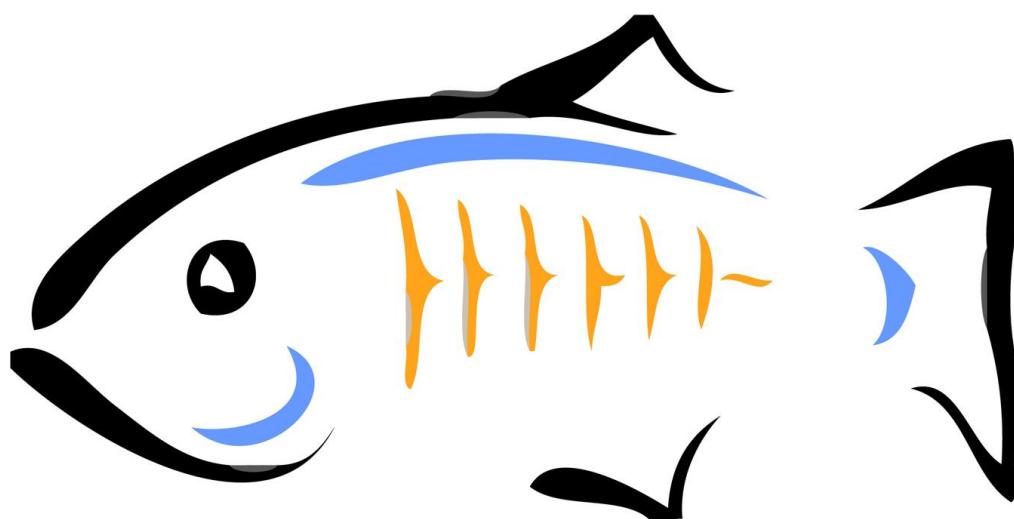


GlassFish架构、启动、配置和监控

GlassFish V3 初探



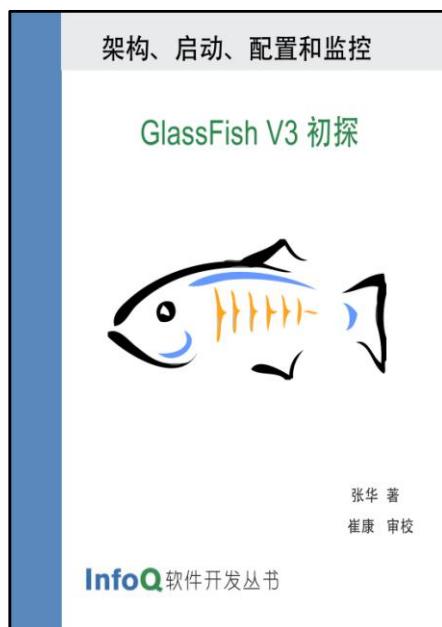
张华 著

崔康 审校

InfoQ 软件开发丛书

免费在线版本

(非印刷免费在线版)



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品



本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 软件开发系列图书。

本迷你书主页为

<http://infoq.com/cn/minibooks/glassfish-v3-glance>

目录

引言	1
第一章 GlassFish 简介与架构	2
1.1 V3 主要特点	2
1.2 V3 基于 HK2 OSGi 的架构	2
1.3 HK2 是什么	3
1.4 GlassFish 的域结构 (Domain)	7
第二章 v3 启动过程与 ClassLoader	11
2.1 启动 OSGi 平台	11
2.2 载入 HK2 及 Adapter	12
2.3 载入 AppServerStartup 并启动相关 HK2 服务	13
2.4 Classloader Hierarchy	14
2.5 OSGi Classloader	14
2.6 V3 中 HK2 Service 分类	15
第三章 v3 中相关重要组件研究	17
3.1 配置组件研究	17
3.2 监控组件研究	18
3.3 自定义 CLI 命令研究	23
第四章 应用部署过程研究	26
4.1 实现 Container	26
4.2 添加 Archive Type	26
4.3 创建 Connector Modules	27
4.4 以 EJB 举例说明部署过程	27

第五章 V3 中集成 WEB , EJB , JMS 三大重要模块研究	29
5.1 集成 Web	29
5.2 集成 JMS	33
5.3 集成 EJB	51
第六章 写在最后--我的一点基于 OSGi 与 JMX 的微内核架构设想	53
关于作者.....	53



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

引言

本文从如何运用现有的 Web Container, EJB Container, JMS Container 集成出一个全新的 Java Application Server 出发 , 研究了 GlassFish V3 的部分源代码 , 研究面覆盖了 GlassFish V3 基于 HK2 OSGi 的架构 , 启动过程 , 配置与监控 , classloader 及集成后的统一部署过程 , 与 web, ejb, jms 三大模块的集成。其中重点探讨了 v3 中集成 web, ejb, jms 的过程。由于时间有限 , 本文仅从集成角度研究了 v3 中启动、 classloader 集成相关的源代码 , 对于具体 Container 的实现的认识还有不足 , 希望与应用服务器有兴趣的朋友一起切磋 , 共同进步。

第一章 GlassFish 简介与架构

GlassFish 是 ORACLE 组织开放源代码 J2EE 应用服务器项目，它使用双许可协议：Common Development and Distribution License (CDDL) 与 GNU General Public License (GPL) 。

目前 GlassFish 已经发布了 v1、v2 和 v3 三个版本：

- v1 被作为官方正式的 Java EE 5 的参考实现。
- v2 最重要的特性是 clustering (包括 grouping、load balancing、data replication)。v2 支持 profile 的概念并支持将同一个可执行应用配置成 developer, enterprise 或是 cluster 的 profile。Enterprise Profile 也可以被配置成使用 HADB (High Availability Data Base) 来达到高可用性 (99.999%)。
- v3 是对 v2 主要功能的重新诠释，代码全部移植到了基于 OSGi 模块化的架构上，同时支持 Java EE6 规范。

1.1 V3 主要特点

GlassFish V3 是对 V2 版本主要功能的重新诠释，全部移植到了基于 OSGi 模块化的架构上(代号为 HK2)，同时支持 Java EE6 规范。OSGi 的新架构会使用延迟加载来启动那些被要求使用的服务。因为 V3 是轻量级的部署，所以它可以快速启动而且占用的资源很少。例如，如果你的应用不使用 EJB Container，它就不会被加载。

1.2 V3 基于 HK2 OSGi 的架构

V3 使用了一个代号为 HK2 的模块化容器，同时它也支持 OSGi 容器。HK2、OSGi 与 GlassFish V3 之间的关系可如下图 1-2 所示：

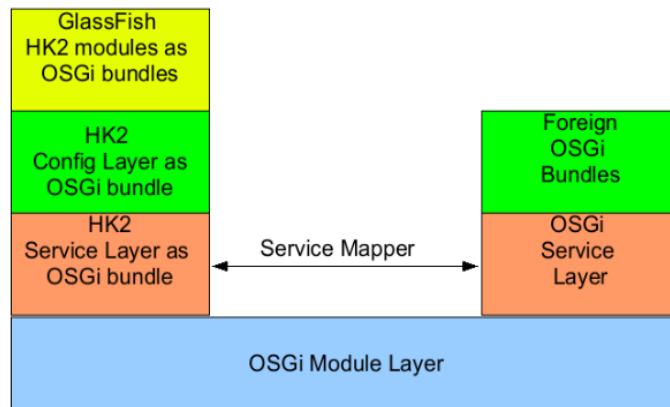


图 1-2 HK2、OSGi 与 GlassFish V3 之间的关系

所有的 GlassFish 包含 JavaEE APIs 模块被打包成 OSGi Bundle。GlassFish 模块不直接使用 OSGi 的 API；他们使用 HK2 API(可以在两种运行机制下使用)。这使得他们可以在没有改变的情况下运行在两种运行机制下。

1.3 HK2 是什么

HK2 是 ORACLE 公司提出的一个类似于 OSGi 的模块化系统规范。HK2 的全称为“Hundred Kilobytes Kernel”，包括 Modules Subsystem 和 Component Model 两部分。据称，该内核将在 JDK 7 中集成，同时，ORACLE 在其开源的 GlassFish J2EE 应用服务器项目 v3 版本中将 HK2 作为其系统内核实现。

模块化系统一般有一个 module repository，在 Equinox OSGi 中，这个 repository 叫做 bundle，它实际上就是加上了元数据描述的 JAR 文件。而 HK2 声称支持下列多种 repository：

- OSGi bundle
- Maven 2 repository
- JAM, module repository of Java SE 7

HK2 的架构图如下图 1-3 所示：

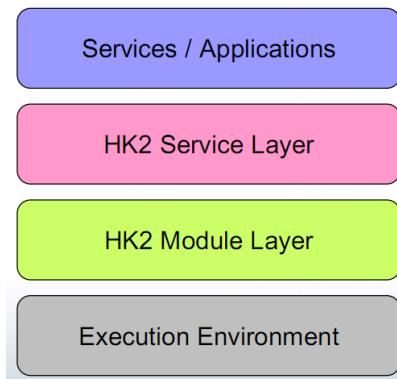


图 1-3 HK2 的架构图

- 模块层-负责类加载和生命周期管理
- 组件层-负责组件的注册，注入等等
- 配置层-负责与 XML 文件同步数据配置

1.3.1 一个小例子直观感受 HK2

我们先看如何开发一个 HK2 服务。

- 1) 接口上使用@Construct 源代码注释
- 2) 实现类上使用@Service 源代码注释
- 3) 依赖注入使用@Inject 源代码注释 (也就是加了@Service 注释的组件)

下面我们来看一个 HK2 Service Client 的例子：

```

@Service
public class MyStartup implements ModuleStartup {
    @Inject Habitat habitat;
    @Inject Foo foo;
    public void setStartupContext(StartupContext context) {
    }
    public void start() {
        Foo lookedUpFoo = habitat.getComponent(Foo.class, null);
        System.out.println("habitat.getByComponent(Foo.class, null) = " + lookedUpFoo);
    }
}
  
```

```

    }

    public void stop(){
        System.out.println("Hello World - My first HK2 Sample Stopped");
    }
}

```

其中 Foo 是一个自定义的 module, 其定义中如下, 着重见红色标注 :

```

<build>
    <plugins>
        <plugin>
            <groupId>com.sun.enterprise</groupId>
            <artifactId>hk2-maven-plugin</artifactId>
            <configuration>
                <archive>
                    <manifestEntries>
                        <HK2-Export-Package>
                            sahoo.hello.api
                        </HK2-Export-Package>
                    </manifestEntries>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>

```

HK2 程序书写完后需要打包 , 打包时比较麻烦的应该是书写 MANIFEST.MF 文件 , 现在我们可以借用 hk2-maven-plugin 来自动生成这个文件。

1) 在 pom.xml 文件中应该有如下片断 :

```

<packaging>hk2-jar</packaging>
<build>
    <plugins>

```

```

<plugin>
    <groupId>com.sun.enterprise</groupId>
    <artifactId>hk2-maven-plugin</artifactId>
    <configuration>
        <archive>
            <manifest>
                <mainClass>
                    com.sun.enterprise.GlassFish.bootstrap.ASMain
                </mainClass>
                <HK2-Export-Package>
                    yahoo.hello.api
                </HK2-Export-Package>
            </manifest>
        </archive>
    </configuration>
</plugin>
</plugins>
</build>
<dependencies>
    <dependency>
        <groupId>com.sun.enterprise</groupId>
        <artifactId>hk2-core</artifactId>
    </dependency>
</dependencies>

```

2) 运行 mvn hk2:run 命令 , OK。

1.3.2 深入 HK2 源码

我们知道 , GlassFish V3 构建在 OSGi 框架之上 , 但是它又不是纯建立在 OSGi 之上的 , 它引入了 HK2 这个内核。 HK2 中有些什么呢 , 有了 HK2 , 使用 OSGi 会更方便吗 , 这些我们都将在本节探索。

在 Maven 存储库中(C:\Documents and Settings\Administrator\.m2\repository\com\sun\enterprise) , 我们发现 :

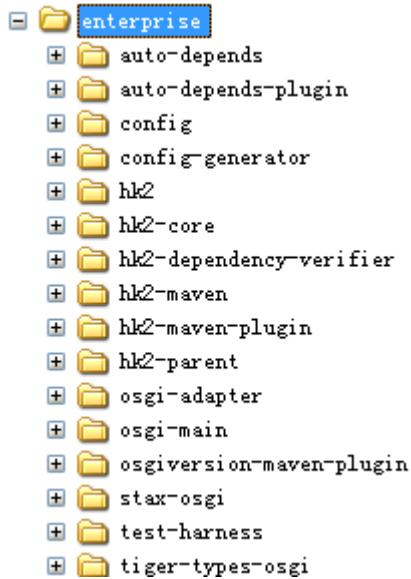


图 1-4 HK2 的目录

从以上目录 , 我们发现 , HK2 主要分为下列三大模块 : Hk2-core Config, 用于方便解析 config.xml 文件 Auto-depend , 提供一些公用包及接口 , 如注解

另外 , HK2 还依赖于下列两个模块 : Stax , 用于解析 XML 的新标准 , 类似于 SAX、DOM Tiger-types, Java SE 7 的 bundle repository 的标准 , 叫做 JAM。在 OSGi 中 , 都是以 JAR 格式封装的 , 在 HK2 中同时支持 JAR、Maven 2 Repository 与 JAM 等。JAM 是 ORACLE 准备在 JAVA 7 中出的新标准。

1.4 GlassFish 的域结构 (Domain)

为了更好的理解 GlassFish 的设计思想 , 我们不得不先提一下 GlassFish 的域结构 , GlassFish 的域结构如下图 1-5 所示 :

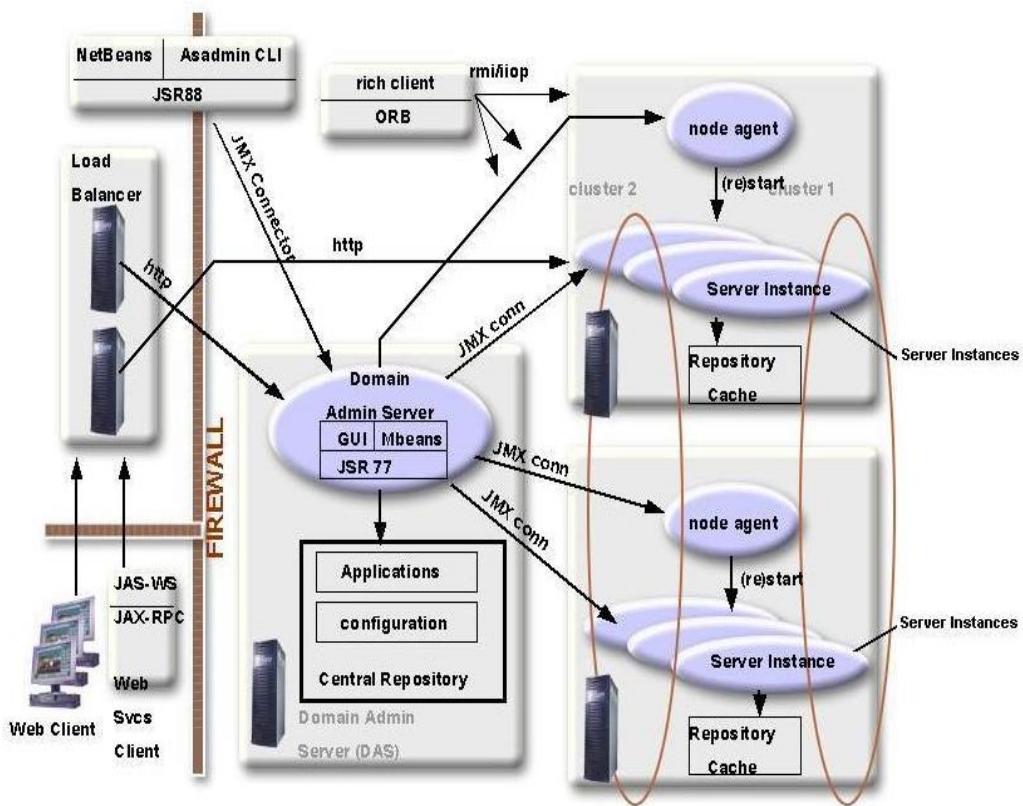


图 1-5 GlassFish 的域结构

这个架构主要由以下一些组件组成：

- **Domain Administrator Server(DAS)**：域管理服务器。域 (Domain) 是 GlassFish 的一个核心概念，我们可以为每个 GlassFish 服务器建立多个域，而域里面又可以建立多个服务实例，但是每个域里面的服 务实例仅仅是为域而服务的，而不能跨域进行服务。DAS 是 GlassFish 的一个核心组件，在集群的环境中，每个 GlassFish 服务器可能会存在多个服务实例 (Server Instance)，DAS 本身也是一个符合 Java EE6 规范的服务实例，主要是为 GlassFish 提供核心管理的功能。所有对域的管理操作，例如 CLI，GUI，Netbeans IDE 以及其他工具的管理请求，都是由 DAS 分发到各个服务器实例去的，而不是直接连接到各个服务器实例。对于一些需要多个实例进行操作的管理请求，DAS 会将操作请求广播到各个实例上去，所以，当 DAS 停止运行后，各种对域的管理操作都不能进行，当然，即使 DAS 已经停止了运行，域的集群和服务器实例仍然可以正常工作，只要域还在正常运行。
- **Administrative Client Applications (CLI , JSR77 , AMX)**：管理客户端包括 asadmin 命令行以及以各种 IDE 的管理模块插件(例如 Netbeans IDE 的管理工具)。这些管理工具都是直接与 DAS 进行通信的，无论是否需要通过防火墙。这些客户端是通过 JMX 连接器来访问

DAS 的。如果需要跨越防火墙 ,连接器实现使用的协议一般是 HTTP 或者 HTTPS。GlassFish 中的管理架构是基于 JMX 技术来构建的。无论是命令行工具还是管理控制台，都是通过服务器端的 MBean 来完成管理功能或相应的服务。另外，也可以使用 AMX 对 GlassFish 进行管理。AMX 是对 JMX 的一个补充，其主要目的是使 MBean 的应用更加方便和面向对象。

- 基于浏览器的 GUI 管理工具。GlassFish 提供了一个非常方便的基于浏览器的 GUI 管理工具，这个管理工具是部署在服务器实例当中的。
- Web 客户端及 Web 服务客户端 :Web 客户端及 Web 服务客户端主要是在浏览器里通过 HTTP 协议或者 Web 服务的调用来与 GlassFish 服务实例进行通信的。在集群环境中，所有的 web 客户端的请求都是经过负载均衡器 (Load Balancer) 来进行请求分发或者故障处理 (Failover) 的。
- 富客户端(RMI/IOP)应用 :富客户端程序通过服务器在客户端生成的客户端存根(stub)用 RMI 或 IOP 协议域服务实体进行远程通信。与 Web 客户端一样，富客户端程序也能通过负载均衡器来进行负载均衡及故障处理 (Failover)。
- 负载均衡器 (Load Balancers)：负载均衡器负责将请求定向到负载量最小的服务实例，检测到失效的节点，适当地重试失效的操作，当会话在某个服务实例上建立后与其维护紧密的联系。当然，除了软件负载均衡器外，还有很多其他的负载均衡解决方案。
- 节点代理 (Node Agent , NA)：节点需要一个轻量级的代理来对服务器实例进行远程生命周期管理。NA 主要负责实例的启动，停止以及创建，同时，也承担监视者以及重启失效进程的责任。和 DAS 一样，NA 也只是进行一些管理的操作，而不需要保证其高可用性。然而，NA 是一个操作系统的系统服务，当本地操作系统启动的时候，它就会一直处于运行的状态。
- 服务器实例 (Server Instance , SI)：一个实例指的是掌管 Java EE 5 应用服务器的 Java 虚拟机。实例间的通信以及和 DAS 之间的通信是依赖于远程 MBean 方法的调用的 (JSR160)。标准的 JSR 160 RMI 连接器用于所有实例与 DAS 的通信。这意味着 DAS 需要管理两种 JMX 连接器——用于管理客户端与 DAS 通信的 HTTP 连接器以及标准的 RMI 管理器。
- 管理接口 :DAS 将 JMX MBean 管理工具的一个子集的接口暴露给用户，这些接口可以用于与 DAS 通过 RMI 连接器进行通信。
- 中央存储库 (Central Repository)：有两个主要的存储库，用于存储实例在域里面共享

的各种信息。其中配置存储库里面存储的是域里面所有的配置信息，而应用程序存储库存储的是在域里面部署的 Java EE 应用程序。中央存储库只能够被 DAS 用 MBean 来维护并持久化到文件系统中。中央存储库被放置在文件系统的一个独立的目录下以便对其进行备份和恢复。

- **本地存储库缓存**：为了减少 DAS 高可用性的限制（因为 DAS 扮演的是一个中央管理者的角色，如果要保证其高可用性的话，代价非常大）以及加速启动的速度，每一个服务器实例在其本地文件系统维护中央存储库的一个缓存。这个缓存在每次实例重启的时候都会与中央存储库进行一次同步。这个缓存是中央存储库的一个子集，用于缓存与该服务器实例相关的一些配置信息或 Java EE 应用。
- **JMX 运行时环境**：JMX 架构提供了用于管理和配置服务器的工具。JMX 运行时环境包括代理服务（MBean 服务器，监视服务等），MBean 应用以及远程连接器等。所有的服务器实例，节点代理以及 DAS 本身都有 JMX 运行时环境。
- **节点**：在这里，一个节点指的是一个独立的主机。注意，一个很大的机器可能会分割为多个独立的节点。
- **管理域（Administration Domain）**：一个域为一个或多个服务器实例提供通用的认证及管理功能。一个管理域内包含多种可管理的资源，包括实例、集群及它们各自资源。需要注意的是，一个可管理的资源，例如一个实例，只能专属于一个域。
- **应用程序（Application）**：Java EE 应用包括 ear，war 及 ejb-jar 文件部署到域中并被域所管理，部署。

第二章 V3 启动过程与 ClassLoader

GlassFish V3 是基于 HK2 与 OSGi 整合之后的架构的，其启动过程分为几步：

- 1) 启动 OSGi 平台
- 2) 载入 HK2 及 Adapter
- 3) Adapter 启动后载入 AppServerStartup 并启动相关 HK2 服务

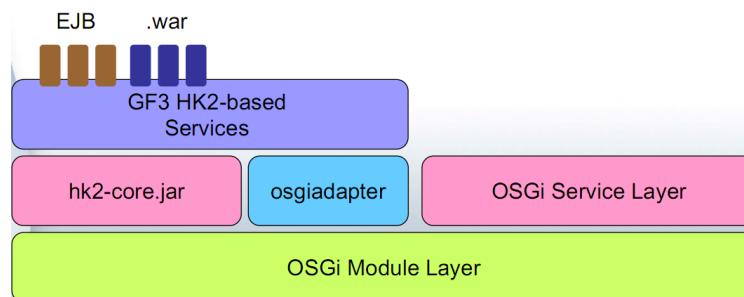


图 2-1 GlassFish V3 的启动过程

2.1 启动 OSGi 平台

入口类为：com.sun.enterprise.GlassFish.bootstrap.ASMain

它位于 GlassFish.jar 中，由于模块化系统中 classloader 的缘故，它并不能直接运行，它必须以 JAR 包的形式运行，方法如下：

```
java -jar GlassFish.jar
```

相关的类位于 GlassFish 子工程下的 com.sun.enterprise.GlassFish.bootstrap 包中,类图如下图 2-2：

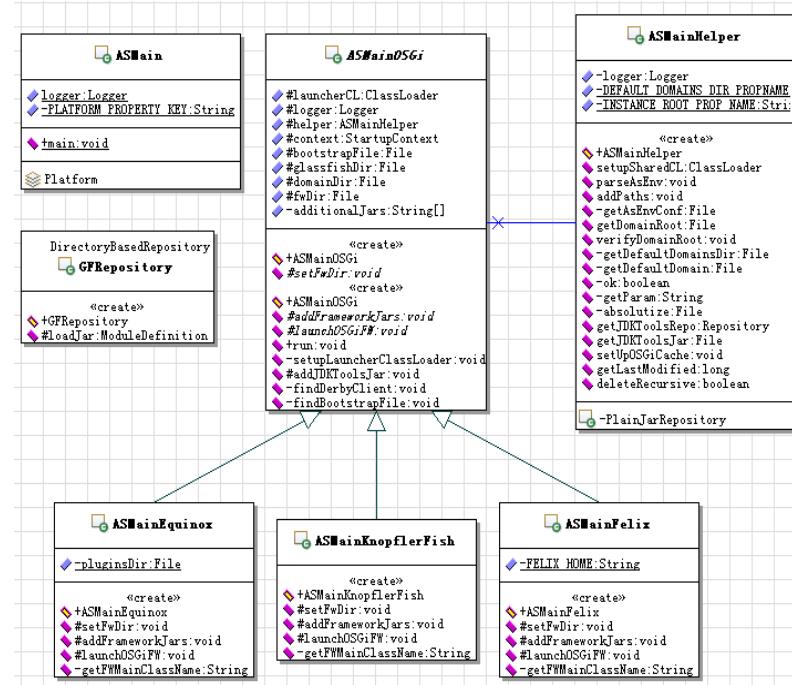


图 2-2 GlassFish V3 启动部分类图

可以看出系统支持 equinox、knopflerFish、felix 三种 OSGi 框架。ASMain 类中通过传进的参数启动相应的 OSGi 框架。

com.sun.enterprise.GlassFish.bootstrap.ASMainFelix 根据 config.properties 启动 Felix，该配置文件关键处如下，可以看到，HK2 被做成了 felix 的一个 bundle：

```

felix.auto.start.1= \
    ${com.sun.aas.installRootURI}/modules/javax.xml.stream.jar \
    ${com.sun.aas.installRootURI}/modules/tiger-types-osgi.jar \
    ${com.sun.aas.installRootURI}/modules/auto-depends.jar \
    ${com.sun.aas.installRootURI}/modules/config.jar \
    ${com.sun.aas.installRootURI}/modules/hk2-core.jar \
    ${com.sun.aas.installRootURI}/modules/osgi-adapter.jar
  
```

图 2-3 V3 启动时要先启动 hk2-core 与 OSGi-adapter

2.2 载入 HK2 及 Adapter

这样，在 Felix OSGi 启动后，会自动启动 hk2-core.jar 与 OSGi-adapter.jar 两个 bundle。

而 hk2-core.jar 的入口类是 Main 类，OSGi-adapter.jar 的入口类是 HK2Main（因为 HK2Main 类实现了 OSGi 框架中控制生命周期的接口 BundleActivator）。

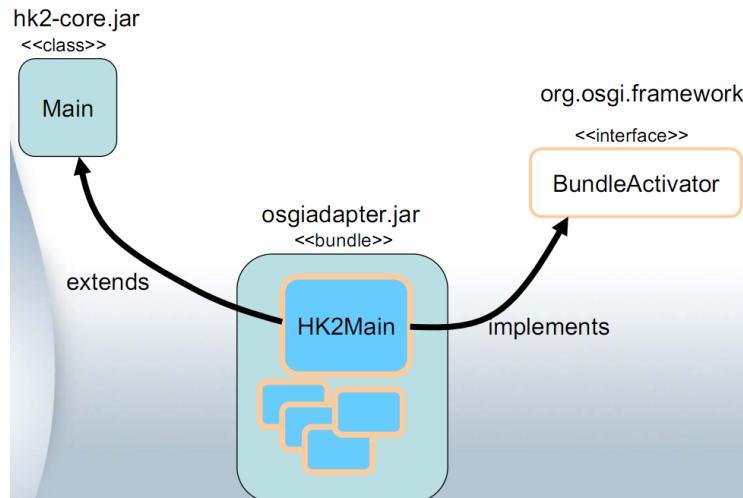


图 2-4 OSGIadapter 既继承了 HK2 又实现了 OSGi 的生命周期接口

2.3 载入 AppServerStartup 并启动相关 HK2 服务

在 OSGIadapter.jar 中的 HK2Main 实现了 OSGi 的 BundleActivator 接口，它载入了 GlassFish 中的 AppServerStartup 服务，而 AppServerStartup(位于 kernel 子工程中)既是 HK2 服务，也是 GlassFish V3 的启动点。

(注意，HK2 里面同样能够通过@Inject 进行依赖注入，当然，我们可以采用开源的 IoC 容器)。

可以看出，启动的是所有实现了 `org.GlassFish.api.Startup` 接口的 HK2 服务，该接口位于 GlassFish-api 子工程中，下列 6 个类实现了该接口：

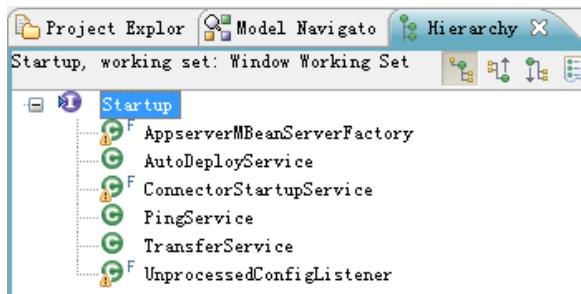


图 2-5 实现了 Startup 接口的 6 个实现类

实现了该接口后就可以像上面一样通过下列语句得到：

```
//run the startup services
final Collection<Inhabitant<? extends Startup>> startups =
habitat.getInhabitants(Startup.class);
```

实现了 HK2 中的 ModuleStartup 接口的类在 HK2 启动时就会被启动，而在 kernel 子工程中的 com.sun.enterprise.V3.server.AppServerStartup 类实现了这个接口，而在该类中会通过 habitat.getInhabitants 取得所有实现了 Startup 接口的类，并启动。

2.4 Classloader Hierarchy

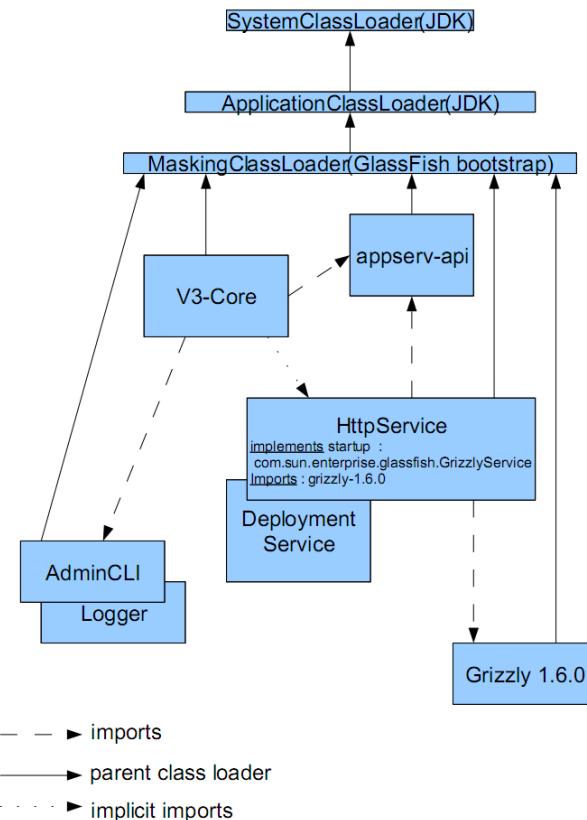


图 2-6 Classloader Hierarchy

- 1) V3-core 导入了 admin-cli、logger modules 等模块
- 2) V3-core 没有导入 HttpService、DeploymentService 模块
- 3) V3-core 仅仅查找实现了 Startup 接口的类。HttpService 与 DeploymentServices 都实现了 Startup 服务接口，它们能够被 HK2 查找，实例，并被注入到 v3-core 中。
- 4) 所有的 modules 有相同的父类加载器。

2.5 OSGi Classloader

OSGi 一个最大的特色就是使用不同的 ClassLoader,让多个 bundle 共享一个虚拟机，而每一个 bundle 使用单独的 ClassLoader。如下图所示：

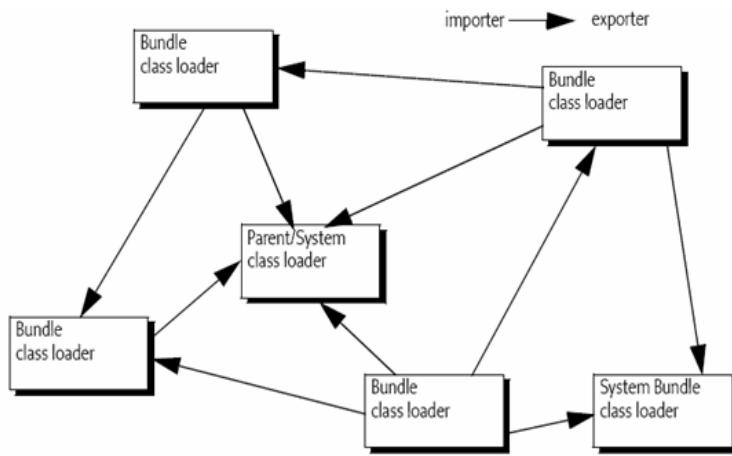


图 2-7 OSGi Classloader

对于资源和类的装载主要包括以下三种途径：

- Boot classpath：启动类环境，包括 `java.*` 包和相应的实现包。
- Framework classpath: OSGi 框架拥有一个独立的 `ClassLoader`，用于装载框架接口类，以及相应的实现类。
- Bundle Space: 每个 bundle 包括了与 bundle 相关的 jar 文件，以及相关的资源。

对于一个类的查询，主要通过以下途径进行查询：

- 从 Parent ClassLoader 中装载
- 从 Import Package 中查询类路径
- 从 Required bundles 中查询类
- 自身 bundle，相关 ClassPath
- 相关的插件片段 Fragment。

2.6 V3 中 HK2 Service 分类

所有的服务接口都在 GlassFish-api 子工程中，如下图所示：

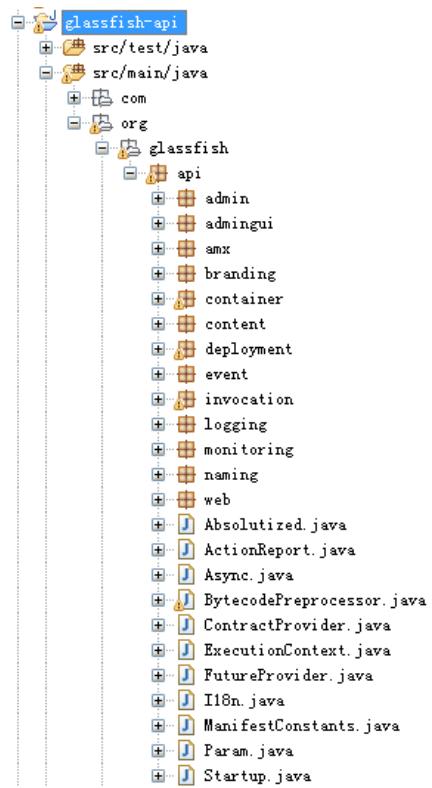


图 2-8 GlassFish V3 中所有的 HK2 服务

以上接口按功能划分为如下几类：

- Startup：该服务负责启动 GlassFish
- Sniffer：该服务用于鉴别应用应该使用哪种部署器，如是 EJB，还是 WEB 等
- ArchiveHandler：鉴别应用类型（WAR, EAR...）
- Deployer：部署服务
- AdminCommnd：CLI 管理命令接口服务

第三章 V3 中相关重要组件研究

3.1 配置组件研究

在 GlasFish 中，有大量的配置文件，很大一部分是自动产生的，为助于理解，我们有必要先将它吃透。它也有一个主域配置文件（位于%GlassFishV3%\domains\domain1\config\config.xml），它主要包括以下几个部分：

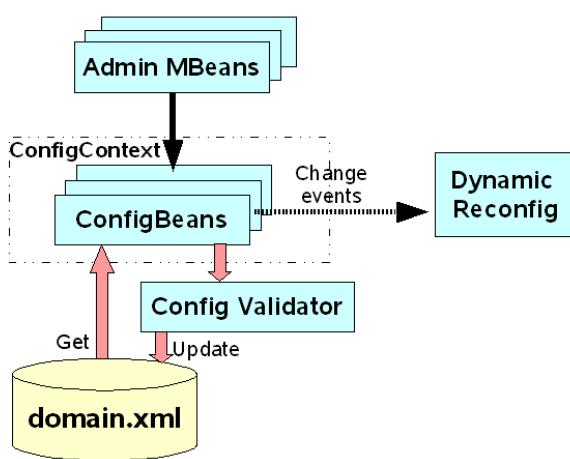


图 3-1 GlassFish config 配置管理组件的组成

- ConfigBeans: 对于对配置文件的 elements、attributes、properties 执行 get/set/create/delete 操作，是一个底层 API
- Admin MBeans: JMX 客户端 API，包括:GUI/CLI/HTML adapters/remote apps.
- Admin Validator: 验证配置，并且防止不合法的配置改变
- Dynamic-reconfig: 重新配置之后，可以不用重启服务器，实时生效

3.1.1 配置元数据

在 GlassFish 的 config-api 模块中包含了大量的配置元数据，有些是自动产生的，有些是手动产生的。

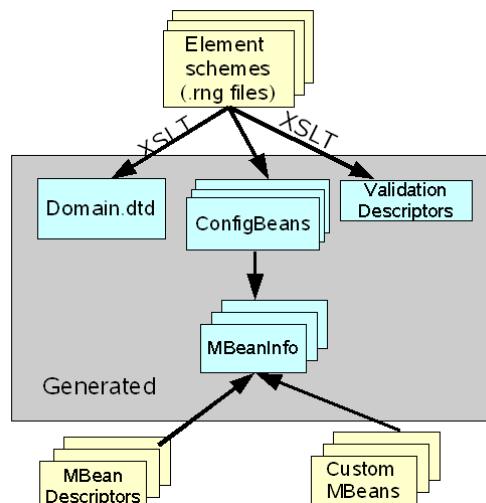


图 3-2 配置元数据

配置文件符合以下三个 schema, 这些 schema 主 Relax NG 语法来进行属性扩展 ,所有的属性扩展在其独立的名空间中 :

- sun-domain-1_2.dtd,
- ConfigBeans
- Validation Descriptors file

3.1.2 源代码分析

- Config Beans: 代码位于 HK2 核中的 config 模块 , 它包含了 ConfigContext、ConfigEvent、Listeners 的接口和实现。在 GlassFish 中的 config-api 模块中的 server beans 是由 sun-domain_1_2.dtd 自动产生的 , 手动产生的是 ApplicationsHelper, ClustersHelper, ResourcesHelper 等。
- Admin MBeans: 提供 JMX 客户端 (CLI/GUI/HTML-adapters/remote apis) 的访问和管理操作。位于 admin-core 中的 admin 子模块中。
- Config Validator : 验证 metadata files , 验证基础类 , 对客户化配置元素的验证等。

3.2 监控组件研究

我们可以通过以下 5 种方式访问 GlassFish 资源 :

- 管理控制台
- 命令行工具 asadmin

- Jconsole 第三方工具
- 标准的 JMX 编程接口
- 面向对象的 AMX 编程接口

JMX 是 GlassFish 管理架构的基础，GlassFish 的设计和实现都遵循了 JMX 规范，因此也完全支持 JMX。这种支持体现在它的命令行管理工具 asadmin 和管理控制台的功能上，体现在第三方管理工具比如 JConsole 的对其访问的支持上，也体现在通过标准的或 GlassFish 特有的编程接口 AMX 对其资源的访问方式上。下面通过实例，来看看 GlassFish 是如何支持这几种方式对其资源进行访问的。

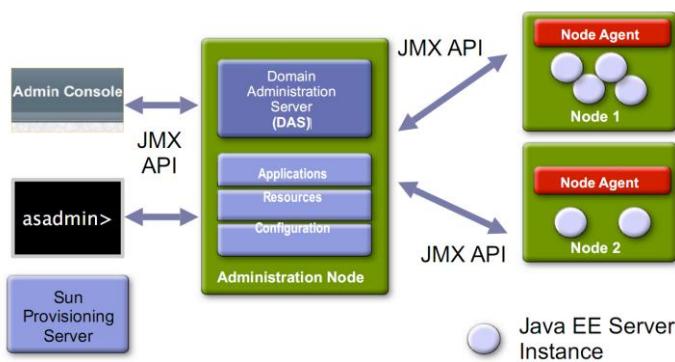


图 3-3 访问 GlassFish 资源的 5 种方式

方法 1，通过管理控制台

首先，先通过管理控制台来创建一个数据库连接池 mypool。在浏览器的输入管理控制台地址：localhost:4848。采用缺省的用户名“admin”及其密码“adminadmin”登录。成功登录后，在左边的树型菜单中，展开“资源” - “JDBC” - “连接池”。在主面板中，点击“新建”。在面板“新建 JDBC 连接池（步骤 1，共 2 步）”中，输入“名称”为 mypool，“资源类型”选为“javax.sql.DataSource”，“数据库供应商”选为“JavaDB”。在接下来的“新建 JDBC 连接池（步骤 2，共 2 步）”中，可以看到数据库连接池的各项缺省设置。将在“池设置”一栏中的“空闲超时”值由缺省的 300 改为 777。点击“完成”。至此，我们通过管理控制台完成了对数据库连接池 mypool 的创建，并修改了其空闲超时的值。

方法 2，通过命令行工具 asadmin

接下来，我们通过命令行的 asadmin 来查看这一资源。

```
asadmin list server.resource*
```

运行结果如下：

```

server.resource-ref.jdbc/_CallFlowPool
server.resource-ref.jdbc/_TimerPool
server.resource-ref.jdbc/_default
server.resources
server.resources.jdbc-connection-pool.DerbyPool
server.resources.jdbc-connection-pool._CallFlowPool
server.resources.jdbc-connection-pool._TimerPool
server.resources.jdbc-connection-pool.mypool
server.resources.jdbc-resource.jdbc/_CallFlowPool
server.resources.jdbc-resource.jdbc/_TimerPool
server.resources.jdbc-resource.jdbc/_default

```

这里列出的 MBean 是用 GlassFish 自己的 DottedName 来标识的。接着通过 asadmin 的子命令 get 来查看对象 mypool 的属性：

```
asadmin get server.resources.jdbc-connection-pool.mypool.*
```

或者进一步查看空闲超时(idle-timeout-in-seconds)的属性值。

```
asadmin get server.resources.jdbc-connection-pool.mypool.idle-timeout-in-seconds
```

结果如下：

```
server.resources.jdbc-connection-pool.mypool.idle-timeout-in-seconds = 777
```

至此，我们完成了使用命令行的管理工具 asadmin 对 mypool 的访问。这里 asadmin 通过 GlassFish 扩展的 Dotted Name 命名方式来访问 MBean 的。Dotted Name 是 GlassFish 命令行工具 asadmin 定义的一套约定。在这套约定的支持下，asadmin 的三个子命令(list、set 和 get)可以通过一个由“.”分隔的字串寻址到 GlassFish 中的 MBean。

方法 3，通过第三方工具 JConsole

接下来，我们要通过 JConsole 来访问对象 mypool。在 JConsole 的登录面板中，选择远程进程：localhost:8686(8686 是 GlassFish 缺省的管理端口)，用户名同样为 admin，密码 admin。登录进来后所看到的是关于 GlassFish 应用服务器运行时的信息，点击“MBean”。展开树型结构 “com.sun.appserv” - “jdbc-connection-pool” - “my pool” - “config” - “属性”。可以看到我们所关心的连接池 mypool 的信息。属性 idle-timeout-in-seconds 的值为 777。修改 777 为 888。在回到管理控制台或命令行工具 asadmin 同样可以看到刚才在 JConsole

所作的修改已经生效。以上说明三种工具对 GlassFish 资源的修改是等效的。接下来通过编程的方式来访问数据库连接池 mypool。

方法 4，通过标准的 JMX 编程方式

标准的 JMX 方式的代码如下：

```

import javax.management.*;
import javax.management.remote.*;
public class JMX_demo {
    public JMX_demo() throws Exception {
        //创建 JMX 的 URL
        JMXServiceURL url = new
JMXServiceURL("service:jmx:rmi://jndi/rmi://localhost:8686/jmxrmi");
        java.util.Map env = new java.util.Hashtable();
        //缺省用户名和其口令
        String[] creds = {"admin","adminadmin"};
        env.put(JMXConnector.CREDENTIALS,creds);
        //建立连接
        JMXConnector connector = JMXConnectorFactory.connect(url,env);
        MBeanServerConnection mbsc = connector.getMBeanServerConnection();
        //要访问的 MBean 的 Object Name
        ObjectName mbeanName = new
ObjectName("com.sun.appserv:type=jdbc-connection-pool,name=mypool,category=config");
        //所要访问的属性 idle-timeout-in-seconds
        System.out.println("Using JMX, jdbc pool idle timeout:"+
mbsc.getAttribute(mbeanName,"idle-timeout-in-seconds"));
    }
    public static void main( final String[] args ) throws Exception{
        new JMX_demo();
    }
}

```

运行结果如下： Using JMX, jdbc pool idle timeout:888

方法 5，通过 AMX 编程方式

AMX 方式的代码如下：

```
import com.sun.appserv.management.DomainRoot;
import com.sun.appserv.management.client.AppserverConnectionSource;
import com.sun.appserv.management.client.TLSParams;
import com.sun.appserv.management.util.misc.ExceptionUtil;
import com.sun.appserv.management.config.*;
import java.net.ConnectException;
import java.util.Map;

/** * 此类为演示使用 AMX 方式访问服务器端的 MBean 的演示代码。 */
public class AMX_demo {
    public AMX_demo() throws Exception {
        //Domain Admin Server 的机器名或 IP 地址
        final String host = "localhost";
        //JMX 管理端口，缺省 8686。
        final int port = 8686;
        //管理员名
        final String user = "admin";
        // 管理员密码
        final String password = "adminadmin";
        TLSParams tlsParams=null;
        //连接到 JMX server
        AppserverConnectionSource conn = new AppserverConnectionSource(
            AppserverConnectionSource.PROTOCOL_RMI, host, port, user,      password,
            tlsParams, null);
        conn.getJMXConnector( true );
    }
}
```

```

//DomainRoot 和 JDBCConnectionPoolConfig 就是所说的 DCP 组件
DomainRoot mDomainRoot = conn.getDomainRoot();

        //获取 JDBCConnectionPool 的列表

        Map pools =
mDomainRoot.getDomainConfig().getJDBCConnectionPoolConfigMap();

        JDBCConnectionPoolConfig mypool =
(JDBCConnectionPoolConfig)pools.get("mypool");

        System.out.println("Using DCP, jdbc pool idle timeout:
"+mypool.getIdleTimeoutInSeconds());

    }

    public static void main( final String[] args ) throws Exception{
        new AMX_demo();
    }
}

```

运行结果如下：

Using DCP, jdbc pool idle timeout: 888

注意，采用 AMX 的方式时，在项目的库路径上要加入 appserv-ext.jar 和 javaee.jar。

3.3 自定义 CLI 命令研究

AdminCommand 也是一个被标注了@Contract 的接口，如果我们要自定义命令接口的话就要实现它。如果要注入的话就用@Inject 注解，如果要将一个字段变成参数的话就用@param 注解。例如：

```

Package test;

@Service(name="aaa")

public class MySuperCommand implements AdminCommand {

    @Inject
    Domain domain;

    @Param
    String name
}

```

```
@Param(optional=true)  
String myoptionalparam  
  
@Param(default=true)  
String path  
  
...  
}
```

可以按照以下方式使用命令：

```
asadmin aaa --path foo.war  
asadmin aaa foo.war
```

@Param 参数除了以上的在命令里指定以外，也可以在 classpath 中去找一个 LocalStrings.properties 文件，它的格式如下：

```
test.MySuperCommand=Some random super command  
test.MySuperCommand.path=path to the file the command applies to
```

QClub

我们影响有影响力的人

北京 上海 广州 大连 西安 太原 成都 杭州 武汉 南京 深圳...

QClub

邀请
业内知名专家

自由开放的
讨论氛围

定期举办的线下活动

结识
圈内技术好友

InfoQ



中文 | 英文 | 日文 | 葡文 |

第四章 应用部署过程研究

应用部署和下列三类服务有关：

- Sniffer：这类服务负责鉴别 archive 由哪类 deployer 部署。
- Deployer：这类服务负责将 archive 发布到特定的 Container。
- ArchiveHandler：这种服务来识别 application bundle types (WAR, EAR...)

4.1 实现 Container

实现一个容器必须同时用@Service 注解及实现 Container 接口，Container 接口定义如下：

```

@Contract
public interface Container{
    public Class<? extends Deployer> getDeployer();
    public String getName();
}

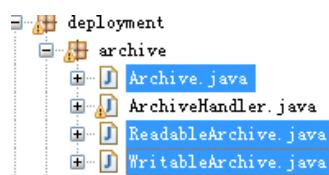
```

- getDeployer() 方法返回一个实现了 org.GlassFish.api.deployment.Deployer 接口的实现类。 Deployer 接口用于管理运行此容器的应用。
- getName() 方法返回一个人类可读的容器名称

Deployer 接口定义如下,其中 DeploymentContext 对象是一个在部署期间有效的 context 对象。

4.2 添加 Archive Type

Archive type 是对 archive 文件格式（如 jar , war 等）的一种抽象，与此相关的三个接口是：



实现了 ReadableArchive 接口，将提供对 archive 的读的功能。实现了 WritableArchive 接口，将提供对 archive 的写的功能。archiveHandler 相当于一个处理 archive 的工具类，每种类型

的 archive (war, jar, rar 等) 对应一个。

4.3 创建 Connector Modules

一个 Connector Modules 是一个基于 HK2 的 service , 只不过这个 service 里有 Sniffer 用于鉴别一个 archive byte 由哪个 Container 来部署。其流程如下 :

- 1) 应用服务器收到一个 deployment request
- 2) 当前的 Sniffer 实现类将决定是哪一种 archive type
- 3) 接着识别出这类 archive type 由哪类 Container 来部署 , 只有在确定是哪类 Container 来部署之后 , 应用服务器调用 Sniffer 中的 setUp 方法来安装 Container, setUp 方法最终返回该 Container 的 HK2 Module
- 4) 容器中的 Deployer 接口的实现了负责最终部署

由于 Sniffer 实际上是一个 HK2 Service , 所以 V3 会知道它。

4.4 以 EJB 举例说明部署过程

4.4.1 OpenEJB 中是如何部署 EJB 的

OpenEJB 是一个基于 OSGi 的开源 EJB Container , 部署时不仅得写 EJB 还得写 Bundle。

4.4.2 V3 中是如何部署 EJB 的

我们认为 , GlassFish V3 作为应用服务器 , 已经将写 Bundle 的步骤在服务器里自己做了 , 我的想法得到了验证。

- 1) 在 Sniffer 接口中按 F4 , 找到 EJBSniffer , 显然它是根据 org.GlassFish.ejb.startup.EjbContainerStarter 启动的
- 2) org.GlassFish.ejb.startup.EjbContainerStarter 类中 ,

```
public Class<? extends org.GlassFish.apt.deployment.Deployer> getDeployer(){  
    return EjbDeployer.class;  
}
```

3) 在 EjbDeployer 中 ,

```
public EjbApplication load(EjbContainerStarter ContainerStarter, DeploymentContext dc){  
    .....  
    EjbApplication ejbApp = new EjbApplication(ebds, dc, dc.getClassLoader());  
    .....  
    return ejbApp;  
}
```

EjbApplication 本来是就是一个实现了 HK2 的 Service , 它实际上往 OSGi 里注册服务的功能 , 至此 , 我的想法得到了验证。

第五章 V3 中集成 WEB , EJB , JMS 三大重要模块研究

5.1 集成 Web

GlassFish V3 中集成 Grizzly 作为它的 Web Container, 如下图 6-1 所示 ,Grizzly 是一个基于 NIO 的应用程序框架 , 开始目的是建构一个 HTTP Web 服务器 , 用来代替 Tomcat 的 Coyote 连接器和 Sun WebServer 6.1.

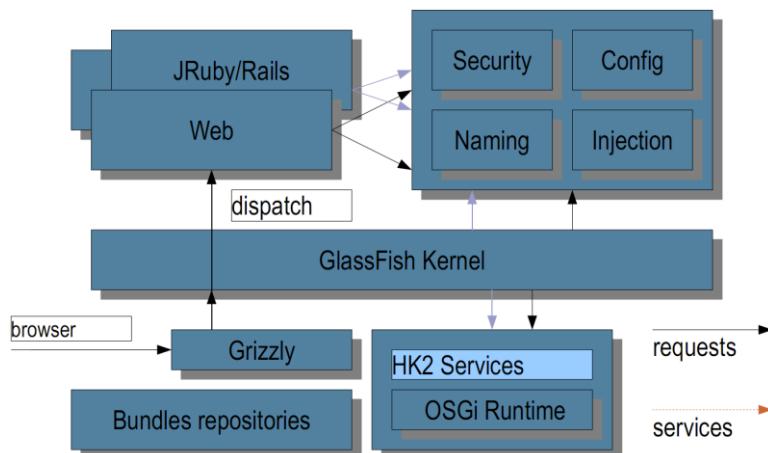


图 6-1 GlassFish V3 集成 Grizzly 架构图

5.1.1 集成 Grizzly

HttpService 接口定义如下 , 该接口可通过源代码注释(@Configured 与 @Element)从 config.xml 文件中读取上述配置), 并且该接口也用了@AMXConfigInfo 这个源代码注释 , 意味着它也能被 JMX 管理哦 :

```

@AMXConfigInfo(amxInterfaceName="com.sun.appserv.management.config.HTTPServiceCo
nfig", singleton=true)

public interface HttpService extends ConfigBeanProxy, Injectable, PropertyBag{

    @Element

    public AccessLog getAccessLog();

}
    
```

5.1.2 集成 Tomcat

在 tomcat-connector 子工程中 , 有一个 TomcatContainer , 如下 :

```

@Service(name="web")
public class TomcatContainer implements Container, PostConstruct, PreDestroy{

```

从@Service 可以看出，它是一个名为 web 的 bundle。

启动 tomcat 非常简单，伪码如下：

```

Embedded tomcat = new Embedded();
.....//根据配置文件生成 tomcat 对象
tomcat.start();           //启动 tomcat

```

所以问题的关键是如何根据配置文件生成 tomcat 对象, tomcat 对象的结构如下图所示：

```

<Server>
  <Service>
    <Connector/>
    <Engine>
      <Host>
        <Context>
        </Context>
      </Host>
    </Engine>
  </Service>
</Server>

```

5.1.3 V3 中集成 tomcat 小实验

- 1) 在 eclipse 中建 plugin-project, 名为 tomcatBundle
- 2) 代码：

```

package org.GlassFish.custom.web;
import org.apache.catalina.Context;
import org.apache.catalina.Engine;
import org.apache.catalina.Host;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.connector.Connector;
import org.apache.catalina.startup.Embedded;

```

```

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
public class StartTomcat implements BundleActivator {
    public void start(BundleContext context) throws Exception {
        final Embedded tomcat = new Embedded();
        tomcat.setCatalinaHome("E://test");
        System.out.print(tomcat.getCatalinaHome());
        Engine engine = tomcat.createEngine();
        engine.setName("WebServer");
        Host host = tomcat.createHost("localhost", tomcat.getCatalinaHome() +
        "/webapps");
        engine.addChild(host);
        engine.setDefaultHost(host.getName());
        Context ctxtRoot = tomcat.createContext("/", host.getAppBase() + "/ROOT");
        System.out.println("Context: " + ctxtRoot.toString());
        ctxtRoot.setPrivileged(true);
        host.addChild(ctxtRoot);
        tomcat.addEngine(engine);
        Connector connector = tomcat.createConnector((java.net.InetAddress) null, 8180,
        false);
        tomcat.addConnector(connector);
        tomcat.start();
        System.out.println("tomcat 已经成功启动!");
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                try {
                    tomcat.stop(); //停止 TOMCAT
                } catch (LifecycleException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
  
```

```
        }
    });
}

public void stop(BundleContext context) throws Exception {
}

}
```

3) MANIFEST.MF 文件 :

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: TomcatBundle Plug-in
Bundle-SymbolicName: tomcatBundle
Bundle-Version: 1.0.0
Bundle-Activator: org.GlassFish.custom.web.StartTomcat
Bundle-Vendor: zhang hua
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.OSGi.framework;version="1.3.0"
Bundle-ClassPath: lib/catalina-optional.jar,
lib/catalina.jar,
lib/commons-el.jar,
lib/commons-logging.jar,
lib/commons-modeler-2.0.1.jar,
lib/jasper-compiler-jdt.jar,
lib/jasper-compiler.jar,
lib/jasper-runtime.jar,
lib/jsp-api.jar,
lib/naming-factory.jar,
lib/naming-resources.jar,
lib/servlet-api.jar,
lib/servlets-default.jar,
```

```
lib/tomcat-coyote.jar,
lib/tomcat-http.jar,
lib/tomcat-util.jar,
```

4) 测试用的 WEB 应用：

Index.jsp 的内容：test

Web.xml 的内容如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

5) 在 GlassFish 中启动该 bundle 即可。

5.2 集成 JMS

因为都是 bundle 化的，在 v3 中可以很容易的集成 JMS，如 OpenMQ 或者 activeMQ。

5.2.1 OpenMQ

5.2.1.1 介绍 openmq

Openmq 是一款功能齐全的消息中间件，其组件结构图如下图 5-2 所示：

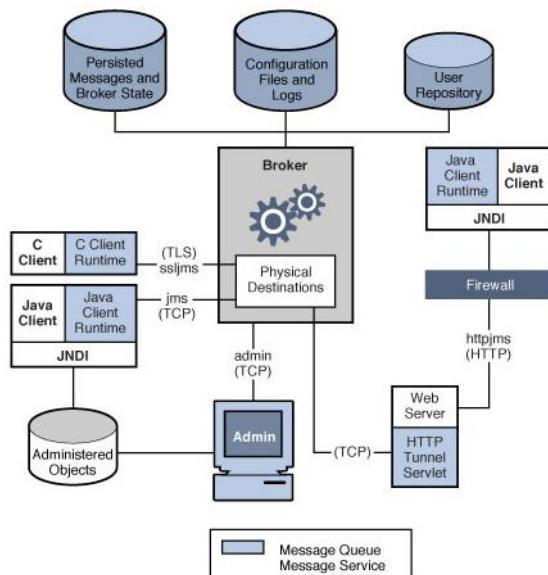


图 5-2 Openmq 组件结构图

- Broker : 它是 MQ 的基础设施 , 它管理所有的消息制造与消息消费 , 它还管理着一些 message desitinations , 这些 desitinations 能被配置成 queue 或者是 topic desitination
- Admin : Open MQ 提供一个简单的内建的管理 GUI , 像一般的启动和停止 broker, 建立 destination。对于更复杂的应用 , 提供了 JMX API 来管理和监视 MQ。能通过这个接口建立自己的管理控制台。
- 消息持久化 : MQ 提供了两个基础的消息存储 (File-Store and JDBC data-source) 。另外 , 在 GlassFish 中用 HADB 来存储消息。
- Java Client: 提供 JAVA 客户端。
- C-Client : 提供 C 客户端。
- Http Client : 提供了通过防火墙的远程访问能力。

另外 , Open MQ 支持集群 , 其结构如下图 5-3 所示 :

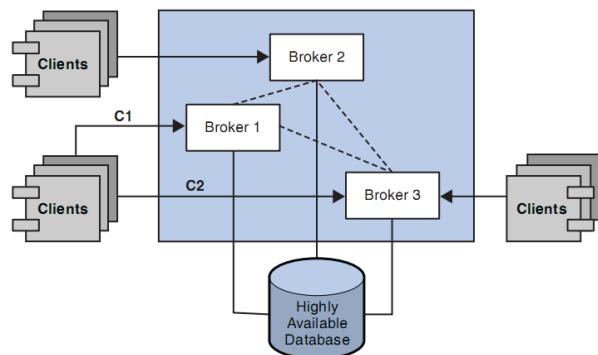


图 5-3 Open MQ 集群配置图

所有的 Broker 共享 HADB 中的配置信息 , HADB 是 ORACLE 经过优化的高性能数据库。Message EJB 可以消费 MQ 中的消息 , 如下图 5-4 所示 :

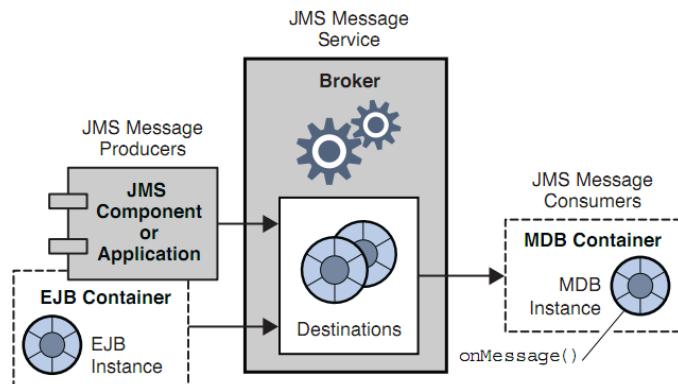


图 5-4 Message EJB 消费 MQ 中消息示意图

5.2.1.2 试用 openmq

用 openmq 写代码与用 activeMQ 写代码除了建立连接工厂的一步不一样以外其他全一样，所以此处的测试代码就不再列出，参见下节“试用 activeMQ”的测试代码，代码要导入 openmq 里的三个 JAR 包，分别是：fscontext.jar, imq.jar, jms.jar。

在 activeMQ 中建立连接工厂的代码是：

```
ConnectionFactory connectionFactory =
new ActiveMQConnectionFactory(
    ActiveMQConnection.DEFAULT_USER,
    ActiveMQConnection.DEFAULT_PASSWORD,
    "tcp://192.168.1.51:61616");
```

而在 openmq 中建立连接工厂的代码是：

```
ConnectionFactory connectionFactory = (javax.jms.ConnectionFactory)
ctx.lookup("quqiConnectionFactory");
```

从上面我们可以看出，我们应该自己配置一个对象存储库，对象存储库存储在 <file:///C:/Temp> 目录下，它的 INITIAL_CONTEXT_FACTORY=com.sun.jndi.fscontext.RefFScontextFactory，这些全是要配置的，另外还要配一个名为 quqiQueue 的 queue destination，下面看看如何在控制台配置这些信息：

- 1) 下载 openmq。http://download.java.net/mq/open-mq/4.2/b17/mq4_2-installer-WINNT_20080627.zip
- 2) 解压后，双击 installer.vbs 开始安装，一路下一步，略。
- 3) 启动 MQ 管理控制台，双击%OpenMQ%\bin\imqadmin.exe
- 4) 启动 Broker, 双击%OpenMQ%\bin\imqbrokerd.exe
- 5) 将 Broker 加入到 MQ 管理控制台以便管理。在 MQ 管理控制台中左侧的“代理”处点击“添加代理”操作(用户名和密码均是 admin)，如下图：

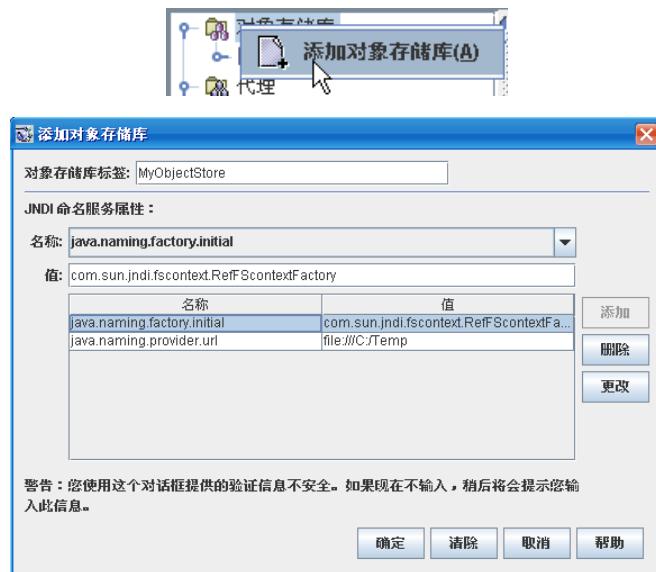




6) 配置一个对象存储库，其信息为：

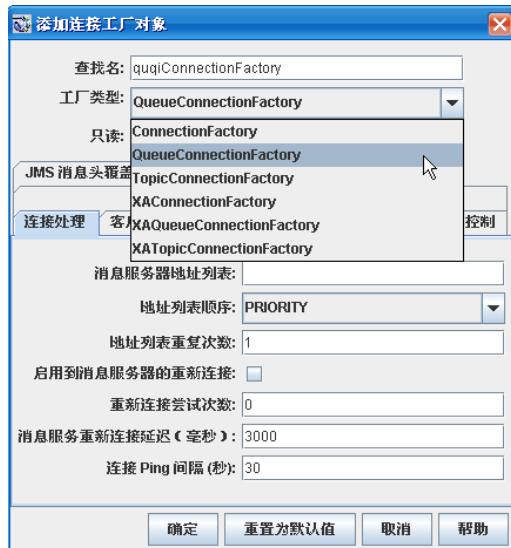
PROVIDER_URL=file:///C:/Temp

INITIAL_CONTEXT_FACTORY=com.sun.jndi.fscontext.RefFscontextFactory,



7) 在该对象存储库上再新建一个名为“quqiConnectionFactory”的连接工厂：





8) 在该对象存储库上再新建一个名为"quqiQueue"的目的地：



9) 配置完毕，直接运行程序即可。OK。

5.2.1.3 在 GlassFish 中如何开发一个 JMS 应用

在 GlassFish 中，可以以两种方式访问连接工厂和目的地：使用 Java 命名和目录接口(Java Naming and Directory InterfaceTM , JNDI)查找或使用注解。

1) JNDI 查找

JMS 客户机使用 JNDI API 查找连接工厂和消息目的地。

```
InitialContext jndi = new InitialContext();
// Lookup queue connection factory
QueueConnectionFactory qFactory = (QueueConnectionFactory)jndi.
```

```

lookup("webTrackerConnFactory");
// Lookup queue
Queue queue = (Queue)jndi.lookup("webTrackerQueue");

```

2) 注释

注释 @Resource 用于查找连接工厂和目的地。在 Web 应用程序中，如果注释放在变量上，那么 servlet 容器将注入请求的资源；也就是，注释变量将在为请求提供服务之前使用适当的值进行预先填充。

```

@Resource(name="connFactory", mappedName="webTrackerConnFactory")
private QueueConnectionFactory qFactory;
@Resource(name="jmsQueue", mappedName="webTrackerQueue")
private Queue queue;

```

除了以上的在获取连接工厂不一样以外，代码中其他的与上节中的“试用 OpenMQ”与下节中的“试用 activeMQ”的写法是一模一样的，略。下面关键看看如何在控制台中配置连接工厂及 queue。

进入 GlassFish V3 的 WEB 控制台：<http://localhost:4848/> 在控制台中建连接工厂，建 queue：



New JMS Connection Factory

The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the factory and a connector resource.

General Settings

JNDI Name: *	<input type="text" value="jms/GlassFishBookConnectionFactory"/>
Resource Type: *	<input type="text" value="javax.jms.ConnectionFactory"/>
Description:	<input type="text" value="Used for book examples."/>
Status:	<input checked="" type="checkbox"/> Enabled

Pool Settings

Initial and Minimum Pool Size:	<input type="text" value="8"/> Connections	Minimum and initial number of connections maintained in the pool
Maximum Pool Size:	<input type="text" value="32"/> Connections	Maximum number of connections that can be created to satisfy client requests
Pool Resize Quantity:	<input type="text" value="2"/> Connections	Number of connections to be removed when pool idle timeout expires
Idle Timeout:	<input type="text" value="300"/> Seconds	Maximum time that connection can remain idle in the pool
Max Wait Time:	<input type="text" value="60000"/> Milliseconds	Amount of time caller waits before connection timeout is sent
On Any Failure:	<input type="checkbox"/> Close All Connections	
Transaction Support:	<input type="text"/>	
Connection Validation:	<input type="checkbox"/> Required	

Home Version

User: admin Domain: domain1 Server: localhost Logout Help

Sun Java™ System Application Server Admin Console

Resources > JMS Resources > Destination Resources

JMS Destination Resources

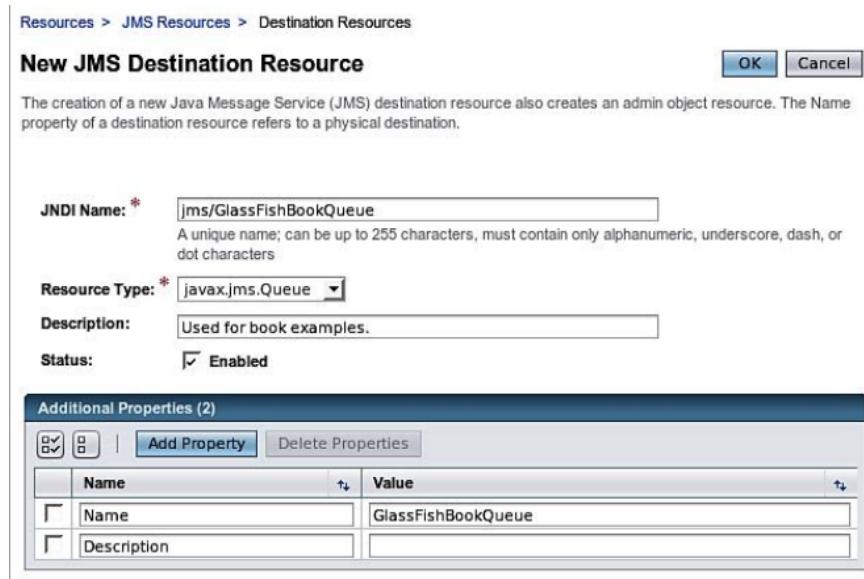
JMS destinations serve as the repositories for messages. Click New to create a new destination resource. Click the name of a destination resource to modify its properties.

Destination Resources (0)

New	Delete	Enable	Disable
JNDI Name	Enabled	Resource Type	Description
No items found.			

Common Tasks

- Application Server
- Applications
 - Enterprise Applications
 - Web Applications
 - EJB Modules
 - Connector Modules
 - Lifecycle Modules
 - Application Client Modules
- Web Services
- JBI
- Custom MBeans
- Resources
 - JDBC
 - JMS Resources
 - Connection Factories
 - Destination Resources
 - JavaMail Sessions



5.2.1.4 通过 JCA 集成 OpenMQ

OpenMQ 有三种启动模式：

- EMBEDDED：在 GlassFish 中可以嵌入式启动 openmq，其他环境启动不了。这种方式启动时，GlassFish 与 openmq 在同一个 JVM 中运行，openmq 自动跟着 DAS 启动或者停止。此种方式不支持集群。
- LOCAL：就是启动 imqbrokerd 命令的方式，当然在程序里也可以调用此命令启动。此方式支持集群，当创建一个 GlassFish server cluster 时，一个 message queue cluster 也自动被创建了。这是所有 GlassFish service instance 的默认模式。
- REMOTE：这种方式 MQ 必须被独立启动。

GlassFish 是通过 JCA(用 resource adapter)集成 openmq 的，遵守 JCA 规范的连接器被称作资源适配器 (resource adapter)。每个资源适配器都被要求支持两套标准接口：一组接口被应用程序服务器使用来与适配器交互作用，JCA 考虑到资源适配器可以把客户端程序作为专有接口的替代另一套由客户/消费者使用与企业信息系统（当然也是通过适配器）相互作用。

我们先简要回顾一下 JCA 的流程与原理：

- 1) Application Component 可以是被 Application Server 管理的组件如 EJB，也可以是未被管理的组件如普通 JAVA 程序。当 Application Component 要通过 JCA 使用 EIS (这里指 openmq) 时，首先要通过部署描述符 ra.xml 描述下列信息：

- Res-ref-name : eis/MyEIS
- Res-type : javax.resource.cci.ConnectionFactory

- Res-auth : Application 或 Container
- 2) Application Component 根据部署描述符里的信息根据 JNDI 查找 ConnectionFactory 实例
- 3) (这一步对于 Application Component 的开发者是透明的 , 既得到了统一的事务、安全、池管理 , 又减少的开发难度 , 这就是 JCA 的好处) ConnectionFactory 这时将创建 Connection 的请求代理给 ConnectionManager 实例 , ConnectionManager 实例收到请求后查找 App Server 提供的连接池 (Connection Pool) , 如果池中没有连接可以满足连接要求 , App Server 则使用 ManagedConnectionFactory 接口 (由 JCA 提供) 创建一个新的与 EIS 的物理连接 , 如果 App Server 在池中找到了匹配的连接 , 则使用匹配的 ManagedConnection 实例来满足连接请求。 Connection Manager 不仅可以提供上面所述的池服务 , 还可以下列各种不同的服务 :
- 事务管理 Transaction Manager
 - 安全 SecurityServiceManager
 - 连接池 PoolManager
 - 错误日志
- 4) Application Component 的开发者通过 Connection Factory 的 getConnection 方法获得实际的物理连接:

```
javax.resource.cci.Connection connection = connectionFactory.getConnection();
```

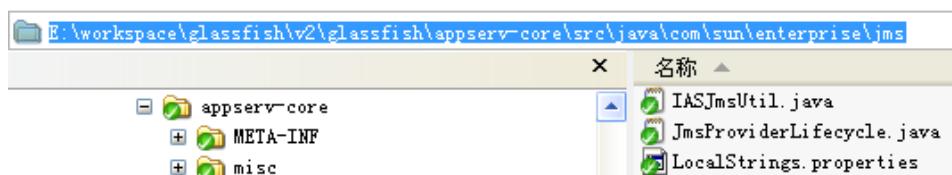
- 5) 当组件完成连接后 , 关闭连接 :

```
connection.close();
```

当知道了上面 JCA 的运行流程之后 , 我们可以很容易知道开发 JCA 究竟要书写哪些类。

具体到 GlassFish 中 , com.sun.enterprise.connectors.system. ActiveJmsResourceAdapter (位于 appserv-core 包中) 实现了 GlassFish 为 OpenMQ 开发的 resource adapter.

这个类中用到了 openmq 的一些工具类 , openmq 的一些工具类全位于 %GlassFish %\appserv-core\src\java\com\sun\enterprise\jms 目录下 , 如下图 :



正因为 glassfish 中有这些 openmq 的工具类，才可以有且只有在 GlassFish 中才能以 Embedded 方式启动 openmq。特别是在 IASJmsUtil 这个工具类中可以通过下列方法得到 openmq 中的：

- com.sun.messaging.jmq.jmsspi.JMSAdminFactory
- com.sun.messaging.jmq.jmsspi.JMSAdmin。

研究此问题将涉及到三块的源代码分别如下：

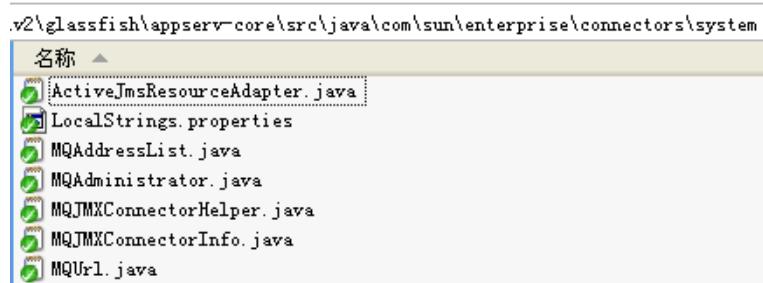
- 1) %GlassFish%\ appserv-core\src\java\com\sun\enterprise\connectors 为应用服务器本身与 JCA 相关的代码：

```

authentication
inflow
system
util
work
ActiveInboundResourceAdapter.java
ActiveOutboundResourceAdapter.java
ActiveRAFactory.java
ActiveResourceAdapter.java
AdministeredObjectResource.java
BootstrapContextImpl.java
ConnectionManagerFactory.java
ConnectionManagerImpl.java
ConnectorAdminObjectAdminServiceImpl.java
ConnectorAdminService.java
ConnectorAdminServicesFactory.java
ConnectorAdminServiceUtils.java
ConnectorConfigurationParserServiceImpl.java
ConnectorConnectionPool.java
ConnectorConnectionPoolAdminServiceImpl.java
ConnectorConstants.java
ConnectorDescriptorInfo.java
ConnectorNamingEvent.java
ConnectorNamingEventListener.java
ConnectorNamingEventNotifier.java
ConnectorRegistry.java
ConnectorResourceAdminServiceImpl.java
ConnectorResourceNamingEventNotifier.java
ConnectorRuntime.java
ConnectorRuntimeException.java
ConnectorSecurityAdminServiceImpl.java
ConnectorServiceImpl.java
DeferredResourceConfig.java
LazyAssociatableConnectionManagerImpl.java
LazyEnlistableConnectionManagerImpl.java
LocalStrings.properties
PoolMetaData.java
ResourceAdapterAdminServiceImpl.java
XATerminatorProxy.java

```

- 2) %GlassFish%\ appserv-core\src\java\com\sun\enterprise\connectors\system 为 openmq 的资源适配器的代码，这是重点：



3) %GlassFish%\appserv-core\src\java\com\sun\enterprise\jms 为 openmq 的一些工具：



为 openmq 提供 JCA 后，openmq 就与 GlassFish 集成了，可以在 GlassFish 的配置页面去使用 openmq 了。

5.2.2 activeMQ

5.2.2.1 试用 activeMQ

使用 activeMQ 相当简单，步骤如下：

- 1) 下载 activeMQ，并解压到任何目录。
- 2) 双击%activeMQ%/bin/activemq.bat 启动 MQ
- 3) 进入 <http://192.168.1.51:8161/admin> 控制台新建一个名称为 quqiQueue 的 queue
- 4) 代码如下（需使用 JAR 包： activemq-all-5.2.0.jar ）：

```
package test.jms;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.DeliveryMode;
import javax.jms.Destination;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;
```

```
public class Sender {  
    public static void main(String[] args) {  
        ConnectionFactory connectionFactory;  
        Connection connection = null;  
        Session session;  
        Destination destination;  
        MessageProducer producer;  
  
        //构造 ConnectionFactory 实例对象，此处采用 ActiveMq 的实现 jar  
        connectionFactory = new ActiveMQConnectionFactory(  
            ActiveMQConnection.DEFAULT_USER,  
            ActiveMQConnection.DEFAULT_PASSWORD,  
            "tcp://192.168.1.51:61616");  
  
        try {  
            connection = connectionFactory.createConnection();  
            connection.start();  
            session = connection.createSession(Boolean.TRUE,  
                Session.AUTO_ACKNOWLEDGE);  
  
            //quqiQueue 是一个服务器的 queue，须在 ActiveMq 的 console 配置  
            destination = session.createQueue("quqiQueue");  
            producer = session.createProducer(destination); //发送者  
  
            // 设置不持久化，此处学习，实际根据项目决定  
            producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);  
            sendMessage(session, producer);  
            session.commit();  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            try {  
                if (null != connection)  
            } catch (Exception e) {  
            }  
        }  
    }  
}
```

```
        connection.close();

    } catch (Throwable ignore) {

    }

}

public static void sendMessage(Session session, MessageProducer producer) throws
Exception {

    for (int i = 1; i <= 5; i++) {

        TextMessage message = session.createTextMessage("消息" + i);

        System.out.println("发送消息：" + "消息" + i);

        producer.send(message);

    }

}

}

package test.jms;

import javax.jms.Connection;

import javax.jms.ConnectionFactory;

import javax.jms.Destination;

import javax.jms.MessageConsumer;

import javax.jms.Session;

import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnection;

import org.apache.activemq.ActiveMQConnectionFactory;

public class Receiver {

    public static void main(String[] args) {

        ConnectionFactory connectionFactory;

        Connection connection = null;

        Session session;
```

```
Destination destination;  
  
MessageConsumer consumer;  
  
connectionFactory = new ActiveMQConnectionFactory(  
    ActiveMQConnection.DEFAULT_USER,  
    ActiveMQConnection.DEFAULT_PASSWORD,  
    "tcp://192.168.1.51:61616");  
  
try {  
    connection = connectionFactory.createConnection();  
    connection.start();  
    session =  
connection.createSession(Boolean.FALSE,Session.AUTO_ACKNOWLEDGE);  
  
    //quqiQueue 是一个服务器的 queue , 须在 ActiveMq 的 console 配置  
    destination = session.createQueue("quqiQueue");  
    consumer = session.createConsumer(destination);  
    while (true) {  
        TextMessage message = (TextMessage) consumer.receive(1000);  
        if (null != message) {  
            System.out.println("收到消息" + message.getText());  
        } else {  
            break;  
        }  
    }  
  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    try {  
        if (null != connection)  
            connection.close();  
    }  
}
```

```
        } catch (Throwable ignore) {  
            }  
        }  
    }  
}
```

5.2.2.2 在应用中嵌入 activeMQ

代码如下：

```
package test.jms;  
  
import org.apache.activemq.broker.BrokerService;  
  
public class EmbededMQ {  
  
    public static void main(String[] args) throws Exception{  
  
        BrokerService broker = new BrokerService();  
  
        // configure the broker  
  
        broker.setBrokerName("brokerName"); //通过指定名称可以同时嵌入多个 broker  
  
        broker.addConnector("tcp://192.168.1.51:61616");  
  
        broker.start();  
  
        // //BrokerService broker = BrokerFactory.createBroker(new URI(someURI));  
        // BrokerService broker = new BrokerService();  
        // broker.setBrokerName("brokerName");  
        // broker.setUseShutdownHook(false);  
        // //Add plugin  
        // broker.setPlugins(new BrokerPlugin[]{new JaasAuthenticationPlugin()});  
        // //Add a network connection  
        // NetworkConnector connector =  
        broker.addNetworkConnector("static://"+"tcp://192.168.1.51:61616");  
        // connector.setDuplex(true);  
        // broker.addConnector("tcp://192.168.1.51:61616");  
        // broker.start();  
    }  
}
```

```
}
```

5.2.2.3 在 V3 中集成 activeMQ

在 V3 中集成 activeMQ 也就是基于 OSGi 写一个 bundle，或者是基于 HK2 写一个 service。下面以写 bundle 为例来集成一下。

1) 在命令行中到 GlassFishV3 的源代码目录(在任何目录都可以)执行：

```
mvn -o archetype:create
-DgroupId=org.GlassFish.custom.activemq -DartifactId=activemq
```

会报下列错误：

```
Unable to read local copy of metadata: Cannot read metadata from 'E:\.m2\repository\org\glassfish\build\maven-glassfish-extension\3.0-Prelude-SNAPSHOT\maven-metadata-maven2.java.net-backup.xml': end tag name </body> must match start tag name <hr> from line 7 <position: TEXT seen ...</address>\n</body>... @9:8>
org.glassfish.build:maven-glassfish-extension:jar:3.0-Prelude-SNAPSHOT
```

按照错误说明将

E:\.m2\repository\org\GlassFish\build\maven-GlassFish-extension\3.0-Prelude-SNAPSHOT\maven-metadata-maven2.java.net-backup.xml 文件中的<hr>标签删掉就行了，注意一定要加-o 参数。

2) 书写 bundle 代码，使用上节中在应用中嵌入 activeMQ 的方式，并删除 test 代码：

```
package org.GlassFish.custom.activemq;
import org.OSGi.framework.BundleActivator;
import org.OSGi.framework.BundleContext;
import org.apache.activemq.broker.BrokerService;
public class StartActivemq implements BundleActivator {
    @Override
    public void start(BundleContext context) throws Exception {
        BrokerService broker = new BrokerService();
        // configure the broker
        broker.setBrokerName("brokerName"); //通过指定名称可以同时嵌入多个 broker
        broker.addConnector("tcp://192.168.1.51:61616");
        broker.start();
    }
}
```

```

@Override
public void stop(BundleContext context) throws Exception {
}

```

3) 使用 felex-maven-plugin

(<http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html>)

在 pom.xml 中添加一些设置，添加后的 pom.xml 文件为：

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.GlassFish.custom.activemq</groupId>
  <artifactId>activemq</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>activemq</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>org.apache.felix</groupId>
      <artifactId>org.OSGi.core</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

```

```

<extensions>true</extensions>

<configuration>
    <instructions>

<Bundle-Activator>org.GlassFish.custom.activemq.StartActivemq</Bundle-Activator>

    <!!--<Bundle-ClassPath>.,E:\workspace\GlassFish\customBundle\activemq\lib\activemq-all-5.2.0.jar</Bundle-ClassPath>-->
    </instructions>
</configuration>
</plugin>
</plugins>
</build>
</project>

```

4) mvn org.apache.felix:maven-bundle-plugin:bundle

(注意：3)4)两步尚未成功，后来改到用eclipse建plugin-in project成功了，生成的JAR名为：qctivemqBundle_1.0.0.jar)

5) 将上述bundle的JAR包拷到%GlassFish%\felix\bundle目录下去。

6) 打开GLASSFISH_HOME/GlassFish/felix/conf/config.properties文件，将这个JAR包添加为启动项：

```

felix.auto.start.1= \
${com.sun.aas.installRootURI}/modules/javax.xml.stream.jar \
${com.sun.aas.installRootURI}/modules/tiger-types-osgi.jar \
${com.sun.aas.installRootURI}/modules/auto-depends.jar \
${com.sun.aas.installRootURI}/modules/config.jar \
${com.sun.aas.installRootURI}/modules/hk2-core.jar \
${com.sun.aas.installRootURI}/modules/osgi-adapter.jar \
${com.sun.aas.installRootURI}/felix/bundle/qctivemqBundle.jar

```

7) 让GlassFish以felix模式启动

```
java -DGlassFish_Platform=Felix -jar modules/GlassFish.jar
```

8) OK，MQ成功启动，如下图所示。现在就可以跑JMS应用了。

```
2009-6-22 19:47:12 org.apache.activemq.transport.TransportServerThreadSupport doStart
信息: Listening for connections at: tcp://192.168.1.51:61616
2009-6-22 19:47:12 org.apache.activemq.broker.TransportConnector start
信息: Connector tcp://192.168.1.51:61616 Started
2009-6-22 19:47:12 org.apache.activemq.broker.BrokerService start
信息: ActiveMQ JMS Message Broker (brokerName, ID:ZHANGHUA-29289-1245671232015-0:0) started
```

5.3 集成 EJB

V3 中的 EJB Container 覆盖的内容包含：

- Stateless Session Bean Container
- Stateful Session Bean Container
- Message Bean Container
- Entity Bean Container
- EJB Timer Service
- Code Generation Framework
- EJB Deployment Descriptors

5.3.1 OpenEJB

OpenEJB 使用了其他一些开源框架，如使用 OpenJPA 提供 JPA 和 CMP 的持久化、使用 ActiveMQ 处理 JMS/MDB、使用 Apache CXF 实现 JAX-WS 特性。OpenEJB 的一些特性主要瞄准了 EJB 3.1 规范，如 Collapsed EAR（在同一个归档及 classloader 中共存的 ejbs 与 servlets）及针对单元测试的嵌入式 EJB 容器。不仅如此，即将成为 JavaEE 6 组成部分的 EJB 3.1 Lite profile 非常类似于 OpenEJB。

与 Java EE 容器的集成：

OpenEJB 3.0 可以以插件的方式集成到 Tomcat 6 服务器中，这就在 Web 应用中增加了对 EJBs 的支持。将 OpenEJB 加入到 Tomcat 中可以为 Servlets 提供新的 Java EE 5 能力，如 JPA、JAX-WS、JMS、J2EE 连接器及事务。OpenEJB 天生就提供了对 GlassFish 部署描述符、Geronimo 及部分 WebLogic 部署描述符的支持。

OSGi 支持：

OpenEJB 框架是作为一个 OSGi 包发布的，这意味着所有 OpenEJB 3.0 的二进制文件与 OSGi 元数据一起被提供并且可用于任何 OSGi 平台上。基于 OSGi（使用 Apache Felix 构建）的开源 ESB 框架 ServiceMix 4 将把 OpenEJB 作为 ServiceMix 的一部分。

当前 OSGi 支持适合使用 OSGi 平台的人，他们渴望以包的方式增加 OpenEJB 以获得 EJB 支持，

或者由类似于 Apache ServiceMix 这样的项目所驱动的人，该项目做的就是提供 EJB WebServices 支持。OpenEJB 可用的含有 EJB jars 的所有包都将被部署。

OpenEJB 3.0 能通过 HTTP 协议调用 EJB，其动因就是让人们能绕过防火墙的限制并使 Tomcat/OpenEJB 用户可以通过一个单一的端口来运行 ejbs 和 servlets。最终的目标就是提供 RESTful ejb 调用。

5.3.2 V3 中集成 openejb

EjbSniffer(EJB 嗅探器)，该类继承了 GenericSniffer 并实现了接口 Sniffer。ContainerStarter:这个类负责容器的启动，将查找容器。在 EJB 容器的启动也注入了其相关的资源。

EJB 应用注入了安全机制，也能管理生存周期。加载并启动容器的方法：

```
boolean loadAndStartContainers(ApplicationContext startupContext){  
    .....  
}
```

EJB 模块的部署：

```
@Service  
public class EjbDeployer extends JavaEEEdeployer<EjbContainerStarter, EjbApplication>{  
    @Inject  
    protected ServerContext sc;  
    @Inject  
    protected Domain domain;  
    @Inject  
    protected ServerEnvironmentImpl env;  
}
```

从上面的程序可以看到 EJB 部署的时候会注入所需的相关信息。

第六章 写在最后--我的一点基于 OSGi 与 JMX 的微内核架构设想

JMX 主要是用来往应用程序里插入管理功能的框架，它能够通过 HTTP/XML 远程管理 MBeans，而 MBeans 模型可以在本地很好的集成被管理资源，但它并不适合作为一个开发系统的框架。而 OSGi 的主要优点在于它能够管理复杂的组件依赖，它有完整的生命周期模型。二者强强结合，我想应该可以实现出优良的系统。

在下图 6-1 中，每一个服务都是一个 OSGi service, JMX service 仅仅是另一个 OSGi Service ,这样我们可以通过 JMX 来访问 OSGi services. 集成 MX4J 到一个 OSGi bundle 中 ,那么其它 OSGi service 就可以引用这个 bundle 了。

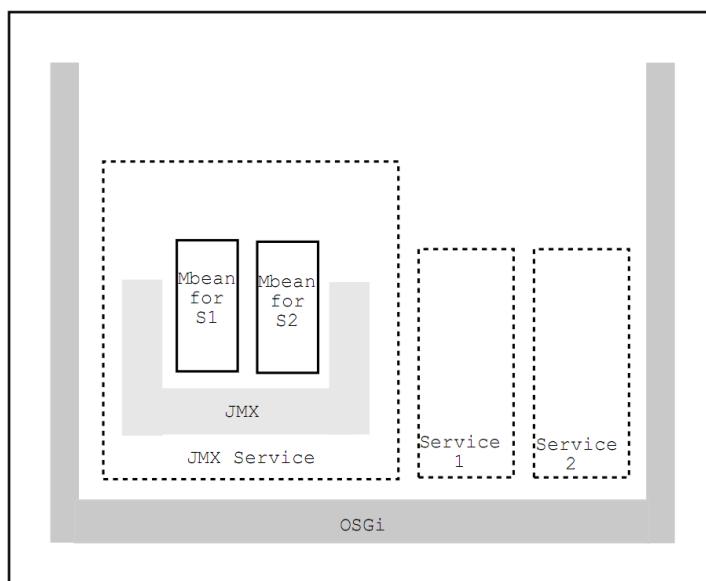
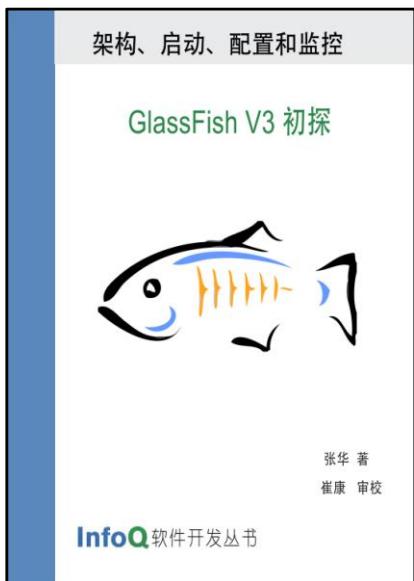


图 6-1 基于 OSGi 与 JMX 的微内核架构设想

关于作者

张华，长期从事 Java 方面的开发工作，有搜索引擎、中间件应用服务器、互联网、云计算等领域的行业经验，目前正在从事基于 Power 的虚拟化技术研发。博客地址：<http://blog.csdn.net/quqi99>



GlassFish V3 初探

作者：张华

审校：崔康

迷你书责任编辑：崔康

迷你书美术编辑：水羽哲

本迷你书主页为

<http://infoq.com/cn/minibooks/glassfish-v3-glance>

本书属于 InfoQ 软件开发丛书。

如果您打算订购 InfoQ 的图书，请联系 books@c4media.com

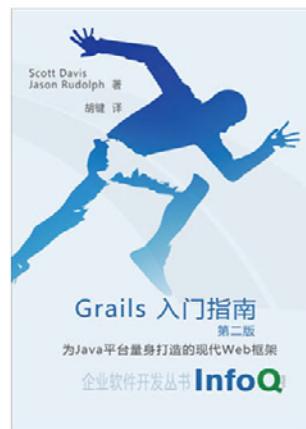
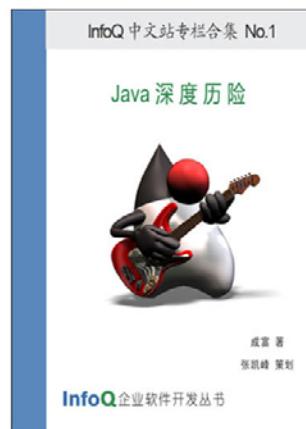
未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

InfoQ 软件开发丛书

欢迎免费下载



商务合作: sales@cn.infoq.com

读者反馈/内容提供: editors@cn.infoq.com