**新开普软件平台（NSP）架构**

|  |  |  |
| --- | --- | --- |
| 文件状态：  【】草稿  【】修改稿  【】正式发布 | 文档密级： |  |
| 当前版本： | 1.0 |
| 作者： | 祁有栋 |
| 完成日期： |  |

机构: 郑州新开普电子股份有限公司

电话: 0371-67579650

网址: http://www.newcapec.com.cn

**文档修订记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本号** | **作者** | **备注** |
| 2013-09-13 | V1.0 | 祁有栋 | 起草 |
| **日期** | **版本号** | **作者** | **备注** |
|  |  |  |  |
| **修改内容** |  | | |
| **增加内容** |  | | |
| **删除内容** |  | | |
| **日期** | **版本号** | **作者** | **备注** |
|  |  |  |  |
| **修改内容** |  | | |
| **增加内容** |  | | |
| **删除内容** |  | | |

**目录**

[1 简介 5](#_Toc367793756)

[1.1 目的 5](#_Toc367793757)

[1.2 范围 5](#_Toc367793758)

[1.3 术语词汇表 5](#_Toc367793759)

[1.4 参考资料 6](#_Toc367793760)

[2 构架表示方式 6](#_Toc367793761)

[3 构架设计目标与约束 7](#_Toc367793762)

[3.1 关键功能需求 7](#_Toc367793763)

[3.2 关键非功能性需求 8](#_Toc367793764)

[3.3 开发策略 8](#_Toc367793765)

[3.3.1 软件复用策略 8](#_Toc367793766)

[3.3.2 使用开源构件 8](#_Toc367793767)

[3.3.3 使用商业构件 9](#_Toc367793768)

[3.4 其它产品依赖 9](#_Toc367793769)

[3.5 其它设计约束 9](#_Toc367793770)

[4 平台总体视图 9](#_Toc367793771)

[4.1 概述 9](#_Toc367793772)

[4.2 平台总体框架 10](#_Toc367793773)

[4.3 产品建设基础平台 10](#_Toc367793774)

[5 逻辑视图 12](#_Toc367793775)

[5.1 概述 12](#_Toc367793776)

[5.2 服务组件 13](#_Toc367793777)

[5.2.1服务组件描述 13](#_Toc367793778)

[5.2.2服务组件封装标准 14](#_Toc367793779)

[5.2.3服务组件调用机制 17](#_Toc367793780)

[5.3 系统层次结构模型 19](#_Toc367793781)

[5.4 主要的设计包和子系统 21](#_Toc367793782)

[5.4.1 子系统划分 21](#_Toc367793783)

[5.4.2 业务子系统 22](#_Toc367793784)

[5.4.3 XXX子系统 26](#_Toc367793785)

[5.4.4 集成门户 26](#_Toc367793786)

[5.5 架构机制（Architectural Mechanism） 30](#_Toc367793787)

[5.5.1 集中部署时的组件化机制 30](#_Toc367793788)

[5.5.2 分布式部署时的组件化机制 33](#_Toc367793789)

[5.5.3 持久化机制（Persistence Mechanism） 33](#_Toc367793790)

[5.5.4 安全机制（Security Mechanism） 33](#_Toc367793791)

[5.5.5 界面个性化机制（View Customize Mechanism） 38](#_Toc367793792)

[5.5.6 数据个性化机制（Data Customize Mechanism） 38](#_Toc367793793)

[5.5.7 功能个性化机制（Function Customize Mechanism） 38](#_Toc367793794)

[5.5.8 高可用机制 38](#_Toc367793795)

[6 实现视图 40](#_Toc367793796)

[6.1 概述 40](#_Toc367793797)

[6.2 系统实现模型总体结构 41](#_Toc367793798)

[6.3 开发框架的特点 43](#_Toc367793799)

[6.4 系统WebRoot结构 45](#_Toc367793800)

[6.5 系统源码包及目录结构 47](#_Toc367793801)

[6.5.1整体包结构 47](#_Toc367793802)

[6.5.2整体目录结构 49](#_Toc367793803)

[6.5.3服务组件包目录结构 51](#_Toc367793804)

[6.5.4配置文件目录结构 52](#_Toc367793805)

[7 进程视图 53](#_Toc367793806)

[7.1 概述 53](#_Toc367793807)

[7.2 大并发集群部署时的请求处理 53](#_Toc367793808)

[7.3 运行统一认证 54](#_Toc367793809)

[7.4 服务集成 54](#_Toc367793810)

[8 部署视图 55](#_Toc367793811)

[8.1 概述 55](#_Toc367793812)

[8.2 大并发应用服务器部署 55](#_Toc367793813)

[8.2.1应用服务器性能设计 56](#_Toc367793814)

[8.2.2应用服务器可靠性设计 57](#_Toc367793815)

[8.2.3应用服务器伸缩性设计 57](#_Toc367793816)

[8.2.4应用服务器部署对设计约束 58](#_Toc367793817)

[8.3 数据库服务器部署 59](#_Toc367793818)

[8.3.1数据服务器性能设计 60](#_Toc367793819)

[8.3.2数据服务器可靠性设计 61](#_Toc367793820)

[8.3.3数据服务器可伸缩性设计 61](#_Toc367793821)

# 简介

## 目的

本文档全面与系统地表述新开普软件平台系统架构，并通过使用多种视图来从不同角度描述系统的各个主要方面，以满足相关涉众（客户、设计人员、开发人员等）对目标系统的不同关注焦点。

本文档记录并表述了架构师对系统构架方面做出的重要决策；项目经理将根据架构定义的组件结构制定项目的开发计划；设计员将据此进行各组件的详细设计；测试设计员按照架构设计系统的总体测试框架；另外架构文档还用于指导各组件的实施、集成及测试。

## 范围

本文档适用于“新开普软件平台”的总体构架。

## 术语词汇表

本文中的词汇表重新定义了各种规范中描述中，比较混淆的词语，这次重新定义是为了架构文档表达的共识性，当前仅限于本文档范围内有效，说明如下：

术语表格

|  |  |
| --- | --- |
| **词汇名称** | **词汇说明** |
| 组件（服务组件） | 本文中组件及组件服务，描述的是同一概念。当强调组件服务的概念时，表示该组件具备远程接口，并且受部署环境中服务治理的约束。  在标准规范中，对组件的定义是多层次、多粒度的。为了明确表达概念。本文中组件（服务组件）定义在一个比较粗的粒度上：一个组件应该能够独立部署对外提供约束，符合功能自治、数据自治的原则。在这个粒度上一个业务服务就成为一个服务组件了。（例如：考勤服务组件等） |
| 构件 | 在一些文档中，构件与组件表达的是同一意思，只是在中文文献中翻译的不同。本文对构件进行重新定义：构件表示一个组件中可以替换的模块单位。构件本身不会独立部署，并对外提供完整的功能服务，它是组件的组成部分，例如一个服务组件往往包含一个远程接口封装的构件。构件本身为了可替换、可移植也具备严格的接口定义，并且同一构件可以在不同服务组件中被重用，它用来描述小粒度的软件复用模块。 |
| 部件 | 为了明确表达系统对通用技术产品的依赖，本文定义了部件的感念：部件定义了一个可以重用的通用的技术产品，在产品建设中会根据情况选择性使用。部件的构件往往依赖于第三方开源产品或商用产品，在产品研发人员来说基于开发框架封装的接口来访问这些部件，保证了开发接口的稳定性。（例如：报表部件、工作流部件等） |

## 参考资料

参考资料

# 构架表示方式

本文档以一系列的视图（View）来表示系统的软件构架，主要包括平台总体视图、逻辑视图、处理视图、部署视图、开发视图等；每个视图拥有一个或多个模型（Model）（例如逻辑视图包含分析模型、设计模型和数据模型等）；并围绕相关视图来描述系统的基本结构、组成机制与工作原理等。

本文档还将系统的构架机制描述也放在了逻辑视图之下。

本文档主要使用统一建模语言（UML）来充当相关模型的表达语言；主要图表（Diagram）引用自目标系统的Rose Model。

# 构架设计目标与约束

描述构架设计必须满足的关键系统功能需求和质量约束，这些功能需求和质量要求对软件构架有重大的影响，并决定了构架的设计。

本节同时还列明影响构架的其他相关因素，如软件的复用策略、使用商业构件、设计与实施的策略等。

## 关键功能需求

1. 基于该平台，能够完成企业级产品的建设。同时，基于该平台公司也能够应对规模性运行产品的建设（大数据、大并发）。
2. 系统提供一个完善、灵活的开发框架。为企业级产品与运行级产品的开发人员提供一致化的开发视图。
3. 系统提供组件开发的规范与环境，提供组件的开发、部署与管理工具。
4. 系统需要提供页面定制、功能定制及数据定制的功能支持。
5. 系统具备页面集成、功能集成、数据集成的能力，满足系统集成第三方产品系统，增强自身功能的需求。
6. 系统提供单点登录、权限控制、基础信息库、报表及页面集成门户五部分基础功能组件

## 关键非功能性需求

***待补充！***

## 开发策略

### 软件复用策略

系统中重要基础层服务组件应当具备较高的设计与构建质量，可以在平台上不同产品中复用。通过对系统构建层次的合理划分、接口的合理定义及服务组件依赖性的管理，实现服务组件间的松散耦合，并保证每个服务组件在产品不同部门以及不同产品间的可复用性。

### 使用开源构件

系统基础框架主要采用业界的一些主流开源框架，包括：Jquery、Restlet、Velocity、spring、hibernate。

为提供运行级产品及云计算产品建设的需要，系统集成并改造了业界成熟的开源产品，包括：amber 、Memcached、Redis、Mongodb、Solr、Storm、Hadoop、PostgreSQL。

系统引擎层建设主要采用业界的主流开源构件：JBPM、[Quartz](http://www.opensymphony.com/quartz/;jsessionid=LDKHONNCOPJC)等。

Java单元测试使用junit框架。Javascript性能测试工具dynaTrace AJAX 、yahoo YSlow、httpWatch。

### 使用商业构件

报表服务组件采用了公司购置的商用包括控件：硕正报表控件。

## 其它产品依赖

技术平台需要有能力支撑整合公司各事业部的产品建设，并且能够与第三方产品整合的能力。

## 其它设计约束

目标构架总体上应采用分层结构，并全面应用面向对象设计、编程技术使系统具有较好的扩展性与重用性。

本系统支持与其他系统进行集成，所以要提取出良好的集成接口。

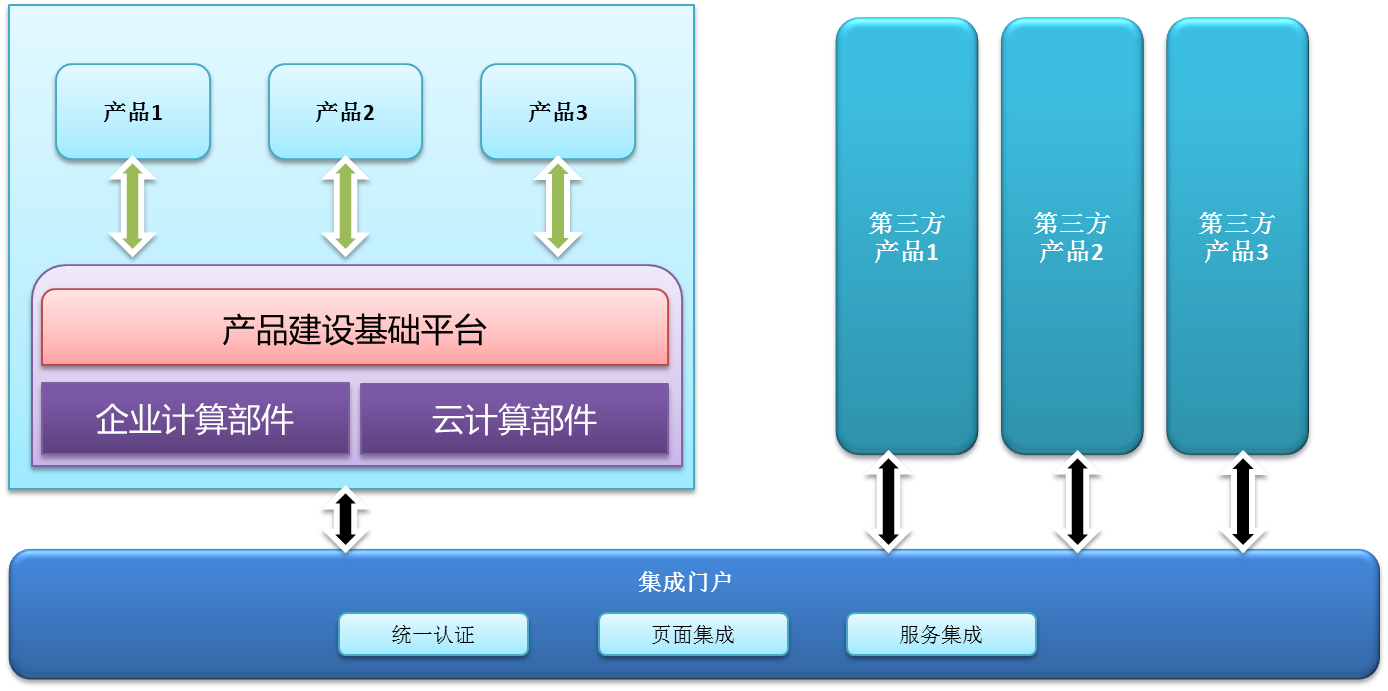
系统为开发人员提供一致、简单的编程模型，屏蔽企业级产品开发、运行级产品开发带来的复杂性，有效降低产品建设、项目实施的成本。

# 平台总体视图

## 概述

平台视图从用户使用的角度描述系统总体构架，描述了平台的模块构成及模块间的层次关系，并描述了模型间的依赖关系及每个模块的使用策略。

## 平台总体框架

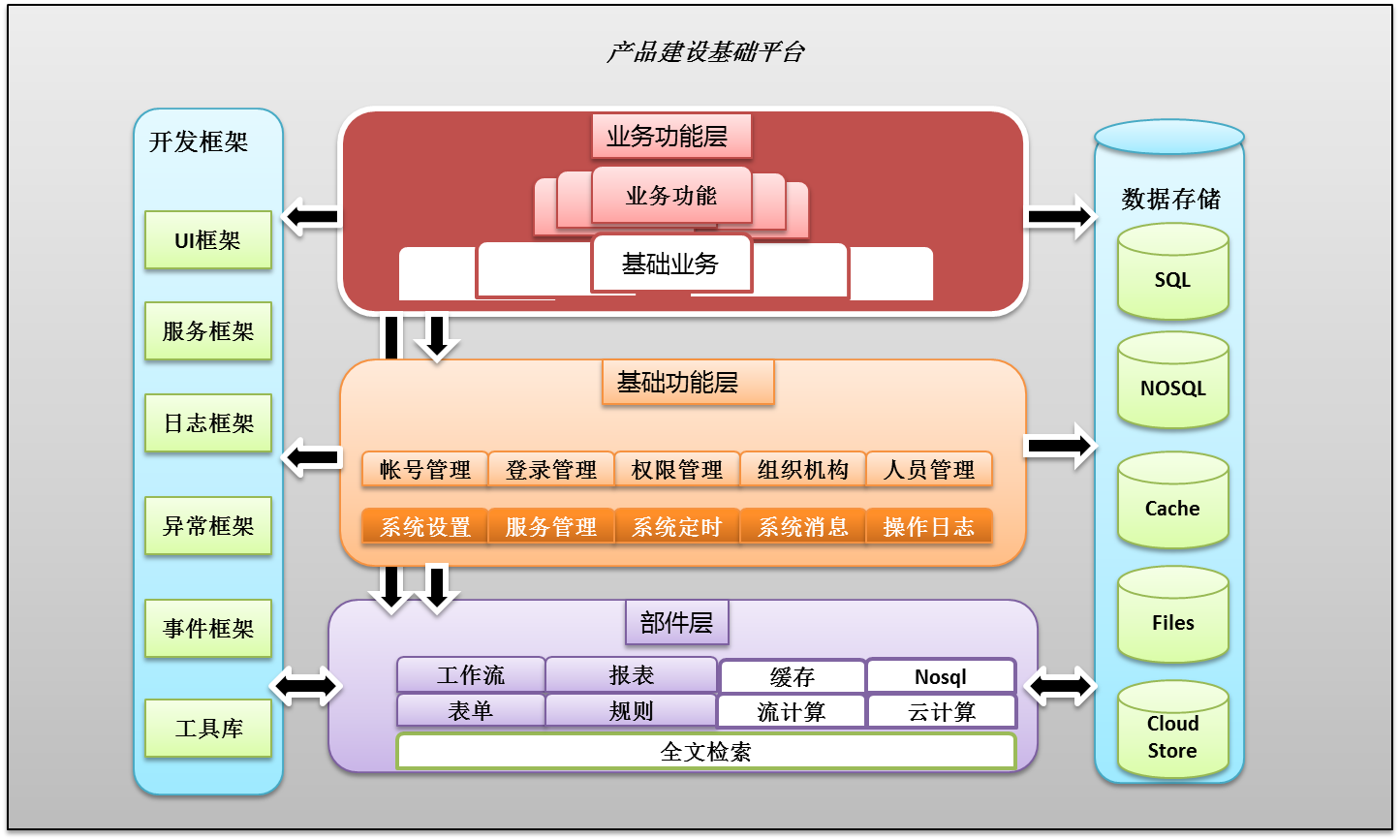


如上图所示，公司软件技术平台为企业计算领域及云计算领域的产品建设，提供了一个分布式的、灵活的、组件化的产品建设平台。其中，基础门户系统本身的定位，不但为公司产品及服务整合提供场所，而且也为第三方产品、服务的整合提供了一个开放的环境。

其中应用集成门户系统主要有统一认证、页面集成、服务集成三个主要部分组成。通过实现不同产品系统间的统一登录、页面集成、功能集成及数据集成，使得不同产品提供的服务能够按需组合为用户提供支持，实现不同产品服务间的按需组合。

## 产品建设基础平台

公司产品的开发基于产品建设基础平台进行，该平台提供一套开发框架，融合了企业计算部件（基础信息库、报表、工作流等）及云计算部件（Redis、Mongodb、Hadoop等）。开发人员基于统一的、简单的开发视图就可以进行不同领域的产品建设，平台本身屏蔽了分布式、云计算方面带来编程的复杂性。



如上图所示，产品建设基础平台由三个层面组成，每个层面的服务组件复用的广度与深度是不同的：部件层为技术重用组件，该层组件的复用率是最高的，左面为企业计算支持部件，右面为云计算支持部件；基础功能层组件包含了信息化系统的基础功能部件，底层为系统级组件，上层为没有领域特征的信息化基础组件；业务功能层的组件服务具备明确的领域特征，为解决特定领域的问题而建设，对于某领域的通用业务逻辑系统将提取成该领域的基础业务组件。

所以三层面的组件的使用与建设，平台提供统一的开发框架。开发框架提供统一的面向资源的开发模式，开发人员利用单一的开发模型进行业务建设，对于生产的产品是面向企业计算还是面向云计算的，对他们来说是透明的，框架本身通过配置改变会话、缓存、存储的等后台模式，保证了两种计算平台的支持。

为了适应各种条件下的系统性能要求（常规、大并发、大数据等），系统的充分利用了各种数据存储方案。这些方案的采用对业务开发人员是透明的，通过开发框架实现各种策略。

# 逻辑视图

## 概述

逻辑视图从系统内在逻辑结构的角度描述系统的基本结构与动态行为，通常包括分析模型（Analysis Model）、设计模型（Design Model）以及数据模型（Data Model）等。

设计模型说明了系统的组成元素、组织架构和关系，并描述了各组成元素的协作以及状态转换关系等（通过用例实现Use Case Realization予以表达）。本节将分别在系统层次结构模型中描述系统的层次组织结构；在主要的包和子系统中说明系统的具体组成；并在架构机制中详述系统中的各种构架机制；最后在关键用例实现中通过描述最重要的用例实现，来说明构架的典型协作（动态行为）。

分析模型对等于设计模型，是在更高的抽象层次上定义系统的结构，作为可选项，本文档将不予说明。

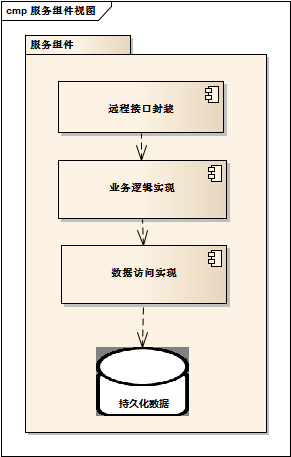
## 服务组件

### 5.2.1服务组件描述

系统根据功能及数据的依赖将划分出不同的服务组件，每个服务组件都是可配置的，且对外提供一类相关的功能服务。这种方式划分出来的服务组件封装相对稳定，很少产生系统运行时的变更。

服务组件的某个功能接口将对外暴露成一个服务。从业务角度来说，随需应变的服务模型为企业提高速度、响应能力和盈利能力创造了新的机会。为了提高灵活性，需要将业务分解为一组离散的流程与服务。如果能够随心所欲地混合和匹配这些服务，那么企业将在市场中赢得巨大的竞争优势。

服务组件定义为：共享相同数据、配置及逻辑实现的服务集合实现构成一个服务组件，它往往对应某个或某类业务实体的业务操作集合。一个服务组件通常由接口封装、业务逻辑实现、 数据访问实现及相应的持久化数据组成，如下图所示：



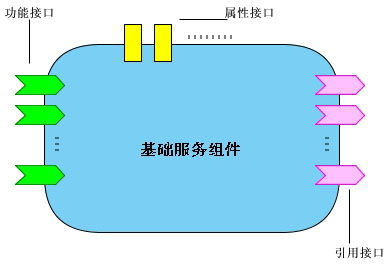
其中每层的具体实现仍然遵循接口隔离的原则，不同层次实现间的交互仍然通过各层实现暴露的接口进行，以此保证服务中的每层实现的可替换性。

其中，上面描述的三个层次实现只有业务逻辑实现是必须具备的，其它两个层次的实现是可选的。只有服务组件需要远程暴露服务接口，才会具备远程暴露封装实现。只有服务组件具备持久化状态时，才具备数据访问层的实现。例如一个复合服务组件的功能是基于其它服务组件的功能协作实现的，此时它本身可能并不具备需要持久化的状态，需要持久化的状态都封装在子服务组件中，由它们分别去维护，此时该复合服务组件就不需要具备对持久化数据的数据访问实现。

### 5.2.2服务组件封装标准

系统将实现为一个层次化系统。系统中服务组件将划分成不同的层次，底层服务组件功能支撑上层服务组件功能的实现，服务组件间将保持松散耦合，服务组件间的通信通过定义良好的接口实现。服务组件定义如下：

* ***基础服务组件***



**服务组件接口定义：**

1. 功能接口：定义了服务组件提供的功能支持，它代表了服务组件的类别。其它软件元素通过调用这类接口获得服务组件的功能服务。
2. 引用接口：服务组件为实现自己的功能，需要调用外部系统提供的功能。引用接口对这些依赖的外部功能进行了定义。这些引用接口的实现者可以是其它服务组件，也可以是系统环境。
3. 属性接口：运行时往往需要查询服务组件的属性以了解服务组件状态，并通过变更属性改变服务组件状态，从而影响服务组件的行为。这些功能都是通过访问服务组件提供的属性接口实现的。

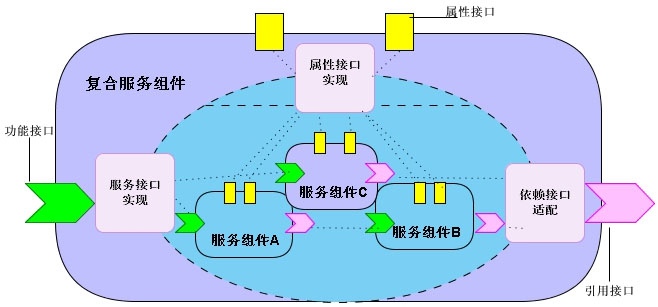
**服务组件交互约束：**外部系统与服务组件的交互，都必须通过严格定义的服务组件接口进行。服务组件接口是外部系统区分服务组件类型的依据，也是服务组件封装的边界。遵循这个原则，只要保持服务组件接口的稳定性，服务组件内部实现的修改对外部系统是不可见的，系统实现时可以针对同一服务组件提供多种实现。需要说明的是，服务组件内部实现中使用的一些接口并不在服务组件接口定义的范围内，原则上它们不应该暴露给服务组件的使用者，使用者也不允许直接使用这些接口。

**数据自治性约束：** 对于依赖于持久化数据的服务组件而言，相同层次中服务组件间的数据访问必须符合数据自治要求。也就是当某个服务组件需要访问另一个服务组件的持久化数据时，它必须通过被访问服务组件的功能接口进行，原则上不允许通过数据访问层实现直接操作相应数据。

**服务组件依赖性管理：**服务组件依赖必须实现松散耦合。系统实现中，系统利用抽象工厂模式，配合Spring的DI机制，实现这个要求，不允许对服务组件的直接实例化。

**服务组件接口实现：**根据需要服务组件接口可以提供两类实现：本地接口与远程接口。其中远程接口包含服务组件功能的远程暴露实现（RESTFUL或SOAP）。调用端可以通过直接远程调用来访问服务组件的远程接口。

* ***服务组件组装***



如上图所示，多个服务组件可以组装成更大的服务组件，称为复合服务组件。参与组装的服务组件可以是基础服务组件，也可以是复合服务组件。

复合服务组件对外接口仍然符合服务组件规范，提供功能接口，引用接口及属性接口。这些接口的实现通常需要调用其内部子服务组件的功能接口。

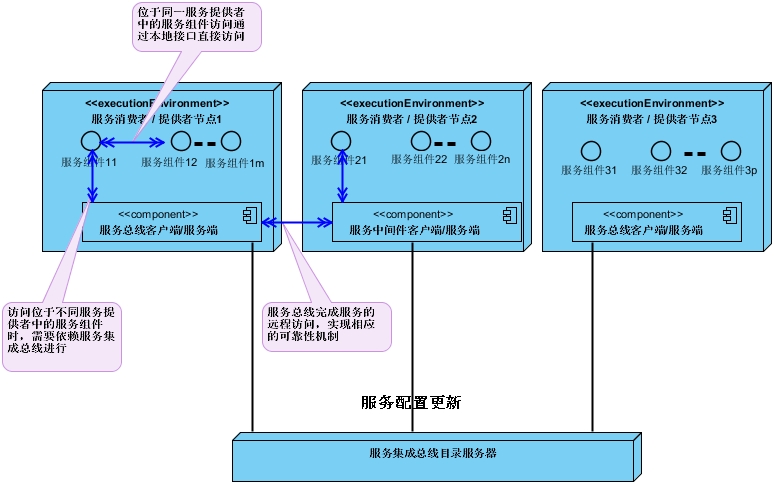
系统服务组件可以通过这种递归的方式组合成不同粒度的服务组件，为系统提供服务。基础服务组件是系统存在原子服务组件，也是功能替换的最小粒度。只要符合服务组件标准定义的规范，原则上各种粒度的服务组件都能够实现替换。

复合服务组件的其它约束与基础服务组件是相同的。

具体的服务组件规范描述将由单独的文档提供，本文不再详述。

### 5.2.3服务组件调用机制

服务组装应该具备很大的灵活性，在部署与运行时都应该能够灵活的定制。本文描述一种服务调用机制，详细的服务编排方案将有单独的文档给出。

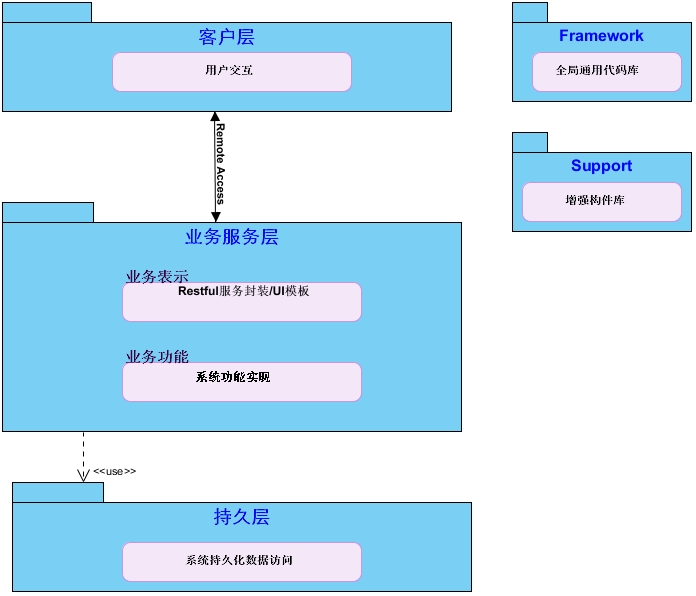


服务调用机制如上图所示：

1. 需要访问某个服务组件的软体所处的部署环境定义为一个服务消费者节点，提供被访问服务组件所处的部署环境定义为一个服务提供者节点。
2. 整个系统会存在多个服务提供者节点，每个服务提供者节点会对应一个产品服务器集群或一个服务器。每个服务提供者节点会提供多个服务组件。
3. 平台会提供服务集成总线，它由目录服务器与客户端/服务端两部分成。服务总线客户端会部署到服务消费者节点中，服务端会部署到服务提供者节点中，它们是以嵌入式的方式部署的。如果一个服务提供者节点同时也是其他服务的消费者节点，则服务总线客户端及服务端都将部署其中。
4. 平台上每个服务组件会向服务集成总线目录服务器注册自己提供的服务。服务集成中间件客户端会向目录服务器查询到相应的服务信息，同时目录服务会向客户端通知服务变更的信息。
5. 当软体需要访问一个目标服务组件提供的服务时，会根据服务提供者节点与服务访问者节点是否相同，而采用不同的访问策略：相同时通过本地接口的直接功能调用进行访问；不同时则依赖服务集成总线进行访问。（具体策略请参考后面关于服务集成的描述）。
6. 服务组件依赖性分为强依赖与弱依赖两种，对于前者访问时不需要进行可访问性的判定，对于后者需要在访问时判断目标服务组件的可访问性（可访问性主要由目标服务组件是否开通来决定，强依赖的服务组件原则上是同时开通的）。
7. 服务集成总线目录服务器会集中管理全局的配置信息及策略，它会向每个客户端发送只与该客户端所需的配置及策略信息。

## 系统层次结构模型

系统主要分成4个逻辑层：客户层、表示层、业务层、持久层。另外还有一个Framework的包作为全局可用的基础设施（包含基础开发框架）。目标系统还使用了其它的增强构件，它们被分在Support包里面。如下图所示：



* 客户层

系统客户层关注点是用户的交换逻辑，主要完成业务数据的展现及采集功能，它主要是在客户端实现。另外，由于客户端与服务器端的信息传递要通过远程访问进行，所以与表示层间通信逻辑也是该层的主要关注点。

* 业务服务层

业务服务层由业务表示与业务功能两部分构件组成。

业务表示构件部署在服务器端，负责接收客户端的客户请求，转换成对业务功能构件调用。同时，负责将业务功能构件的处理结果格式化成客户层可以接受的数据格式，并且利用模板技术生成UI界面，或暴露成Restful的远程接口。

系统业务功能是实现业务功能的地方，它同业务表示层代码将共同部署在服务器端，业务功能层完成业务功能逻辑，处理业务表示层的调用请求，完成事务处理。

* 持久层

业务持久层主要负责系统持久化数据的访问、缓冲、加密等功能。它将实现各种持久化数据的生命周期管理，为业务层提供一个简单、统一的数据访问接口。该层代码将与业务层一起部署在服务器端，业务层以直接功能调用的方式访问持久化层的功能接口。

* framework包

主要包含开发框架、配置文件读取、状态机、通用异常类、特殊类型等全局通用的代码，这个包里面的内容很多并不是专为目标系统而做的，是可以在其它系统中进行重用的。

* Support包

包含系统用到的增强构件，包括Redis、Mongodb等访问构件。

## 主要的设计包和子系统

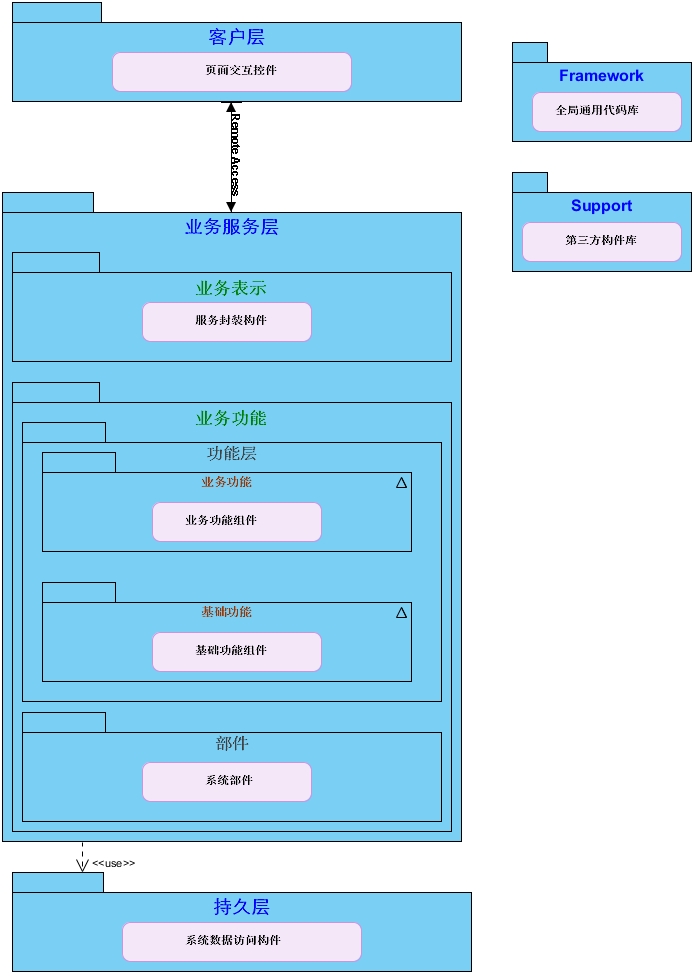
本节描述各层与服务包中具体的设计包与子系统组织结构。

### 子系统划分

公司软件系统主要划分为通信子系统、集成门户子系统及业务子系统。

### 业务子系统

#### 层次说明

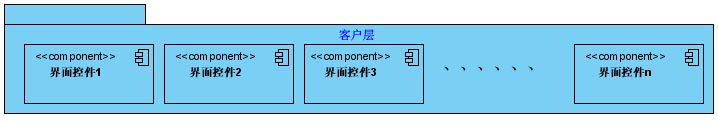


应用子系统实现为由服务组件构成的层次化系统，这可以满足系统对灵活性、扩展性、可重用性及易维护性的要求。其中，业务层根据功能不同划分功能层与部件层两大类，表示层提供RESTFUL远程封装。

部件层服务组件为其上层服务组件提供基本的技术功能支持，上层服务组件通过部件层服务组件暴露的简单、统一接口访问其功能。

功能层包含了系统的业务服务组件，该层根据系统运行的功能依赖性又划分成两个子层次：基础功能层与业务功能层。前者包含所有业务系统都必须存在的服务组件（如组织机构），后者包含的服务组件往往针对某一领域业务（如：数字校园系统、企业一卡通、城市一卡通等），系统可以根据业务需要选择是否启用。

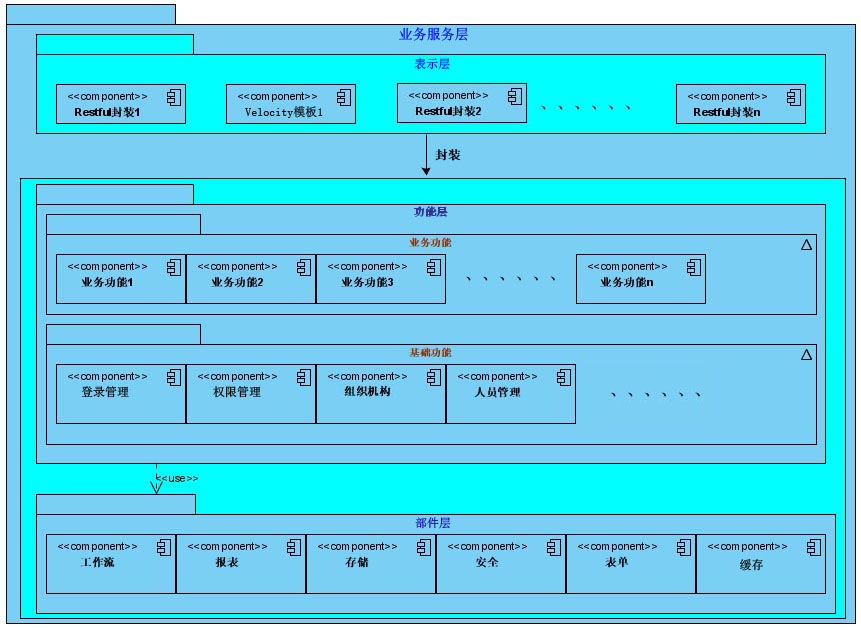
#### 客户层



客户层主要由各种界面交互控件组成。这些控件由客户端执行语言实现（JAVASCRIPT），根据信息展现及交互体验的需求将构建不同种类的界面控件（例如：表格、面板、窗体、树及布局等）。另外需要实现控件之间的通信机制，以及针对服务器的远程访问机制。

由于客户层代码需要通过网络传递到客户端执行，所以必须对代码的尺寸及执行效率进行优化，满足系统的性能要求。

#### 业务服务层



如上图所示，系统业务服务层分成两个层次：表示层、功能层。层次之间的服务构件功能是向下依赖的关系，具体说明如下：

部件层服务组件主要为业务功能实现提供基础的技术支持。其中工作流部件将基于开源工作引擎建立；报表部件将基于硕正报表控件建立一个报表服务组件；安全部件封装数据加解密及数字证书相关的功能，为整个系统提供统一的安全接口；表单引擎为系统提供自定义表单的功能支持。缓存部件融合优秀的分布式缓冲产品，为系统提供高性能的缓存支撑；存储部件融合的云计算产品，提供了数据的路由、缓冲及存储功能。

功能层是用户提供最终的价值的地方，根据用户需求的不同系统功能可以按需提供。但有些服务组件是其它服务组件的基础，也是每个业务应用中都需要的，系统设计中将它们定义为基础功能服务组件。登录管理、权限管理、组织机构、人员管理等基础功能服务组件，将以更加泛化的方式来定义功能接口，最理想的情况是以元模型的方式来建设这些模型，以适应不同客户的要求。功能层的另一部分组件定义为业务功能服务组件，这部分服务组件能够根据用户需要选择性地提供给用户，它们之间的依赖性也将尽可能地松散。

服务组件功能的远程暴露接口采用RESTFUL风格。原则上，一个需要远程暴露的服务组件会对应一个RESTFUL封装。当需要改变或增加远程暴露方式时，能够通过替换（例如替换现有的rest封装包）或增加新的封装包（例如增加新的soap暴露封装包）来实现。Restful服务封装方式利用数据接口的暴露，对于需要人机交互UI的场景，框架利用Velocity模板引擎生成UI界面。

基于服务组件化的系统设计、接口的明确划分将为系统提供良好的可扩展性及可维护性。

#### 持久层



持久层将提供对结构化数据、非结构化数据及系统缓冲的各种访问功能，该层将实现各种的DAO，完成对不同类别数据资源的访问及管理功能。

### XXX子系统

//具体说明待出。

### 集成门户

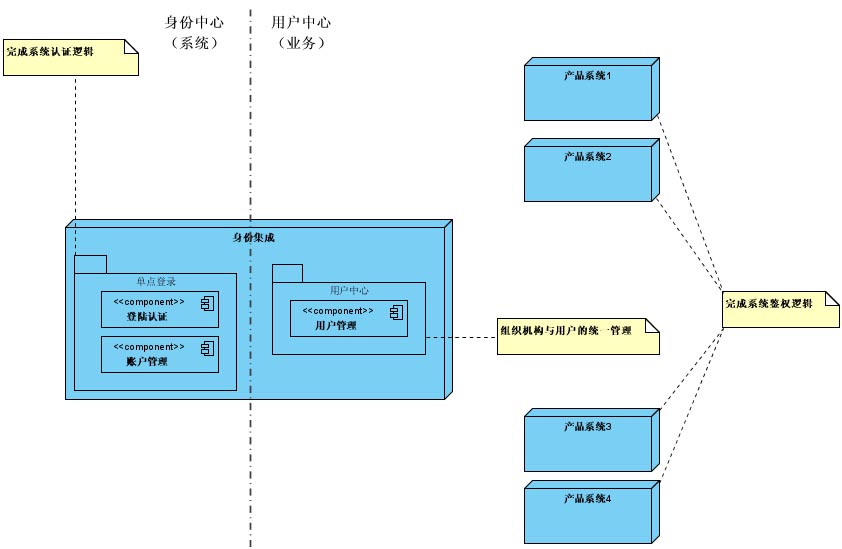
集成门户平台为公司产品及第三方产品提供统一的访问口支持，系统建设的目标是最终能够利用应用集成门户的基础设施，实现公司产品间的，以及与第三方产品间在界面、功能乃至数据层面的集成。使得系统间可以在功能层面互相增强，在展现层面实现一体化体验，在资源方面实现灵活的共享。

如上图所示，应用集成门户包含四大部分的建设：身份集成、页面集成、服务集成，具体描述如下：

* 页面集成：可以实现不同系统提供的功能界面元素在同一界面的集成展现。
* 身份集成：用户使用平台产品提供的服务时，并不关心他使用的服务来自于平台上的那个系统，交换过程中系统间的切换对用户是透明的。而每个运行系统都需要保证自己系统资源的安全，这就需要实现不同系统间安全上下文的平滑传播。统一用户中心完成了不同系统间用户身份的归一化，使得平台上不同系统间的用户身份保持唯一，这也为事务控制等功能的实现奠定了基础。
* 服务集成：页面集成实现了不同产品来源的服务的统一展现，但具体功能实现时需要不同系统在功能及数据层次上能够实现高效的交换。服务集成中间件，将提供一个消息渠道，通过消息驱动的方式实现不同系统间的服务集成。而服务本身，可以是功能服务也可以是数据服务。

集成门户组成包含：有基础服务、规范、嵌入包三个部分。基础服务提供了基础的支撑功能；规范作为系统统一和互通的标准、方法论和要求；嵌入包提供具体认证和服务访问的封装，是集成户提供的工具。

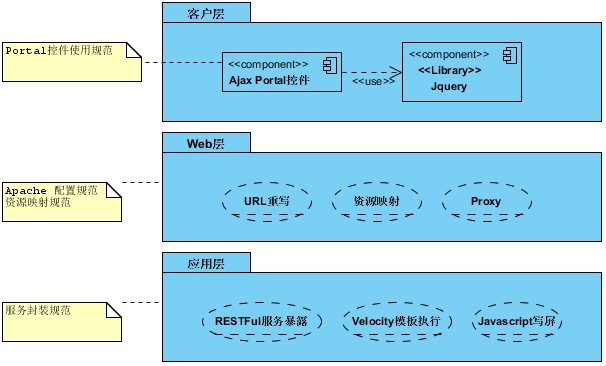
#### 5.4.4.1身份集成



如上图所示，身份集成需要完成两个中心的建设，及身份中心与业务中心。前者主要实现平台上所有产品系统的单点登录，后者主要是实现产品系统间业务层面的用户统一。

单点登录服务器将维护系统中所有的登录账户，并通过SSO机制实现系统间的单点登录，它职位与账户登录相关的信息。需要说明的是，SSO只负责用户登录的认证，具体的鉴权实现是在每个产品系统中完成的，它们之间保持统一的用户ID。

#### 5.4.4.2大并发环境下的界面集成



如上图所示界面集成实现包含三个层次：客户端Ajax控件库及相应使用规范；Web层规则配置规范及资源映射规范；应用层服务封装规范。具体实现机制描述如下：



客户端界面根据需要，将通过Ajax或直接Http请求方式发送对不同产品资源的请求。这些请求，经过Apache服务处理后存在两种处理方式：当请求可以映射到Apache服务器上的静态资源，而且这个资源存在时，将直接返回请求资源；而第一种情况不满足时，请求将被路由到后端的应用服务器来处理，处理结果将被返回到客户端。

页面Mashup方式，要求客户界面集成某产品页面时，客户端发送的产品资源URL需要指向产品的外网地址。这是因为此请求是有客户端界面发出的，它需要经过Apache的映射处理。因此，当产品需要暴露自己的服务或页面时，需要将相应资源的外网Url暴露出来。并且，当某个产品应用需要集成其它产品的资源时，这些资源Url映射将通过读取配置文件的方式来获得，保证了部署环境变化时资源Url的灵活变更。

#### 5.4.4.3服务集成

//待补充

## 架构机制（Architectural Mechanism）

描述系统实现中的主要构架原理，为每个子系统或组件的开发提供指导，并阐述各部分的实现机制，同时讨论相应设计的适应性、局限性、可变性及与其它备选方案等方面内容。

### 集中部署时的组件化机制

组件化是我们重用业务逻辑的基础，而有效重用是快速响应市场，有效降低成本的前提。为了实现上述目标，必须从开发、部署与执行三方面进行分析，保证每个环节的有效性。

对集中部署的组件需要解决的问题描述如下：

* 1. 服务组件的效率问题：组件效率包括开发效率与执行效率两个方面，前者需要开发模型简洁性与代码管理的方便性；后者对组件开发采用的技术策略的合理性。
  2. 组件依赖性问题：服务组件的实现需要保证软件的高内聚低耦合需求，如何处理组件间依赖性，实现组件的可插拔是组件化的关键因素。
  3. 公共依赖模块，库的管理问题：集中部署会遇到的另一个问题是，不同服务自己实现会依赖相同的软件模块与第三方库，而这些模块与库会加载到相同的执行环境中（相同的进程、相同的类加载器），这时会产生两个问题：一个是依赖相同的模块与库但是依赖类库的版本不同；另一个是相同的功能却依赖的不同的模块与库，造成程序执行空间需要初始化更多的对象到内存中，消耗主机资源。
  4. 集中部署组件时的冲突问题：一个完整的服务组件由类文件、库文件、配置文件、页面文件、模板文件、图片、JS脚本、才CSS样式文件等不同类型资源构成。当集中部署时，如果保证组件间的资源不会互相覆盖。当程序加载时，如何保证不同组件间的代码、脚本等资源执行不会产生冲突也是需要考虑的。
  5. 组件的服务的添加、升级与更新问题：组件如何发布，升级与更新，而不影响已部署的服务组件，如何简化实施操作降低实施成本。

针对上述问题的解决机制如下：

1. 基于统一的开发框架，封装通用的功能逻辑；简化数据传递逻辑、调用寻址逻辑；合理分层与MVC模型结合清晰化研发人员分工都可以有效提高服务组件的开发效率。针对远程数据接口的暴露通过Restlet提供的原生接口；针对人机交互界面基于高效的模板引擎实现；业务服务Bean依赖管理基于轻量级的Spring框架实现；ＯＲＭ映射基于Hibernate实现，同时为查询操作提高更直接的执行通道；对通用功能封装（例如分页等），并在框架内部进行持续优化，这些都保证了服务组件的执行效率。
2. 为了实现组件间的弱依赖，首先在服务组件的设计、实现想保证组件的数据自治与功能自治，其次服务组件间的访问以接口调用的方式进行。并且，在软件平台层面提供全局、单例工厂，当服务组件A需要访问服务组件B时，它需要从全局工厂获得组件B的组件工厂，并通过该组件工厂获得组件B中的对象实例。全局工厂返回NULL时，组件A可以判断组件B没有安装并做出相应的处理。这就保证了组件间本地调用情况下功能依赖的弱耦合性，在远程调用情况下，弱耦合性更加容易保障。
3. 对于公共模块、第三方库依赖性问题需要规范与管理并行来实现。首先服务组件实现时的依赖库要符合平台架构的选型要求，服务组件开发者不能随意变革依赖版本，同时平台架构应该尽可能的封装通用逻辑。其次，服务组件在选择个性化依赖库时，需要进行平台架构的兼容性测试。
4. 服务组件的集中部署可以简化部署过程，提高系统运行效率，轻量化实施，对大部分项目比较适合。为了解决不同服务组件集中部署时的冲突问题，需要解决部署与执行两个阶段的问题。在开发视图层面，基于整个公司产品视图隔离不同服务组件的资源，不同资源文件采取目录隔离方式，代码采用包隔离与名字空间隔离的方式。基于这种规范控制，就可以避免服务组件集中部署时的冲突问题。
5. 服务组件目录、名称、代码的规范化与规则化，为组件的部署、更新、升级奠定了自动化的基础。基于这些规则，可以开发公司的组件部署、更新、升级工具。每个组件一个包，包中定义组件的版本，可用来做更新时的比对。另外，可以利用工作队包中的类文件大小进行比对，进行选择性提供。自动化工具将大大降低组件部署、更新的成本。

### 分布式部署时的组件化机制

基于上述规则实现的服务组件，很容易采取分布式部署的方式。唯一需要考虑的就是组件的间访问可能需要通过远程接口进行。此时，需要解决两个问题：远程接口调用的复杂性；服务本身的治理。

前面对组件调用机制有明确描述，服务治理这部分可用依赖于zookeeper这种成熟的开源产品改造实现。

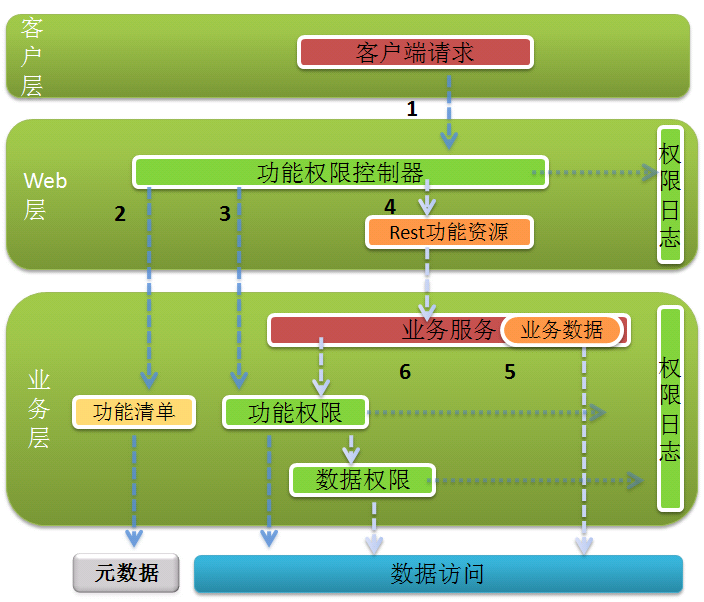
### 持久化机制（Persistence Mechanism）

//持久化机制有待补充

### 安全机制（Security Mechanism）

产品系统的安全机制通过四种途径来实现：访问控制、信道加密、数据加密与审计功能。

#### 5.5.3.1访问控制



客户层：

页面中的可操作元素（按钮、下拉列表、文本框）的操作权限全部来源在应用系统中的权限管理模块，页面通过Ajax方式向系统发出请求，由系统返回当前页面中可操作元素的具体使用权限。

Web层：

WEB层内为功能权限过滤请求。WEB层过滤URI均需要在权限系统资源管理中注册，注册过视为受限URI。不受限的URI请求不会被权限控制所过滤。

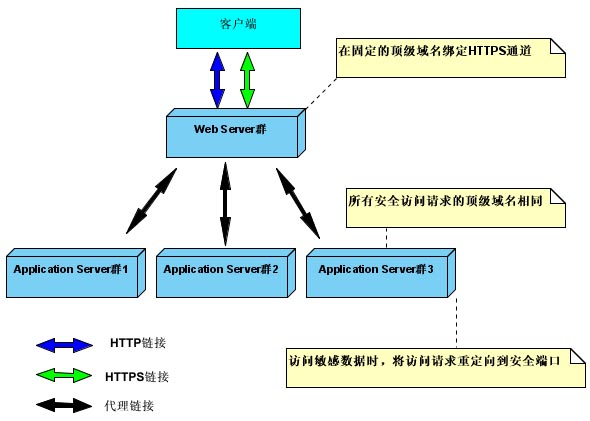
Biz业务层：

这个层次包含为功能这下的数据级权限（对于数据集合行列权限控制）两部分，以实现业务进行过程中对业务数据细粒度的控制。

关于访问控制的具体实现请见系统权限基础组件的设计。

。

#### 5.5.3.2信道加密

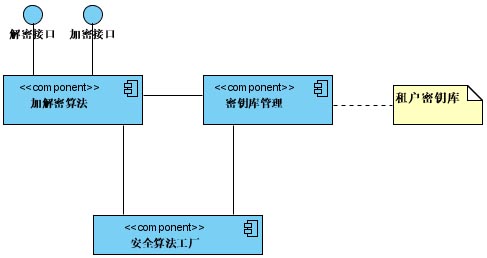


信道加密用来保证用户敏感数据在网上传输时的安全性，系统通过提供HTTPS通道来实现。具体方式如上图所示：

HTTPS通道部署在前端Web Server中，由于加密证书绑定是按域名来收费的，所以实现中所有加密通道的顶级域名是相同的，次级域名则根据租户域名来构造， Web Server利用URL重写策略及代理策略来处理安全请求。

当一个访问敏感数据的HTTP请求达到应用服务器时，应用服务器利用HTTP重定向（setRedirector）方法将请求重定向到安全通道上，安全通道上的请求会执行上述的处理方式。

#### 5.5.3.3数据加密



数加密主要包含三部分：加解密算法模块，密钥库管理模块及算法工厂。

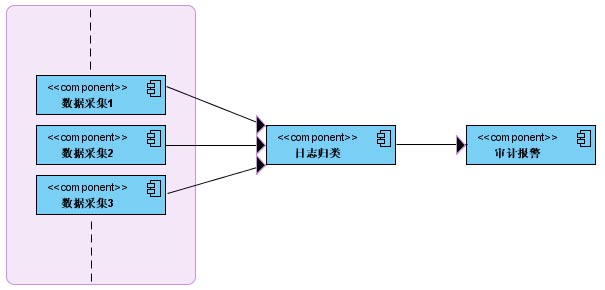
其中算法工厂提供管理功能，主要是对各种算法模块、密钥库管理模块的生命周期进行管理，根据用户请求构造不同的算法模块。

密钥库管理模块负责租户密钥的生成与维护。

算法模型是最终提供数据加密及数据解密功能的模块。

具体应用中，会根据业务需求对租户资源进行安全级别的划分。只对安全级别达到一定程度的数据，实施相应的加密，以保证系统性能。

#### 5.5.3.4审计功能



系统审计实现主要包括数据采集、日志归类、日志的审计与报警等几大基本功能。

数据采集模块，即日志的采集部件。为了实现日志记录的多层次化，需要记录网络、系统、应用和用户等各种行为来满足系统不同层次行为审计的需求，所以在产品系统中设置了多个数据捕获点。

日志归类模块主要是为了简化审计时的工作量而设计的，它的主要功能是根据日志记录行为的不同层次来进行分类，将其归为网络行为、系统行为、应用行为、用户行为中的一种，同时进行时间归一化。进行日志分类目的是对海量信息进行区分，以提高日志审计时的分析效率。

日志审计与报警模块侧重对日志信息的事后分析。该模块的主要功能是对日志信息进行审计分析，即将收到的日志信息通过特定的策略进行对比，以检测出不合规则的异常事件。随着审计过程的进行，若该异常事件的可疑度不断增加以致超过某一阈值时，系统产生报警信息。该模块包括日志信息的接收、规则库的生成、日志数据的预处理、日志审计等几个功能。

系统日志机制主要使用log4j框架来实现，仅仅是简单的直接应用，用于记录上述各种层次的信息，

### 界面个性化机制（View Customize Mechanism）

//有待补充

### 数据个性化机制（Data Customize Mechanism）

//有待补充

### 功能个性化机制（Function Customize Mechanism）

//有待补充

### 高可用机制

软件系统的高可用性除了软件本身设计与实现的容错能力，主要是指软件运行环境的高可用性，此处是指系统集群的高可用。

系统集群高可用是一个整体的解决方案，包括网络高可用、主机高可用、应用服务高可用、数据库高可用，存储高可用等，任何一个节点都要考虑到故障时可自动切换功能，否则任何一个单点发生故障，都将导致整个集群不可用。



系统部署结构如上图所示：Web服务器层为Apache服务器集群，应用服务器层为Jboss服务器集群，数据库层为Oracle服务器的RAC集群，另外将部署Redis服务器提供分布式缓冲。其中每一台Apache服务器，Jboss服务器，Oracle服务器分别位于不同的物理机上。一台物理机发生故障，另一台物理机上的应用服务器可以接管服务，保证了主机和应用服务的高可用问题。Oracle数据库服务器也位于不同的物理机上，在数据库访问层面通过Oracle的RAC保证了数据访问的高性能，通过系统级别的HA方案，保证了单点故障时的访问切换的，从而实现数据库端的高可用功能。

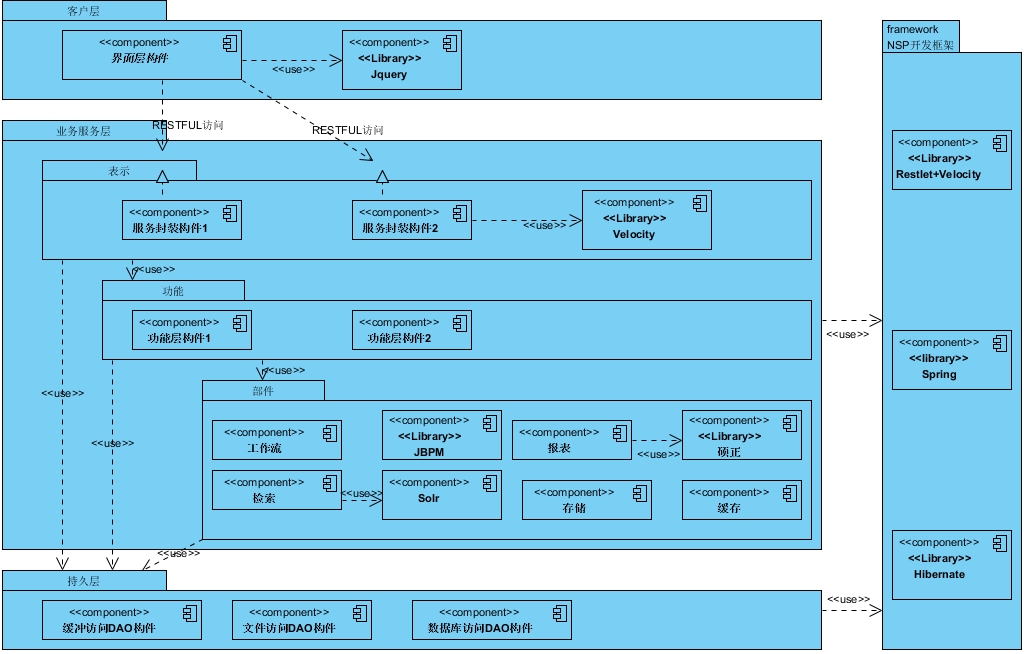
对网络高可用描述如下：如果两台物理服务器是在同一机柜，连接的是同一台交换机，在这种情况下，交换机可能成为单一故障点，如果把物理机分放到不同的机柜上，使两台互备的物理机分别连接不同的交换机，这样，一台交换机出故障，整个集群还能提供服务。该方案需要从部署与运行成本考虑，以确定是否采用。

# 实现视图

## 概述

实现视图从软件编译与构建的角度，描述系统实施构件的组织结构与依赖关系（主要是编译依赖）。模型包括实施子系统和构件结构，及其依赖关系。同时还表达了逻辑视图中各个包和类分配到实施视图中的子系统和构件的映射关系。

## 系统实现模型总体结构



如上图所示，公司软件系统架构实现将以公司NSP开发框架为基础。NSP开发框架封装Spring、Hibernate、Velocity及Restlet框架，并封装了通用的数据处理及功能处理逻辑。它依赖Spring实现依赖注入，依赖Hibernate实现O/R映射，依赖于Restlet实现Restful服务接口封装，依赖于Velocity完成UI人机界面实现，为NSP系统框架实现提供了很好的基础设施。

其中持久层将根据访问资源类别不同分为缓冲访问DAO、文件访问DAO及数据库访问DAO三类，而其中的数据库访问DAO将通过Hibernate实现O/R映射。

业务服务层与持久层的依赖关系通过Spring提供的DI机制来管理，两层服务构件间的协作是通过直接功能调用的方式进行的。业务层包含三个子层次，它们之间存在相同的依赖管理策略及调用方式，具体描述如下。

部件层中，工作流部件将依托开源的工作流引擎JBPM来建立。报表部件依赖于硕正报表控件来建设。检索部件依赖于开源全文检索引擎Solr来建设。通信引擎依赖Spring提供的各种通信基础设施来建设。缓冲部件提供Memcached、Redis等实现分布式缓冲。存储部件提供对Mongodb、Hadoop等Nosql、云计算部件的支持。

功能层各种服务组件的业务逻辑实现为一组Spring管理的Bean。它们封装系统的业务逻辑，依托Spring实现生命周期管理，并利用容器提供的AOP实现相应事务、安全及日志等系统功能。

表示层，将依托Restlet框架来实现系统功能的Restful资源封装。该层实现的是系统的功能的Restful暴露，而具体功能则是由相应服务的功能层构件实现的，设计方式将依据面向资源的理念进行。开源框架Restlet为面向资源设计及Restful服务封装都提供了成熟的基础设施支持。Restful服务封装方式比较适合数据服务接口，对于人机交互的UI实现，框架提供了基于Velocity模板引擎的实现，Velocity模板引擎会在内存缓冲编译后的AST，所以其执行效率高于JSP等传统实现方式。

客户层构件将运行在客户端，它将以客户端执行语言实现（Javascript）。表示层主要实现各种数据展现及用户交互控件，它们以Ajax的方式访问服务器端的Restful资源服务，并完成数据展现与数据采集的功能。完成通信的数据协议以JSON为主，对部分复杂数据则以XML形势提供。

## 开发框架的特点

**1、采用成熟、轻量级开源框架构建，采用业界先进的编程模式与技术**

开发框架整体思路采用了面向资源的Restful编程模式，这种编程模型具备结构清晰、符合标准、易于理解、扩展方便的特点，代表了互联网时代的发展方向。

基于资源的请求映射更加直观，开发人员编程时的请求URI字符编码与服务器端的服务方法一对一映射，可以有效简化编程负担。而且这种直观表示使程序员理解、跟踪、处理程序问题时，可以更快地理解程序。Restful编程模式本身就保证了程序本身具备良好的伸缩性，面对规模性问题可以很容易部署成集群模式、云计算模式。当前主流的软件企业都已经转向或正在转向这种编程模式。

开发框架选择了成熟、优秀、轻量的开源框架来构建：采用Restlet支持请求处理；采用Velocity支持UI界面生成；采用Spring支持业务Bean的管理；采用Hibernate完成OR映射；采用Jquery支持客户端组件实现。采用这些成熟框架可以使得我们利用软件领域的优秀成果，轻量级保证我们可以按需使用，开源保证了我们对程序的掌控力。

**2、既适应企业级应用开发，也适应运营级应用的开发**

企业级开发关注的主要是业务实现，当遇到规模性问题时(高并发、大量数据等)，往往采用传统的集群（应用集群、数据库集群等）即可以解决。但是对运营级应用而言，需要面对超大规模性问题（超大并发、海量数据等），此时由于传统技术限制了系统I/O、存储与主机计算能力的发挥，需要采用Nosql、云存储、云计算等技术来解决。

NSP开发框架，封装了会话管理等缓冲操作接口，基于标准数据访问对象（DAO）完成持久化操作。并且提供了两部分的技术部件：面向企业计算的部件（例如：报表等）与面向运营级部件（例如：Redis、Mongodb、Hadoop等）。通过配置改变缓冲与持久化策略后台，就可以再两类部件中选择，来适应企业级与运营级系统建设的要求。

**3、即适应机机接口的开发，也适应人机UI界面的开发**

Restful架构模式更适合封装应用间交换的远程接口，关注点在数据表示与消息交换，所以Restlet本身最适合的工作是封装机机接口，它可以更高效地完成这类工作。

当访问服务的是人时，系统需要提供友好的UI界面。此时，开发框架采用了高效的模板引擎Velocity，它处理UI部件、样式、交换机数据格式化更加有效，而且模板本身的缓存功能也使执行效率大大提高。

以上两种方式，使得开发框架不仅适合接口编程，而且也适合UI编程。

**4、通用处理逻辑的封装，简化开发者编程**

请求路由、对象与Json串的转换、翻页等通用逻辑都封装在框架中，开发人员集成开发框架提供的各层基类后，可以方便地使用这些功能。

系统利用通用的对象数据表示方式（HashMap），简化了不同层次间数据传递中的对象转换与格式化操作，只有在必要的时候（例如：持久化前）进行对象转换，以完成数据校验等其它操作。这种方式不但大大简化了编程模型、提高了效率，而且也为开发工具的建设奠定了良好的基础。

**5、自带丰富的工具类，简化开发者编程**

字符编码转换、日期处理、对象串行化与反串行化、UUID生成、HTTP访问、文件上传下载等等这些业务开发人员经常需要工具类，在开发框架中都进行了封装。开发人员可以直接使用，简化了编程，同时也规范化了系统建设过程，避免统一功能不同研发人员采用不同的第三方库区实现。根据使用情况，框架能够不断积累这样的通用工具类，实现公司的技术积累。

**6、具备良好的异常处理与日志处理体系**

开发框架提供了系统级异常与业务级异常的封装，而且提供了通用的异常提示处理资源，开发人员根据规范抛出相应异常，客户端就可以出现友好提示。

日志处理框架提供了灵活的配置模型，可以根据需要配置日志的采集点、存储目标、存储格式等，这部分为系统日志与业务日志的采集、存储提供了强大的支持。（日志部分还没有建设完成）

## 系统WebRoot结构

WebRoot目录结构针对架构研发人员、产品研发人员及集成平台研发人员会有一定的差别，其中产品研发人员的目录结构与产品生产期结构是相同的，分别说明如下：

**基础架构开发视图：**



(服务器端代码库)

（UI基础框架）

(Web图片资源)

(Web脚本资源)

（UI基础组件）

（样式资源）

（UI模板资源）

如上图所示， images目录存放系统图片资源，logs目录存系统执行中形成的日志问题，scripts目录存放前端脚本资源，styles目录存放系统样式描述文件。

其中scripts目录中，base子目录放置UI框架代码，component子目录放置UI基础组件代码，lib子目录放置依赖的第三方JS库。

WEB-INF目录下的classes目标存放编译后的类文件，lib目录存储编译后的程序库与第三方库文件（jar包）。其中的pages目录中，存放前端展示的Velocity模板文件。

组件化对项目型公司的产品基本要求，为了实现组件的高内聚低耦合，同时支持组件间的集中部署与分布式部署，整个系统基于递归的目录规则隔离不同组件间在资源。如前所述，构成一个组件的资源是多层次的，包括图片、客户端脚本（JS、Flash等）、样式（CSS等）、UI模板（Velocity模块文件等）等多方面，同时它们共同依赖NSP框架提供的公共资源，所以NSP框架中的目录规则也是在不同的层次被复用，具体说明如下：

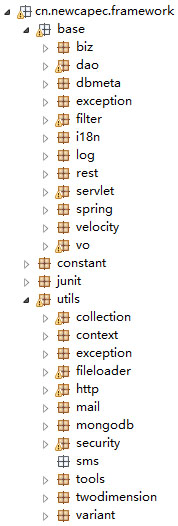
客户端资源目录（JS、Css等）：foundation目录作为基础服务组件资源的根目录，其中的每个子目录代表一个组件资源，目录名称以组件的英文名称标识；function目录作为业务服务组件资源的根目录，其中每个子目录代表一个组件资源，目录名称以组件的英文名称标识；lib目录包含了框架依赖的第三方库，所有服务组件实现都可以对其产生依赖；其它的同级目录可以根据资源类型不同进行扩展定义，但是这些目录中资源都是公共资源，不隶属于某个组件所有。

服务器端资源目录（class、jar包、Velocity UI模板）：class与Jar的隔离策略将在下一节介绍，这里对UI模块的隔离策略进行描述。其中目录定义与客户端定义完全相同，而common目录中存放了NSP框架的公共模板。

## 系统源码包及目录结构

### 6.5.1整体包结构

**基础架构开发包：**



系统建设的整体包结构如上图所示，整个系统的顶级包为：cn.newcapec

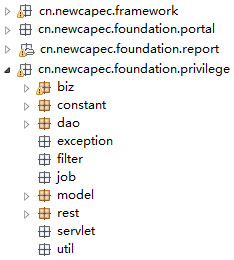
framework中包含基础架构代码，包含在base包中。

framework中包含系统常量定义，包含在constant包中。

framework中包含系统开发的测试基类,包含在junit包中。

framework中包含系统提供的工具类，包含在untils包中。

**基础功能开发包：**



系统建设的整体包结构如上图所示，整个系统的顶级包为：cn.newcapec.foundation

每个组件定义一个jar包，jar包的前缀相同，后面以组件描述的英文拼写区分。每个组件中实现中的子包格式符合开发框架的层次定义，例如：biz代表业务层实现、dao代表数据访问层实现、rest代表rest层实现等。

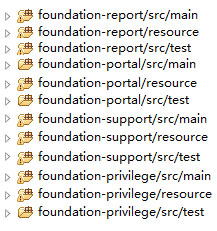
上面描述的系统的整体包结构划分，具体研发过程会有不同的小组成员分别负责不同模块及组件的研发。所以，整体研发的目录结构需要实现研发与部署的模块化，具体目录结构描述如下。

### 6.5.2整体目录结构

**基础开发框架层：**



**基础功能层：**



**公共配置、通用功能：**



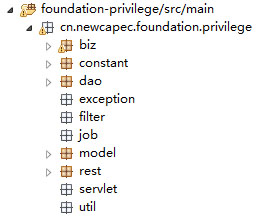
在以上目录结构中，每个应用程序包包含两个源代码树和一个资源包，两个源代码树分别是Java应用[程序](http://www.miiceic.org.cn/phrase/200604232224305.html)的源代码和单元测试源代码，资源包用来管理本模块的配置文件。

**业务功能层：**



在以上目录结构中，每个应用程序包包含两个源代码树和一个资源包，两个源代码树分别是Java应用[程序](http://www.miiceic.org.cn/phrase/200604232224305.html)的源代码和单元测试源代码，资源包用来管理本模块的配置文件。

### 6.5.3服务组件包目录结构



服务组件包目录结构规范了服务组件逻辑实现的物理拓扑，编译研发人员对代码的分析与继承。

1、biz包

该包中放置业务功能的定义及实现的代码，一般情况将某类业务操作提取为一组接口，例如考勤管理业务，然后在biz包下生成一个impl包，在impl包中才放置操作接口的实现类。

2、constant包

业务服务的常量定义类，该包中的常量是针对某个业务服务有效的常量。

3、dao包

该包放置各类dao(data access object)，也就是放置对数据库及文件访问的接口和实现类。

4、exceoption包

业务服务异常类，针对该业务服务的异常类定义的场所。

5、filter包

业务服务过滤器类，针对该业务服务的过滤器定义的场所。

6、job包

业务服务定时器类，针对该业务服务的定时器类实现场所。

7、model包

业务服务中业务实体类定义的场所，定义业务服务中的数据模型。

8、rest包（RESTful Resource）

该包中放置对biz包中业务功能的面向RESTful方式进行服务暴露的包装，此包中不应包含具体业务逻辑运算。

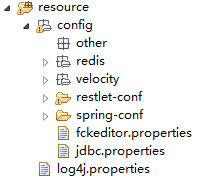
9、servlet包

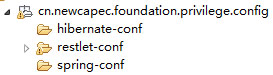
业务服务中Servlet类定义的场所；

10、util包

业务服务中工具类定义的场所；

### 6.5.4配置文件目录结构





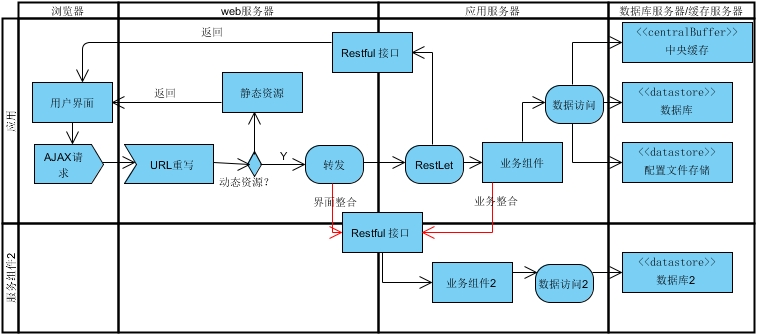
resource目录管理公共的配置文件，每个应用程序包中也有一个resource目录用来管理本模块的配置文件，这样在开发的过程中各模块的程序代码和配置文件是相对独立的，互不影响。同时所有配置文件的目录结构是一样的，可以做到合并和拆分的便捷性。

# 进程视图

## 概述

进程视图从系统运行时刻的角度，描述系统划分为进程、线程的结构，及其动态关系。模型主要说明进程、线程的分类，系统构架敏感的主要边界类、控制类对象等在进程、线程中的分布，以及它们之间的创建、交互与消息通讯关系等。

## 大并发集群部署时的请求处理



系统请求处理过程如上图所示。

首先浏览器以Ajax方式发送用户请求。请求首先到达系统的Web服务器（Apache服务器），收到请求后Web服务器利用预设的规则对请求的URL进行重写。重写后的URL将与Web服务器上的静态资源进行匹配（重写URL与服务器静态资源的目录结构进行匹配），如果匹配成功，将直接返回静态资源给客户端。

如果不存在匹配的静态资源，Web服务器将转发请求给后台的应用服务器。应用服务器分为两大类：应用服务器及外部应用服务器。前者将部署应用系统，而后者为第三方产品，用来与系统集成以增强系统功能。集成规范中，将要求两者的服务都以Restful方式暴露。

请求分发到系统时，以Restlet实现的Restful封装层将处理请求，并调用相应的业务功能层来完成请求，而功能层逻辑的执行则需要依赖持久层的数据访问逻辑来实现数据访问。当请求依赖分布式部署的第二个服务组件来完成时，系统将调用该服务组件的RESTFUL远程服务处理请求。最终的处理结果也需要通过Restful封装暴露给客户端。

对于外部应用的请求处理，也要求存在Restful的服务封装规范，但是并不强制规定外部应用的实现机制。

## 运行统一认证

//有待补充

## 服务集成

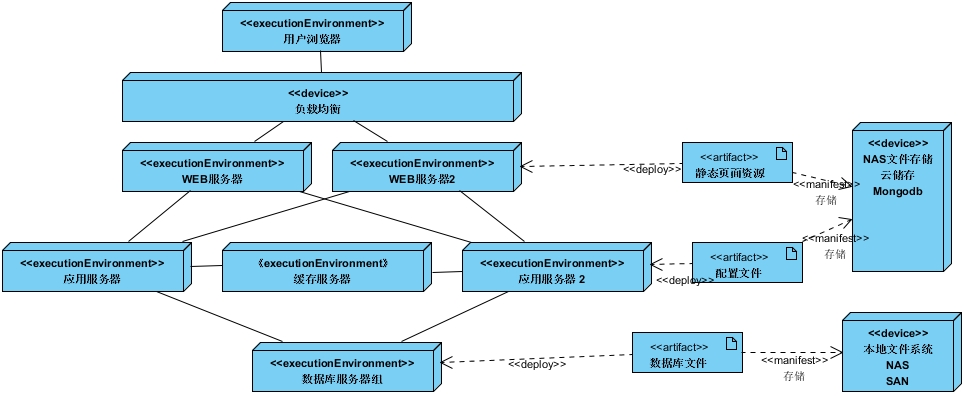
//有待补充

# 部署视图

## 概述

部署视图从系统软硬件物理配置的角度，描述系统的网络逻辑拓扑结构。模型包括各个物理节点的硬件与软件配置，网络的逻辑拓扑结构，节点间的交互与通讯关系等。同时还表达了进程视图中的各个进程具体分配到物理节点的映射关系。

## 大并发应用服务器部署



如上图所示，系统服务器集群分为Web服务器集群、应用服务器集群及数库服务器集群。用户请求经过负载均衡设备分发到不同的Web服务器，Web服务器可以根据应用服务器的负责情况，将请求分派给不同的应用服务器来处理。它们的文件存储空间由统一的NAS\云存储\Mongodb提供（根据性能需求进行具体方案设计），结构化数据存储由统一的数据库集群提供，数据库的数据存储由相应的本地文件系统\NAS\SAN提供（根据数据量、性能需求进行具体方案设计）。

### 8.2.1应用服务器性能设计

1. 利用web服务器分流静态请求，减轻应用服务器压力：对于静态资源，例如图片，脚本，样式表，HTML页面（包括原始页面和静态化以后的）等，被访问时不需要经过应用服务器的运算，只需从文件中读出起内容即可。我们利用性能较高的web服务器（我们选用Apache2），对这些内容提供服务，这样将只有很少的“动态请求”才必须由应用服务器接收并处理。我们在web服务器中利用mod\_proxy模块将这些动态请求转发到应用服务器中。
2. web服务器的高可靠：利用LoadBalancer，在多个web服务器间建立负载均衡集群。此集群中各节点互为镜像，任何一个节点失败时，LoadBalancer可以将请求分发到其他节点。由于我们选用的web服务器Apache本身具有较好的高可用性，可以将此集群减少到一个节点，蜕化成单一服务器。
3. 应用服务器的高可靠：利用web服务器（Apache2）中的mod\_jk模块，在多个应用服务器件建立负载均衡集群。由于应用服务器上通常执行一些相对复杂的运算等，稳定性相对较差，为了保证可靠性，最少不能低于两台服务器。
4. 在应用服务器集群中采用Session stick策略而不采用session复制策略：在服务器工作正常的情况下，同一个session的请求会分配到同一个服务器去处理。而当某台服务器故障时，认为原session已失效，请求会被分发到其他的服务器后建立新的session。避免session复制带来的性能开销。
5. 缓存服务：采用集中式或分布式缓存对数据进行缓存，并使之于数据库中的内容保持一致，减少为取得同一项数据而对数据库的访问。｛集中式缓存需要额外的服务器设备，不占用应用服务器的内存，应用服务器需通过网络存取缓存，不需要缓存数据同步；分布式缓存嵌入在应用服务器内容，占用应用服务器的缓存，应用服务器直接在进程内存取缓存，各缓存节点间必须保持缓存数据的一致｝

### 8.2.2应用服务器可靠性设计

1. web服务器的高可靠：利用LoadBalancer，在多个web服务器间建立负载均衡集群。此集群中各节点互为镜像，任何一个节点失败时，LoadBalancer可以将请求分发到其他节点。
2. 应用服务器的高可靠：利用web服务器（Apache2）中的mod\_jk模块，在多个应用服务器件建立负载均衡集群。由于应用服务器上通常执行一些相对复杂的运算等，稳定性相对较差，为了保证可靠性，最少不能低于两台服务器。

### 8.2.3应用服务器伸缩性设计

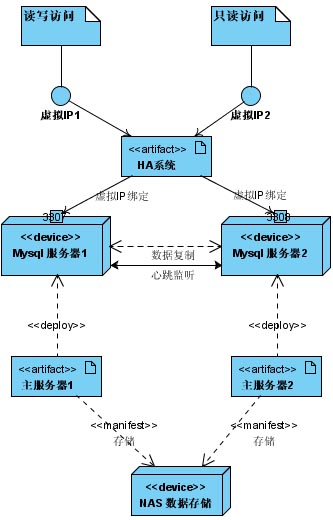
1. web服务器的可伸缩：利用LoadBalancer，在多个web服务器间建立负载均衡集群。此集群可以根据运行期间具体的访问负载增减web服务器的个数，最小可减少到一台，蜕化成单一服务器。
2. 应用服务器的可伸缩：利用web服务器（Apache2）中的mod\_jk模块，在多个应用服务器件建立负载均衡集群。此集群可以根据运行期间具体的访问负载增减应用服务器的个数，但为了保证可靠性，最少不能低于两台应用服务器。

### 8.2.4应用服务器部署对设计约束

为了配合负载均衡及缓存的应用， 在设计上需要注意以下几点:

1. 避免在session中保存过多的信息：需要保存的话， 也尽量保存ID， 不要把整个实例都放进去。 这样可以有效避免内存泄漏， 也便于出现Failover时，在新节点上快速恢复原session中的内容。
2. 数据修改的有效性：这个问题对于单个服务器的部署也存在， 对同样的数据修改的时候， 要保证多个线程或者多个进程数据的一致性和有效性。
3. 缓存数据的一致性：如果有缓存数据的话， 必须使用分布式缓存， 保证多个服务器上缓存的数据是一致的。

## 数据库服务器部署



数据库采用MYSQL数据库，运用数据库的复制技术来实现数据库的高性能，利用系统级HA架构实现数据库的高可用。每个应用群集中将部署两个数据库的主服务器，它们都可以接受读操作与写操作。每个数据库服务器部署会应用一台虚拟服务器，两个虚拟服务器会位于两个不同的主机中。数据库的数据库文件将放置在NAS存储设备中。

在同一时间，只能有一台数据库有写服务器，然后可以把数据同步到另外一台服务器上，保持两台服务器之间的同步，另一台服务器如果状态正常，那么可以承担读事务操作。

在这种基于HA的架构模式中，将利用heartbeat来直接监控主机、网卡及mysql服务。heartbeat将为每台服务器生成一个虚拟IP地址（VIP），Jboss不再直接连接mysql服务器的物理IP地址，而是连接每台主机上的虚拟IP，一旦某台mysql服务器DOWN机或者mysql服务终止，该台上的VIP将会漂到正常工作的另一台服务器上，此时，这台服务器将同时承担读写功能，实现完全的自动故障切换功能。等到发生故障的主机修复后，再手工启动修复机上的Heartbeat， 把该主机上的虚拟IP切换回来，这样又可以切换回两台服务器都同时承担负载的正常状态。

### 8.3.1数据服务器性能设计

1. 按客户拆分：为了让数据库能够应对数据量和访问量的持续增长，拆分数据库已经几乎是一个必然趋势。在不少大型B/S应用的架构中都可以见到，不论使用是Oracle还是MySQL。系统也采用同样需要拆分数据库。拆分数据库的目的是要达到scale out（增加硬件设备，比如服务器数量）而非scale up（提高硬件配置，比如使用有更多CPU和更大内存的机器）
2. 读写分流：每组服务器中同一时间只有一台是可写的，其他全是只读的，并利用Mysql的复制技术从主服务器向从服务器同步数据。
3. 选用不同的存储引擎：根据具体情况选用不同的存储引擎
4. 需要支持事务的表使用InnoDB
5. 需要大量UPDATE的表使用InnoDB
6. 不需要支持事务并且没有大量UPDATE的表使用MyISAM（比如广告、新闻都可以考虑使用）
7. 按功能拆分：保存历史数据的数据库，如果需要保存较大数据量的历史数据，并且提供给用户查询，那么可能需要独立出保存历史数据的数据库。

### 8.3.2数据服务器可靠性设计

1. 集群容错：集群中的任一台服务器发生故障后，故障服务器上的负载自动切换到另一台正常服务器上，待故障机修复后，需要把服务切换回正常状态。由于mysql已经配置成了双向复制，所以数据不再需要进行完全手工同步，重启故障机的mysql服务后，数据会自动同步回来。

### 8.3.3数据服务器可伸缩性设计

1. 按客户拆分：如果写操作的性能超过了现有主（写）服务器的能力，可以很方便地增加一组新服务器，而不会影响客户数据的完整性。