，forward比sendRedirect更有效
  - 在init()方法里缓存静态数据，而在destroy()中释放它

- EJB代码
  - 不需要EJB服务的时候,建议使用普通Java类
  - 有效使用设计模式
  - 缓存Home接口
    - JNDI是个远程对象，通过RMI方式调用
    - 可在系统初始化时就获得需要的Home接口并缓存

- 封装实体Bean
  - 用会话Bean封装对实体Bean的访问
- 释放有状态会话Bean
  - 通常有状态会话Bean的释放是在超时时发生的
  - 确认使用完不再需要时，应显式的将其释放掉

- JMS代码
  - 使用接收程序能直接使用的最简单、最小的消息类型
  - 为应用程序定义专用的JMS连接工厂
  - 尽量使用异步消费者

- 数据库访问
  - 使用速度快的JDBC驱动
    - 选择不同的JDBC驱动，在效率上会有差别
  - 使用JDBC连接池
    - 使用连接池，避免频繁打开、关闭数据库连接
    - 循环利用连接可以显著的提高性能

  - 缓存DataSource
    - 一个DataSource对象代表一个实际的数据源
    - 为了避免JNDI调用，可在系统中缓存要使用的DataSource
  - 关闭所有使用的资源
    - 每次申请和使用完资源后，应该释放供别人使用

- 批数据的处理
  - 使用直接访问数据库的方法，用SQL直接存取数据
- 缓存经常使用的数据
  - 数据经常使用，但又不经常变化，可以在系统中缓存起来
  - 缓存工作可以在系统初始化时一次性读取数据

- JDBC代码
  - 严格资源使用
  - 关闭自动提交
  - 使用PreparedStatement
  - 使用RowSet
  - 优化SQL语句
    - 显式获取列
    - 实现分页查询
    - SQL重用

- 通用代码调优
  - 减小没有必要的操作
  - 选择适当的类型
  - 尽量使用pool、buffer和cache

其它性能考虑

- 综合考虑性能：
  - 服务器软硬件环境
  - 集群技术
  - 系统构架设计
  - 系统部署环境
  - 数据结构
  - 算法设计