模块化系统的构建专家

OSGI

如果你一直在构建模块化系统，但是，只有当你使用OSGi来构建模块化系统时，你才会体验到什么是真正的模块化系统。

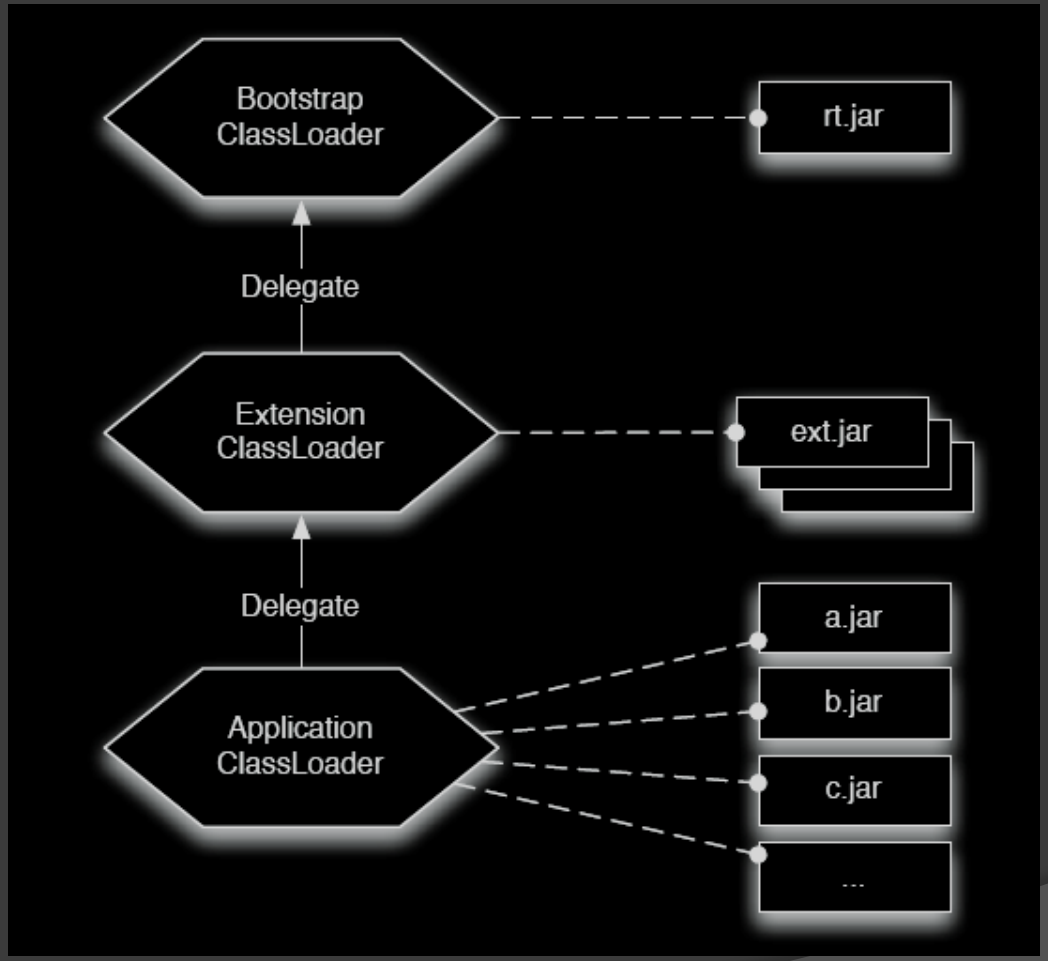
大纲

- 传统项目打包方式分析
  - Java类加载器
  - 类冲突
  - Jar包依赖性管理
  - 包可见性管理
  - Jar包版本管理
- 传统项目模块结构分析
  - 从硬编码到模块化的演变
- OSGi方式重构传统项目
  - 完全的模块化、动态化
- OSGi简介
  - 概述
  - OSGi Module Layer
  - OSGi Life Cycle Layer
  - OSGi Service Layer
- OSGi相关实现
  - Felix，Equinox，Spring DM，Eclipse Plugin
- OSGi应用
  - Demo演示

JAR打包方式分析

- Java类加载器
- 类冲突
- Jar包依赖性管理
- 包可见性管理
- Jar包版本管理

Java类装载器



Java类装载器

- 类加载模型：双亲委派模型

  1. JRE向应用类加载器（ApplicationClassLoader）请求加载一个类
  2. 应用类加载器向扩展类加载器（ExtensionClassLoader）请求加载这个类
  3. 扩展类加载器向启动类加载器（BootStrapClassLoader）请求加载这个类
  4. 如果启动类加载器找到这个类，则将它装入内存，如果没有找到，则扩展类加载器会尝试加载它
  5. 如果扩展类加载器找到这个类，则将它装入内存，如果没有找到，则应用类加载器会尝试加载它
  6. 如果应用类加载器找找到这个类，则将它装入内存，如果没有找到，则抛出ClassNotFoundException

类冲突

- 如果两个应用jar文件（ty.jar, hw.jar）中同时包含com.boco.Alarm类，类加载器应该如何加载这个类？

jar包依赖性管理

- 经常碰到的一种情况：

  例如我们引入了spring.jar，结果使用的时候发现它还依赖于common-logging.jar。

  在传统的开发模式下这种依赖性只能通过文档来规范

包可见性管理

考虑下面这种场景：

Jar包中包含以下三个包：

com.boco.tongyong.kernal

com.boco.tongyong.service

com.boco.tongyong.service.impl

假设impl和kernal包中是我们的实现，应用模块并不需要关心，我们只需要将service包暴露出来，应该如何处理？

Jar包版本管理

- 场景：

GEVT1.5中依赖GAUS2.0，而系统中引入的是GAUS1.0，假设GAUS1.0与GAUS2.0的包结构和类结构一致，这个错误在将只会在系统运行期被发现。

在传统模式下只能使用文档来作规范。

题外话

- Maven可以管理jar包依赖性和jar包版本问题，从项目构建管理的角度上讲与OSGi还存在差距。

传统项目模块结构分析

一个传统项目

- 项目需求：构建一个简单的系统自身管理程序，通过Socket连接系统，根据用户输入的命令对系统各个关键点进行维护。

  例如：　输入date显示系统当前时间；

  输入memory显示系统占用内存

  输入thread显示系统线程数
- 项目特点：命令是变化的，系统开发完成后可能需要增加新的命令来满足不同的系统管理功能

一个传统项目-方式一

- 方式一：对命令进行硬编码，最原始的方式

```
if(cmd.equals(“date”)){
        showDate();
}else if(cmd.equals(“memory”)){
        showUsedMemory();
} else if(cmd.equals(“thread”)){
        showThreads();
} else if(….) {}
```

一个传统项目-方式一

- 优点：

  封装了socket连接，后续的开发只需要新增else if(cmd.equals(“”))语句即可实现相关的业务管理功能，而不需要关注socket连接的细节
- 缺点：每次增加新的命令或修改旧的命令都需要修改代码

一个传统项目-方式二

- 方式二：采用一个ActionCommand类负责命令的分发，根据命令请求的不同将命令发送到不同的处理类中进行处理。

  具体的命令类可以通过配置文件配置。

  UML：TODO

一个传统项目-方式三

- 优点：每次新增新的命令只需要在配置文件中配置对应的命令处理类，不需要修改任何现有代码
- 缺点：每次增加新的命令需要修改配置文件

一个传统项目-方式三

- 方式三：将配置文件分离，不同的命令处理类在不同的配置文件中配置，类结构与方式二相同。项目结构类似于：

```
SysManager
        |--lib
                |--kernal.jar
                |--ext1-huawu.jar
                |--ext2-shuju.jar
                |--…
        |--cfg
                |--kernal(cmd-kernal.xml，…)
                |--ext1-huawu(cmd-hw.xml，…)
                |--ext2-shuju(cmd-sj.xml，…)
        |--resouces(…)
```

一个传统项目-方式三

- 优点：
- 缺点：

项目的OSGi实现

OSGi简介

- OSGi简介
  - 概述
  - OSGi类加载器
  - OSGi Module Layer
  - OSGi Service Layer

OSGi简介

- OSGi服务平台提供一个通用、安全并且可管理的Java框架；它可以动态管理部署在框架内的Bundle，在不重启系统的情况下对Bundle进行安装和移除
- OSGi框架层次
  - 安全层 Security Layer
  - 模块层 Module Layer
  - 生命周期层 Life Cycle Layer
  - 服务层 Service Layer

安全层

- OSGi安全层是OSGi服务框架的一个可选的层。它基于Java 2 安全体系结构和jar签名机制，提供了对精密控制环境下的应用部署和管理的基础架构。

模块层

- Bundle
- Bundle描述文件
- Bundle类加载器
- Bundle国际化

模块层

- Java平台只提供了对打包、部署和对Java应用和组件检验的最小支持。很多基于java的项目，如J2EE AppServer，常常借助于专用的类加载器来创建用户模块层，以实现打包、部署和对Java应用和组件检验。而OSGi框架提供了对Java模块化的标准的解决方案。

Bundle

- Bundle是一个框架定义的模块化单元，它包含模块运行时需要的class文件和资源文件，通过Manifest.MF文件组织起来。
- Bundle在OSGi框架中以jar的形式进行部署，Bundle也是框架中需要部署的唯一实体。
- 框架中的各个Bundle是物理隔离的，它们通过Export-Package、Import-Package和service的方式进行交互。

Bundle

- 一个Bundle的组织结构

  Bundle-Alarm

  |--classes(编译的类文件)

  |--OSGi-OPT(可选目录，DS的描述文件放在这个目录)

  |--META-INF(manifest文件放在这里)

  |--….（自定义）

Bundle描述文件

- MANIFEST.MF

Bundle-ContactAddress:	bundle发行者的联系信息。
Bundle-Copyright: BOCO(c) 2002	bundle的版权信息。
Bundle-Description: 告警拓扑基础	bundle的描述信息。
Bundle-DocURL: http:/www.boco.com.cn/tongyong/doc	Bundle文档的链接地址。
Bundle-Localization: OSGI-INF/l10n/bundle	描述bundle的本地文件地址
Bundle-ManifestVersion: 2	定义了bundle遵循的OSGi规范版本。默认值为1，表示R3的bundle，2表示R4及其后发布的版本。
Bundle-Name: 基础工具	定义了一个具有可读性的名字来标识bundle。
Bundle-NativeCode: /lib/runtime.dll; osname = Solaris; osversion = 8	对bundle中包含的本地代码库的规范。
Bundle-RequiredExecutionEnvironment:	描述在服务平台的执行环境
Bundle-SymbolicName: com.boco.tongyong	提供了bundle的一个全局的惟一的标志符，名称应该是基于反域名解析的。这个标记是必须的。
Bundle-UpdateLocation:	bundle的更新地址。
Bundle-Vendor: BOCO	bundle的发行者信息。

Bundle描述文件

- Export-Package和Import-Package

Bundle之间交互的一种主要方式；如果系统内不同版本的Bundle需要共存，版本标识是非常重要的。

Bundle-Kernal:

Export-Package：com.boco.tongyong.service;version="2.0"

Bundle-Kernal-bak:

Export-Package：com.boco.tongyong.service;version="1.0"

Bundle-Alarm:

Export-Package：
com.boco.tongyong.alarm.model;version="1.0"

Import-Package：com.boco.tongyong.service;version="2.0"

Bundle-App:

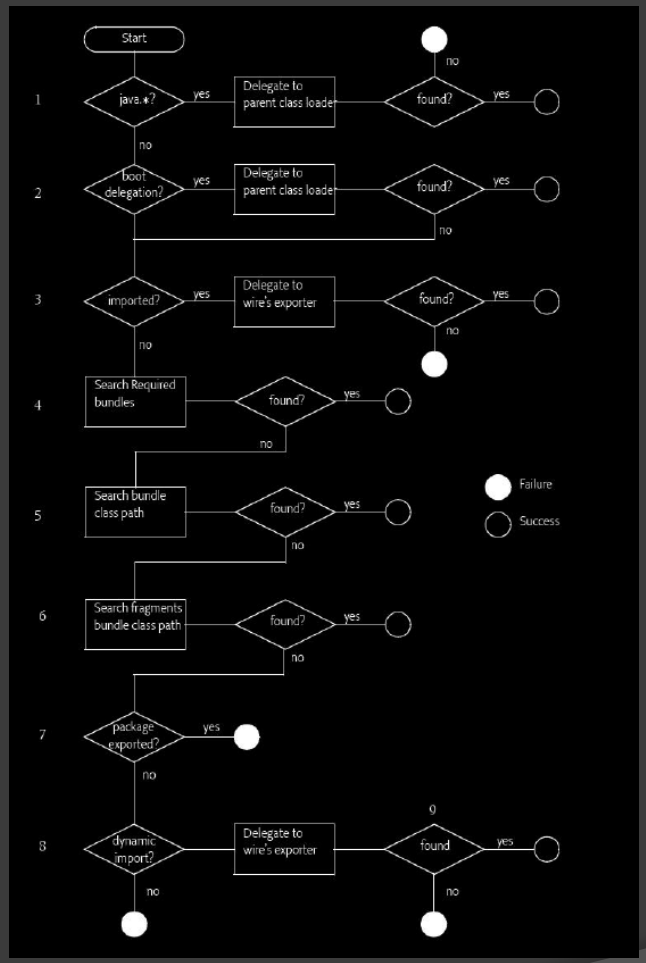
Import-Package: com.boco.tongyong.alarm.model,

com.boco.tongyong.service;version="1.0"

Bundle类加载器

- 每一个Bundle(除了Fragment Bundle)都有自己的类加载器，这种机制保证了Bundle间的物理隔离，也是OSGi动态加载能力的基础。

Bundle类加载器



Bundle国际化

- 采用标准的Java国际化方案，在Manifest文件中配置如下：

  Bundle-Name=%bundleName

  在OSGI-INF/l10n目录下建立bundle_zh_cn.properties文件：

  bundleName=告警拓扑

一种特殊的Bundle

- Fragment Bundle

  Bundle片段，没有独立的类加载器，必须附属在其它的Bundle上，一个应用场景是使用它来更好的实现国际化。

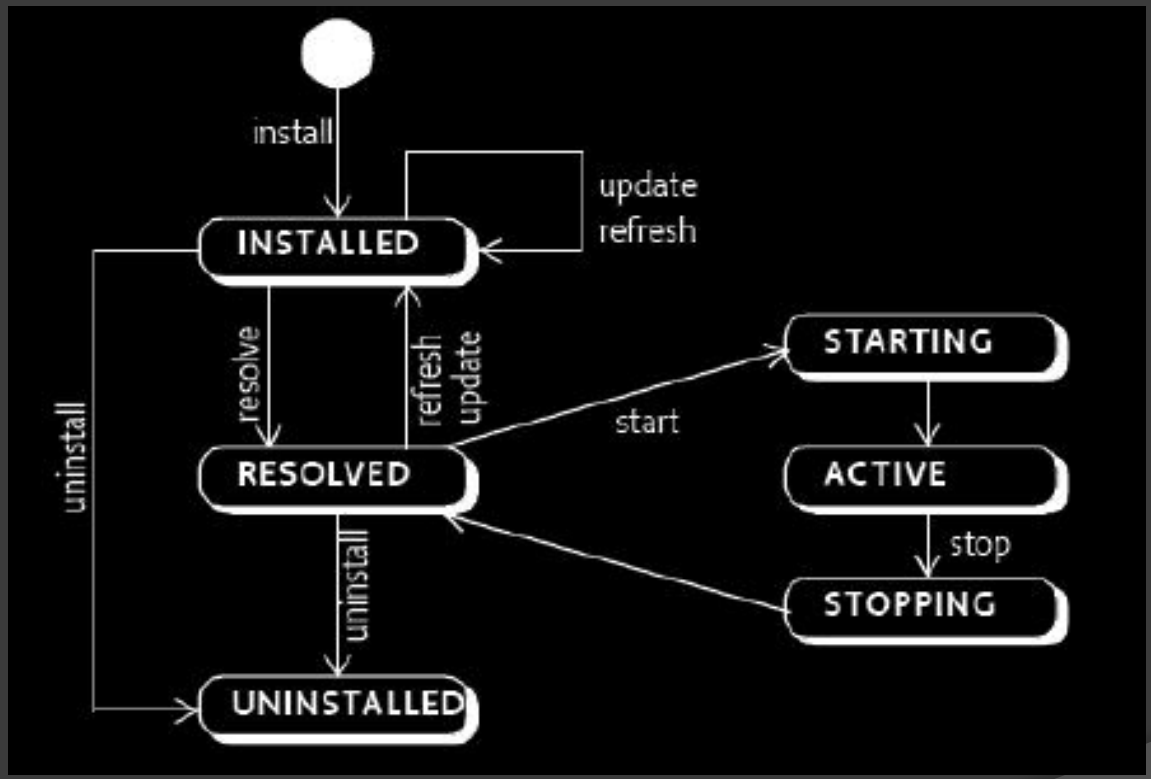
生命周期层

- Bundle状态
- 系统Bundle
- BundleContext
- Bundle事件

生命周期层

- **生命周期层**
  生命周期层建立在安全层和模块层之上，提供Bundle的生命周期管理功能。
- **Bundle状态**
  Bundle状态共有六种：
  INSTALLED
  Bundle已经成功的安装。
  RESOLVED
  Bundle中所需要的类都已经可用，RESOLVED状态表明Bundle已经准备好了用于启动或者Bundle已被停止。
  STARTING
  Bundle正在启动中，BundleActivator的start方法已经被调用，不过还没返回。BundleActivator是可选的，可以在其中初始化资源，启动线程。
  ACTIVE
  Bundle已启动，并在运行中。
  STOPPING
  Bundle正在停止中，BundleActivator的stop方法已经被调用，不过还没返回。
  UNINSTALLED
  Bundle已经被卸载了。

生命周期层



系统Bundle

- 系统Bundle是一种特殊的Bundle，它的启动在任何其它的Bundle之前，并且不能被卸载（Uninstall），系统Bundle状态的变化会发出FrameworkEvent事件。系统Bundle也不能通过start命令进行启动。

BundleContext

- BundleContext是框架和其它Bundle之间联系的一个纽带。
- 每个Bundle启动的时候，框架会生成对应的BundleContext对象，通过BundleActivator的start方法传给对应的Bundle。
- 在多个Bundle之间不能传递BundleContext。
- 通过BundleContext，Bundle可以向框架注册自己提供的服务，获取其它Bundle提供的服务，监听其它Bundle的状态等。

Bundle事件

- 在Bundle的状态发生变更时，框架会发出相应的事件，事件分为BundleEvent和FrameworkEvent，而对应的监听器也有两种：BundleListener和FrameworkListener。应用系统可以通过注册监听器来跟踪关心的事件，以在动态系统中表现出正确的行为。

服务层

- R3服务层
- 声明式服务（ Declarative Services）

R3服务层

- 服务层提供服务的注册、查找定位、服务状态的监听等功能
- 注册
  通过BundleContext的registerService方法进行服务的注册
- 查找
  通过BundleContext的getServiceReference方法进行服务的查找
- 状态监听
  通过BundleContext的addServiceListner方法注册ServiceListener，进行服务状态的监听
- 服务的取消注册
  通过BundleContext的unregister方法进行服务的取消注册

声明式服务Declarative Services（DS）

- DS是一个面向服务的组件模型，它制订的目的是更方便地在 OSGi 服务平台上发布、查找、绑定服务，对服务进行动态管理。对服务组件的描述采用XML来实现。
- 在 DS中，Component 可以是 Service 的提供者和引用者，一个 Component 可以提供 0 至多个 Service，也可以引用 0 至多个 Service，并且采用component 方式封装 Service，方便了对 Service 的复用。
- 在DS中发布的服务的生命周期由OSG框架管理

Component.xml

- Component.xml是DS的服务配置文件，像spring的applicationContext.xml一样，在这个文件中可以定义服务，配置服务之间的依赖关系

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="AlarmService">
    <implementation class="com.boco.tongyong.AlarmServiceImpl"/>
    <service>
        <provide interface="com.boco.tongyong.service.AlarmService"/>
    </service>
    <reference name="DaoService"
    interface="com.boco.tongyong.DaoAervice" bind="setDaoAervice"
    unbind="unsetDaoAervice" policy="dynamic"/>
</component>
```

启动级别服务

- 在OSGi框架中可以定义Bundle的启动级别，启动级别是一个非负整数，值越小的启动级别越高，系统Bundle的启动级别为0
- 应用场景：

1. 可能有某些Bundle需要先启动，例如消息服务、C/S中的SplashScreen，这时启动级别要设高一点
2. 某些Bundle在启动时依赖于其他Bundle提供的服务，这时启动级别要设低一些

OSGi相关实现

- Felix：OSGi规范的标准实现，Apache项目
- Equinox：在OSGI规范的基础上增加了扩展点的支持，该项目在Eclipse社区
- Spring DM：Spring和OSGi的结合，在OSGi规范的基础上构建，支持Felix和Equinox。与OSGi的DS是互补的，同时Spring中的其它特性均可以在OSGi中使用
- Eclipse RCP：基于Equinox开发的客户端平台

OSGi应用

- 目前业界基于OSGi构建和支持OSGi的系统已经非常多了，比如GlassFish、WAS AppServer、Spring DM Server（一个基于OSGi实现的OSGi应用服务器，通过了许多类库的测试，可以为我们的应用提供很好的参考）
- 与Spring一样，OSGi框架是低侵入性的，同时学习的曲线并不陡峭，就开发来说除了需要关注系统动态性带来的变化外，仅仅是打包方式的不同
- 使用OSGi构建应用系统时具有以下特点的类库需要谨慎使用：
  - 具备动态类生成能力的类库；
  - 从classpath加载资源的类库；
- 不适合使用OSGi的系统：比较依赖于J2EE应用服务器的系统

OSGi应用

- 在OSGi中使用JMX、Spring、JMS、RMI、AOP

- 问题？

感谢