# 基于Spring的WEB功能组件化详细设计

## 目标背景

精灵管家本身的目标是希望作为一个智能的企业公众号管理平台，为企业管理提供便利的公众号管理功能，包括众多智能化的功能应用，减轻用户管理公众号负担，提高精灵管家的用户体验。

但是随着精灵管家系统功能的复杂，原有的系统结构在功能扩展和开发方面都非常困难，主要体现在：各个功能模块耦合度过高；功能的内部代码块分散在不同的package，造成功能纵向分散，横向聚合（分层系统的特点，一个完整的功能被切分为多个层次），这样的缺点就是在进行扩展时，需要对整个系统的各个层次都进行修改，分别添加新功能的相应部分，代码也难于管理，同时功能之间的依赖关系也无法控制。最明显的缺点就是，添加新功能需要在原系统的代码结构中添加新的类，相当于直接嵌入到系统内部成为系统的一部分（就好像是长出来了的新芽，而不是放上去的积木），无法将精灵管家看作是一个平台对外开放扩展。另外，还有就是每次发布一个新功能系统必须重启，稳定性堪忧！

因此，为了能够提高系统功能的可扩展性、代码的可维护性、系统的稳定性，以及良好的用户体验，需要对系统进行组件化重构。最终，系统作为一个平台提供通用的应用服务，而业务逻辑则是在平台之上的一个个可插拔式（热插拔）组件，组件的增删只需要简单的上传删除操作即可，无需重启系统。而开发人员（包括第三方），按照组件开发的规范，进行组件的独立开发。

## 术语与约定

* **PILLOW项目**：本项目
* **组件（WEB组件、应用）**：在web系统中的功能模块，主要特点是包含完整的MVC结构（也可以不完整），即该功能模块具有自身的模型、视图、控制器，类似一个子系统；
* **PILLOW**：枕头，组件一词在代码中的名称；
* **根上下文（ROOT Context）**：由ContextLoaderListener所创建的spring的根上下文环境，在web.xml中配置，作为平台的根上下文环境；
* **核心请求处理器（Core Dispatcher）**：Spring核心控制器（DispatcherServlet），在web.xml中配置，用来处理除组件请求之外的平台本身的请求；
* **组件上下文（PillowContext）**：组件拥有独立的spring上下文环境；
* **~~组件通用请求处理器（PillowPillowDispatchController）~~**~~：所有组件的请求处理都会先由该servlet响应，然后由组件分发控制器发送到具体的组件中处理。另外还提供默认的请求处理方案；~~
* **组件分发控制器（PillowDispatcherController）**：从通用请求处理器中获取url请求，根据请求连接获取对应的组件并调用处理；
* **组件请求处理器（PillowDispatcherServlet）**：组件本身的请求处理器，在自己的配置文件中配置。

## 基本原理

一般系统只需要一个根上下文和一个dispatcher 上下文即可。而要达到组件化的目标，我们需要对spring的父子上下文进行扩展，自动为每一个组件都生成自身的上下文，并根据依赖关系（树形，只能有一个父级），构造自身的组件依赖体系。在这个依赖树中，根节点为系统根上下文（即spring容器的根上下文）。因此需要对上下文初始化进行干预，重写组件上下文加载及初始化过程。

对于组件dispatcher的加载初始化则比较麻烦。这是因为web容器需要所有的servlet（dispatcher即servlet）都必须配置在web.xml中，而我们不可能将所有组件的dispatcher都配置到web.xml中。因此，采用配置一个所有组件公用的servlet（PillowDispatchController，继承DispatcherServlet），这个servlet和spring的默认前端控制器（DispatcherServlet）配置上没有任何差别，主要差别在于PillowDispatchController的初始化和请求处理，这两个地方都被重写，初始化时不仅初始化自身，对所有组件的dispatcher进行创建并初始化。

因为PillowDispatchController响应所有组件请求（通过前缀匹配，如/plugin/\*），请求过来后会进入到该类继承于FrameworkServlet的doService方法，我们需要重写改方法，让请求分发到具体的组件中，这个分发工作由DispatchController.doService进行。DispatchController.doService会根据url中的组件名称查找对应的组件并调用其doService方法完成请求响应。

## 架构设计

正题口口口

### 设计目标

虽然PILLOW是因精灵平台而发展的，但是我们的目标不仅仅是服务于精灵平台，因此除在目标背景一节中所讲到的目标之外，我们还努力将PILLOW做成一个通用性的技术方案，所有符合要求的项目引入PILLOW后即可获得所有PILLOW提供的服务，将自身从专用系统提升为平台！

因此PILLOW的架构需要达到以下几个目标：

* 易于开发；
* 易于扩展；
* 高可靠；
* 非入侵；
* 通用性高；

### 依赖基础

spring + spring mvc

PILLOW是建立在spring和spring MVC之上的，因此要求使用PILLOW的项目必须使用这两种技术框架。

### 系统结构图

基础功能服务

ROOT context

Core dispatcher

一般组件

Pillow context

Pillow dispatcher

父级为一般组件的组件

Pillow context

Pillow dispatcher

DispatcherController

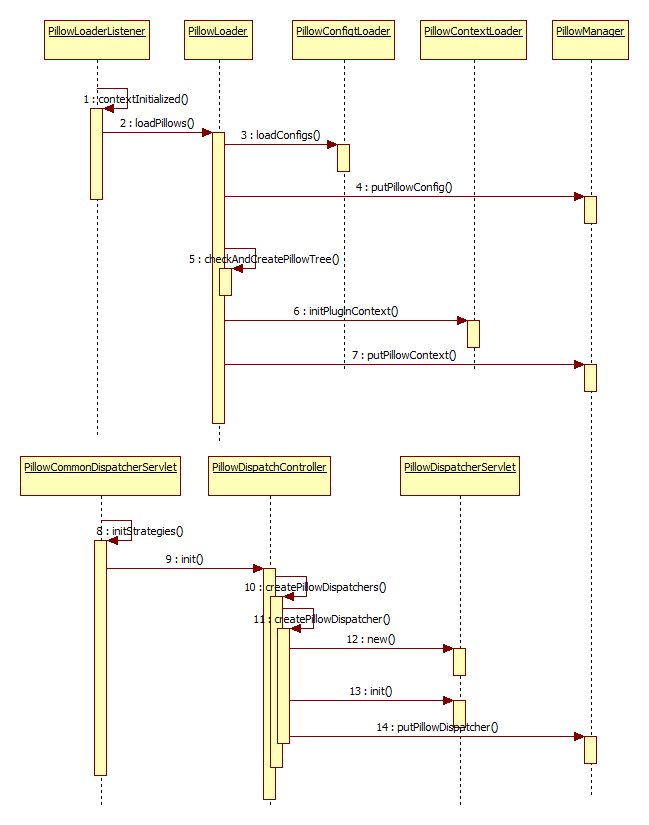
箭头表示父子关系，起点为父级，指向为子级。

### 类图

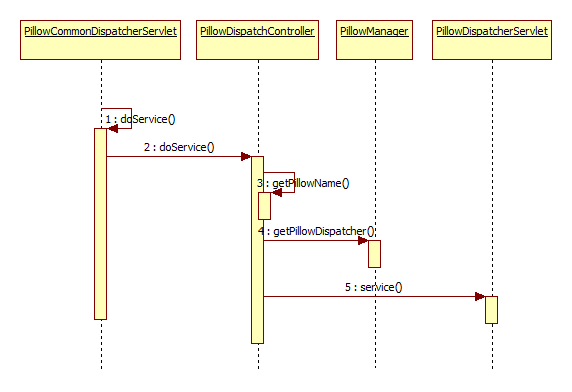
图3-1 类图



### 初始化流程



### 请求处理



### 关键类设计

#### PillowLoaderListener

作为组件加载初始化入口，作用类似spring中的contextLoaderListener，被配置在web.xml中（springcontextLoaderListener之后）作为一个监听器。在系统启动时启动，通知PillowLoader进行组件环境加载初始化。

该类作用比较简单，内持有pillowLoader对象。源码大致如下：

**publicclass** PillowLoaderListener **implements**ServletContextListener {

**private** PillowLoader pillowLoader = **new** PillowLoader();

**privatestaticfinal** Logger *log* = Logger.*getLogger*(PillowLoaderListener.**class**);

/\*\*

\* Initialize the root web application context.

\*/

@Override

**publicvoid** contextInitialized(ServletContextEvent event) {

**try** {

**this**.pillowLoader.loadPillows(event.getServletContext());

} **catch** (IllegalAccessException e) {

*log*.error("Init pillow context failed!",e);

e.printStackTrace();

}

}

@Override

**publicvoid** contextDestroyed(ServletContextEvent arg0) {

//do nothing

}

}

实现了ServletContextListener接口的contextInitialized方法，该方法在启动时会被tomcat容器（或其他web容器）调用。

#### PillowLoader

组件加载类，用于根据指定方式，搜索组件并解析加载到系统中。主要完成定位功能，具体的解析和加载组件上下文交由PillowConfigtLoader和PillowContextLoader负责。

**publicclass** PillowLoader{

**privatestaticfinal** Logger *log* = Logger.*getLogger*(PillowLoader.**class**);

/\*\*

\* 配置文件路径变量

\*/

**privatestatic** String *PILLOW\_LOCATION\_ATTRIBUE* = "pillowConfigLocation";

/\*\*

\* 默认路径

\*/

**privatestatic** String *DEFULT\_PILLOW\_LOCATION* = "/WEB-INF/lib/plugins";

/\*\*

\* The root WebApplicationContext instance that this loader manages.

\*/

**private** WebApplicationContext rootContext;

/\*\*

\* 组件管理器

\*/

**private** PillowManager pillowManager = PillowManager.*getPillowManager*();

//配置文件加载解析，暂时仅支持配置文件在jar包中，并且名称固定为pillow.xml

**private** PillowConfigtLoader configLoader;

//组件上下文环境加载器

**private** PillowContextLoader contextLoader;

/\*\*

\* 从JAR\_PLUGIN\_LOCATION加载组件

\* **@param** servletContext

\* **@throws** IllegalAccessException

\*/

**publicvoid** loadPillows(ServletContext servletContext) **throws** IllegalAccessException {

//根context，确保在pillowcontext之前被加载

rootContext = (WebApplicationContext) servletContext.getAttribute(

WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*);

**if** (rootContext == **null**) {

**thrownew** IllegalStateException(

"Cannot initialize pillow context because can't find a root application context present - " +

"check whether you have configured ContextLoader\* definitions in your web.xml " +

"and make sure it is defined before PillowLoader\*!");

}

//从DEFAULT\_PLUGIN\_LOCATION中加载jar并生成PillowConfig

List<PillowConfig> configList = loadConfigs(servletContext);

//组件名及其配置，组件名全局唯一。检查组件配置是否有误，并生成组件依赖树

List<PillowConfig> availableConfigList = checkAndCreatePillowTree(pillowManager,configList);

//根据配置信息，逐个加载组件

**for**(PillowConfig config:availableConfigList){

loadContext(servletContext,config);

}

}

/\*\*

\* 根据指定目录，获取所有的组件配置文件，并加载

\* **@return**

\*/

**private** List<PillowConfig> loadConfigs(ServletContext sc) {

String location = sc.getInitParameter(*PILLOW\_LOCATION\_ATTRIBUE*);

**if**(location == **null**){

location = *DEFULT\_PILLOW\_LOCATION*;

}

//根据路径，获取配置文件，并创建配置信息对象。利用spring中的resourcereader

List<PillowConfig> list = configLoader.loadConfigs(location);

**return** list;

}

/\*\*

\*

\* **@param** pillowManager

\* **@param** configList

\* **@return**

\* **@return**

\*/

**private** List<PillowConfig> checkAndCreatePillowTree(PillowManager pillowManager, List<PillowConfig> configList) {

//check and create tree

//set available config into manager

**for**(PillowConfig config:availableConfigList){

pillowManager.putPillowConfig(config.getPillowName(), config);

}

**return**availableConfigList;

}

/\*\*

\* 加载初始化组件上下文环境，若组件有父级，则先加载初始化父级

\* **@param** servletContext

\* **@param** config

\*/

**privatevoid** loadContext(ServletContext servletContext, PillowConfig config) {

ApplicationContext father;

**if**(config.getFather() != **null**){

father = pillowManager.getPillowContext(config.getPillowName());

//父级尚未初始化，先初始化父级

**if**(father == **null**){

loadContext(servletContext, config.getFather());

}

father = pillowManager.getPillowContext(config.getPillowName());

}

**else**{

//根上下文为所有组件的顶级上下文

father = rootContext;

}

//进行组件上下文创建及初始化

ApplicationContext pillow = contextLoader.initPluginContext(servletContext, father, config);

//放入到管理器中

pillowManager.putPillowContext(config.getPillowName(), pillow);

}

}

#### PillowConfig

组件配置信息对象，保存从配置文件中解析出来的组件的一些信息，主要包括：

**publicclass** PillowConfig {

//组件名，全局唯一

**private** String pillowName;

//父级组件若为null，则父级为根上下文

**private** PillowConfig father;

//上下文配置文件路径

**private** String contextConfigLocation;

//组件dispatcher的配置文件路径

**private** String dispatcherConfigLocation;

}

#### PillowConfigtLoader

组件配置文件加载类，目前默认从/WEB-INF/lib/plugin/下加载名为pillow.xml的文件。可以通过扩展支持从其他指定路径进行加载。（注意，同一包路径下同名文件只会加载第一个）

**publicclass** PillowConfigtLoader{

**privatestaticfinal** Logger *log* = Logger.*getLogger*(PillowConfigtLoader.**class**);

/\*\*

\* 默认加载路径

\*/

**privatestaticfinal** String *DEFAULT\_PILLOW\_LOCATION*= "/WEB-INF/lib/plugins";

/\*\*

\* 默认配置文件名

\*/

**privatestaticfinal** String *DEFAULT\_PILLOW\_XML\_NAME* = "pillow.xml";

**public** List<PillowConfig> loadConfigs(String pluginLocation) {

// do some something

}

}

#### PillowContextLoader

加载并初始化组件上下文，组件上下文和根上下文一样是一个WebApplicationContext对象，默认以根上下文为父级。

**publicclass** PillowContextLoader{

**privatestaticfinal** String *DEFAULT\_PILLOW\_LOCATION* = "WEB-INF/lib/plugins";

/\*\*

\* 组件上下文环境变量显示名后缀

\*/

**privatestaticfinal** String *CONTEXT\_DISPLAY\_NAME\_SUFFIX* = " WebApplicationContext";

/\*\*

\* Initialize Spring's web application context for the given servlet context,

\* according to the "{@link #CONTEXT\_CLASS\_PARAM contextClass}" and

\* "{@link #CONFIG\_LOCATION\_PARAM contextConfigLocation}" context-params.

\* **@param** servletContext current servlet context

\* **@return** the new WebApplicationContext

\* **@see** #CONTEXT\_CLASS\_PARAM

\* **@see** #CONFIG\_LOCATION\_PARAM

\*/

**public** WebApplicationContext initPluginContext(ServletContext servletContext, ApplicationContext parent, PillowConfig config) {

*log*.info("Initializing Pillow '" + config.getPillowName() + "' WebApplicationContext");

**if** (*log*.isInfoEnabled()) {

*log*.info("'" + config.getPillowName() + "' WebApplicationContext: initialization started");

}

**long** startTime = System.*currentTimeMillis*();

WebApplicationContext context = **null**;

**try** {

// Store context in local instance variable, to guarantee that

// it is available on ServletContext shutdown.

context = createWebApplicationContext(servletContext, parent, config);

**if** (*log*.isDebugEnabled()) {

*log*.debug("Published '" + config.getPillowName() + "' WebApplicationContext.");

}

**if** (*log*.isInfoEnabled()) {

**long** elapsedTime = System.*currentTimeMillis*() - startTime;

*log*.info("'" + config.getPillowName() + "' WebApplicationContext: initialization completed in " + elapsedTime + " ms");

}

**return** context;

}

**catch** (RuntimeException ex) {

*log*.error("Context initialization failed", ex);

servletContext.setAttribute(WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*, ex);

**throw** ex;

}

**catch** (Error err) {

*log*.error("Context initialization failed", err);

servletContext.setAttribute(WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*, err);

**throw** err;

}

}

/\*\*

\* Instantiate the root WebApplicationContext for this loader, either the

\* default context class or a custom context class if specified.

\* <p>This implementation expects custom contexts to implement the

\* {@link ConfigurableWebApplicationContext} interface.

\* Can be overridden in subclasses.

\* <p>In addition, {@link #customizeContext} gets called prior to refreshing the

\* context, allowing subclasses to perform custom modifications to the context.

\* **@param** sc current servlet context

\* **@param** parent the parent ApplicationContext to use, or <code>null</code> if none

\* **@return** the root WebApplicationContext

\* **@see** ConfigurableWebApplicationContext

\*/

**protected** WebApplicationContext createWebApplicationContext(ServletContext sc, ApplicationContext parent, PillowConfig config) {

Class<?> contextClass = determineContextClass(sc);

**if** (!ConfigurableWebApplicationContext.**class**.isAssignableFrom(contextClass)) {

**thrownew** ApplicationContextException("Custom context class [" + contextClass.getName() +

"] is not of type [" + ConfigurableWebApplicationContext.**class**.getName() + "]");

}

ConfigurableWebApplicationContext wac =

(ConfigurableWebApplicationContext) BeanUtils.*instantiateClass*(contextClass);

//下面用到sc的可能存在冲突问题

// Assign the best possible id value.

**if** (sc.getMajorVersion() == 2 && sc.getMinorVersion() < 5) {

// Servlet<= 2.4: resort to name specified in web.xml, if any.

String servletContextName = sc.getServletContextName();

wac.setId(ConfigurableWebApplicationContext.*APPLICATION\_CONTEXT\_ID\_PREFIX* +

ObjectUtils.*getDisplayString*(servletContextName));

}

**else** {

// Servlet 2.5's getContextPath available!

**try** {

String contextPath = (String) ServletContext.**class**.getMethod("getContextPath").invoke(sc);

wac.setId(ConfigurableWebApplicationContext.*APPLICATION\_CONTEXT\_ID\_PREFIX* +

ObjectUtils.*getDisplayString*(contextPath));

}

**catch** (Exception ex) {

**thrownew** IllegalStateException("Failed to invoke Servlet 2.5 getContextPath method", ex);

}

}

wac.setParent(parent);

wac.setServletContext(sc);

// wac.setServletConfig(getServletConfig());

wac.setConfigLocation(config.getContextConfigLocation());

wac.setNamespace(config.getNamespace());

customizeContext(sc, wac);

//传说中的refresh

wac.refresh();

**return** wac;

}

}

#### PillowManager

Manager作用其实和context差不多，都是用来保存一些信息对象。这里的manager主要用来保存组件的配置信息对象、组件上下文、组件dispatcher

**publicclass** PillowManager {

/\*\*

\* 单例，饥饿模式

\*/

**publicstatic** PillowManager *manager* = **new** PillowManager();

/\*\*

\* 组件名与组件配置对象映射

\*/

**private** Map<String,PillowConfig>pillowConfigMap = **new** HashMap<String,PillowConfig>();

/\*\*

\* 组件名与组件上下文对象映射

\*/

**private** Map<String,ApplicationContext>pillowContextMap = **new** HashMap<String,ApplicationContext>();

/\*\*

\* 组件名与组件dispatcher对象映射

\*/

**private** Map<String,PillowDispatcherServlet>pillowDispatcherMap = **new** HashMap<String,PillowDispatcherServlet>();

}

#### PillowPillowDispatchController

组件url请求通用处理器，其作用有二：

1、作为初始化入口，和组件请求入口；

2、为组件提供通用的请求处理方案；

**publicclass**PillowPillowDispatchController**extends** DispatcherServlet{

/\*\*

\* 组件请求分发控制器

\*/

**private** PillowDispatchController controller = **new** PillowDispatchController();

/\*\*

\* Initialize the strategy objects that this servlet uses.

\* <p>May be overridden in subclasses in order to initialize further strategy objects.

\*/

**protectedvoid** initStrategies(ApplicationContext context) {

//本身的配置初始化完毕

**super**.initStrategies(context);

// getWebApplicationContext();

//创建并初始化组件

controller.init(**this**);

}

/\*\*

\* Exposes the DispatcherServlet-specific request attributes and delegates to {@link #doDispatch}

\* for the actual dispatching.

\*/

@Override

**protectedvoid** doService(HttpServletRequest request, HttpServletResponse response) **throws** Exception {

controller.doService(request, response);

}

}

#### PillowDispatchController

所有组件的请求都会先通过PillowPillowDispatchController.doService() ，然后交由controller.doService()进行分配到具体的组件。

PillowDispatchController有两个作用，首先是加载并初始化组件的PillowDispatcherServlet；第二就是在用户请求过来时，查找正确的组件并响应请求。

**publicclass** PillowDispatchController {

/\*\*

\*

\*/

**private** PillowManager manager = PillowManager.*getPillowManager*();

**private** PillowPillowDispatchController common = **null**;

**publicvoid** init(PillowPillowDispatchController pillowPillowDispatchController) {

common = pillowPillowDispatchController;

createPillowDispatchers(common.getContext());

}

/\*\*

\* 创建并初始化组件的dispatcher

\*/

**privatevoid** createPillowDispatchers(ApplicationContext context) {

Map<String, PillowConfig> configMap = manager.getPillowConfigMap();

Iterator<String> it = configMap.keySet().iterator();

**while**(it.hasNext()){

String name = it.next();

PillowConfig config = configMap.get(name);

**try** {

createPillowDispatcher(context,config);

} **catch** (ServletException e) {

**continue**;

}

}

}

/\*\*

\* 创建组件dispatcher

\* **@param** config

\* **@throws** ServletException

\*/

**privatevoid** createPillowDispatcher(ApplicationContext context,PillowConfig config) **throws**ServletException {

//

PillowDispatcherServlet dispatcher = **new** PillowDispatcherServlet();

dispatcher.setPillowName(config.getPillowName());

dispatcher.setNamespace("pillow-servlet");

dispatcher.setContextConfigLocation(config.getDispatcherConfigLocation());

dispatcher.init(common.getServletConfig());

pillowDispatcherMap.put(config.getPillowName(), dispatcher);

}

/\*\*

\* 确定请求响应组件，并处理请求

\* **@param** request

\* **@param** response

\*/

**publicvoid** doService(HttpServletRequest request, HttpServletResponse response) {

String name = getPillowName(request);

**try** {

manager.getPillowDispatcher(name).service(request, response);

} **catch** (ServletException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

/\*\*

\* 根据url确定组件名

\* **@param** request

\* **@return**

\*/

**private** String getPillowName(HttpServletRequest request) {

**return null**;

}

}

有两种组件dispatcher处理方式：

1. 像上面的代码，根据给定的servlet文件，生成一个PillowDispatcherServlet对象，该对象有自己的各种handler、mapping等。用户请求过来时，直接将请求交给组件的dispatcher进行处理即可；
2. 第一种方式优点是简单、可靠，缺点是所有组件都必须要有自己的配置文件，配置所有dispatcher必须的bean，这导致大量重复冗余配置。
3. 因此有第二种方式：重写DispatcherServlet中所有的get\*方法，提供默认实现；
4. 个人偏向第二种；

#### PillowDispatcherLoader

上面createPillowDispatcher完成了该类的功能，需要移到该类来。

#### PillowDispatcherServlet

组件上下文类

**publicclass**PillowDispatcherServlet**extends** DispatcherServlet{

**private** PillowManager manager = PillowManager.getPillowManager();

**private** String pillowName = **null**;

**public** String getPillowName() {

**return** pillowName;

}

**publicvoid** setPillowName(String pillowName) {

**this**.pillowName = pillowName;

}

/\*\*

\* Initialize and publish the WebApplicationContext for this servlet.

\* <p>Delegates to {@link #createWebApplicationContext} for actual creation

\* of the context. Can be overridden in subclasses.

\* **@return** the WebApplicationContext instance

\* **@see** #setContextClass

\* **@see** #setContextConfigLocation

\*/

**protected** WebApplicationContext initWebApplicationContext() {

setContextConfigLocation(manager.getPillowConfig(pillowName).getDispatcherConfigLocation());

WebApplicationContext wac = findWebApplicationContext();

**if** (wac == **null**) {

// No fixed context defined for this servlet - create a local one.

WebApplicationContext parent =

(WebApplicationContext) manager.getPillowContext(pillowName);

wac = createWebApplicationContext(parent);

}

//refresh!!!

onRefresh(wac);

// Publish the context as a servlet context attribute.

String attrName = getServletContextAttributeName();

getServletContext().setAttribute(attrName, wac);

**if** (**this**.logger.isDebugEnabled()) {

**this**.logger.debug("Published WebApplicationContext of servlet '" + getServletName() +

"' as ServletContext attribute with name [" + attrName + "]");

}

**return** wac;

}

}

重写了initWebApplicationContext方法主要是加入

setContextConfigLocation(manager.getPillowConfig(pillowName).getDispatcherConfigLocation());

语句，该语句作用是设置当前dispatcher配置文件路径，若不重写，由于所有组件都是通过commondispatcher进行初始化的，将使用commondispatcher的配置路径。

## 使用配置

* 在web,xml中配置PillowLoaderListener和PillowPillowDispatchController和pillowConfigLocation变量；
* 在根spring中配置PillowManager；
* 编写具有特定目录结构的功能应用；
* 将应用打成jar放到要求的目录下；
* 打jar包时需要包含路径信息（否则无法使用注解扫描），在eclipse打包时需要勾选Add directory entires

## 其他

热加载卸载；

版本升级；

## 问题与改进

接口编程；

不同的使用场景；

## 附录

### 参考资料

spring 源码

### 配置文件格式

### 例子