 作为轻量级的容器，Spring常常被认为是EJB的替代品。我们也相信，对于很多（不一定是绝大多数）应用和用例，相对于通过EJB容器来实现相同的功能而言， Sping作为容器，加上它在事务，ORM和JDBC存取这些领域中丰富的功能支持， Spring的确是更好的选择。 不过，需要特别注意的是，使用了Spring并不是说我们就不能用EJB了， 实际上，Spring大大简化了从中访问和实现EJB组件或只实现（EJB组件）其功能的复杂性。 另外，如果通过Spring来访问EJB组件服务，以后就可以在本地EJB组件，远程EJB组件， 或者是POJO（简单Java对象）这些变体之间透明地切换服务的实现，而不需要修改 客户端的代码。 本章，我们来看看Spring是如何帮助我们访问和实现EJB组件的。尤其是在访问 无状态Session Bean（SLSBs）的时候，Spring特别有用，现在我们就由此开始讨论。 1.访问EJB 1.1. 概念要调用本地或远程无状态Session Bean上的方法，通常客户端的代码必须 进行JNDI查找，得到（本地或远程的）EJB Home对象，然后调用该对象的"create" 方法，才能得到实际的（本地或远程的）EJB对象。前后调用了不止一个EJB组件 上的方法。 为了避免重复的底层调用，很多EJB应用使用了服务定位器（Service Locator） 和业务委托（Bussiness Delegate）模式，这样要比在客户端代码中到处进行JNDI 查找更好些，不过它们的常见的实现都有明显的缺陷。例如： 通常，若是依赖于服务定位器或业务代理单件来使用EJB，则很难对代码进 行测试

。 在仅使用了服务定位器模式而不使用业务委托模式的情况下，应用程序 代码仍然需要调用EJB Home组件的create方法，还是要处理由此引入的异常。 导致代码仍然保留了与EJB API的耦合性以及EJB编程模型的复杂性。 实现业务委托模式通常会导致大量的冗余代码，因为我们不得不编写 很多方法，而它们所做的仅仅是调用EJB组件的同名方法。 Spring采用的方法是允许创建并使用代理对象，一般是在Spring的ApplicationContext或BeanFactory里面进行配置，这样就和业务代理类似，只需要 少量的代码。我们不再需要另外编写额外的服务定位器或JNDI查找的代码，或者是手写 的业务委托对象里面冗余的方法，除非它们可以带来实质性的好处。 1.2. 访问本地的无状态Session Bean（SLSB） 假设有一个web控制器需要使用本地EJB组件。我们遵循前人的实践经验， 于是使用了EJB的业务方法接口（Business Methods Interface）模式，这样， 这个EJB组件的本地接口就扩展了非EJB特定的业务方法接口。让我们假定这个 业务方法接口叫MyComponent. public interface MyComponent {…… }（使用业务方法接口模式的一个主要原因就是为了保证本地接口和bean的实现类 之间方法签名的同步是自动的。另外一个原因是它使得稍后我们改用基于POJO（简单Java对象） 的服务实现更加容易，只要这样的改变是有利的。当然，我们也需要实现 本地Home接口，并提供一个Bean实现类，使其实现接口SessionBean和业务方法接口 MyComponent.现在为了把我们Web层的控制器和EJB的实现链接起来，我们唯一要写 的Java代码就是在控制器上公布一个形参为MyComponent的setter方法。这样就可以把这个引用保存在控制器的一个实例变量中。 private

MyComponent myComponent. public void setMyComponent(MyComponent myComponent) { this.myComponent = myComponent. } 然后我们可以在控制器的任意业务方法里面使用这个实例变量。假设我们现在从Spring的ApplicationContext或BeanFactory获得该控制器对象，我们就可以在 同一个上下文中配置一个LocalStatelessSessionProxyFactoryBean 的实例，它将作为EJB组件的代理对象。这个代理对象的配置和控制器的属性myComponent的设置是使用一个配置项完成的，如下所示：class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean"&gt. myComponent com.mycom.MyComponent 这些看似简单的代码背后隐藏了很多复杂的处理，比如默默工作的Spring AOP框架，我们甚至不必知道这些概念，一样可以享用它的结果。Bean myComponent 的定义中创建了一个该EJB组件的代理对象，它实现了业务方法接口。这个EJB组件的 本地Home对象在启动的时候就被放到了缓存中，所以只需要执行一次JNDI查找即可。 每当EJB组件被调用的时候，这个代理对象就调用本地EJB组件的create方法，并调用该EJB组件的相应的业务方法。 在Bean myController的定义中，控制器类的属性 myController的值被设置为上面代理对象。这样的EJB组件访问方式大大简化了应用程序代码：Web层（或其他EJB客户端） 的代码不再依赖于EJB组件的使用。如果我们想把这个EJB的引用替换为一个POJO， 或者是模拟用的对象或其他测试组件，我们只需要简单地修改Bean myComponent 的定义中仅仅一行Java代码，此外，我们也不再需要在应用程序中编写任何JNDI查找 或其它EJB相关的代

码。 评测和实际应用中的经验表明，这种方式的性能负荷极小，（尽管其中 使用了反射方式以调用目标EJB组件的方法），通常的使用中我们几乎觉察不出。请记住 我们并不想频繁地调用EJB组件的底层方法，虽然如此，有些性能代价是与应用服务器 中EJB的基础框架相关的。 关于JNDI查找有一点需要注意。在Bean容器中，这个类通常最好用作单件 （没理由使之成为原型）。不过，如果这个Bean容器会预先实例化单件（类似XML ApplicationContext的变体的行为），如果在EJB容器载入目标EJB前载入bean容器， 我们就可能会遇到问题。因为JNDI查找会在该类的init方法中被执行并且缓存结果， 这样就导致该EJB不能被绑定到目标位置。解决方案就是不要预先实例化这个工厂对象， 而允许它在第一次用到的时候再创建，在XML容器中，这是通过属性 lazy-init来控制的。 尽管大部分Spring的用户不会对这些感兴趣，但那些对EJB进行AOP的具体应用 的用户则会想看看LocalSlsbInvokerInterceptor. 1.3. 访问远程的无状态Session Bean（SLSB） 基本上访问远程EJB与访问本地EJB差别不大，除了前者使用的是SimpleRemoteStatelessSessionProxyFactoryBean.当然， 无论是否使用Spring，远程调用的语义都相同，不过，对于使用的场景和错误处理 来说，调用另外一台计算机上不同虚拟机中的对象的方法其处理有所不同。 与不使用Spring方式的EJB客户端相比，Spring的EJB客户端有一个额外的 好处。通常如果客户端代码随意在本地EJB和远程EJB的调用之间来回切换，就有一个问题。这是因为远程接口的方法需要声明其会抛出RemoteException ，然后客户端代码必须处理这种异常，但

是本地接口的方法却不需要这样。 如果要把针对本地EJB的代码改为访问远程EJB，就需要修改客户端代码，增加对RemoteException的处理，反之就需要去掉这样的 异常处理。使用Spring 的远程EJB代理，我们就不再需要在业务方法接口和EJB的 代码实现中声明会抛出RemoteException，而是定义一个 相似的远程接口，唯一不同就是它抛出的是RemoteAccessException， 然后交给代理对象去动态的协调这两个接口。也就是说，客户端代码不再需要与RemoteException这个显式（checked）异常打交道，实际运行中 所有抛出的异常RemoteException都会被捕获并转换成一个隐式（non-checked）的RemoteAccessException，它是RuntimeException的一个子类。这样目标服务端就可以 在本地EJB或远程EJB（甚至POJO）之间随意地切换，客户端不再需要关心甚至 根本不会觉察到这种切换。当然，这些都是可选的，我们并不阻止在业务接口中声明 异常RemoteExceptions. 2. 使用Spring提供的辅助类实现EJB组件Spring也提供了一些辅助类来为EJB组件的实现提供便利。它们是为了倡导一些 好的实践经验，比如把业务逻辑放在在EJB层之后的POJO中实现，只把事务隔离和 远程调用这些职责留给EJB. 要实现一个无状态或有状态的Session Bean，或消息驱动Bean，我们的实现 可以继承分别继承AbstractStatelessSessionBean， AbstractStatefulSessionBean，和AbstractMessageDrivenBean/AbstractJmsMessageDrivenBean.