EJB 3.1 教程

学习笔记(转载 http://jerry-wu.gicp.net/)

MashupEye

目 录

JBOSS AS 7 入门了解

前两天 jboss as 7.0.0 Final 正式发布，与 rc1 不同，这次发布了两个版本 Web Profile Only 与 Everything，这两版本其实与 java ee 6 的 Web Profile 与 Full Profile 相对应，Web Profile 与 Full Profile 的介绍参考 javaee6:Web Profile&Full Profile。

jboss as 7 与以前的版本不通，这个版本完全是重新开发的，引入了 modules 的概念，各种 jar 包被放置在 modules 目录下，分成了不同的 module(datasource 也被认为是一个 module,默认的 datasource 在 modules 目录下的 com\h2database\h2 模块，以后要添加自己的数据源也要仿照了添加一个 module)

jboss as 7 启动模式有两种：domain 与 standalone，在 bin 目录下有各自的启动脚本，两种模式的配置分别在 domain 目录与 standalone 目录下

standalone 的配置文件主要是 standalone/configuration/standalone.xml
整个 jboss as 7 是又许多个 subsystem 组成的，每个 subsystem 又各自对应一个或多个 module
我们在配置文件中可以看到各种 modules：

```
<extensions>
    <extension module="org.jboss.as.clustering.infinispan"/>
    <extension module="org.jboss.as.connector"/>
    <extension module="org.jboss.as.deployment-scanner"/>
    <extension module="org.jboss.as.ee"/>
    <extension module="org.jboss.as.ejb3"/>
    <extension module="org.jboss.as.jaxrs"/>
    <extension module="org.jboss.as.jmx"/>
    <extension module="org.jboss.as.jpa"/>
    <extension module="org.jboss.as.logging"/>
    <extension module="org.jboss.as.naming"/>
    <extension module="org.jboss.as.osgi"/>
    <extension module="org.jboss.as.remoting"/>
    <extension module="org.jboss.as.sar"/>
    <extension module="org.jboss.as.security"/>
    <extension module="org.jboss.as.threads"/>
    <extension module="org.jboss.as.transactions"/>
    <extension module="org.jboss.as.web"/>
    <extension module="org.jboss.as.weld"/>
</extensions>
```

还有各种 subsystem，其中一个 datasource 的 subsystem：

```
<subsystem xmlns="urn:jboss:domain:datasources:1.0">
    <datasources>
        <datasource jndi-name="java:jboss/datasources/ExampleDS"
pool-name="ExampleDS">
```

```
5
6  <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-url>
7            <driver>h2</driver>
8            <pool>
9                <min-pool-size>10</min-pool-size>
10               <max-pool-size>20</max-pool-size>
11               <prefill>true</prefill>
12           </pool>
13           <security>
14               <user-name>sa</user-name>
15               <password>sa</password>
16           </security>
17       </datasource>
18       <xa-datasource jndi-name="java:jboss/datasources/ExampleXADS"
19 pool-name="ExampleXADS">
20           <driver>h2</driver>
21           <xa-datasource-property
22 name="URL">jdbc:h2:mem:test</xa-datasource-property>
23           <xa-pool>
24               <min-pool-size>10</min-pool-size>
25               <max-pool-size>20</max-pool-size>
26               <prefill>true</prefill>
27           </xa-pool>
28           <security>
29               <user-name>sa</user-name>
30               <password>sa</password>
31           </security>
32       </xa-datasource>
33       <drivers>
34           <driver name="h2" module="com.h2database.h2">
35
   <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
           </driver>
       </drivers>
   </datasources>
  </subsystem>
```

各个 subsystem 如何配置，还得去看 subsystem 的 doc，比如：

- datasource 采用的项目是 IronJacamar

IronJacamar homepage: http://www.jboss.org/ironjacamar

Project Documentation: http://www.jboss.org/ironjacamar/docs

- messaging 采用的项目是 HornetQ

HornetQ homepage: http://www.jboss.org/hornetq

Project Documentation: http://www.jboss.org/hornetq/docs

- web 采用的项目是 JBossWeb

JBossWeb homepage: http://www.jboss.org/jbossweb

Project Documentation: http://docs.jboss.org/jbossweb/7.0.x/

- web service 采用的项目是 JBossWS

JBossWS homepage: http://www.jboss.org/jbossws

Project Documentation: http://www.jboss.org/jbossws/docs

jboss as 7 文档位置为 http://www.jboss.org/jbossas/docs/7-x，目前文档还没完善

JavaEE6 EJB3.1 简介

ejb 的种类
JavaEE 平台定义了几种 EJB：

- 会话 Bean（Session bean）在业务逻辑的实现中使用的频率很高，它在 EJB 技术中成了很重要的一部分。Session bean 有一下几种特征：
  • Stateless：这种 Session bean 在方法调用中不包含会话状态，任何实例都可以被任意的客户端调用。
  • Stateful：这种 Session bean 包含会话状态，一个实例只能被一个客户端持有。
  • Singleton：单例 session bean，只存在一个实例，并被所有客户端共享，并支持同步调用。
- 消息驱动 Bean(Message-driven beans (MDBs)) 通过使用 JMS 接收异步消息来与外部其他系统集成。这个组件模型主要通过消息中间件(MOM)用来集成系统，MDBs 通常带业务逻辑给会话 Bean

EJB3.1 比上个版本更加简单，一个简单的 java 类和一个标注就能构成一个 ejb，例如：

```
@Stateless
public class BookEJB {
  ……
  public Book findBookById(Long id) {
    ……
  }
  public Book createBook(Book book) {
    ……
  }
}
```

EJB 容器
ejb 是一个服务器端的组件，ejb 与普通 java 类的一个区别就是，ejb 必须要运行在 ejb 容器中，ejb 容器提供 ejb 所需要的运行环境

- 远程客户端通讯（Remote client communication）：不需要编写额外的代码，一个 EJB 客户（例如：另一个 EJB、一个用户接口，一个进程等等）端就可以通过标准的协议来远程调用方法。
- 依赖注入(Dependency injection)：容器可以注入各种资源到 EJB 中(例如:JMS destinations and factories, datasources, 其他 EJB, 环境变量等等)
- 状态管理（State management）：对于有状态会话 Bean，容器管理着它们的状态。你可以为你的客户端保持状态，例如你开发一个桌面程序。
- 池（Pooling）：对于无状态会话 Bean 和消息驱动 Bean，容器池化一些实例，这些实例被多个客户端共享。每次调用完毕，实例被放回到池中，而不是被销毁。
- 组件生命周期（Component life cycle）：容器可靠地管理着组件的生命周期
- 消息（Messaging）：容器允许 MDBs 侦听消息源并消费消息，并不使用太多的 JMS 探测
- 事务管理（Transaction management）：通过声明式事务管理，EJB 可以使用标注来告诉容器它所需要的事务级别，容器管理着事务的提交与回滚。
- 安全（Security）：EJB 可以指定类或者方法级别的访问控制，要求强制验证用户或者角色
- 并发同步（Concurrency support）：除了单例，一些并发申明是需要的，其他所有类型的 EJB 是默认线程安全的。你可以开发性能好的应用而不必担心线程问题。

- 拦截器（Interceptors）：切入点（Cross-cutting concerns）可以放到拦截器中，容器会自动调用它。
- 异步方法调用（Asynchronous method invocation）：在 EJB3.1 中，可以不使用消息而实现异步调用

内嵌容器（Embedded Container）

像 GlassFish, JBoss, Weblogic 等等，在你部署并使用你的 EJB 前，必须要先启动。EJB 3.1 可以指定了内嵌的容器，这样可以轻易地跨服务器，并且内嵌容器可以执行 ejb 在 javase 上并允许客户端程序运行在同一个 JVM 和类加载环境中。内嵌容器 api（在 javax.ejb.embeddable 中定义）提供了 javaEE 容器同样的服务

依赖注入和 JNDI（Dependency Injection and JNDI）

EJB 使用依赖注入来访问各种各样的资源（其他 EJB，datasources，JMS destinations,环境资源等等）以下例子，客户端使用@EJB 标注来获取注入的 EJB 引用

```
1 @EJB
2 private static BookEJB bookEJB;
```

在 EJB 3.1 中, JNDI 名称被规范化，这样代码的可移植性将大幅度提高

java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]

EJB3.1 新特性

EJB3.1 包含了如下新特性：

- 无需接口（No-interface view）：会话 Bean 本地访问无需额外的业务逻辑接口
- War 部署（War deployment）：现在 ejb 组件可以直接打包部署到 war 里边
- 内嵌容器（Embedded container）：新增加了内嵌 api，用来在 JavaSE 环境中执行 ejb 组件(可用于单元测试、批处理等等)
- 单例（Singleton）：这是一个新的组件，它可以更加容易了共享状态
- 丰富的定时服务（Richer timer service）：这个特性允许自动创建 EJB 定时器，并且可以基于日期表达式
- 异步（Asynchrony）：现在可以不使用 MDBs 而进行异步调用
- 轻量级 EJB（EJB Lite）：定义了一个轻量级的 JavaEE 功能子集（例如 JavaEE Web Profile）
- 便携式的 JNDI 名称（Portable JNDI name）：现在规定了查找 EJB 组件的语法，有效地防止了不同 EJB 容器间不同的 EJB 组件查找方式。

EJB3.1 单例会话 Bean（Singleton）

Singleton 是 ejb 中会话 bean 的一种，容器会确保它在每个应用中只会实例化一次。它实现了单例模式。Singleton 在所有客户端之间共享状态。

Singleton 是不能感知群集的。集群是一组容器紧密地合作（共享相同的资源，EJB 等等）。因此，在某些情况下，需要在好几台机器上部署分布式容器集群，每个容器将有它自己的 Singleton 实例。

初始化

当客户端需要访问一个单例会话 bean 的方法时，ejb 容器要确定是否需要实例化单例会话 bean 还是使用在容器中已经存在的实例。但是，某些时候实例化一个单例会话 bean 是比较耗时的（例如，我们需要连接数据库，并且需要查询数千个对象，并缓存起来）。

为了避免这个现象，我们就需要要求 ejb 容器在启动的时候就预先实例化好单例会话 bean。将单例会话 bean 打上@Startup 的标注，ejb 容器就会在启动的时候实例化单例会话 bean，而不是在客户端第一次调用的时候去实例化。如下所示：

```
@Singleton
@Startup
public class CacheEJB {
  // ...
}
```

链接 Singleton（Singleton 依赖）

在某些情况下，存在一些 Singleton，而这些 Singleton 的初始化顺序很重要。@javax.ejb.DependsOn 这个标注就用来描述 Singleton 之间的依赖关系。如下：

```
@Singleton
public class CountryCodeEJB {
  ...
}

@DependsOn("CountryCodeEJB")
@Singleton
public class CacheEJB {
  ...
}
```

@DependsOn 可以有多个字符串参数，每个字符串指定一个单例会话 bean。以下例子展示了 CacheEJB 的初始化依赖于 CountryCodeEJB 和 ZipCodeEJB 的初始化。

```
@Singleton
public class CountryCodeEJB {
  ...
}

@Singleton
```

```
public class ZipCodeEJB {
  ...
}

@DependsOn("CountryCodeEJB", "ZipCodeEJB")
@Singleton
public class CacheEJB {
  ...
}
```

有时，依赖的 Singleton 在同一个应用的另外一个模块中，并且被打包进了不同的 jar 包中（例如：CacheEJB 依赖于 CountryCodeEJB，并且他们分别在 technical.jar 和 business.jar 中），此时的依赖如下：

```
@DependsOn("business.jar#CountryCodeEJB")
@Singleton
public class CacheEJB {
...
}
```

并发

单例会话 bean 只存在一个实例，并且被所有客户端共享。这样有时就需要控制客户端之间的并发访问。有两种方式通过@ConcurrencyManagement 标注控制并发访问：

- 容器管理并发（Container-managed concurrency (CMC)）：容器根据元数据（标注或者 xml 描述文件）控制对实例的并发访问.
- bean 管理并发（Bean-managed concurrency (BMC)）：容器允许并发访问与同步由 bean 来控制

如果没有指定并发管理，CMC 为默认的管理。

1.*容器管理并发*

CMC，作为默认的并发管理，容器负责控制对单例会话 bean 实例的并发访问。你可以使用@Lock 标注来指定当客户端调用方法时容器如何管理并发.@Lock 标注可以有的值为 READ（共享）和 WRITE（独享）

- @Lock(LockType.WRITE)：带有排它锁的方法将不允许并发调用，直至这个方法被调用完毕。例如：客户端 C1 调用了一个带有排它锁的方法，客户端 C2 将不能调用这个方法直至 C1 调用完成。
- @Lock(LockType.READ)：带有共享锁的方法将允许任意多个并发调用。例如：两个客户端 C1 和 C2，可以同时调用带有共享锁的方法。

@Lock 标注可以在类、方法上标注，或者可以同时标注。标注在类上是指它对类中的所有方法都起效果。如果未指定标注属性，默认为@Lock(LockType.WRITE)。

以下例子中，除了 getFromCache()外的所有方法都是 READ 的。

```
@Singleton
@Lock(LockType.READ)
public class CacheEJB {
  private Map<Long, Object> cache = new HashMap<Long, Object>();

  public void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
```

```
8       cache.put(id, object);
9     }
10
11    public void removeFromCache(Long id) {
12      if (cache.containsKey(id))
13        cache.remove(id);
14    }
15
16    @AccessTimeout(value = 20, unit = TimeUnit.SECONDS)
17    @Lock(LockType.WRITE)
18    public Object getFromCache(Long id) {
19      if (cache.containsKey(id))
20        return cache.get(id);
21      else
22        return null;
23    }
24  }
```

getFromCache()方法上还使用了@AccessTimeout 标注，并发访问时，假如调用 getFromCache()方法时未获取到锁并被阻塞超过 20 秒，调用客户端将获得 ConcurrentAccessTimeoutException 异常。

2.*bean 管理并发*

在这种状况下，可以使用 Java 原生态的同步方式，例如：synchronized 和 volatile。BMC (@ConcurrencyManagement(BEAN))状态下使用 synchronized，如下：

```
1   @Singleton
2   @ConcurrencyManagement(ConcurrencyManagementType.BEAN)
3   public class CacheEJB {
4     private Map<Long, Object> cache = new HashMap<Long, Object>();
5
6     public synchronized void addToCache(Long id, Object object) {
7       if (!cache.containsKey(id))
8         cache.put(id, object);
9     }
10
11    public synchronized void removeFromCache(Long id) {
12      if (cache.containsKey(id))
13        cache.remove(id);
14    }
15
16    public Object getFromCache(Long id) {
17      if (cache.containsKey(id))
18        return cache.get(id);
19      else
20        return null;
21    }
```

```
}
```

3.*并发访问超时于不允许并发访问*

@AccessTimeout 用来指定访问被阻塞时的超时时间，@AccessTimeout 的值是-1 时指客户端调用可以无限阻塞直至方法被调用，@AccessTimeout 的值是 0 时指并发访问不被允许，假如方法被另外一个客户端调用时，当前客户端调用结果将会抛出一个 ConcurrentAccessException 异常。

以下例子中，CacheEJB 不允许在 addToCache()方法上并发访问。假如客户端 A 添加一个对象道缓存中，客户端 B 在同一个时间想做同样的事情，客户端 B 将得到一个异常。另外两个是默认的 CMC @Lock(WRITE)

```
@Singleton
public class CacheEJB {
  private Map<Long, Object> cache = new HashMap<Long, Object>();

  @AccessTimeout(0)
  public void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
  }

  public void removeFromCache(Long id) {
    if (cache.containsKey(id))
      cache.remove(id);
  }

  public Object getFromCache(Long id) {
    if (cache.containsKey(id))
      return cache.get(id);
    else
      return null;
  }
}
```

EJB3.1 业务接口与 bean 类

接口与 bean 类

- 业务接口：接口包含了业务方法的申明，可以被客户端访问，并且被 bean 类实现。会话 bean 可以有本地接口，远程接口或者根本没有接口（没有接口的会话 bean 只能被本地访问）
- bean 类：bean 类包含了业务方法的实现，可以实现零个或多个业务接口。会话 bean 必须根据它本身的类型标注上@Stateless,@Stateful 或者 @Singleton

远程、本地和无接口会话 bean

根据客户端调用会话 bean 的位置，bean 类需要实现远程接口或本地接口，或者根本就没有接口。假如你的调用 EJB 的客户端运行在 EJB 容器的 jvm 实例外，那么你必须使用远程接口。当会话 bean 于客户端运行在同一个 jvm 内，你可以本地调用。你可以用一个 EJB 调用另外一个 EJB 或者运行于同一个 JVM 上的 web 容器中的 web 组件（servlet，JSF）。

一个会话 bean 可以实现零个或几个接口。一个业务接口是一个不继承任何 EJB 指定接口的纯 JAVA 接口。和 java 接口一样，业务接口定义一些供客户端调用的方法。你可以使用以下标注：

- @Remote：表示一个远程业务接口。作为 RMI 协议的一部分，方法参数按值传递，并且需要序列化
- @Local：表示一个本地业务接口。方法参数从客户端到 bean 使用引用传递

你不能在同一个接口上打多于一个标注（意思就是只能打一个标注，要么本地，要么远程）。

以下代码展示了被 ItemEJB 无状态会话 bean 实现的一个本地接口（ItemLocal）和一个远程接口（ItemRemote）。代码中，findCDs()方法可以被远程调用，也可以被本地调用，因为它被两个接口申明了。而 createId（）方法只能通过 RMI 被远程访问。

```
@Local
public interface ItemLocal {
  List findBooks();
  List findCDs();
}

@Remote
public interface ItemRemote {
  List findBooks();
  List findCDs();
  Book createBook(Book book);
  CD createCD(CD cd);
}

@Stateless
public class ItemEJB implements ItemLocal, ItemRemote {
  ...
}
```

另为对于上边的代码，你可能想在 bean 类里边指定接口类型。这样你需要在@Local 和@Remote 标注中包含接口的名称，如下：

```
public interface ItemLocal {
  List findBooks();
  List findCDs();
}

public interface ItemRemote {
  List findBooks();
  List findCDs();
  Book createBook(Book book);
  CD createCD(CD cd);
}

@Stateless
@Remote (ItemRemote.class)
@Local (ItemLocal.class)
public class ItemEJB implements ItemLocal, ItemRemote {
  ...
}
```

Web Service 接口

除了通过 RMI 远程调用，无状态会话 bean 也可以通过 SOAP web service 或者 RESTful web service 被远程调用。

以下代码展示了一个带有本地接口、SOAP web service（@WebService）和 RESTful web service（@Path）的无状态会话 BEAN，需要注意的是这两个标注不是 EJB 的一部分，他们分别来自 JAX-WS 和 JAX-RS

```
@Local
public interface ItemLocal {
  List findBooks();
  List findCDs();
}

@WebService
public interface ItemSOAP {
  List findBooks();
  List findCDs();
  Book createBook(Book book);
  CD createCD(CD cd);
}

@Path(/items)
public interface ItemRest {
  List findBooks();
}

```

```
@Stateless
public class ItemEJB implements ItemLocal, ItemSOAP, ItemRest {
  ...
}
```

bean 类

一个会话 bean 是一个实现了业务逻辑的标准 java 类。开发一个会话 bean 的要求如下：

- 这个类必须被@Stateless, @Stateful, @Singleton 标注，或者在部署描述文件中等价的 xml 配置
- 必须实现它接口中的任意方法
- 类必须被定义为 public 的，并且不能是 final 或者 abstract
- 类必须有一个 public 的无参构造函数，以便容器使用它来创建实例
- 类不能定义 finalize()方法
- 业务方法的名字不能以 ejb 开头，并且他们不能是 final 或者 static
- 一个远程方法的参数和返回值必须是合法的 RMI 类型

EJB3.1 ejb 客户端 与 会话上下文

ejb 客户端

会话 bean 的客户端可以是任意组件（一个 POJO，一个图形界面（swing），一个受管 bean（managed bean 带有@javax.annotation.ManagedBean 标注的 bean），一个 servlet，一个 JSF 管理的 bean，一个 web service（SOAP 或者 REST）或者是另一个 EJB）。
调用一个会话 bean 的方法，客户端不是直接实例化这个 bean（使用 new）。它需要一个这个 bean 的引用（或者是它接口的引用）。它可以通过依赖注入（@EJB 标注）或者查找 JNDI 获得。除非特殊指定，客户端调用会话 bean 是同步地。EJB3.1 允许很好地异步调用方法了。

@EJB
JavaEE 使用了数个标注来注入资源(@Resource)、entity managers（@PersistenceContext）和 web servcie（@WebServiceRef）等的引用。但@javax.ejb.EJB 标注是专门用来将会话 bean 的引用注入到客户端中。依赖注入只能在被管环境中使用，例如（ejb 容器，web 容器和客户端应用容器）。
客户端如果要调用无接口的会话 bean，它需要获得会话 bean 类实例的引用。例如：下面的代码，客户端使用@EJB 标注来获得 ItemEJB 的引用

```
@Stateless
public class ItemEJB {
  ...
}

//客户端代码
@EJB ItemEJB itemEJB;
```

假如会话 bean 实现了多个接口，客户端需要指定指定它需要的引用。下边的代码 ItemEJB 实现了两个接口。客户端可以通过它的本地或者远程接口调用，但是不能通过无接口的方式调用了。

```
@Stateless
@Remote (ItemRemote.class)
@Local (ItemLocal.class)
public class ItemEJB implements ItemLocal, ItemRemote {
  ...
}
// 客户端代码
@EJB ItemEJB itemEJB; // 部署的时候会抛出一个异常
@EJB ItemLocal itemEJBLocal;
@EJB ItemRemote itemEJBRemote;
```

假如会话 bean 至少暴露了一个接口，它可以在这个 bean 上标注@LocalBean 来显示地暴露它为一个无接口 bean。下边的代码中，客户端可以通过远程、本地和无接口方式来调用 bean

```
@Stateless
@Remote (ItemRemote.class)
@Local (ItemLocal.class)
@LocalBean
```

```
public class ItemEJB implements ItemLocal, ItemRemote {
  ...
}
// 客户端代码
@EJB ItemEJB itemEJB;
@EJB ItemLocal itemEJBLocal;
@EJB ItemRemote itemEJBRemote;
```

基于你客户端的环境，你可能不能使用注入（组件不被容易管理），这种情况下，你可以通过他们统一的 JNDI 名称使用 JNDI 来查找会话 bean

轻便的 JNDI 名字

会话 bean 也可以使用 JNDI 来查找。JNDI 多用于客户端不被容器管理并且不能使用依赖注入时的远程调用。但是 JNDI 也被本地客户端使用，尽管依赖注入更加方便。查找会话 bean，客户端程序需要使用 JNDI API 来与目录名称服务通讯。

一旦一个会话 bean 被部署到容器中，它会自动地被绑定一个 JNDI 名字。JavaEE6 之前，这个名字不是标准的，所以会话 bean 被部署到不同的服务器（GlassFish, JBoss, WebLogic 等等）中，会有不同的名字。JavaEE6 定义了统一了命名规范：

java:<scope>[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]

JNDI 名称中的每部分都有各自的含义,如下：

- <scope>定义了一系列对应到 JavaEE 应用范围的标准命名空间：
1. global: java:global 前缀允许一个在 JavaEE 应用外执行的组件访问一个全局命名空间
2. app: java:app 前缀允许一个 JavaEE 应用内执行的组件访问一个应用指定的命名空间
3. module: java:module 前缀允许一个 JavaEE 应用内执行的组件访问一个模块指定的命名空间
4. comp: java:comp 前缀是一个私有的组件指定的命名空间，它不能被其他组件访问

- <app-name> 仅仅在会话 bean 被打包进一个 ear 文件中时才需要。默认名称为 ear 文件的名称(不带有.ear 后缀)
- <module-name> 是会话 bean 被打包进去的模块的名称。它可以是一个在单独 jar 包中的 EJB 模块，也可以是一个在 war 包中的 web 模块。默认名称为不带有文件后缀的打包文件名称
- <bean-name> 就是这个会话 bean 的名字
- <fully-qualified-interface-name> 是所有定义的业务接口的名称，没有业务接口的，就是 bean 的类名

为了说明这个名称转化，我们举个 ItemEJB 的例子。ItemEJB 是，被打包到 cdbookstore.jar 中().这个 EJB 有一个远程接口和一个无接口视图（使用@LocalBean 标注）

```
package com.apress.javaee6;
@Stateless
@LocalBean
@Remote (ItemRemote.class)
public class ItemEJB implements ItemRemote {
  ...
}
```

一旦被部署，一个其他的组件以下的全局 JNDI 名称来访问 ItemEJB

java:global/cdbookstore/ItemEJB
java:global/cdbookstore/ItemEJB!com.apress.javaee6.ItemEJB
java:global/cdbookstore/ItemEJB!com.apress.javaee6.ItemRemote

需要注意的是，假如 ItemEJB 被部署道 ear 文件中（例如：myapplication.ear），你必须要使用，如下：

java:global/myapplication/cdbookstore/ItemEJB
java:global/myapplication/cdbookstore/ItemEJB!com.apress.javaee6.ItemEJB
java:global/myapplication/cdbookstore/ItemEJB!com.apress.javaee6.ItemRemote

容器也被要求成生 java:app 和 java:module 命名空间的 JNDI 名称。所以一个组件被部署道与 ItemEJB 相同的应用中，可以使用以下的 JNDI 名称来查找：

java:app/cdbookstore/ItemEJB
java:app/cdbookstore/ItemEJB!com.apress.javaee6.ItemEJB
java:app/cdbookstore/ItemEJB!com.apress.javaee6.ItemRemote
java:module/ItemEJB
java:module/ItemEJB!com.apress.javaee6.ItemEJB
java:module/ItemEJB!com.apress.javaee6.ItemRemote

会话上下文

会话 bean 是运行在容器中的业务逻辑组件。通常他们不访问容器或者直接使用容器服务（事务、安全、依赖注入等）。但是有些时候，在 bean 代码中明确地需要使用容器资源（例如：创建一个事务用于回滚）。这些可以用 java.ejb.SessionContext 接口来实现。SessionContext 允许编程来访问提供给会话 bean 实例的运行时上下文。SessionContext 继承了 javax.ejb.EJBContext 接口。一些 SessionContext API 的方法在下边描述：

1. getCallerPrincipal
   返回与调用相关的 java.security.Principal associated
2. getRollbackOnly
   查看当前事务是否被标记为回滚
3. getTimerService
   返回 java.ejb.TimerService 接口。只有 stateless beans 和 singletons 可以使用这个方法。Stateful session beans 不能是定时的对象。
4. getUserTransaction
   返回 javax.transaction.UserTransaction 接口来标志事务。只有使用 bean-managed transaction (BMT)的 session beans 可以使用这个方法。
5. isCallerInRole
   查看调用者是否有安全权限
6. lookup
   允许会话 bean 在 JNDI 命名上下文中查找它的环境变量值
7. setRollbackOnly
   允许 bean 标志当前事务为回滚状态。
8. wasCancelCalled
   检查客户端是否在客户端的 Future 对象对应的正在执行的异步业务逻辑方法上调用了 cancel()方法。

一个会话 bean 可以通过使用@Resource 标注注入一个 SessionContext 的引用来访问它的环境上下文

```
@Stateless
public class ItemEJB {
  @Resource
  private SessionContext context;
  ...
  public Book createBook(Book book) {
    ...
    if (cantFindAuthor())
      context.setRollbackOnly();
  }
}
```

EJB3.1 内嵌容器

会话 bean 是容器管理的组件，这是他们的优势。这导致你需要总是要在容器中运行 EJB，甚至你需要测试他们。
在 EJB3.1 中，通过创建一个内嵌的 EJB 容器来解决这个问题。EJB3.1 定义了一些标准的 API 来在 JavaSE 中执行 EJB。这个可嵌入的 API（javax.ejb.embeddable 包）允许客户端在自己的 JVM 中实例化一个 EJB 容器。这个内嵌的容器提供了同 JavaEE 容器一样的环境和服务：注入、事务、生命周期等等。内嵌式 EJB 容器只提供 EJB Lite 容器 API 的服务，不是 full EJB 容器。

下边展示了一个 JUnit 测试类，它使用了 javax.ejb.embeddable.EJBContainer 来启动容器，查找 EJB，调用 EJB 方法。

```
public class ItemEJBTest {
  private static EJBContainer ec;
  private static Context ctx;

  @BeforeClass
  public static void initContainer() throws Exception {
    ec = EJBContainer.createEJBContainer();
    ctx = ec.getContext();
  }

  @AfterClass
  public static void closeContainer() throws Exception {
    if (ec != null)
    ec.close();
  }

  @Test
  public void shouldCreateABook() throws Exception {
    // 创建一个 book 的实例
    Book book = new Book();
    book.setTitle("The Hitchhiker's Guide to the Galaxy");
    book.setPrice(12.5F);
    book.setDescription("Science fiction comedy book");
    book.setIsbn("1-84023-742-2");
    book.setNbOfPage(354);
    book.setIllustrations(false);
    // 查找 EJB
    ItemEJB bookEJB = (ItemEJB) ctx.lookup("java:global/classes/ItemEJB");
    // 存储 book 到 EJB
    book = itemEJB.createBook(book);
    assertNotNull("ID should not be null", book.getId());
    // 从数据库中查询所有 book
```

```
33      List<Book> books = itemEJB.findBooks();
34      assertNotNull(books);
35    }
36  }
```

在 initContainer()方法中，EJBContainer 使用了一个工厂方法（createEJBContainer()）来创建一个容器实例。默认情况下，内嵌容器查找客户端的 classpath 来寻找 EJB 并实例化。一旦容器实例化了，方法获得容器上下文（使用 EJBContainer.getContext()类返回一个 javax.naming.Context）来查找 ItemEJB（通过 JNDI 名称）。

需要注意的是 ItemEJB 是一个无状态会话 bean，并通过无接口方式暴露了业务逻辑方法。它使用了注入、容器管理事务和一个 JPA Book entity。closeContainer()方法调用了 EJBContainer.close()来关闭内嵌容器的实例。

JavaEE6 EJB3.1 异步调用

默认状况下，通过远程接口、本地接口或无接口视图调用会话 bean 是同步的：一个客户端调用一个方法，然后被阻塞直到被处理完成，然后返回结果，客户端继续以下的工作。但是许多长时间执行的任务都有需要异步调用的需求。例如：打印一个订单，依赖于打印机是否在线，是否有足够的纸，文档是否已经在打印池中准备好，这是一个很花时间的任务。当客户端调用一个方法来打印文档，它希望调用后不用等待，直接继续执行下边的任务。

在 EJB3.1 之前，异步调用只能通过 JMS 和 MDBs 来实现。在 EJB3.1 中，你可以在一个会话 bean 的方法上添加一个 @javax.ejb.Asynchronous 标注来实现异步调用。下边例子中，OrderEJB 有一个方法用来给客户发送 e-mail，另一个用来打印订单。这两个方法都是很耗时的，他们都标注上了 @Asynchronous。

```
@Stateless
public class OrderEJB {
  @Asynchronous
  public void sendEmailOrderComplete(Order order, Customer customer) {
    // 发送 E-mail
  }

  @Asynchronous
  public void printOrder(Order order) {
    // 打印订单
  }
}
```

当客户端调用 printOrder()和 sendEmailOrderComplete()时，容器立即返回控制权给客户端，客户端可以继续下边的任务，而被调用的任务会在另外一个线程中被执行。上边的两个方法中都返回了 void，但有时候我们需要有返回值。异步调用可以返回 void，也可以返回 java.util.concurrent.Future 对象，其中 V 代表了返回值。Future 对象允许在另外一个线程中执行的方法返回一个值。客户端可以使用 Future API 来获得结果，设置可以终止调用。

下边例子中的方法返回了一个 Future。sendOrderToWorkflow()方法使用了工作流来处理 Order 对象。假设它调用了多个企业组件，每一步都返回一个状态（一个 integer）。当客户端异步调用 sendOrderToWorkflow()方法时，它希望得到这个工作流的状态。客户端可以使用 Future.get()方法来得到这个结果，或者基于其他原因，它想取消调用，它可以使用 Future.cancel()。假如客户端调用了 Future.cancel()，容器只有在执行还没开始的时候将尝试取消这个异步执行。sendOrderToWorkflow()方法使用了 SessionContext.wasCancelCalled()方法来检查客户端是否发出了取消调用的请求。这个方法返回实现了 Future 的 javax.ejb.AsyncResult。

有一点需要注意的是，AsyncResult 是作为一种结果值返回给容器的，而不是直接给调用者。

```
@Stateless
@Asynchronous
public class OrderEJB {
  @Resource
```

```
  SessionContext ctx;

  public void sendEmailOrderComplete(Order order, Customer customer) {
    // 发送 E-mail
  }

  public void printOrder(Order order) {
    // 打印订单
  }

  public Future<Integer> sendOrderToWorkflow(Order order) {
    Integer status = 0;
    // 处理
    status = 1;
    if (ctx.wasCancelCalled()) {
      return new AsyncResult<Integer>(2);
    }
    // 处理
    return new AsyncResult<Integer>(status);
  }
}
```

@Asynchronous 标注同样也可以标注在类上。这表示这个类里边的所有方法都是异步调用的。

当客户端调用了 sendOrderToWorkflow()方法，它需要执行 Future.get()来得到最终的返回结果

```
Future<Integer> status = orderEJB.sendOrderToWorkflow (order);
Integer statusValue = status.get();
```

EJB3.1 部署描述文件、环境变量以及依赖注入

部署描述文件

xml 部署描述文件（ejb-jar.xml）是标注的替代方式，任何标注都相当于是一个 xml 标签。假如标注与部署描述都使用了，在部署的时候部署描述文件里边的设置将会覆盖标注。下边我们看下 ItemEJB 在 ejb-jar.xml 文件里边描述的样子。它定义了 bean 类，远程和本地接口，类型（Stateless）和它使用的容器管理事务（CMT）。描述了会话 bean 的环境变量值。

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
version="3.1">
  <enterprise-beans>
    <session>
      <ejb-name>ItemEJB</ejb-name>
      <ejb-class>com.apress.javaee6.ItemEJB</ejb-class>
      <local>com.apress.javaee6.ItemLocal</local>
      <local-bean/>
      <remote>com.apress.javaee6.ItemRemote</remote>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>aBookTitle</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Beginning Java EE 6</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

假如会话 bean 被部署到 jar 文件中，部署描述文件的位置应在 META-INF/ejb-jar.xml。假如部署进了 war 里边，它应存在 WEB-INF/ejb-jar.xml。XML 配置在处理特定环境而不是特定代码标注时非常有用（例如：在开发环境中使用一种 ejb 而在测试和产品环境中使用另一种 ejb）

依赖注入

依赖注入在 JavaEE6 中将资源的引用注入到属性中非常简单同时也非常强大。不同于应用在在 JNDI 中查找资源，他们是被容器注入的。

容器可以使用不同的标注（或者部署描述）来向会话 bean 中注入各种各样的资源：

- @EJB：向被标注的属性中注入一个本地、远程或者无接口的 EJB 引用
- @PersistenceContext 和@PersistenceUnit：表述了 EntityManager 与 EntityManagerFactory 的依赖关系
- @WebServiceRef：注入一个 web service 的引用

- @Resource：注入各种资源，例如：数据源（JDBC datasources），会话上下文（session context），事务（user transactions），JMS 连接工厂和目的地（JMS connection factories and destinations），环境变量（environment entries），定时服务（the timer service）等等
以下是一个无状态会话 bean 的代码片段，它使用了各种标注来向属性中注入各种资源。
这些标注可以放在实例的属性上，也可以放在 setter 方法上。

```
@Stateless
public class ItemEJB {
  @PersistenceContext(unitName = "chapter07PU")
  private EntityManager em;
  @EJB
  private CustomerEJB customerEJB;
  @WebServiceRef
  private ArtistWebService artistWebService;
  private SessionContext ctx;
  @Resource
  public void setCtx(SessionContext ctx) {
    this.ctx = ctx;
  }
  ...
}
```

环境命名上下文

当你开发企业应用时，从一个部署环境换到另一个时，有一些参数需要修改（依赖于你开发所在的国家，应用的版本等等）。例如，在 CD-BookStore 应用中，ItemEJB 把条目的价格转化到当前应用部署所在国家的价格。假如你在欧洲部署这个无状态会话 bean，你需要你需要把条目的价格乘以 0.8 以及修改本地地址为欧洲。

```
@Stateless
public class ItemEJB {
  public Item convertPrice(Item item) {
    item.setPrice(item.getPrice() * 0.80);
    item.setCurrency("Euros");
    return item;
  }
}
```

你应该知道，hard-coding 这些参数的问题是每次修改区域，你都需要修改你的代码，重新编译，重新部署。还有一种方法是将这些参数放入到数据库中，每次调用 convertPrice()方法的时候都去访问一次数据库。这样很浪费资源。此时你很想把这些参数放在某些地方，这样在每次部署前修改下就行了。部署描述文件就是这些放置这些参数的地方。
部署描述文件（ejb-jar.xml）在 EJB3.1 中是可选的，但是在写环境变量时使用它是很正常，同时也是推荐使用的。在部署描述文件中定义的环境变量条目可以通过依赖注入（或者通过 JNDI）来访问。环境变量支持以下 java 类型：String, Character, Byte, Short, Integer, Long, Boolean, Double, and Float。
下边展示了为 ItemEJB 配置了两个环境变量的 ejb-jar.xml：带有 Euros 值的 String 类型的 currencyEntry 和 带有 0.80 值的 Float 类型的 changeRateEntry。

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd" version="3.1">
  <enterprise-beans>
    <session>
      <ejb-name>ItemEJB</ejb-name>
      <ejb-class>com.apress.javaee6.ItemEJB</ejb-class>
      <env-entry>
        <env-entry-name>currencyEntry</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>Euros</env-entry-value>
        </env-entry>
      <env-entry>
      <env-entry-name>changeRateEntry</env-entry-name>
        <env-entry-type>java.lang.Float</env-entry-type>
        <env-entry-value>0.80</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

现在参数被放到了外边的部署描述文件中，ItemEJB 可以使用依赖注入来得到各个环境变量值。下边例子中，@Resource(name = “currencyEntry”)向 currency 属性注入了 currencyEntry 的值。需要注意的是，环境变量中的数据类型与被注入属性的类型要兼容，不然容器会抛出异常。

```
@Stateless
public class ItemEJB {
  @Resource(name = "currencyEntry")
  private String currency;
  @Resource(name = "changeRateEntry")
  private Float changeRate;
  public Item convertPrice(Item item) {
    item.setPrice(item.getPrice() * changeRate);
    item.setCurrency(currency);
    return item;
  }
}
```

JavaEE6 EJB3.1 定时服务

EJB 定时服务是一个容器服务，它允许 EJB 被注册成回调的服务。EJB 通知是基于日历的计划服务，可以在指定的时间，在指定的一段时间以后，也可以是循环的时间间隔。容器保存了所有的定时记录，在指定的时间调用适当的 bean 实例的方法。

注意：无状态会话 bean，单例，MDB 可以被注册成定时服务，但是有状态会话 bean 不能也不应当使用定时 API

假如 bean 中存在被@Schedule 标注的方法，定时服务可以在部署的时候自动被容器创建。同时，定时服务也可以编写代码来创建，但必须提供一个被@Timeout 标注的回调方法。

基于日历的表达式

定时服务使用了基于日历的语法，它是从 Unix 的 cron 工具演化而来。这个表示式用在编程式创建定时服务（使用 ScheduleExpression 类），也可以用在自动定时服务（通过@Schedule 标注或者部署描述）。以下是创建基于日历的表达式属性：

属性	描述	可能值	默认值
second	秒数（一分钟内可以的秒数）	[0,59]	0
minute	分钟数（一小时内可以的分钟数）	[0,59]	0
hour	小时数（一天内可以的小时数）	[0,23]	0
dayOfMonth	天数（一个月内可以的天数）	[1,31] 或者{"1st", "2nd", "3rd", . . . , "30th", "31st"} 或者{"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"} 或者 "Last" (一个月的最后一天) 或者 -x (一个月的倒数第 x 天)	*
month	月数（一年内的可以的月数）	[1,12] 或者 {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"}	
dayOfWeek	一星期内的天数	[0,7] 或者 {"Sun", "Mon", "Tue", "Wed", Thu", "Fri", "Sat"}—"0" 和"7" 都是指星期天	*
year	特定的年份	一个四位数的年份	*
timezone	特定的时区	由 zoneinfo（或 TZ）数据库提供的时区列表	

每个属性（秒，分钟，小时）都支持以不同的形式来表示。例如：你有一些日期或者一个年份范围。下边描述下不同的表达形式.

形式	描述	例子
单个值	属性只有一个可能的值	year = "2010" month= "May"
通配符	属性所有可能的值	second = "*" dayOfWeek = "*"
列表	属性有多个值，以,隔开	year = "2008,2012,2016" dayOfWeek ="Sat,Sun" minute = "0-10,30,40"
范围	属性有在一定范围内的值，以-隔开	second = "1-10". dayOfWeek = "Mon-Fri"
增量	属性有一个起点和一个时间间隔，以/隔开	minute = "*/15" second = "30/10"

如果你之前用过 Unix 的 cron，这将很熟悉也很简单，有了这丰富的语法，几乎可以表达出任何类型的日期，例如：

例子	表达式
每周三的午夜	dayOfWeek="Wed"
每周三的午夜	second="0", minute="0", hour="0", dayOfMonth="*", month="*", dayOfWeek="Wed", year="*"
每个工作日的 6:55	minute="55", hour="6", dayOfWeek="Mon-Fri"
每个工作日的 6:55,巴黎时区	minute="55", hour="6", dayOfWeek="Mon-Fri", timezone="Europe/Paris"
每天每一小时的每一分钟	minute="*", hour="*"
每天每一小时每一分钟的每一秒	second="*", minute="*", hour="*"
每周一和周五的中午过后 30 秒	second="30", hour="12", dayOfWeek="Mon, Fri"
一小时中的每 5 分钟	minute="*/5", hour="*"
一小时中的每 5 分钟	minute="0,5,10,15,20,25,30,35,40,45,50,55", hour="*"
12 月最后一个星期一的下午 3 点	hour="15", dayOfMonth="Last Mon", month="Dec"
每个月倒数第 3 天的下午 1 点	hour="13", dayOfMonth="-3"

在每月的第二个星期二中午开始后每隔一小时	hour="12/2", dayOfMonth="2nd Tue"
早上 1 点和 2 点时，每隔 14 分钟	minute = "*/14", hour="1,2"
早上 1 点和 2 点时，每隔 14 分钟	minute = "0,14,28,42,56", our = "1,2"
在 30 秒的时候开始，每隔 10 秒钟	second = "30/10"
在 30 秒的时候开始，每隔 10 秒钟	second = "30,40,50"

自动创建定时服务

基于元数据，定时服务可以在容器部署的时候自动被创建。每个带有@javax.ejb.Schedule 或者@Schedules（或者在 ejb-jar.xml 部署描述中的 xml 标签）的方法，容器都会自动创建一个定时服务。每一个@Schedule 标注对应一个持久化的计时器，但是你也可以定义非持久化的定时器。

下边的 StatisticsEJB 定义了一些方法，statisticsItemsSold()方法创建了一个每月第一天上午 5:30 被调用的计时服务。generateReport()定义了两个定时服务：每天上午 2 点和每星期 3 的下午 2 点。refreshCache()创建了一个非持久化的定时服务，每 10 分钟刷新下缓存。

```
@Stateless
public class StatisticsEJB {
  @Schedule(dayOfMonth = "1", hour = "5", minute = "30")
  public void statisticsItemsSold() {
    // ...
  }
  @Schedules({
     @Schedule(hour = "2"),
     @Schedule(hour = "14", dayOfWeek = "Wed")
  })
  public void generateReport() {
     // ...
  }

  @Schedule(minute ="*/10", hour = "*", persistent = false)
  public void refreshCache() {
    // ...
  }
}
```

编程式创建定时服务

如果要编程式创建定时服务，EJB 需要依靠依赖注入、EJBContext (EJBContext.getTimerService()) 或者 JNDI 查找来访问 javax.ejb.TimerService 接口。TimerService 的 API 有几种方法，可以允许你创建不同种类的定时服务，一般分一下 4 类：

- createTimer：基于日期、时间间隔、持续时间来创建定时服务。这些方法不是使用基于日历表达式的。

- createSingleActionTimer：创建一个在给定时间点或者过指定时间的单一的定时器。当超时回调方法被成功调用后，容器将移除这个定时服务。
- createIntervalTimer：创建一个基于时间间隔的定时服务，在第一个给定的时间点调用后，每隔一段给定的时间将调用一次。
- createCalendarTimer：使用 ScheduleExpression 类创建基于日历表达式的定时服务

ScheduleExpression 帮助类允许你通过编程式来创建一个基于日历表达式的定时服务。下边是一些例子：

```
new ScheduleExpression().dayOfMonth("Mon").month("Jan");
new ScheduleExpression().second("10,30,50").minute("*/5").hour("10-14");
new ScheduleExpression().dayOfWeek("1,5").timezone("Europe/Lisbon");
new ScheduleExpression().dayOfMonth(customer.getBirthDay())
```

所有 TimerService 的方法（createSingleActionTimer, createCalendarTimer 等等）都返回一个包含了已创建定时信息（什么时候被创建，是否持久化）的 javax.ejb.Timer 对象。Timer 还允许 EJB 在定时点之前取消定时服务。当过了时间点，容器调用 bean 中标注了@javax.ejb.Timeout 的方法，并传递 Timer 对象。一个 bean 只能有一个@Timer 方法。

下边例子中，当 CustomerEJB 在系统中创建了一个新的客户（通过 createCustomer()方法），它创建了一个在客户生日上的定时服务。这样，每年容器都会触发 bean 去创建和发送生日 e-mail 给客户。为了做到这个，无状态会话 bean 首先需要注入一个定时服务的引用（通过使用@Resource）。createCustomer()方法持久化客户信息到数据库中，并使用客户的生日来创建一个 ScheduleExpression。

```
@Stateless
public class CustomerEJB {
  @Resource
  TimerService timerService;
  @PersistenceContext(unitName = "chapter07PU")
  private EntityManager em;
  public void createCustomer(Customer customer) {
    em.persist(customer);
    ScheduleExpression birthDay = new ScheduleExpression().
        dayOfMonth(customer.getBirthDay()).month(customer.getBirthMonth());
    timerService.createCalendarTimer(birthDay, new TimerConfig(customer,
true));
  }

  @Timeout
  public void sendBirthdayEmail(Timer timer) {
    Customer customer = (Customer) timer.getInfo();
    // ...
  }
}
```

一旦定时器被创建，容器每年将调用@Timeout 方法（sendBirthdayEmail()），并传递 Timer 对象给它。由于定时服务同客户对象被序列化，这个方法可以通过 getInfo()来访问它。

EJB3.1 会话 bean 的生命周期

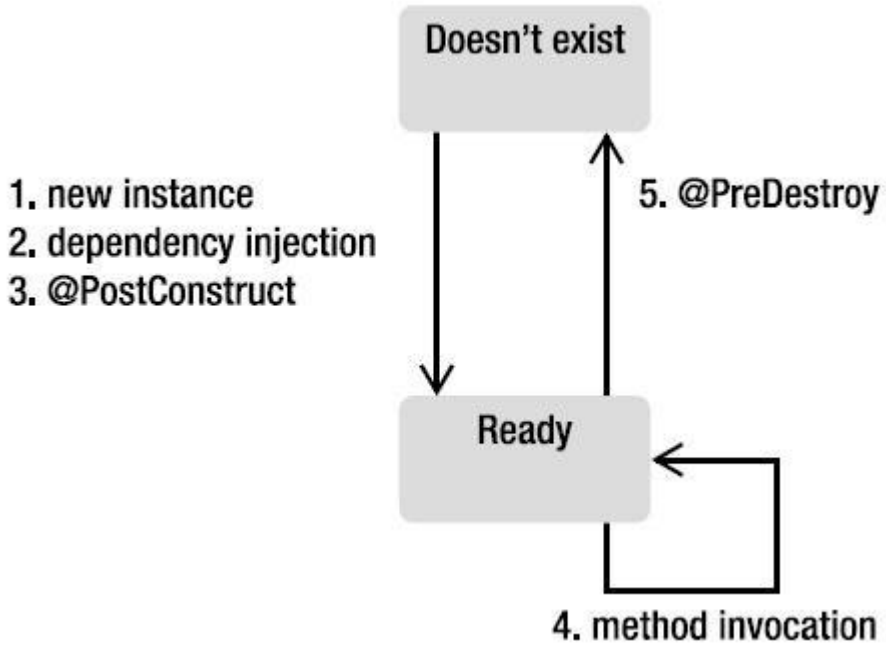
ejb 客户端并不使用 new 来创建会话 bean 的实例。它通过依赖注入或者 JNDI 查找来获得会话 bean 的引用。仅仅只有容器才能创建和销毁实例。这意味着容器有责任来管理 bean 的生命周期。

所有会话 bean 在它们的生命周期中有两个明显的阶段：创建和销毁。另外，有状态会话 bean 还有钝化和活化两阶段。

Stateless 和 Singleton

Stateless 和 Singleton 事实上有共同点，它们都没有客户端来保持会话状态。两个会话 bean 都允许被任意客户端访问，有相同的生命周期，如下：

1. 一个 stateless 或者 singleton 的会话 bean 的生命周期开始于当一个客户端请求一个 bean 的引用（通过依赖注入或者 JNDI 查询）。在 singleton 的情况下，它也可以在容器启动的时候开始（使用了 @Startup 标注）。容器创建一个新的会话 bean 实例。
2. 假如新创建的实例通过标注（@Resource, @EJB, @PersistenceContext 等等）或者部署描述使用了依赖注入，容器注入所有它所需要的资源。
3. 假如实例有一个标注了@PostContruct 的方法，容器将调用这个方法
4. bean 实例处理客户端调用，保持在就绪模式以处理接下来的调用。Stateless beans 将一直呆在就绪模式直到容器从池中释放一些空间。Singletons 将一直保持在就绪模式直到容器关闭。
5. 容器不再需要这个实例，它调用标注了@PreDestroy 的方法（假如这样的方法存在），然后结束这个 bean 实例的生命周期



Stateless 和 singleton beans 使用同样的生命周期，但是在创建和销毁上有一些不同。

当一个 stateless 会话 bean 被销毁，容器创建数个实例，并将他们加入到池中。当一个客户端调用 stateless 会话 bean 的方法，容器从池中选取一个实例，代理对这个实例方法的调用，然后把它返回到池中。当容器不在需要这个实例的时候（通常容器希望缩小池中实例的数量），容器将销毁它。

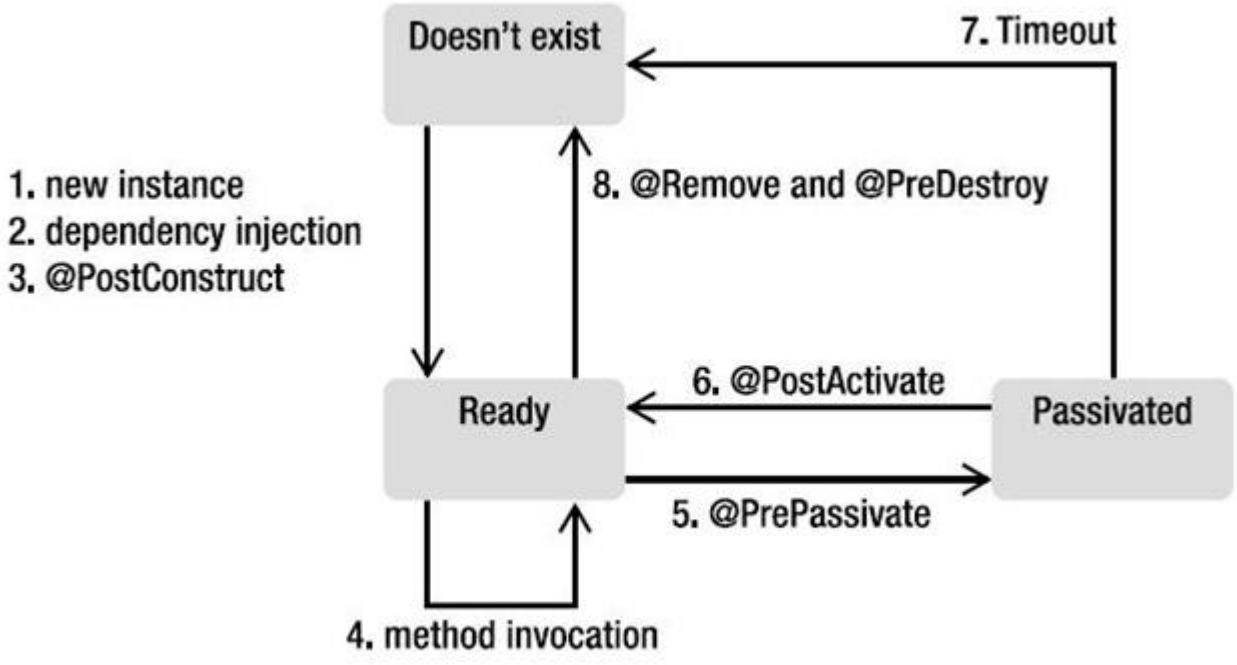
对于 singleton 会话 bean，创建依赖于他们是否急切需要实例化（@Startup），或者他们被依赖（@DependsOn）的另外一个 singleton 是否急切需要创建。如果答案是肯定的，实例将在部署的时候被创建。如果答案是否定的，容器将在一个业务方法被客户端调用的时候创建实例。因为 singleton 持续在应用的整个过程中，它的实例在容器关闭的时候被销毁。

Stateful

Stateful 会话 bean 在代码编写上与 stateless 和 singleton 会话 bean 没有太大的区别：仅仅标注不同（@Stateful 代替了@Stateless 和@Singleton）。但是真正的不同在于 stateful bean 保持与客户端会话的状态，因此它有了完全不同的生命周期。容器生成一个实例，并只将它指定给一个客户端。因此，每个从这个客户端来的请求都会被传递给同一个实例。假如客户端足够长时间没有调用它的 bean 实例，容器将在 JVM 内存溢出前把实例清除，持久化这个实例的状态，当它需要的时候再拿回来。容器管理着钝化和活化。

钝化是将 bean 的实例从内存中持久化到永久存储介质中。活化是相反的过程，它在 bean 实例被客户端重新使用的时候激活，容器从持久化存储介质中反序列化到内存中，并激活它。这意味着这个 bean 的属性必须是可被序列化的（它必须是 Java 原始数据类型或者实现了 java.io.Serializable 接口）。stateful bean 的生命周期描述如下：

1. stateful bean 的生命周期在客户端请求一个 bean 的引用（依赖注入或者 JNDI 查找）时开始。容器创建一个新的会话 bean 实例，并存储在内存中。
2. 假如新创建的实例通过标注（@Resource, @EJB, @PersistenceContext 等等）或者部署描述使用了依赖注入，容器注入所有它所需要的资源。
3. 假如实例有一个标注了@PostContruct 的方法，容器将调用这个方法
4. bean 执行被请求的调用并保持在内存中，等待客户端的后续调用
5. 假如客户端依然保持空闲一段时间，容器将调用标注了@PrePassivate 的方法（假如存在），钝化这个 bean 的实例到持久化存储介质中。
6. 假如客户端调用一个已钝化的 bean，容器将活化它到内存中，并调用标注了@PostActivate 的方法（假如存在）
7. 假如客户端在会话超时期间还不调用钝化的 bean，容器将销毁它。
8. 与第 7 步不同，客户端调用了一个带有@Remove 标注的方法，容器将调用带有@PreDestroy 标注的方法（假如存在），然后结束这个 bean 实例的生命周期。



在某些情况下，一个有状态的 bean 包含了一些开放资源，例如 sockets 连接，数据库连接。由于容器不能对每个 bean 都保持这些资源，你将不得不在钝化前后关闭和重新打开。这时候，生命周期的回调方法就可以使用了。

回调方法

每个会话 bean 都由它自己的容器管理的生命周期。容器可以让你在 bean 状态改变时编写自己的业务代码。当状态从一个改变为另一个时，可以被容器拦截，并且容器可以调用被下边列表中标注的方法：

- @PostConstruct 标注一个方法，在一个 bean 实例被容器创建并且完成了依赖注入后立即调用，这个标注经常用来执行各种初始化。
- @PreDestroy 标注一个方法，在一个 bean 被容器销毁前立即调用。被@PreDestroy 标注的方法经常用来释放之前被初始化的资源。在 stateful bean 中，这将在带有@Remove 标注的方法被调用完成后发生。
- @PrePassivate 标注一个方法，在容器钝化一个实例前调用。它经常给 bean 提供时间来准备序列化和不能被序列化的资源（例如：关闭数据库连接，消息 broker，网络 socket 等等）
- @PostActivate 标注一个方法，在容器重新激活一个实例后立即调用。这将给 bean 一个机会，可以重新初始化在钝化时关闭的资源。

一个回调方法必须具有以下签名：

```
void <METHOD>();
```

以下规则适用于一个回调方法：

- 方法必须没有参数，并且必须返回 void
- 方法不能抛出一个经检查异常（a checked exception），但是可以抛出一个运行时异常。抛出一个运行时异常将回滚事务（假如存在事务）
- 方法可以使 public，private，protected 或者默认，但不能是 static 或者 final。
- 一个方法可以打上多种标注。一种类型的标注在一个 bean 中只能出现一次（例如：在同一个会话 bean 中你不能出现两个@PostConstruct 标注）

- 回调方法不能访问 bean 的环境变量。

下边例子中,CacheEJB 使用了@PostConstruct 标注来初始化它的缓存。在创建单例 CacheEJB 后，容器调用 initCache()方法。

```
@Singleton
public class CacheEJB {
  private Map<long  , object> cache = new HashMap<long  , object>();

  @PostConstruct
  private void initCache() {
    // Initializes the cache
  }

  public Object getFromCache(Long id) {
    if (cache.containsKey(id))
      return cache.get(id);
    else
      return null;
  }
}
```

下边代码展示了一个 stateful bean。在创建一个 stateful bean 的实例后，容器向 ds 属性注入一个 datasource 的一个引用。一旦注入完成，容器将调用带有@PostConstruct 标注的方法（init(),创建数据库连接）。假如容器要钝化实例，close()方法将被调用（带有@PrePassivate 标注）。这目的是为了关闭数据库连接，数据库连接在钝化期间是不再需要的。当客户端调用 bean 上的业务方法，容器将活化它并调用 init()方法（@PostActivate 标注）。当客户端调用 checkout()方法（带有@Remove 标注），容器将销毁这个实例，但之前将先再次调用 close()方法（@PreDestroy 标注）

```
@Stateful
public class ShoppingCartEJB {
  @Resource
  private DataSource ds;
  private Connection connection;
  private List<item> cartItems = new ArrayList<item>();

  @PostConstruct
  @PostActivate
  private void init() {
    connection = ds.getConnection();
  }

  @PreDestroy
  @PrePassivate
  private void close() {
```

```
17      connection.close();
18    }
19
20    // ...
21
22    @Remove
23    public void checkout() {
24      cartItems.clear();
25    }
26  }
```

为了提高可读性，我省略了 SQL 异常处理的回调方法。

EJB3.1 拦截器

你可以把容器想象成是一个拦截器链。当你开发一个会话 bean 的时候，你仅仅关注于你的业务逻辑代码。但在这背后，当一个客户端调用了一个 EJB 的方法，容器拦截了这个调用，并提供不许多不同的服务（生命周期管理，事务，安全等等）。以下是一些在客户端与 EJB 之间被调用的拦截器：

- 调用环绕拦截器
- 业务方法拦截器
- 生命周期回调拦截器

调用环绕拦截器

有多种方式来定义一个业务方法的拦截器，最简单的方式是在 bean 中添加一个 @javax.interceptor.AroundInvoke 标注方法（或者<around-invoke>部署描述元素）。如下边例子，CustomerEJB 在 logMethod()方法上标注了@AroundInvoke。logMethod()用来当进入一个方法时记录一条消息日志，当出这个方法时记录另一条日志消息。一旦这个 EJB 被部署了，客户端调用 createCustomer()或者 findCustomerById()方法时将被拦截，而 logMethod()方法将被应用上。需要注意的是，这个拦截器的作用范围仅限于这个 bean。

```
@Stateless
public class CustomerEJB {
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;
  private Logger logger = Logger.getLogger("com.apress.javaee6");

  public void createCustomer(Customer customer) {
    em.persist(customer);
  }

  public Customer findCustomerById(Long id) {
    return em.find(Customer.class, id);
  }

  @AroundInvoke
  private Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
      return ic.proceed();
    } finally {
      logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
  }
}
```

由于带有@AroundInvoke 标注，logMethod()必须遵循以下的签名：

@AroundInvoke
Object <METHOD>(InvocationContext ic) throws Exception;

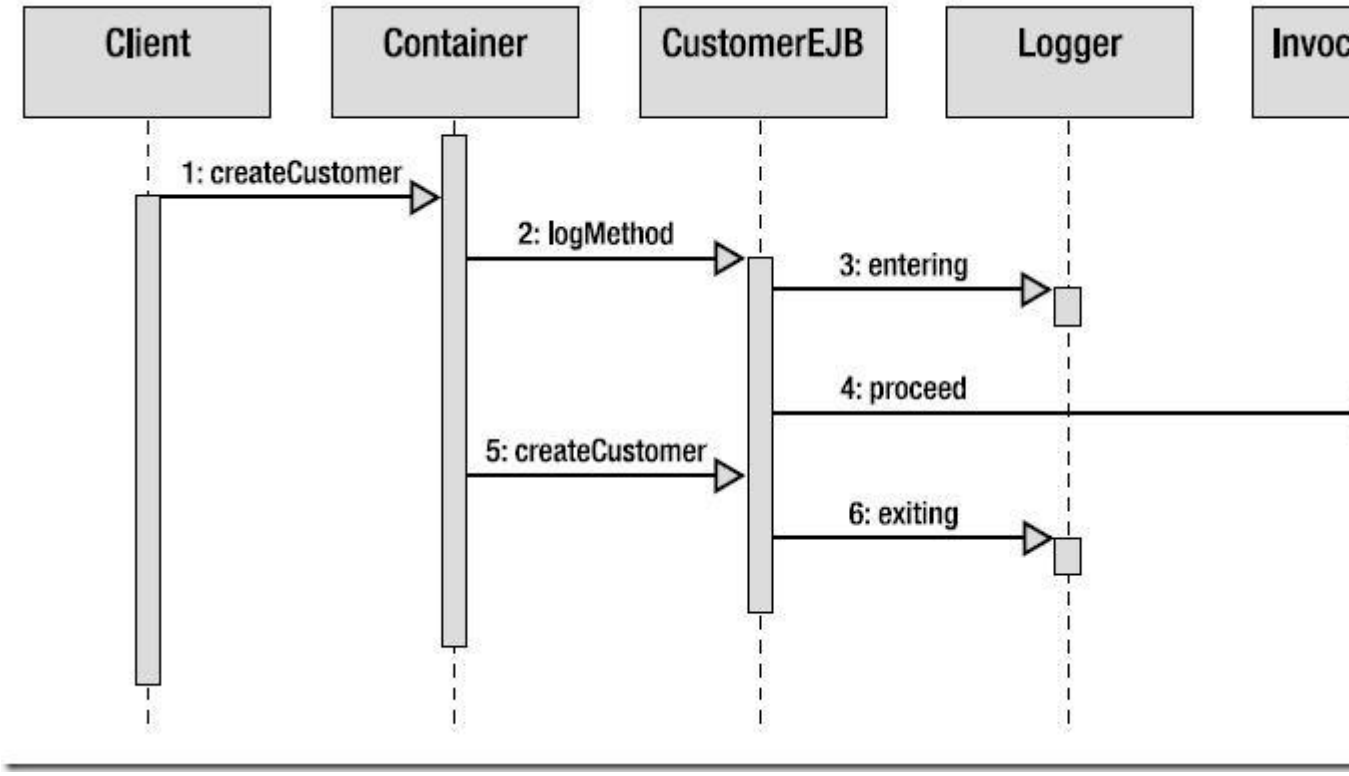
以下规则适用于环绕调用方法：

- 方法可以是 public、private、protected 或者是默认的包级别，但不能是 static 或者 final
- 方法必须有一个 javax.interceptor.InvocationContext 参数并且必须返回 Object，返回值就是被调用的方法的返回值（假如方法返回 void，这将返回 null）
- 方法可以抛出一个经过检查的异常（checked exception）

InvocationContext 对象允许拦截器控制调用链的行为。假如有几个拦截器链接，每个拦截器将接收到同样的 InvocationContext 实例，实例可以添加上下文数据给其他拦截器处理。下边描述了 InvocationContext API:

- getContextData 允许在拦截器之间传递数据，数据使用 Map 保存在 InvocationContext 实例中
- getMethod 返回被调用拦截器的 bean 类的方法
- getParameters 返回被用于调用业务方法的参数
- getTarget 返回拦截器方法隶属的 bean 实例
- getTimer 返回与@Timeout 关联的定时器方法
- Proceed 触发在链中下一个拦截方法。它返回下一个方法调用的结果。假如方法是 void 的，将返回 null
- setParameters 修改用于目标类方法调用的参数值。参数的类型与数量必须与 bean 方法签名相符，否则抛出 IllegalArgumentException 异常

下边解释下 CustomerEJB 代码是如何工作的，先看下边的时序图，看看当客户端调用 createCustomer()时发生了什么。首先，容器拦截了请求，不是直接去调用 createCustomer()，而是首先调用了 logMethod()方法。logMethod()方法使用了 InvocationContext 接口来获得调用的 bean（ic.getTarget()）和调用的方法（ic.getMethod()），然后把它们记录到日志里边（logger.entering()）。然后 proceed()方法被调用。调用 InvocationContext.proceed()是极为重要的，它将告诉容器去处理下一个拦截器或者调用业务逻辑方法。createCustomer()是最后被调用的，一旦它返回了，拦截器完成了它的调用并记录退出信息日志（logger.exiting()）。当客户端调用 findCustomerById()方法时会发生同样的调用顺序。



方法拦截器

上边定义的环绕拦截器只能作用于 CustomerEJB。但是大多数情况下，你希望单独地定义一个类，然后告诉容器要再数个回话 bean 上拦截请求。日志就是一个典型的例子，你希望你的 EJB 上所有的方法都记录进入和离开的信息。

如下例子，LoggingInterceptor 就是一个简单的 POJO，带有一个标注了@AroundInvoke 的方法：

```
public class LoggingInterceptor {
  private Logger logger = Logger.getLogger("com.apress.javaee6");
  @AroundInvoke
  public Object logMethod(InvocationContext ic) throws Exception {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
      return ic.proceed();
    } finally {
      logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
  }
}
```

LoggingInterceptor 现在可以显式地被任何对这个拦截器感兴趣的 EJB 所使用。回话 bean 需要使用 @javax.interceptor.Interceptors 标注来通知容器。下边例子中，这个标注被放在了 createCustomer() 方法上。这意味着任意对这个方法的调用将被容器拦截，然后 LoggingInterceptor 类将被调用（在进出方法时记录日志）

```
@Stateless
public class CustomerEJB {
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;

  @Interceptors(LoggingInterceptor.class)
  public void createCustomer(Customer customer) {
    em.persist(customer);
  }

  public Customer findCustomerById(Long id) {
    return em.find(Customer.class, id);
  }
}
```

上边代码中@Interceptors 仅仅指定给了 createCustomer()方法。这意味着假如客户端调用 findCustomerById()，容器将不进行拦截。假如你希望对这两个方法的调用都进行拦截，你可以在这两个方法上都添加@Interceptors 标注，或者直接在 bean 上添加。

```
@Stateless
@Interceptors(LoggingInterceptor.class)
public class CustomerEJB {

  public void createCustomer(Customer customer) { ... }

  public Customer findCustomerById(Long id) { ... }

}
```

假如你的 bean 有数个方法，但你希望提供一个拦截器给这个 bean 中一个特殊方法以外的所有方法，这时你可以使用 javax.interceptor.ExcludeClassInterceptors 标注来排除。下边代码中的 updateCustomer()将不会被拦截：

```
@Stateless
@Interceptors(LoggingInterceptor.class)
public class CustomerEJB {

  public void createCustomer(Customer customer) { ... }
  public Customer findCustomerById(Long id) { ... }
  public void removeCustomer(Customer customer) { ... }

  @ExcludeClassInterceptors
  public Customer updateCustomer(Customer customer) { ... }
}
```

生命周期拦截器

生命周期拦截器很像上边的方法拦截器。和@AroundInvoke 不同，方法可以被回调标注标志。下边的 ProfileInterceptor 类带了两个方法：logMethod()在构造完成后使用，profile()在销毁前使用

```
public class ProfileInterceptor {

  private Logger logger = Logger.getLogger("com.apress.javaee6");

  @PostConstruct
  public void logMethod(InvocationContext ic) {
    logger.entering(ic.getTarget().toString(), ic.getMethod().getName());
    try {
      return ic.proceed();
    } finally {
      logger.exiting(ic.getTarget().toString(), ic.getMethod().getName());
    }
  }

  @PreDestroy
  public void profile(InvocationContext ic) {
    long initTime = System.currentTimeMillis();
    try {
      return ic.proceed();
    } finally {
      long diffTime = System.currentTimeMillis() - initTime;
      logger.fine(ic.getMethod() + " took " + diffTime + " millis");
    }
  }
}
```

正如你看到上边的代码，生命周期拦截器方法带有一个 InvocationContext 参数，返回 void 而不是 Object（因为生命周期方法返回 void），不能抛出检查的异常（checked exceptions）

会话 bean 使用@Interceptors 标注来使用这个拦截器。下边例子中，CustomerEJB 定义了 ProfileInterceptor.当 EJB 被容器实例化，拦截器的 logMethod()方法将在 init（）被调用前调用。假如客户端调用 createCustomer()或者 findCustomerById()方法，没有拦截器会被调用。但是在 CustomerEJB 被容器销毁前，profile()方法将被调用。

```
@Stateless
@Interceptors(ProfileInterceptor.class)
public class CustomerEJB {
  @PersistenceContext(unitName = "chapter08PU")
  private EntityManager em;

  @PostConstruct
```

```
  public void init() {
    // ...
  }

  public void createCustomer(Customer customer) {
    em.persist(customer);
  }

  public Customer findCustomerById(Long id) {
    return em.find(Customer.class, id);
  }
}
```

生命周期方法和@AroundInvoke 方法可以定义在同一个拦截器中。

链和排除拦截器

事实上，@Interceptors 标注可以附上多个拦截器，它可以用一个带逗号的拦截器列表作为参数。当数个拦截器被定义了，他们被调用的顺序是他们在@Interceptors 标注中被定义的顺序。下边的例子中。@Interceptors 用在了 bean 和方法上。

```
@Stateless
@Interceptors(I1.class, I2.class)
public class CustomerEJB {
  public void createCustomer(Customer customer) { ... }

  @Interceptors(I3.class, I4.class)
  public Customer findCustomerById(Long id) { ... }

  public void removeCustomer(Customer customer) { ... }

  @ExcludeClassInterceptors
  public Customer updateCustomer(Customer customer) { ... }
}
```

当客户端调用了 updateCustomer()方法，没有拦截器被调用（方法被标注了@ExcludeClassInterceptors）。当 createCustomer()方法被调用，拦截器 I1 被调用后接着 I2 被调用。当 findCustomerById()方法被调用，拦截器 I1,I2,I3 和 I4 按顺序被调用。

除了给方法和类指定拦截器，Interceptors1.1 允许你创建一个默认的拦截器，给所有 EJB 的所有方法使用。这样就没有标注，只有一个应用范围。假如你希望在你的应用中提供一个默认的拦截器，你需要在你的部署描述文件中（ejb-jar.xml）定义它。例如你希望定义默认拦截器为 ProfileInterceptor，下边的 XML 你需要加入到你的部署描述里边：

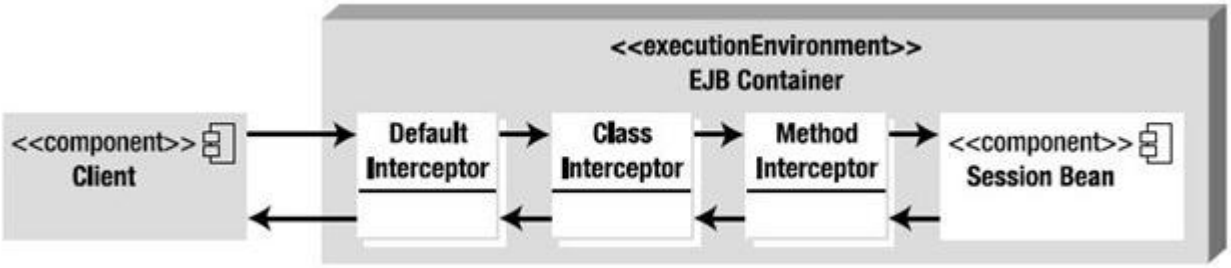
```
<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>*</ejb-name>
    <interceptor-class>
```

```
    com.apress.javaee6.ProfileInterceptor
  </interceptor-class>
 </interceptor-binding>
</assembly-descriptor>
```

正如你看到的，字符×在<ejb-name>中意思是所有的 EJB 将使用在<interceptor-class>中定义的拦截器。假如你销毁带有默认拦截器的 CustomerEJB，ProfileInterceptor 将在任意拦截器前被调用。

假如一个会话 bean 定义了多个类型的拦截器，容器将从最大范围（默认拦截器）到最小范围（方法拦截器）调用。

这个执行顺序如下图：



假如你希望在某个 EJB 上不使用默认拦截器，你可以在类或者方法上使用 @javax.interceptor.ExcludeDefaultInterceptors，如下：

```
@Stateless
@ExcludeDefaultInterceptors
@Interceptors(LoggingInterceptor.class)
public class CustomerEJB {
  public void createCustomer(Customer customer) { ... }
  public Customer findCustomerById(Long id) { ... }
  public void removeCustomer(Customer customer) { ... }

  @ExcludeClassInterceptors
  public Customer updateCustomer(Customer customer) { ... }
}
```

JavaEE6 上下文依赖注入(CDI)介绍（1）

CDI 概述

CDI 提供了如下基本的服务：

- 上下文(Contexts)：使得生命周期和有状态组件之间有良好的定义并且生命周期上下文可扩展
- 依赖注入(Dependency injection)：可以通过类型安全的方式向应用中注入组件，并且可以在部署的时候选择接口的哪个实现来注入

另外，CDI 提供了如下的服务：

- 整合了表达式语言(EL),允许直接在 JSF 或者 JSP 页面直接使用任何组件
- 装饰注入的组件
- 使用类型安全的拦截器绑定来关联组件和拦截器
- 事件通知模型
- 在 Java Servlet 中定义的除了 3 个标准的范围（request, session 和 application）以外的 web 会话范围
- 完整的服务提供接口(SPI)，允许很好的整合第三方框架

CDI 最主要的作用是松耦合，如下：

- 通过良好定义的类型和限定符来解耦服务端和客户端，这样使得服务器端实现是可变的
- 解耦正交关注的 JavaEE 拦截器
- 通过如下解耦协作组件的生命周期
  - 组件上下文化，使用自动生命周期管理
  - 允许有状态组件像服务样交互，纯粹由消息传递

随着松耦合，CDI 提供了如下强类型：

- 消除使用基于字符串名称的相关查找，这样编译器可以检测输入错误
- 允许使用声明的 Java 注解来指定一切，在很大程度下消除了 xml 部署描述的需要，使得易于提供简略代码和在开发的时候了解依赖结构的工具

关于 Bean

CDI 重新定义了再其他 Java 技术中得 bean 概念，例如 JavaBeans 和 EJB 技术。在 CDI 中，bean 是一个定义了应用状态或者逻辑的上下文化的对象。假如一个 JavaEE 组件实例的生命周期由容器基于 CDI 中定义的生命周期上下文模型管理的，这个组件就是一个 bean。

更具体的，一个 bean 有以下属性：

- 一组非空的 bean 类型
- 一组非空的限定符
- 一个范围
- 一个 bean 的 EL 名称
- 一组拦截器绑定
- 一个 bean 的实现

一个 bean 类型定义了 bean 的客户端可见的类型。几乎任何 java 类型可以是一个 bean 的 bean 类型

- 一个 bean 类型可能是一个接口、一个具体的类或者一个抽象类，并且可能申明为 final 或者有 final 的方法
- 一个 bean 类型可能是一个带有类型参数和类型变量的
- 一个 bean 类型可能是一个数组类型。只有当元素的类型是相同时，两个数组类型才被认为是相同的。
- 一个 bean 类型可能是一个原始基本数据类型。原始基本数据类型被认为与他们相对应的再 java.lang 中得包装类是一样的
- 一个 bean 类型可能是一个原生态类型

关于受管 bean(Managed Beans)

一个受管 bean 由一个 java 类实现，这个 java 类被称为它的 bean 类。假如一个顶级类被其他 JavaEE 技术（如：JavaServer Faces 技术）定义为一个受管 bean，或者它符合下边所有条件，那它就成了一个受管 bean。

- 它不是一个非静态内部类
- 它是一个具体的类或者被标注上了@Decorator
- 它没有 EJB 组件定义的标注或者在 ejb-jar.xml 中被定义为 EJB bean
- 它带有一个合适的构造器，就是以下情况之以：
  - 这个类带有无参构造函数
  - 这个类定义了一个带有@Inject 标注的构造函数

不需要特别的声明（例如一个标注）来定义一个受管 bean。

bean 作为可注入对象

注入的概念有时候已经成为了 java 技术的一部分。自从 JavaEE5，标注已经使得它可以向容器管理的对象中注入资源和一些其他对象。CDI 使得它可以注入更多类型的对象，可以注入到不是容器管理的对象中。

以下这些对象可以被注入：

- 几乎任意的 java 类
- 会话 bean
- JavaEE 资源：数据源，JMS 主题、队列、连接工厂等等
- 持久化上下文（Persistence contexts）(JPA EntityManager 对象)
- 生产者字段（Producer fields）
- 生产者方法（producer methods）返回的对象
- Web service 的引用
- 远程企业 bean 引用

例如：假设你创建了一个简单的 Java 类，并且带有一个返回 string 的方法：

```
1 package greetings;
2 public class Greeting {
3   public String greet(String name) {
4     return "Hello, " + name + ".";
5   }
6 }
```

这个类成了一个 bean，你可以把它注入到另一个类中。这种形式，这个 bean 并没有暴露给 EL。