Оглавление

RMI

RMI - Remote Method Invocation (удаленный вызов метода). RMI служит для связи 2-х Java-процессов (работающих в различных экземплярах JVM) — клиентского и серверного. Смысл данной технологии в том, чтобы сделать вызов метода удаленного объекта таким же, как обычный локальный вызов внутри одной JVM. Реализуется это CORBA'образно, созданием представителей нужного серверного объекта на стороне сервера (скелет - skeleton) и клиента (заглушка - stub), в которые зашита вся работа по удаленному взаимодействию. И скелет и заглушка представляют часть функционала класса этого серверного объекта, реализуя некоторый его интерфейс. Клиентский процесс общается с заглушкой (она же посредник или proxy), а сервер — со скелетом, удаленное взаимодействие происходит путем общения заглушки и скелета. Когда клиент дергает метод интерфейса, реализующая его заглушка пакует (сериализует) параметры, адреса и отправляет их скелету на сервер. Скелет все это распаковывает (десериализует) и уже вызывает соответствующий метод класса, принимает ответ, запаковывает его и отправляет обратно заглушке. Заглушка распаковывает ответ и пересылает его клиенту. Для получения заглушки поднимается спец. сервер с реестром, хранящим записи типа имя-ресурс.

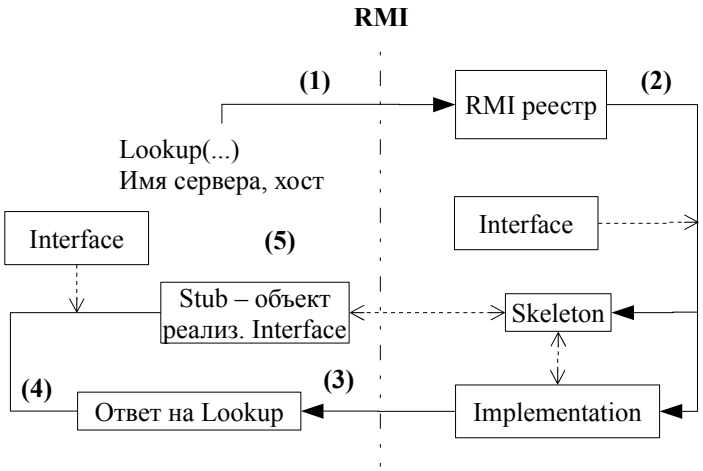

Схема работы RMI: (1) удаленный запрос lookup на сервер для получения заглушки-stub; (2) создание на сервере экземпляра требуемого класса и экземпляров заглушки и скелета, реализующих некоторый требуемый интерфейс этого класса; (3) сериализация заглушки и передача ее клиенту; (4)-(5) десериализация заглушки клиентом и получение результата вызова lookup в виде объекта-заглушки, приведенной к реализуемому ею интерфейсу.

[rmi_ref] В качестве параметров и результатов вызова, передаваемых по RMI, можно использовать простые типы и сериализуемые объекты, в т.ч. заглушки. Все они сериализуются автоматически перед отправкой и десериализуются после получения. Т.о. передача происходит по значению, т.е. копированием. Передача объекта, который в добавок реализует заглушечную функциональность (extends UnicastRemoteObject), также происходит по значению. Но при этом любое клиентское обращение к интерфейсу такого объекта трактуется как обращение к заглушке, которая транслирует запрос на сервер и принимает ответ. Отправка клиентом заглушки в качестве параметра приводит к созданию на сервере ее копии, но связанной с нею. Поскольку клиент имеет дело только с интерфейсными методами заглушек, транслирующими запросы на сервер, а в случае передачи заглушек на сервер, они становятся связаными со своими серверными копиями, то фактически обмен заглушками эмулирует передачу по ссылке, где любое изменение на одной стороне, отражается и на другой.

Т.о синхронизация между клиентским и серверным процессами возможна только при общении skeleton-stub при помощи заглушек. Если передавать объекты сериализацией по значению без обертки заглушечной функциональностью, то на серверной реализации класса клиентские изменения автоматически не отобразятся.

На стороне сервера. Интерфейс класса для удаленного вызова по RMI должен быть расширен от интерфейса java.rmi.Remote. Сам класс должен быть расширен от класса java.rmi.server.UnicastRemoteObject и реализовывать интерфейс удаленного вызова. Конструктор должен пробрасывать RemoteException.

```
Ex: public interface Inter extends Remote {…}
public class Impl extends UnicastRemoteObject implements Inter {
…
public Impl() throws RemoteException { … }
```

На стороне клиента. Предварительно должен быть загружен в перманентную область (PermGen/Metaspace) клиента тот же интерфейс, что используется на сервере. Клиентский процесс может получить заглушку (stub) через вызов java.rmi.Naming.lookup с указанием адреса RMI-сервера и алиаса заглушки, который предварительно создается в реестре RMI. Заглушка приводится к указанному интерфейсу и используется обычным образом, т.е. так же, как использовался бы целевой класс.

```
Ex: … Inter inter = (Inter)Naming.lookup(“rmi://localhost/RMI_INSTANCE”);
inter.exMethod(...);
```

Rem: *... RMI_INSTANCE – алиас заглушки stub в RMI-реестре*

Для запуска службы RMI и регистрации в нем ресурсов можно применять разные способы.

1. Существует автономная утилита rmiregistry из состава jdk
2. Можно создать экземпляр службы прямо в java-коде java.rmi.registry …
3. В Application Server'ax могут существовать собственные RMI-службы

Для регистрации в RMI-реестре ресурса, надо указать алиас и требуемый объект реализации класса (ресурс).

```
Ex: java.rmi.Naming.rebind("ex_alias",impl);
```

Rem: Для обмена запросами RMI использует специальные протоколы прикладного уровня. На данный момент употребим протокол RMI-IIOP, который представляет собой Java-надстройку над протоколом CORBA и позволяет серверному объекту RMI связываться с клиентским CORBA-объектом, написанном на любом языке (имеющим ORB). В JEE7 поддержка IIOP в RMI помечена как кандидат на отмену в JEE8.

Rem: Application server Weblogic имеет свою реализацию RMI, поддерживающую специальный транспортный протокол t3://

Rem: В IDE Jdeveloper подключение удаленных ресурсов запрещено в настройках домена по-умолчанию. Для отмены этого ограничения надо изменить конфигурационный файл. В Jdeveloper 12 для этого надо найти файл вида C:\Users\my_user\AppData\Roaming\JDeveloper\system12.1.2.0.40.66.68\DefaultDomain\bin\setStartupEnv.cmd и переназначить все (4 шт) -Dweblogic.jdbc.remoteEnabled с false на true.

Rem: для десериализации объекта требуется иметь загруженный в перманентную область класс. Надо следить, чтобы uid класса на сервере и клиенте совпадали, т.к. иначе JVM выдаст ошибку Поэтому либо надо следить за полным соответствием пакетов на клиенте и сервере, либо вручную, назначая свой serialVersionUID и отслеживая фактическую функциональную совместимость классов.

Пример поднятия и использования реестра RMI в JSE

Для тестирования удаленных вызовов создается пара классов, их удаленные интерфейсы, класс создания RMI реестра и тестовый клиент.

Удаленный интерфейс IMyObject создан для тестирования простых удаленных запросов и передачи его реализаций в виде заглушек.

```
package test.rmitest;
import java.rmi.*;
```

```
public interface IMyObject extends Remote {
  void say(String message) throws RemoteException;
  void say() throws RemoteException;
  void setPhrase(String phrase) throws RemoteException;
}
```

Удаленный интерфейс ISharedObject содержит свойство myObject параметрического типа и нужен для тестирования передачи заглушек “по ссылке”.

```
package test.rmitest;
import java.rmi.*;

public interface ISharedObject<T> extends Remote {
  T getMyObject() throws RemoteException;
  void setMyObject(T t) throws RemoteException;
  void clearMyObject() throws RemoteException;
  void changeMyObject(String phrase) throws RemoteException;
  public void changeMyObjectPhrase(String phrase) throws RemoteException;
}
```

Класс MyObject реализует на серверной стороне IMyObject. Кроме того он расширяется заглушечной функциональностью UnicastRemoteObject, что позволяет передавать его заглушку клиенту в виде результата и обратно на сервер в виде параметра метода.

```
package test.rmitest;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class MyObject extends UnicastRemoteObject implements IMyObject {
  private String phrase = "Init";
    private int count = 0;

    public MyObject() throws RemoteException {
      super();
    }
    @Override
    public void say(String message) throws RemoteException {
      System.out.println(message);
    }
    @Override
    public void say() throws RemoteException    {
      System.out.println("[Count = "+count + "] "+phrase);
      count++;
    }
    @Override
    public void setPhrase(String phrase) throws RemoteException {
      this.phrase = phrase;
    }
}
```

Класс SharedObject реализует на серверной стороне ISharedObject. Он содержит свойство myObject типа MyObject с get/set методами для тестирования передачи «объектов по ссылке» заглушками.

```
package test.rmitest;
import java.rmi.*;

public class SharedObject implements ISharedObject<IMyObject> {
  private IMyObject myObject;

  public SharedObject() {
    super();
    try {
      myObject = new MyObject();
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }
  @Override
  public IMyObject getMyObject() throws RemoteException {
    return myObject;
  }
  @Override
  public void setMyObject(IMyObject myObject) throws RemoteException {
    this.myObject = myObject;
  }
  @Override
    public void clearMyObject() throws RemoteException {
    this.myObject = null;
  }
  @Override
  public void changeMyObject(String phrase) throws RemoteException {
    this.myObject = new MyObject();
    myObject.setPhrase(phrase);
  }
  @Override
  public void changeMyObjectPhrase(String phrase) throws RemoteException {
    this.myObject.setPhrase(phrase);
  }
}
```

Реестр RMI поднимается в отдельной JVM. После этого поток main можно убить.

```
package test.rmitest;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.io.IOException;
```

```
public class RunServer {
  public static void main(String[] args) throws InterruptedException, IOException {
// Поднятие реестра на порту 2099
    Registry registry = LocateRegistry.createRegistry(2099);
// Экспорт в заглушку Remote stub объекта SharedObject для удаленных вызовов, связанных с
//заданным портом - 0
    Remote stub = UnicastRemoteObject.exportObject(new SharedObject(),0);
// Создание алиаса "RMI_Server" для поиска и получения клиентом заглушки
    registry.rebind("RMI_Server", stub);
    System.out.println("Press enter to exit...");
    System.in.read();
    registry = null;
    System.out.println("RMI Registry Thread have to be collected.");
  }
}
```

Клиент находит поднятый реестр RMI, получает оттуда заглушку ShredObject по зарегистрированному алиасу, тестирует удаленные вызовы, заглушечный обмен.

```
package test.rmitest;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class RunClient {

  public static void main(String[] args) throws RemoteException, NotBoundException {
//Поиск поднятого на порту 2099 реестра RMI
    Registry registry = LocateRegistry.getRegistry(2099);
//Получение в RMI реестре Remote-заглушки удаленного объекта
    Remote srv = registry.lookup("RMI_Server");
    ISharedObject<IMyObject> service = (ISharedObject<IMyObject>) srv;
//Получение из заглушки sevice объекта SharedObject заглушки myObject его (SharedObject)
//внутреннего поля myObject типа MyObject,
//расширенного заглушечной функциональностью UnicastRemoteObject
    IMyObject myObject = service.getMyObject();
// Тестирование удаленного вызова метода через заглушку. Передается строковый сериализуемый
//параметр
  myObject.say("Хрю!");
// Тестирование изменения состояния удаленного объекта через его заглушку. Передается
строковый сериализуемый параметр
  myObject.setPhrase("Хрю-хрю!");
// Удаленный вызов печати состояния myObject
  myObject.say();
// Присвоение null полю myObject в серверном объекте SharedObject
  service.clearMyObject();
// Объект myObject на сервере уничтожен не будет, т.к. он расширяет UnicastRemoteObject и при
создании помещается в реестр и висит там.
  myObject.say();
// Удаленное инициализация поля myObject серверного объекта SharedObject прежней клиентской
```

```
//заглушкой myObject.
// Данная клиентская заглушка связана с прежним серверным экземпляром MyObject и поле будет
//вновь указывать на этот же экземпляр.
    service.setMyObject(myObject);
// Заглушка текущего поля myObject SharedObject
    myObject = service.getMyObject();
// Тестирование этого поля через его заглушку. Экземпляр прежний, что и до clearMyObject()
    myObject.say();
// Смена myObject в SharedObject
    service.changeMyObject("Мяу");
// Получение заглушки на поле myObject после обновления
    IMyObject myObject2 = service.getMyObject();
// Тестирование обновленного поля через заглушку
    myObject2.say();
// Тестирование старого myObject через заглушку
    myObject.say();
// Изменение (локальное на сервере) состояния myObject в SharedObject
    service.changeMyObjectPhrase("Гав");
    myObject2.say();
  }
}
```

Результат теста:

```
Хрю!
[Count = 0] Хрю-хрю!
[Count = 1] Хрю-хрю!
[Count = 2] Хрю-хрю!
[Count = 0] Мяу
[Count = 3] Хрю-хрю!
[Count = 1] Гав
```

JNDI - Основы

JNDI = Java Naming & Directory Interface (интерфейс службы имен и каталогов Java).

Служба имен в общем случае занимается отображением множества имен во множество программных объектов. JNDI отображает имена в Java-объекты, при этом имена структурируются в древовидную структуру (JNDI tree), что позволяет проводить скоростной поиск объектов.

Обычно служба JNDI интегрирована в JEE-сервер. В JEE5 и JEE6 для унификации удаленных вызовах к различным реализациям JEE-серверов был введен переносимый синтаксис. В соответствии с Remote и Local доступом был введен глобальный и локальный JNDI. Глобальный JNDI реализует удаленный поиск и получение объектов по сети. Для этого служба JNDI используется в связке с протоколом RMI. Локальный JNDI ENC (Enterprise/Environment Naming Context) используется в рамках Application Server, точнее в рамках одного домена (соответствующему в современных версиях JEE

одному управляющему процессу). Иметь локальную службу имен полезно для усиления платформонезависимости, поскольку боевая платформа может быть отлична от платформы разработчика и на ней может не быть доступа к исходникам. Кроме того возможна кластеризация, когда Application Server будет работать на нескольких JVM. В этих случаях для переконфигурирования проекта достаточно будет изменить только xml-дескрипторы, не трогая ни исходный код, ни метаинформацию (аннотации).

JNDI - Global JNDI

Глобальный JNDI доступен через унифицированный вендор-независимый синтаксис. Имена глобального JNDI имеют вид:

java:global[/app-name]/module-name/bean-name[!FQN]

- java:global = глобальное пространство имен
- app-name = имя приложения (ear)
- module-name = имя war/jar
- FQN = Full Qualified Name (полное имя класса с пакетами)

На стороне сервера. Регистрацию global JNDI имени можно сделать непосредственно в службе JNDI. В службе JNDI может изначально существовать иерархия предопределенных ресурсов и для регистрации объекта одного из них достаточно определить только ряд параметров.

Ex: регистрация объекта источника данных Data Source в JNDI-службе сервера Weblogic через

(web-консоль):
DataSource = ExDataSource
services -> data sources -> new -> generic data source
name = ExDataSource
jndi name = jdbc/ExDataSource
db = orcl
hostname = localhost
user = hr
pass = 12345678
server = default server
тестирование: monitoring -> testing -> server = Defaultserver -> test data source
найти созданное соединение можно в ресурсах приложения

На стороне клиента. Доступ к корню глобального JNDI tree можно получить в объекте javax.naming.InitialContext, при создании которого в конструкторе надо указать параметры подключения к службе JNDI, используя строковые константы родительского интерфейса javax.naming.Context. После этого можно осуществлять поиск зарегистрированных объектов по JNDI-имени, используя метод lookup объекта InitialContext. Параметры подключения к global JNDI задаются или в Hashtable или в файле настроек (ресурсов).

Ex: получение объекта (RMI-заглушки на самом деле) источника данных при помощи JNDI:

```
java.util.Hashtable<String,String> ht = new java.util.Hashtable<String,String>();
ht.put(Context.INITIAL_CONTEXT_FACTORY,"weblogic.jndi.WLInitialContextFactory");
ht.put(Context.PROVIDER_URL,"http://localhost:7101");
Context  context = new InitialContext(ht);
DataSource  ds = (DataSource)context.lookup("jdbc/ExDataSource");
```

Ex: получение объектов (RMI-заглушек) темы и фабрики соединений JMS при помощи JNDI в Glassfish 4:

```
   java.util.Hashtable<String,String> ht = new java.util.Hashtable();
   ht.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.enterprise.naming.SerialInitContextFactory");
   ht.put(Context.URL_PKG_PREFIXES, "com.sun.enterprise.naming");
   ht.put(Context.STATE_FACTORIES,
"com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
   ht.put(Context.PROVIDER_URL, "http://localhost:3700");
   javax.naming.Context context = new InitialContext(ht);
   Destination topic = (Destination)context.lookup("jms/example/Topic");
   ConnectionFactory factory = (ConnectionFactory)context.lookup("jms/example/ConnectionFactory");
  }
```

Rem: при использовании сервера Weblogic подключение к JNDI происходит при помощи библиотеки Weblogic Remote Client (jar файл библиотеки подключается в свойствах проекта). Переменная подключения 'weblogic.jndi.WLInitialContextFactory' задает путь к классу в этой библиотеке, объект этого класса создается при помощи рефлексии.

Rem: в JDeveloper для использования файла ресурсов надо его создать:

```
java.naming.factory.initial = weblogic.jndi.WLInitialContextFactory
java.naming.provider.url = http://localhost:7101
```

затем подключить связку ресурсов через Application -> application/project properties.
Настройки: в project properties: Multiple shared bundles
Добавление resource bundles: application -> resource bundles -> add
Для создания контекста следует использовать вызов new InitialContext()

Rem: Для просмотра JNDI-дерева в сервере Weblogic надо зайти в консоль http://localhost:7101/console/Environment -> Servers, кликнуть по нужному серверу приложений, нажать в открывшемся окне на ссылку «View JNDI Tree».

JNDI — JNDI ENC

Локальный JNDI ENC работает только в JEE и использует внутренние ENC имена, которые Application Server автоматически отображает на глобальные JNDI-имена в определенном контексте (узле JNDI tree). Для каждого JEE-управляемого объекта создается свой ENC контекст. По сути ENC – карманный справочник JEE-управляемого

класса. Для регистрации JNDI ENC имен используется специальный xml-дескриптор развертывания META-INF/ejb-jar.xml или аннотации внедрения вроде @Resource, которые позволяют исключить прямое использование метода lookup объекта InitialContext. При одновременном определении аннотации и соответствующего элемента xml-дескриптора, последний имеет приоритет, что удобно при переконфигурировании приложения без изменения исходного кода.

Существует перечень объектов, которые могут быть зарегистрированы в JNDI ENC:

- EJB интерфейсы
- JMS (queue/topic/connection factory)
- DataSource
- простые типы-оболочки
- String

и пр.

Для каждого типа объекта существует своя аннотация и свой синтаксис в дескрипторе развертывания ejb-jar.xml

JNDI — @EJB

@javax.ejb.EJB - аннотация RUNTIME уровня FIELD, METHOD, TYPE. Она служит для внедрения в MBean компонентов EJB и регистрации соответствующих точек внедрения в JNDI ENC этого MBean. Для нее могут определяться следующие атрибуты:

- name = ENC-имя точки внедрения EJB-компонента в MBean (если аннотация применяется к классу, то это неявная точка внедрения в MBean, доступная только через JNDI)
- description = описание
- beanInterface = FQN.class внедряемого интерфейса, идет только в паре с beanName
- beanName = имя внедряемого EJB-компонента, используемое как конечная часть имени Global JNDI переносимого синтаксиса. Соответствует параметру name аннотаций @Statefull/@Stateful/@Singleton и значению элемента <ejb-name> дескриптора ejb-jar.xml. По умолчанию задается неквалифицированным именем класса внедряемого EJB. Идет только в паре с beanInterface
- mappedName = id внешнего вендорного реестра, часто совпадает с global JNDI именем
- lookup = JNDI-имя, по которому может быть найдена ссылка на бин

Для использования множества @EJB на уровне класса существует аннотация @EJBs: @EJBs({@EJB..., …, @EJB...})

Регистрация в ENC. Для регистрации в JNDI ENC объектов EJB можно использовать дескриптор ejb-jar.xml, в котором в тегах <ejb-local-ref> описываются ссылки на объекты. Также можно использовать аннотацию @EJB. Причем @EJB определенная на

уровне класса только регистрирует EJB в ENC компонента внедрения. А @EJB на уровне свойства как регистрирует, так и внедряет объект EJB: обрабатывая ее первый раз, JEE-контейнер создает соответствующий объект и регистрирует его в ENC под указанным в аннотации именем, а также и под именем свойства внедрения: xxx/yyy, где xxx – FQN класса-хозяина, yyy – имя свойства для внедрения. ENC-имя по-умолчанию — FQN регистрируемого класса EJB. Это имя можно переопределить на собственное xxx: @EJB(name=”xxx”) или <ejb-ref-name> в ejb-jar.xml. При одновременном определении, ejb-jar.xml имеет больший приоритет.

Использование ENC. При поиске объекта EJB через ENC-имя, JEE-контейнер будет вычислять соответствующее глобальное JNDI-имя. При этом для вычисления будет использоваться соответствующий контекст ENC, автоматически определяемый классом вызывающего JEE-управляемого объекта. Получение объекта через JNDI ENC имени можно делать по-разному.

1. При помощи обычного InitialContext:

```
@EJB(name="referenceToMyEJB2", beanInterface=MyEJB2LocalBusiness.class,
beanName="MyEJB2")
public class MyEJBBean implements MyEJBLocalBusiness {
…
javax.naming.InitialContext ctx = new InitialContext();
otherBean = (MyEJB2LocalBusiness)
ctx.lookup("java:comp/env/referenceToMyEJB2");
```

java:comp/env — контекст (узел JNDI tree) в компонентном пространстве имен ejbs/referenceToMyEJB2 — ENC имя ресурса, предварительно зарегистрированное в ENC бина-хозяина.

2. При помощи локального контекста EJBContext (или его расширения SessionContext), которым обладает любой сессионный EJB. Этот контекст можно получить через @Resource. При этом запрашиваемый EJB-компонент уже должен быть внедрен в JNDI ENC текущего EJB при помощи @EJB или дескриптора ejb-jar.xml:

```
@EJB(name="referenceToMyEJB2", beanInterface=MyEJB2LocalBusiness.class,
beanName="MyEJB2")
public class MyEJBBean implements MyEJBLocalBusiness {
@Resource private javax.ejb.SessionContext ctx;
…
public void lookEjbFromContext() {
MyEJB2LocalBusiness otherBean = (MyEJB2LocalBusiness)ctx.lookup("referenceToMyEJB2");
```

В lookup сессионного контекста надо передавать лишь ENC-имя, предварительно определенное в ENC бина-хозяина, все остальное будет найдено автоматически.

3. При помощи @EJB можно внедрять бины напрямую в свойства, при это во время первого обращения бин будет создан и зарегистрирован в JNDI ENC под указанным в

@EJB именем, а также под именем xxx/yyy, где xxx – FQN класса-хозяина, а yyy – имя свойства для внедрения. Заполнение самого свойства произойдет автоматически сразу после создания экземпляра бина. При повторных запросах по этим именам будет выдан уже созданный экземпляр бина.

Вместо @EJB можно использовать дескриптор ejb-jar.xml Там помимо описания JNDI ENC — ссылки можно указать параметры внедрения: класс-хозяин и свойство в теге <injection-target>.

```
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>MyEJB</ejb-name>
<ejb-local-ref>
<ejb-ref-name>MyEJB2</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<local>org.ejb3book.examples.MyEJB2LocalBusiness</local>
<ejb-link>MyEJB2Bean</ejb-link>
<injection-target>
<injection-target-class>
org.ejb3book.examples.MyEJBBean
</injection-target-class>
<injection-target-name>otherBean</injection-target-name>
</injection-target>
</ejb-ref>
</enterprise-beans>
</ejb-jar>
```

- <ejb-local-ref> = @EJB
- <ejb-ref-name> = name в @EJB
- <local> и <remote> = beanInterface в @EJB
- <ejb-link> = beanName в @EJB
- <injection-target-class> = FQN класса, содержащего свойство для внедрения (дает возможность использовать сужение/расширение интерфейса EJB);
- <injection-target-name> = имя свойства внедрения

Наследование @EJB. Аннотации вроде @EJB свойств наследуются вместе с их классом хозяином. В наследниках их можно переопределить за исключением случая, когда свойства предка было private – в этом случае аннотация не переопределяется.

- расширенное ENC-имя. Если существует несколько одноименных EJB, расположенных в одном ear-архиве приложения, в разных его jar/war-подархивах, то для <ejb-link> или параметра beanName @EJB надо использовать расширенное имя вида xxx.jar#yyy где xxx – путь к jar из корня ear, yyy – имя EJB.

Rem: В рамках единого ejb-jar.xml, т.е единого jar-архива допустимо использовать только уникальные EJB-имена.

Rem: В сервере Weblogic параметры ENC можно использовать для поиска EJB в глобальном контексте JNDI (из внешнего клиента): ctx.lookup("xxx#yyy"), где xxx – вендорное mappedName, yyy – FQN бизнес-интерфейса.

Алгоритм поиска EJB по его ENC-имени для сервера JBOSS в случае простейшего аннотированного свойства — поля @EJB private ExBean bean:

1. Поиск в ejb-jar.xml EJB-реализации интерфейса ExBean. Если существует единственный ответ, то поиск удачно завершен. Если существует >1 ответа, то выбрасывается Exception. Если ничего не найдено — п.2.
2. Поиск в других jar, принадлежащих ear. Если существует единственный ответ, то поиск удачно завершен. Если существует >1 ответа, то выбрасывается Exception. Если ничего не найдено — п.3.
3. Поиск в global JNDI контексте (корень JNDI tree)

Rem: Если существует beanName, то это дополнительный id для поиска в дескрипторе.

Rem: Если присутствуют mappedName или lookup параметры, то поиск идет в global JNDI.

JNDI — @PersistenceUnit

@javax.persistence.PersistenceUnit - аннотация RUNTIME уровня FIELD, METHOD, TYPE. Для нее могут определяться следующие атрибуты:

- name = ENC имя фабрики. Требуется только на уровне класса.
- unitName = алиас PersistenceUnit в дескрипторе persistence.xml. Требуется только на уровне класса и обязателен там, если в дескрипторе определено множество PersistenceUnit (в случае единственного он будет подставлен по-умолчанию).

Аннотация аналогична @EJB, но служит для работы с объектами EntityManagerFactory (EMF) и имеет несколько другой синтаксис.

Регистрация в ENC. @ PersistenceUnit уровня класса регистрирует объект EM в ENC. На уровне свойства и регистрирует (первый раз) и внедряет объект аналогично @EJB.

Использование ENC. Используется @PersistenceUnit уровня свойства без параметров (вся информация берется из свойства внедрения).

Для использования множества @PersistenceUnit существует аннотация @PersistenceUnits:
@PersistenceUnits({@PersistenceUnit..., …, @PersistenceUnit...})

Вместо @PersistenceUnit можно использовать дескриптор ejb-jar.xml Там помимо описания JNDI ENC — ссылки можно указать параметры внедрения: класс-хозяин и свойство в теге <injection-target>.

```
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>TravelAgentBean</ejb-name>
<persistence-unit-ref>
<persistence-unit-ref-name>persistence/MyDB</persistence-unit-ref-name>
<persistence-unit-name>MyDB</persistence-unit-name>
<injection-target>
<injection-target-class>org.ejb3book.examples.MyEJBBean
</injection-target-class>
<injection-target-name>pu</injection-target-name>
</injection-target>
</persistence-unit-ref>
</enterprise-beans>
</ejb-jar>
```

- <persistence-unit-ref> = @PersistenceUnit
- <persistence-unit-ref-name> = name в @PersistenceUnit
- <persistence-unit-name> = unitName в @PersistenceUnit
- <injection-target-class> = FQN класса, содержащего свойство для внедрения (дает возможность использовать сужение/расширение интерфейса EMF);
- <injection-target-name> = имя свойства внедрения

- расширенное ENC-имя. В рамках одного приложения (ear-архива) возможно определить множество PersistenceUnit как в jar/war-архивах, так и в ear/lib. Для задания правильного пути в этом случае следует использовать для <persistence-unit-name> или параметра unitName @PersistenceUnit надо использовать расширенное имя вида xxx.jar#yyy где xxx – путь к jar из корня ear, yyy – имя PersistenceUnit.

JNDI — @PersistenceContext

@javax.persistence.PersistenceContext - аннотация RUNTIME уровня FIELD, METHOD, TYPE. Для нее могут определяться следующие атрибуты:

- name = ENC имя EM. Требуется только на уровне класса.
- unitName = алиас PersistenceUnit в дескрипторе persistence.xml. Обязателен, если в дескрипторе определено множество PersistenceUnit (в случае единственного он будет подставлен по-умолчанию).
- type = Enum {TRANSACTION, EXTENDED}. По-умолчанию EXTENDED. Тип транзакционного механизма, поддерживаемый внедряемым EM.
- properties = массив аннотаций, содержащих пары name-value для вендор-специфических дополнений.

Аннотация аналогична @EJB, но работает с объектами EntityManager (EM) и имеет несколько другой синтаксис.

Регистрация в ENC. @PersistenceContext уровня класса регистрирует объект EM в ENC.

На уровне свойства и регистрирует (первый раз) и внедряет объект аналогично @EJB.

Использование ENC. Используется @PersistenceContext уровня свойства, требуется указать атрибут unitName.

Для использования множества @PersistenceContext существует аннотация @PersistenceContexts:
@PersistenceContexts({@PersistenceContext..., …, @PersistenceContext...})

Вместо @PersistenceContext можно использовать дескриптор ejb-jar.xml Там помимо описания JNDI ENC — ссылки можно указать параметры внедрения: класс-хозяин и свойство в теге <injection-target>.

```
<persistence-context-ref>
   <persistence-context-ref-name>calculator.MyBean/myEM</persistence-context-ref-name>
   <persistence-unit-name>bar-unit</persistence-unit-name>
   <persistence-context-type>Transaction</persistence-context-type>
   <injection-target>
      <injection-target-class>calculator.MyBean</injection-target-class>
      <injection-target-name>myEM</injection-target-name>
   </injection-target>
</persistence-context-ref>
```

- <persistence-context-ref> = @PersistenceContext
- <persistence-context-ref-name> = name в @PersistenceContext
- <persistence-unit-name> = persistenceName в @PersistenceContext
- <persistence-context-type> = type в @PersistenceContext
- <injection-target-class> = FQN класса, содержащего свойство для внедрения (дает возможность использовать сужение/расширение интерфейса EMF);
- <injection-target-name> = имя свойства внедрения

JNDI — @Resource

@javax.annotation.Resource - аннотация RUNTIME уровня FIELD, METHOD, TYPE. Служит для работы с объектами внешних ресурсов, источников JMS, EJB-контекстов, элементов среды (environment entry), сервисов и т.д.

Для нее могут определяться следующие атрибуты:

- name = JNDI ENC имя ресурса
- type = F.Q,N, класса ресурса
- mappedName = вендор-специфический идентификатор для поиска, часто это global JNDI-имя
- lookup = специальный механизм указания ресурса в global JNDI контексте
- authentificationType = Enum {CONTAINER, APPLICATION}. По-умолчанию CONTAINER. Тип защиты доступа. CONTAINER – организация доступа из

самого ресурса или конфигурации Application Server. APPLICATION – организация доступа программно, в классе-носителе данной аннотации.
- shareable = вкл/выкл совместный доступ к ресурсу на период транзакции. По-умолчанию ресурс разделяется, т.е. возможен совместный доступ на период транзакции по одному соединению к нему, ссылки на которое и внедряются. В противном случае доступ к фактическому соединению будет выдаваться только одному пользователю из пула соединений, что считается более затратным.

Регистрация в ENC. @Resource уровня класса регистрирует объект ресурса в ENC, при этом требуются атрибуты name и type. На уровне свойства и регистрирует (первый раз) и внедряет объект в это свойство.

Использование ENC. Используется @Resource уровня свойства, из параметров может применяться только lookup, остальное берется из сигнатуры свойства.

Для использования множества @Resource существует аннотация @Resources: @Resources({@Resource..., …, @Resource…})

JNDI - @DataSourceDefenition

@javax.annotation.sql.DataSourceDefenition – JEE-аннотация RUNTIME уровня TYPE. Служит для создания, конфигурации источника данных РБД (DataSource) и регистрации его в Global JNDI. Может создавать объекты интерфейсов javax.sql.DataSource, javax.sql.XADataSource, javax.sql.ConnectionPoolDataSource.

Содержит следующие обязательные атрибуты:

- className — строка класса реализации интерфейса создаваемого объекта
- name — строковое JNDI-имя для хранения в Global JNDI

Некоторые необязательные атрибуты:

- url – адрес для связи с РБД в форме URL (может содержать параметры, конкурирующие с атрибутом properties)
- user — имя пользователя для аутентификации соединения с РБД
- password — пароль для аутентификации соединения с РБД
- databaseName — имя БД
- properties — строковый массив вендор-специфических и не очень атрибутов соединения с РБД {"p1=v1, …, pN=vN"}

Ex: пример неявного трудолюбиовго создания и инициализации единого для приложения объекта javax.sql.DataSource, который потом будет использоваться EM для связи с РБД ExDB СУБД Derby при объектно-реляционном отображении:

@Singleton

```
@Startup
@DataSourceDefinition(
  className = "org.apache.derby.jdbc.EmbeddedDataSource",
  name = "java:global/jdbc/exDS",
  url="jdbc:derby://localhost:1527/exDB",
  user = "user01",
  password = "pass01",
  properties = {"connectionAttributes=", "create=false"}
)
public class AppInit { … }
```

Аналог в XML-дескрипторе JPA persistence.xml будет иметь вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
          xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
          http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
          version="2.1">
  <persistence-unit name="exPU" transaction-type="JTA">
    <jta-data-source>java:global/jdbc/exDS</jta-data-source>
    <properties>
      <property name="eclipselink.target-database" value="DERBY"/>
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
      <property name="eclipselink.logging.level" value="INFO"/>
    </properties>
  </persistence-unit>
</persistence>
```

Rem: также создавать источники данных и запихивать их в JNDI-дерево можно вендорными средствами JEE-серверов.

```
Ex:  создание пула соединений exDBPool, связанного с источником данных exDB в консоли
GlassFish:

$ asadmin create-jdbc-connection-pool
  --datasourceclassname=org.apache.derby.jdbc.ClientDataSource
  --restype=javax.sql.DataSource
  --property portNumber=1527:password=pass01:user=user01:serverName=localhost:
databaseName=exDB:connectionAttributes=;create\=false  ExDBPool

Проверка соединения:
$ asadmin ping-connection-pool ExDBPool

Связывание источника данных exDS с пулом соединений ExDBPool
$ asadmin create-jdbc-resource --connectionpoolid ExDBPool jdbc/exDS

Список активных источников данных:
$ asadmin list-jdbc-resources
```

JNDI — Environment Entries & Administered Objects

Environment Entries или элементы среды — помещенные в JNDI бъекты Enum, String и объекты-оболочки Integer, Long, Double, Float, Byte, Boolean, Short. Подобные простые объекты обычно помещают в JNDI для настройки поведения каких-то сложных ресурсов. Внедрять их можно через @Resource с использованием атрибута name. Чаще всего применяется совместное описание через элемент <env-entry> дескриптора ejb-jar.xml и аннотации @Resource уровня свойства, что позволяет задавать значение по-умолчанию, не попадающее в JNDI и переопределять его при необходимости в xml, не трогая исходников.

```
Ex:
…
// Параметр  passphrase по-умолчанию =  defaultPassphrase, оно не попадет в JNDI
@Resource(name="ciphersPassphrase") private String passphrase = "defaultPassphrase";
...

<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>EncryptionEJB</ejb-name>
<!-- Переопределение пароля -->
<env-entry>
<env-entry-name>ciphersPassphrase</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>OverriddenPassword</env-entry-value>
<injection-target>
<injection-target-class>
org.ejb3book.examples.MyEJBBean
</injection-target-class>
<injection-target-name>ciphersPassphrase</injection-target>
</injection-target>
</env-entry>
</session>
</enterprise-beans>
</ejb-jar>
```

Administered Objects – некие управляемые объекты, которые входят в состав некоторых сложных ресурсов и которые необходимо внедрять или получать через JNDI в run-time (во время выполнения программы). Эти управляемые объекты создаются и инициализируются в deploy-time (во время установки) и управляются затем JEE-сервером в run-time.

Подобные элементы и объекты обычно применяют для получения ссылок на сервисы вроде javax.transaction.UserTransaction или javax.transaction.TransactionSynchronizationRegistry.

Ругательства

- RMI = Remote Method Invocation = Удаленный вызов метода
- JNDI = Java Naming & Directory Interface = Интерфейс службы имен и каталогов Java)
- MBean = Managed Bean = Управляемый компонент — компонент, управляемый одним из контейнеров JEE. В JEE7 любой управляемый компонент поддерживает технологию CDI, т.е. является также и CDI MBean.
- ENC = Enterprise/Environment Naming Context = Корпоративный контекст именования — окружение имен для EJB и MBean

Литература

Enterprise JavaBeans 3.1 (6th edition), Andrew Lee Rubinger, Bill Burke, стр. nnn

Изучаем Java EE 7, Энтони Гонсалвес, стр. nnn

EJB3 в действии, (2е издание), Дебу Панда, Реза Рахман, Райан Купрак, Майкл Ремижан, стр. nnn

Java EE 6 и сервер приложений GlassFish v3, Дэвид Хэффельфингер

http://www.sql.ru/forum/1044613-2/rmi-peredacha-po-ssylke