**NSP组件化**

|  |  |  |
| --- | --- | --- |
| 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** |
| 2013-10-8 | 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 4](#_Toc370110635)

[1.1 目的 4](#_Toc370110636)

[1.2 范围 5](#_Toc370110637)

[1.3 术语词汇表 5](#_Toc370110638)

[1.4 参考资料 6](#_Toc370110639)

[2 组件化思路 6](#_Toc370110640)

[2.1 重用性思路 7](#_Toc370110641)

[2.1.1重用模块的粒度划分 7](#_Toc370110642)

[2.1.2组件重用策略 9](#_Toc370110643)

[2.1.3构件重用策略 10](#_Toc370110644)

[2.1.4定制化策略 11](#_Toc370110645)

[2.1.5更细粒度的重用策略 11](#_Toc370110646)

[2.2 建设规划 11](#_Toc370110647)

[2.2.1先融合（第一步） 12](#_Toc370110648)

[2.2.2再统一（第二步） 12](#_Toc370110649)

[2.2.3后平台 13](#_Toc370110650)

[3 组件化目标与约束 13](#_Toc370110651)

[3.1 组件化目标 14](#_Toc370110652)

[3.2 组件化的约束 14](#_Toc370110653)

[3.3 开发策略 15](#_Toc370110654)

[3.3.1 软件复用策略 15](#_Toc370110655)

[3.3.2 使用开源构件 15](#_Toc370110656)

[3.3.3 使用商业构件 16](#_Toc370110657)

[3.4 其它产品依赖 16](#_Toc370110658)

[3.5 其它设计约束 16](#_Toc370110659)

[4 平台总体视图 16](#_Toc370110660)

[4.1 概述 16](#_Toc370110661)

[4.2 平台总体框架 17](#_Toc370110662)

[4.3 产品建设基础平台 17](#_Toc370110663)

[5 逻辑视图 19](#_Toc370110664)

[5.1 概述 19](#_Toc370110665)

[5.2 服务组件 20](#_Toc370110666)

[5.2.1服务组件描述 20](#_Toc370110667)

[5.2.2服务组件封装标准 21](#_Toc370110668)

[5.2.3服务组件调用机制 24](#_Toc370110669)

[5.2.3组件的划分 26](#_Toc370110670)

[5.3 系统层次结构模型 29](#_Toc370110671)

[5.4 组件包与构件包说明 30](#_Toc370110672)

[5.4.1 打包粒度划分 31](#_Toc370110673)

[5.4.2 打包资源说明 32](#_Toc370110674)

[5.4.3 可重用条件 37](#_Toc370110675)

[5.5 组件化、构件化机制 39](#_Toc370110676)

[5.5.1 组件化的版本管理 39](#_Toc370110677)

[5.5.2 组件重用机制 39](#_Toc370110678)

[5.5.3 构件版本管理 40](#_Toc370110679)

[5.5.4 构件重用机制 44](#_Toc370110680)

[5.5.5 组件开发模式 45](#_Toc370110681)

[5.5.6 项目开发模式 45](#_Toc370110682)

[5.5.7 工具支撑 45](#_Toc370110683)

[5.5.8 XXX 45](#_Toc370110684)

[6 实现视图 45](#_Toc370110685)

[6.1 概述 45](#_Toc370110686)

[6.2 系统实现模型总体结构 46](#_Toc370110687)

[6.3 开发框架的特点 48](#_Toc370110688)

[6.4 系统WebRoot结构 50](#_Toc370110689)

[6.5 系统源码包及目录结构 52](#_Toc370110690)

[6.5.1整体包结构 52](#_Toc370110691)

[6.5.2整体目录结构 54](#_Toc370110692)

[6.5.3服务组件包目录结构 56](#_Toc370110693)

[6.5.4配置文件目录结构 57](#_Toc370110694)

# 简介

## 目的

本文档表述新开普软件平台的组件化思路，并结合组件的开发、发布、部署、实施、升级的整个生命周期，阐明了系统设计的关键策略与机制，以满足相关涉众的不同关注焦点。

## 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） |
| UI构件 | 本文中UI资源构成如下：   1. Velocity模板资源； 2. 图片、页面、脚本、样式等客户端界面资源； |
| 代码构件 | 组件中可以被复用的代码模块（例如：一个排课算法模块）。代码构件只包含相应的功能逻辑实现代码与支持执行的配置文件等资源，不包含UI等其它资源，以接口方式提供功能支撑，可以整体打包到代码包中（Jar包）。 |

## 参考资料

参考资料

# 组件化思路

组件化编程的关键目的是为了将程序模块化，使各个模块之间可以单独开发，单独测试，并根据客户要求按需组合、独立升级，从而有效公司降低软件开发、实施、维护过程中的总体运营成本，增强公司对快速变化的客户需求的响应能力。

## 重用性思路

组件化的根本目的是重用性。对一个软件来说，重用本身是分层次、有粒度的，现实场景中希望针对所有粒度的重用都用统一的方式，往往是不现实或是得不偿失的。经常遇到的组件粒度划分不当，模块分割不合理的问题也经常是由于这个原因造成的。

为了解决上述问题，针对产品及项目中涉及的问题进行分析，针对不同粒度的重用进行分类，并根据重用资源的存在形式与可变性场景分析，提出公司重用方案，具体说明如下：

### 2.1.1重用模块的粒度划分

从实际场景分析可以得出公司产品与项目需要的软件模块重用粒度的划分。项目签订后，会根据客户需求选择性部署相应的产品组件（例如：考勤系统、门禁系统等）。项目实施中，项目中产品组件的实施都会根据客户需求进行定制化，包含界面、交互的定制化，部分功能的定制化，以及一些数据的定制化（表单等）。项目上线后，需求会随着时间发生变化，用户希望更友好的交互、更合理功能，还希望系统能够适应业务规则变更而灵活调整，甚至期望不同版本功能可以同时提供。

在这个过程中公司期望：产品组件可以满足不同的客户需求，减少项目实施中的定制开发工作量；产品组件针对项目定制化的合理控制与管理，包括一个项目中软件的定制化与版本管理，以及项目中产生的定制化软件模块能否在不同项目中被复用；产品版本管理与项目版本管理如何进行，以降低公司的总体运营成本。

针对上述问题，下面对公司软件体系中的可重用软件粒度进行了说明：

1. 组件（服务组件）：本文定义的组件是可独立提供某类功能服务的软件模块，客户往往在这个粒度上选择公司产品。例如针对一个项目，客户会选择购买“考勤系统”,“门禁系统”等，这就是对“考勤系统”组件与“门禁系统”组件的选择。一个产品组件往往由一个团队开发，具备良好的配置模型，可以服务于不同的项目。
2. 构件：本文定义的组件粒度是很粗狂，无法满足软件模块细粒度的复用要求，也不可能将组件配置模型完善到不同项目通过配置就能满足客户要求的程度。那么针对项目实施中与项目上线后的定制化与不断功能升级应如何管理，这里提出“构件”的定义来解决这个问题。组件是由多个构件组成的，针对组件中“构件”的划分也是从实际项目情况分析得出的。产品组件实施与上线后，客户会提出系统交互、系统功能的定制化与升级需求，所以可以抽象出“UI构件”与“代码构件”两种类型。具体定义在本文词汇表中已有描述。根据系统架构规则，业务程序员可以开发出符合用户需求的各种构件，无缝扩展到用户购买的产品组件中，为用户提供服务。用户的交互的定制化需求由UI构件满足，用户的功能定制化需求由代码构件满足。由于这些构件遵循系统架构规则，所以它们能够与产品组件灵活组合，而在不同项目被重用。
3. 不同粒度的复用模块需要不同的配置管理模式：组件与构件的划分都是为重用而抽象出来的软件模块，但是两者在重用的场景存在根本上的区别。对于产品组件来说，其版本变更不会频繁，会根据市场需求稳定发展。对于构件来说，其版本变更与定制化就频繁的多，针对每个项目或多或少的都会产生定制化的构件，而且随着项目上线，项目中构件的升级也是比较频繁的，而统一构件被用于不同项目时也会发生定制化变更。一些重用度高的定制化构件，会提取出来成为产品组件中的产品构件，它们可能会添加到当前版本的产品组件中，也可能出现在新版本的产品组件中。所以对组件与构件的配置管理模式是完全不同的。

### 2.1.2组件重用策略

组件重用从合理划分、有效封装、规范管理三个维度进行考虑：

1. 合理划分：对基础功能组件的划分需要根据技术通用性进行划分，对于业务功能组件根据领域通用性进行划分。当定义跨领域的业务功能组件时，需要保证不同领域需要数据结构不存在巨大差异。
2. 有效封装：基于统一的技术架构、良好的接口规范进行组件建设是必要的，这样才能有效保证组件的灵活组合。另外，对业务功能组件而言，准确的需求设计与优秀的配置模型设计是保证其成功的关键。
3. 组件的设计、开发、发布、升级管理需要公司级别的一致化管理模型。组件的版本管理基于代码版本管理工具（例如：SVN）进行管理就可以很好地实现。

### 2.1.3构件重用策略

构件重用需要合理划分，有效封装，可拆分、可组合、可跟踪实现，及依赖性管理：

1. 构件的合理划分是对组件中可变性才抽取，原则是尽量在一个大粒度上进行抽取，例如：一个参数管理模块的部分需要定制化，那么可以抽取一个完整的参数管理定制构件，在项目中取代标准的参数管理定制模块。
2. 有效封装指构件模块建设需要基于统一的技术架构与规范，以保证不同环境与组件及其它构件的无缝组合。同时，对于一些通用性比较高的构件也会存在自己的配置模型。
3. 可拆分、可组合、可跟踪是构件能够灵活重用的核心。因为构件不必受遵循组件那样严格的约束，所以只要方便重用，它本身可以灵活定义。对于UI构件可以根据目录隔离方式管理不同版本的构件，其中不同版本的UI构件资源会放到不同的目录中，目录名称以版本号定义。对于项目定制化的构件，项目编码作为隔离目录的名称。对于代码构件，其包名称可包含版本名，扩展包名称包含ext\_版本号，项目定制构件包名称包含ext\_项目编码­\_版本号，同理可以定义Jar包的名称。根据这些规则与信息，就可以方面跟踪每个构件了。基于目录与名称隔离，我们可以方便的打包每个构件，移植到不同项目中使用。
4. 依赖性管理是一个架构性的问题。由于构件不是自治的，不能独立存在提供服务，它要依附于一个组件，其功能实现需要依赖于组件提供的功能。另外一些依赖的基础功能服务是NSP基础平台统一提供的。在不同项目间移植构件时，可以认为NSP环境是所以项目都存在的，那个其依赖的产品组件也应该是存在的。但是，为了保证构件的移植性，必须规定构件功能实现只能依赖产品组件核心功能，而不能依赖于另一个定制构件提供的功能，否则就把这两个构件合成一个。

### 2.1.4定制化策略

定制化是所有项目都需要做的工作，如果产品组件的配置模型完善，那么通过配置可以满足大部分的定制化需求。对于需要定制化开发的部分，上述构件重用策略中基本已经描述。需要指明的是，对于定制化开发的需求，是通过向产品组件添加定制化构件实现的，不是去修改产品组件的构件代码。基于软件“开闭原则”，既可以满足项目定制化需求，也不会扰乱产品与项目的代码管理。

### 2.1.5更细粒度的重用策略

产品、项目开发最忌讳重新发明轮子，公司应该可以不断沉淀自己的技术资产，提升团队的生成效率。界面的Window空间、Tree控件、Combox控件，已经后台日历、字符转换、文件压缩等都是可以不断沉淀，并在所有开发活动中重用的。这些更细粒度的软件工具，与NSP的统一开发框架统一提供给研发人员，研发人员而言根据需要选择使用。

## 建设规划

组件化本身是一个分阶段实施、逐步完善的过程。公司需要根据自身的人员、产品、市场情况制定出合理的实现步骤，保证组件化工作的推进与阶段性成果的有效性。根据对公司各事业部的调研情况，以及结合公司产品与市场的了解，整理出公司组件化的思路是：先融合、再统一，后平台。具体说明如下：

### 2.2.1先融合（第一步）

当前公司各事业部产品数量多，采用的技术平台、技术框架也不同。而市场竞争的压力要求公司整合产品优势、提升产品质量，组件化是公司的必经之路。但是组件化的粒度与深度的选择会严重影响组件化的时间与成本。从可行性角度分析，前期摒弃理想化的、完整的组件化思想，在粗粒度上实现功能重用，尽量重用已有的成熟软件成果，对核心模块与新项目、新产品可以选择更完善的开发框架实现。基于这个思路，前期组件化通过抽取整合核心组件，规范化应用组件间的通信、接口机制，在应用组件粒度上尽快实现重用，减轻部署与升级负担。所以前期融合策略：核心组件抽取与实现（基于新开发框架），基于远程文档型接口的应用组件集成模式建立，公司级别基于组件化的配置管理模式建立，组件化产品管理模式建立。

### 2.2.2再统一（第二步）

在融合阶段，应用组件粒度上的复用比较容易实现。但是存在一些问题：首先应用组件整合方式强制系统本身必须是分布式系统，对一些项目来说不适合；其次，如果技术框架与开发框架不同统一，那些细粒度的构件复用就无法有效实现；最后，最终平台化也希望规范的精细化与技术的统一，这样对平台工具的开发，以及平台工具的自动化也是必要条件。所以，第二阶段的统一主要是技术平台与技术框架的统一，这个阶段公司会定制主流技术平台（当前结果应该是J2EE平台），并且考虑与其它技术的融合性（考虑一些关键技术点需要C、C++,C#这样的技术实现）。这一阶段的工作主要是：将公司的核心产品都迁移到统一的技术平台与框架上；完善组件化方案，支持集中部署与分布式部署两种方式，完善构件级别的重用方案；完善组件的开发、测试、发布、升级流程与配套工具，完善公司级配置管理与产品管理模式；

### 2.2.3后平台

平台化是公司组件化进程进入成熟阶段的标致。基于前两个阶段的工作，在技术与经验上公司已经达到了一定的成熟度。结合公司的市场规划、产品策略与项目性质，从总体上进行软件体系与机制的优化，对基础服务的封装与关键过程（开发、实施、升级中的重要环节）自动化是这个阶段在重要工作，同时完成平台的开发环境的建设也一个核心工作。这个阶段完成后，将为公司软件体系提供完整的规范，为研发人员提供统一、简洁的开发环境，为实施人员提供编辑的部署、升级及监控工具，从而整体提高公司的生成效率与产品质量。

# 组件化目标与约束

描述公司组件化的整体目标，并且为了保证可行性需要满足的客户约束。

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。

## 组件化目标

1. 基于组件化建设，保证公司产品建设与项目建设中软件模块的灵活组合，使得在产品开发与项目建设中可以最大限度地利用已有的软件模块，而不必重新开发，有效降低公司软件生产的整体成本。
2. 基于组件化建设，延长软件模块的生命周期。根据产品与项目的实际需要，在不同粒度的软件模块上实施版本管理。
3. 基于组件化建设，保证产品建设与项目建设可以有效维护，独立升级。使产品建设与项目建设互相不受影响，可以有效跟踪各自的版本，并且清晰化项目对产品版本的依赖性管理。
4. 基于组件化建设，实现组件间的灵活组合，实现不同组件间软件模块（构件）的有效复用，以及基于统一技术平台与架构实现更细粒度的软件功能复用。
5. 基于组件化建设，实现不同团队间的并行开发，并且可以无缝整个各自的成果。
6. 基于组件化建设，统一公司的技术管理与技术体系。
7. 基于组件化建设，完成公司成熟技术平台的建设。

## 组件化的约束

1. 组件化程度与方案的不同，对时间与成本投入存在巨大差别，在选择组件化策略师必须充分考虑公司市场、技术与人员的实际情况。必须能够跟上市场节奏，为公司的市场活动服务，同时有效控制成本（人员与时间成本）。
2. 组件建设是一个长期持续的过程。如何由简入繁地构建公司的组件体系，使公司的技术团队逐步统一是个关键问题。必须尊重公司实际情况（产品、项目、技术、人员等情况），既不能建设速度过慢而跟不上市场节奏，也不能过急打乱当前的工作节奏。
3. 组件化建设涉及到技术管理体系的建设与完善，这方面也必须从公司的实际情况入手，逐步调整公司的相关管理体系。

## 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、接口的合理定义及软件依赖性的管理，实现服务组件间的松散耦合，并保证服务组件能够灵活组合。基于规则（目录、包、命名）等有效隔离扩展、定制化及不同版本的构件，结合软件模块依赖性管理，保证构件可以灵活打包与重用。

### 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

组件平台需要有能力支撑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

组件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。

支持与其他系统进行集成，所以要提取出良好的集成接口。

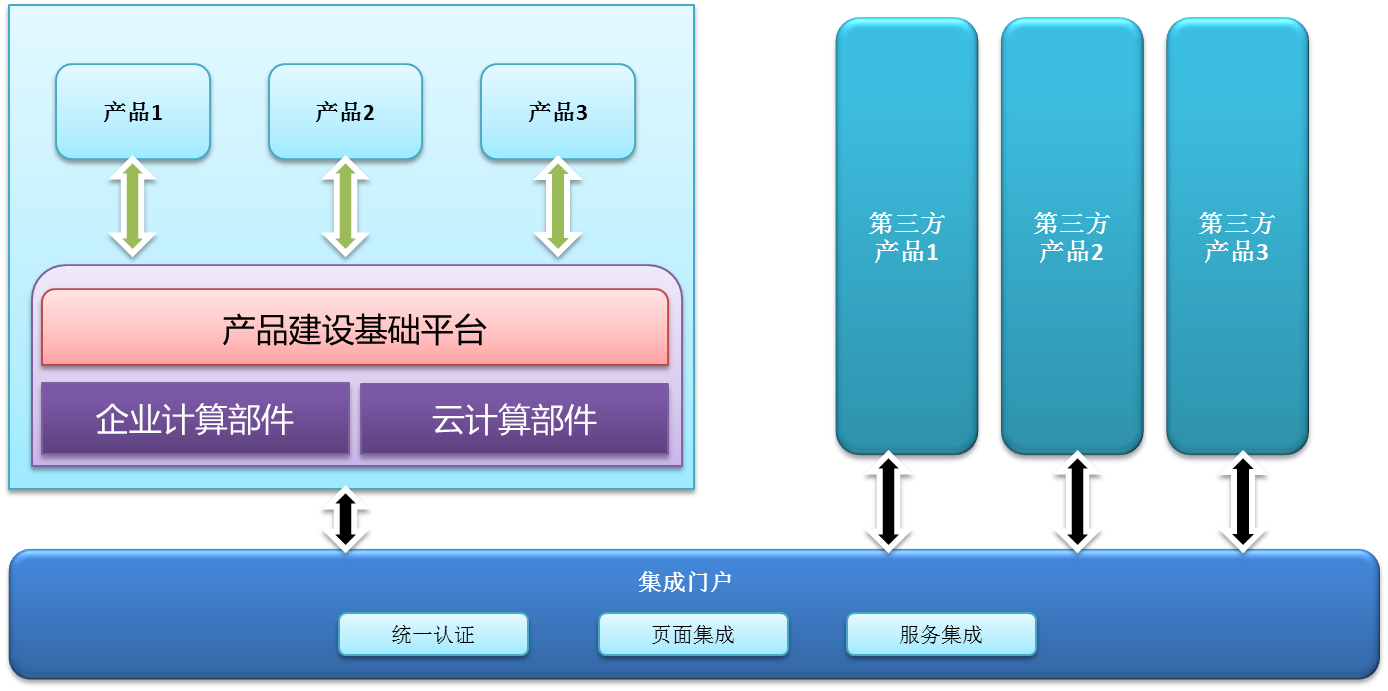
开发人员提供一致、简单的编程模型，屏蔽企业级产品开发、运行级产品开发带来的复杂性，有效降低产品建设、项目实施的成本。

# 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

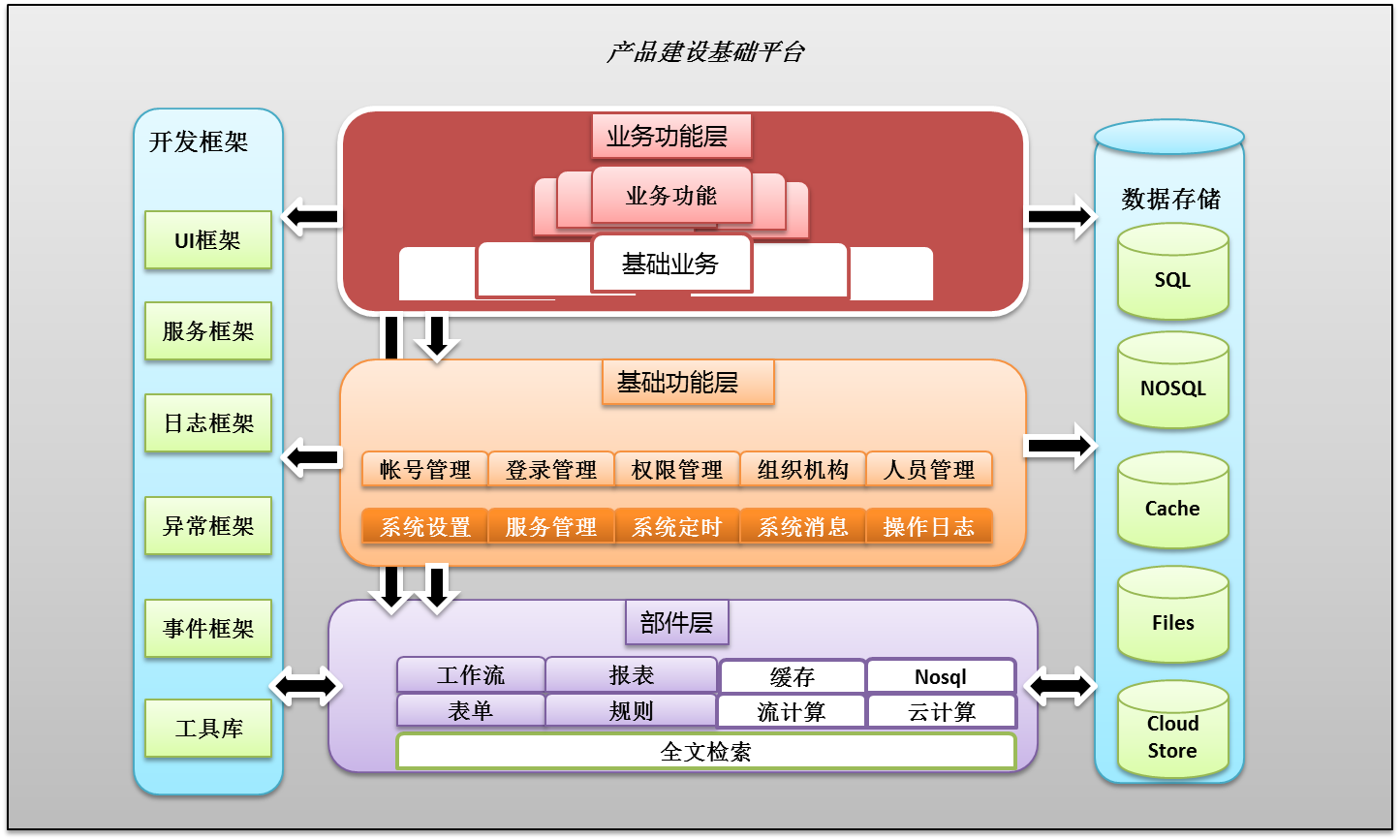


如上图所示，公司软件技术平台为企业计算领域及云计算领域的产品建设，提供了一个分布式的、灵活的、组件化的产品建设平台。其中，基础门户系统本身的定位，不但为公司产品及服务整合提供场所，而且也为第三方产品、服务的整合提供了一个开放的环境。

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。

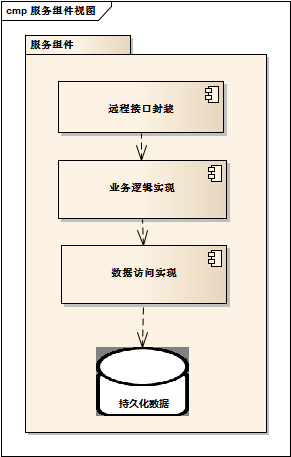
## 服务组件

### 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：



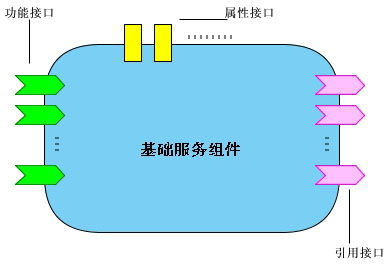
其中每层的具体实现仍然遵循接口隔离的原则，不同层次实现间的交互仍然通过各层实现暴露的接口进行，以此保证服务中的每层实现的可替换性。

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。

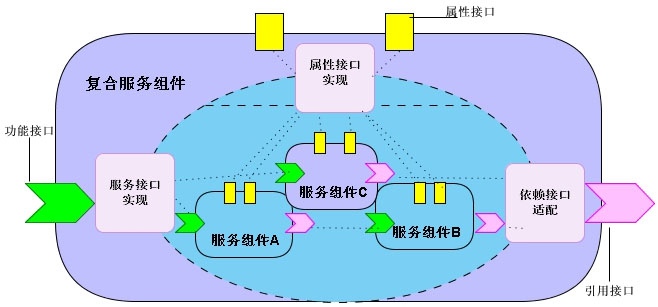
**服务组件交互约束：**外部系统与服务组件的交互，都必须通过严格定义的服务组件接口进行。服务组件接口是外部系统区分服务组件类型的依据，也是服务组件封装的边界。遵循这个原则，只要保持服务组件接口的稳定性，服务组件内部实现的修改对外部系统是不可见的，系统实现时可以针对同一服务组件提供多种实现。需要说明的是，服务组件内部实现中使用的一些接口并不在服务组件接口定义的范围内，原则上它们不应该暴露给服务组件的使用者，使用者也不允许直接使用这些接口。

**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。

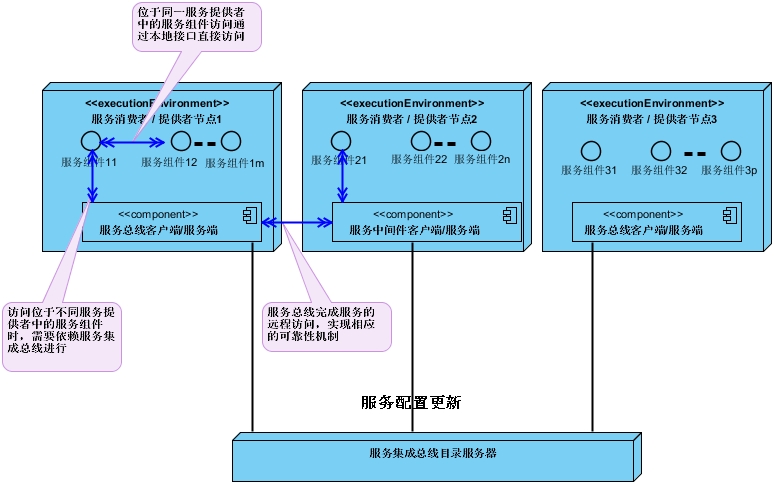
系统服务组件可以通过这种递归的方式组合成不同粒度的服务组件，为系统提供服务。基础服务组件是系统存在原子服务组件，也是功能替换的最小粒度。只要符合服务组件标准定义的规范，原则上各种粒度的服务组件都能够实现替换。

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。

### 5.2.3组件的划分

应用服务组件的划分存在于一个粗粒度上。在组件定义时，需要从开发者视角进行考虑：一个组件应该对应一个单独的代码库，并且有一组开发人员负责维护，这样便于团队的分工与协助。本节描述了组件划分的三种常见方式，以及组件与应用产品的之间的关系，说明如下：

**一个组件就是一个产品：**



这种情况下，一个产品的开发由一个研发小组负责，开发、调试、部署都比较简单，目标产品可以方便地与其他产品组合成完整的解决方案，服务于客户。

**多个组件形成一个产品，严格遵守组件规范：**



这种情况下，一个产品由多个组件构成，开发可以由一个或多个产品研发小组负责，产品部署是由多个分布式部署的组件构成的。除了业务独立性外，组件划分考虑的产品销售与实施需求。

对于产品销售来说：产品销售可以根据客户需求，有选择的部署组件，以支持市场的差异化定价策略。

对于产品实施来说：可以把需求稳定的业务与需求不稳定的业务封装成不同的组件，这样实施过程中只要对不稳定的组件进行定制，而不影响其它组件。

对于开发来说：对于一个复杂产品的开发，可以划分成不同业务组件，由不同的研发小组并行生成，以提高产品研发的效率。

**多个组件形成一个产品，不严格遵守组件规范：**



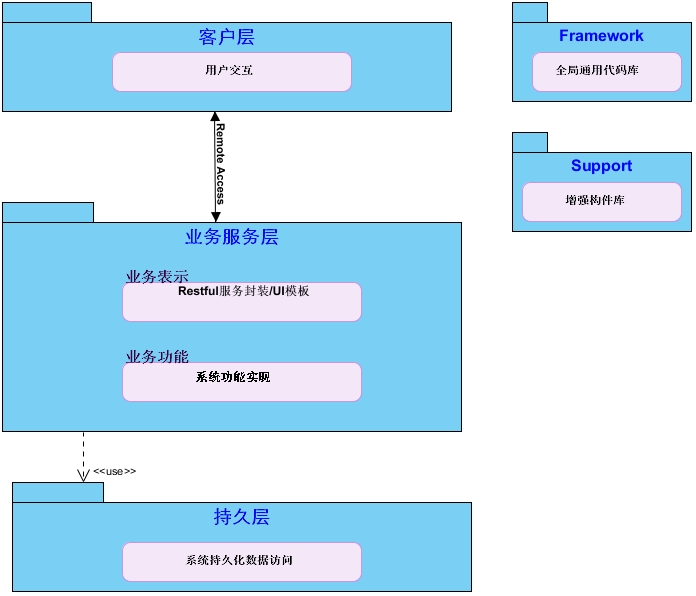
这种情况下，一个产品由多个组件组成，部署方式与特点同上所述。但是与上述方式不同，多个组件是共享存储的，因此组件之间是存在数据依赖。这种模式实际破坏了架构要求的组件约束，但是在以下情况下是一种可行的方案：

1. 组件逻辑，涉及到不同组件产生数据的复杂关联；
2. 不同组件，其需求的稳定性具有很大差异；
3. 不同组件，对计算能力、存储、并发等非功能特性的要求存在很大差异。

上述方案中，所有部署组件都依赖同一个数据存储。

## 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。

## 组件包与构件包说明

本节描述组件打包形式与构件打包形式说明，这些打包规则可以基于Maven或Ant之类的构建工具实现自动化。

### 打包粒度划分

程序打包分成三个粒度：多个服务组件组成的应用包，服务组件包，构件包。具体说明如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| **包粒度** | **适应场景** | **包构成** | **包形式** |
| 应用包 | 适合多个服务组件集中部署，为用户提供一个单例应用实例的场景。（根据实际情况，单例应用能够集群部署） | 基础服务组件、所需要多个应用组件及相应资源（第三方依赖库、图片等）都打包到同一个应用包中。 | WAR |
| 组件包 | 适合于分布式部署方式。 | 单个服务组件及相应资源（第三方依赖库、图片等）都打包到一个组件包中。 | WAR |
| 适合于集中部署方式。 | 单个服务组件及个性化依赖资源（集中部署中，公共的第三方依赖库、图片等资源已经在部署环境中存在，所以只选择那些组件特有的资源进行打包即可）打包成一个组件包。 |  |
| 构件包 | UI构件包，用以在不同组件间、不同项目间重用UI构件。 | UI构件的客户端资源（JS、CSS、图片等资源）与界面模板资源（Velocity模板）。 | WAR |
| 代码构件包，用以在不同的组件间，不同项目间重用代码构件。 | 构件的代码与相应的其它构件资源（配置文件或其它资源等） | JAR |

### 打包资源说明

#### 应用包说明



如上图所示，应用包主要应用与多组件集中部署，提供一个单例应用的情况下。此时，构建过程将第三方依赖包、NSP平台基础包、基础服务组件和所有用户购买的服务组件都打包在一起，发布一个war包。此时，实施过程是针对一个应用的安装，不涉及每个组件的安装过程，以及组件接口间的配置。

#### 组件包说明

**分布式部署组件包：**



如上图所示，分布式部署组件包，需要将第三方依赖库、平台基础包、目标服务组件统一打包，发布成一个war包。由于分布式环境中每个组件都是独立的应用上下文环境，所以此时第三方依赖库与平台基础包与集中部署时不同，不能被公用，所以需要一起打包。实施过程中，每个组件时独立部署的，实施人员需要配置、调试好组件间协作的远程接口，会带来一定的实施负担。

**集中部署组件包：**



如上图所示，集中部署环境下的打包的组件是最精简的。此时，由于集中部署应用上下文环境中第三方依赖包与平台基础包已经存在，并且被所有组件所共有，所有只需打包服务组件发布即可。对于实施过程，则需要自动化工具的帮助，来基于架构规则进行组件的部署与升级，来减轻实施人员的工作复杂性。

#### 构件包说明

**UI构件包：**



如上图所示，UI构件表示一组可重用的UI资源，它有JS、CSS、图片等客户端资源以及一组服务器端的Velocity模板资源构成，其发布成一个资源包，可以被用于其它项目建设与组件的建设中。

**代码构件包：**



如上图所示，代码包表示可以被重复利用的一组代码模块。由构件的程序代码与模块执行需要的其它资源构建形成，其发布形式为一个Jar，可以用于其它项目与组件的建设中。

#### 通用控件与库



如上图所示，各类通用界面与执行逻辑，存在于更细的粒度上，它们的通用性更强，没有领域特征或，不存在外界环境的强依赖，可以为所有产品、项目开发所重用。这类通用模块将封装成通用的界面控件库或代码库，存在于NSP平台中，支持所以服务组件的执行，并且通过NSP的开发框架提供给开发人员使用。

### 可重用条件

上文描述的四种类型的重用模块，每种模块的重用性是受到部署环境与开发环境限制的，下面将具体说明。

#### 5.4.4.1组件重用性说明

在NSP中的组件是定义在粗粒度上，本身是数据自治与服务自治的。

在分布式部署环境中，只要组件依赖的远程接口在部署环境已经提供，组件原则上就可正常提供服务。由于NSP平台基于J2EE实现，原则上基于其实现的各种组件也是跨平台的，所以，分布式部署组件的重用性是很好的。分布式部署带来的问题是远程接口编程、远程访问效率以及多服务治理的负担问题，这些问题会带来开发、实施的复杂性，需要良好的工具来减轻负担。

在集中部署环境中，部署环境需要提供组件执行需要的条件，也就是NSP上。并且，组件间的访问基于平台规范进行，保证组件间的弱依赖。集中部署时组件的安装与升级，涉及到文件的拷贝、比对、替换，并且可能会存在部分全局配置文件的变更。需要基于NSP的架构规则，生产自动化工具减轻实施负担。

#### 5.4.4.2UI构件重用性说明

UI构件封装了一组可重用的界面与交互。其本身实现会依赖与NSP平台的公共脚本与样式库，所以UI构件的重用范围是在基于NSP开发的组件与项目中。

另一方面会限制UI构件的重用范围，就是它的实现是否依赖于衍生出它的组件的个性化部分资源，如果存在依赖性，那么在打包时需要将其依赖的资源一同打包，或者其部署环境中需要该组件同时存在。

#### 5.4.4.3代码构件重用性说明

代码构件封装了一组可重用的功能逻辑。其实现本身会依赖NSP平台提供的基础功能，所以代码构件的重用范围也是在基于NSP开发的组件与项目中。

另一方面会限制代码构件的重用范围，就是它的实现是否依赖于衍生出它的组件的个性化功能，如果存在依赖性，那么其部署环境中需要该组件同时存在。为了保证代码构件的可重用性。NSP架构规定，原则上每个构件实现只依赖于其衍生组件的功能接口，尽量不要依赖另外的扩展构件的功能，这样可以保证构件的可移植性。只要依赖的组件在部署环境中存在，该构件就可以在这个环境中被重用。

#### 5.4.4.4通用控件与库重用性说明

平台提供的统一控件（例如界面的树形控件）和功能库，会随同开发框架一起提供给开发人员。开发人员可以在NSP平台中的所有组件开发与项目开发中，使用这些通用控件与功能库。

## 组件化、构件化机制

组件、构件与通用库是NSP定义的三种粒度上是复用模块。通用库的使用与维护与开发框架一同进行，这里不再展开说明，下面就组件与构件开发、生成、部署、升级的核心机制进行说明，并对阐述了其中的关键性策略。

### 组件化的版本管理

NSP的组件定义可以理解为存在于应用级别，例如：考勤应用组件、门禁应用组件等。组件的版本管理依赖常用版本管理工具（SVN）完成即可，每个应用组件形成一个代码仓库，通过版本管理工具Tag标识组件吧版本。

### 组件重用机制

为了提高组件的重用性，组件的服务接口都暴露成远程的文档性接口，从通用性与效率两方面考虑，组件远程接口暴露成HTTP+JSON的方式。基于NSP平台提供统一认证、统一权限、门户基础产品组件，整合到一起为项目提供服务，如下图所示。



这种方式每个组件都可以单独开发与维护，基于NSP基础服务组件组合成一个整体解决方案提供给客户使用。

对于部分组件需要集中部署的场景，NSP提供了组件间本地访问的工厂机制，保证组件间服务弱耦合性，如下图所示：



当组件A需要访问组件B提供的功能服务时，它不是直接实例化组件B的相关类，并进行接口调用（这种方式会产生强依赖）。而是，向NSP的全局工厂NSPComponentFactory请求组件B的组件厂组件BFactory实例。此时，NSPComponentFactory会根据“组件装配配置文件”中的信息，检测相应的组件工厂是否存在，如果不存在则返回NULL，如果存在则返回工厂实例引用。

当返回值为NULL是表示组件B没有在本地环境安装。当不位NULL时，组件A可以通过组件B工厂实例获得组件B的服务实现对象的引用，完成相关服务调用。

### 构件版本管理

构件存在与一个组件中，其版本的更新比较组件本身要频繁的多。所以，NSP构件的版本管理是基于自定义的目录与命名规则进行的，说明如下。

#### 5.5.3.1UI构件版本管理

UI构件由脚本、图片、样式等客户端资源与后台的Velocity模板文件构成。对它的版本管理可以通过目录隔离的方式进行，也就是不同版本的构件资源放置到不同的目录中，目录的编码规则可以很容易的跟踪、打包、部署相应版本的构件，具体说明如下：

**客户端资源放置方式**

应用根目录/资源类型目录/组件名规则目录/版本号/

例如对权限组件（privilege），规划对其表示构件进行版本控制，那么其1.0版本客户端JS脚本将放置在以下目录中：



相同规则，该构件对应的图片资源放置在以下目录中：

应用根目录/images/foundation/privilege/V1.0/

对于客户端其它资源基于相同的规则进行目录定义。

**服务器端资源放置方式**

WEB-INF/pages/foundation/privilege/department/pagelet/v1.0

如上所示，服务器端模板复用的粒度是页面片段级（pagelet）别的，每个页面片段的版本号指明了其依赖的客户端资源的版本号。上面目录，表示了department（部门）这个pagelet的1.0版本的模板资源。同理可以定义更多的可服用pagelet：

WEB-INF/pages/foundation/privilege/user//v1.0

WEB-INF/pages/foundation/privilege/role/pagelet/v1.0

WEB-INF/pages/foundation/privilege/resource/pagelet/v1.0

、、、、、、

**项目定制UI构件如何管理**

对于项目定制构件，由于它本身不是组件、产品安计划生成，往往是根据项目需要，按需开发的。这里需要强调一个NSP项目定制的架构规则：当一个产品组件应用于项目时，需要对其进行定制修改时，不能修改组件本身的代码，而是开发一定制化的构件，补充到项目中，实现客户的定制化需求。基于这样的“开闭原则”，就可以有效管理公司产品与项目研发，减少冲突。

对于定制构件的管理，也是基于目录隔离的，拿客户端脚本扩展构件为例，说明如下：

应用根目录/scripts/foundation/privilege/ext/项目编码/构件编码/V1.0/

构件其它资源是以相同的规则进行目录管理的。

**组件产品扩展构件如何管理**

组件产品扩展构件，不与具体项目绑定，所以其目录定义如下：

应用根目录/scripts/foundation/privilege/ext/构件编码/V1.0/

构件其它资源是以相同的规则进行目录管理的。

#### 5.5.3.2代码构件版本管理

代码构件有可重用的功能逻辑代码与功能执行依赖的其它资源（配置文件等）组成。对它的版本控制是通过包名称的编码规则来实现的，具体策略与UI构件的思路是一致的。具体说明如下：

代码包：组件包名.ext.构件编码.版本编码

资源包：组件包名.ext.构件编码.版本编码.config

举例说明如下，还是以权限组件（privilege）为例，如果计划针对当前版本的权限组件开发一个扩展构件（例如：针对餐饮领域增加一种授权方式），对这个餐饮领域授权构件的编码为catering\_authorize，那么这个构件的包结构为：

代码包：

cn.newcapec.foundation.privilege.ext.catering\_authorize.v1\_0

资源包：

cn.newcapec.foundation.privilege.ext.catering\_authorize.v1\_0.config

当产品组件用在项目中时，项目定制化的原则同上所述，构件的包结构为：

代码包：组件包名.ext.项目编码.构件编码.版本编码

资源包：组件包名.ext.项目编码.构件编码.版本编码.config

仍以权限组件为例，例如我们对餐饮企业全聚德项目进行定制一个个性化的授权方式，这里的项目编码是quanjude,构件编码是authorize,对应的构件包结构为：

代码包：

cn.newcapec.foundation.privilege.ext.quanjude.authorize.v1\_0

资源包：

cn.newcapec.foundation.privilege.ext.quanjude.authorize.v1\_0.config

基于这种规则化控制，可以通过构件工具，从产品或项目库中抽取打包不同的构件，用于其它的产品与项目建设中。

### 构件重用机制

首先通过上述机制，就可以方便地从某个产品组件或项目中，打包出需要重用的构件。例如以上面的例子来说明，可以将餐饮领域授权模型打包成jar包：cn.newcapec.foundation.privilege.ext.catering\_authorize.v1\_0.jar。

当获取了这个构件Jar包后，它代表了权限组件的针对餐饮领域授权的一个扩展。下一步，需要判断它可以重用的条件：

1. 构件可以重用的系统时基于NSP平台建设的；
2. 适合与权限组件的那个版本。作为组件扩展，构件依赖于组件提供功能，所以构件执行需要部署环境中存在已有组件，而且在一些情况下对组件版本也是有约束的。

上面描述的情况是构件功能的直接重用，在一些情况下可以重用构件的源代码，这是可以基于规则打包构件源代码，导入到新项目的开发环境，重构包名，并且根据需要对代码进行调整，就可以产生新的构件。

### 组件开发模式

//有待补充

### 项目开发模式

//有待补充

### 工具支撑

//有待补充

### XXX

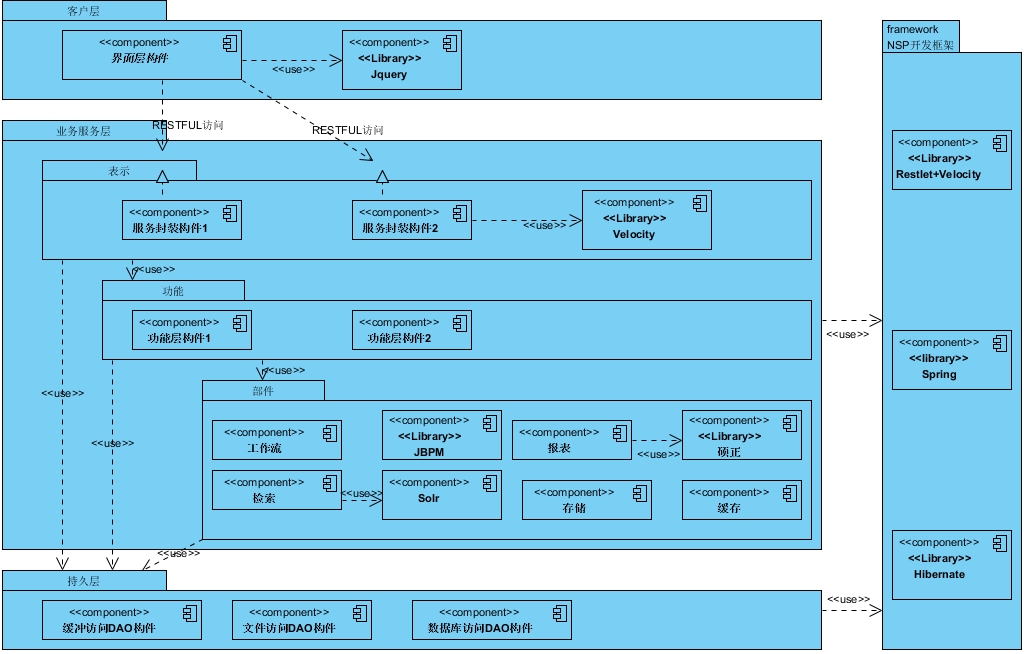
软件实现视图

# 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：

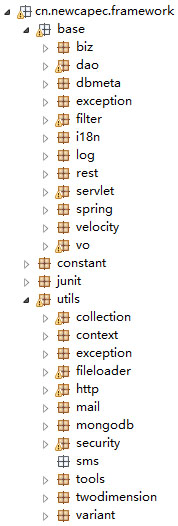
客户端资源目录（JS、Css等）：foundation目录作为基础服务组件资源的根目录，其中的每个子目录代表一个组件资源，目录名称以组件的英文名称标识；function目录作为业务服务组件资源的根目录，其中每个子目录代表一个组件资源，目录名称以组件的英文名称标识；lib目录包含了框架依赖的第三方库，所有服务组件实现都可以对其产生依赖；其它的同级目录可以根据资源类型不同进行扩展定义，但是这些目录中资源都是公共资源，不隶属于某个组件所有。

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

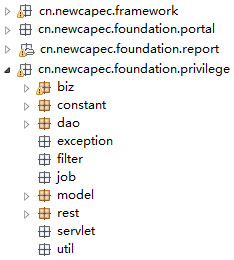
framework中包含基础架构代码，包含在base包中。

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。

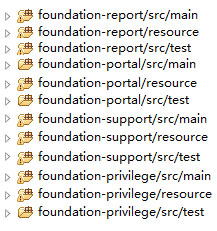
上面描述的系统的整体包结构划分，具体研发过程会有不同的小组成员分别负责不同模块及组件的研发。所以，整体研发的目录结构需要实现研发与部署的模块化，具体目录结构描述如下。

### 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：**



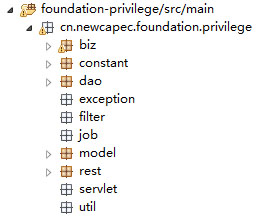
在以上目录结构中，每个应用程序包包含两个源代码树和一个资源包，两个源代码树分别是Java应用[程序](http://www.miiceic.org.cn/phrase/200604232224305.html)的源代码和单元测试源代码，资源包用来管理本模块的配置文件。

**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。

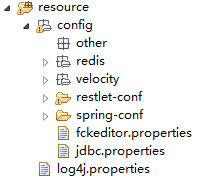
9、servlet包

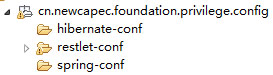
业务服务中Servlet类定义的场所；

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。