[**Spring 中JCA CCI分析--转载**](https://www.cnblogs.com/davidwang456/p/4482922.html)

转载地址：http://blog.csdn.net/a154832918/article/details/6790612

J2EE提供JCA（Java Connector Architecture）规范来标准化对EIS（Enterprise Information System）的访问。这个规范被分为几个不同的部分：   
  
SPI（Service provider interfaces）是连接器提供者（connector provider）必须实现的接口。 这些接口组成了一个能被部署在J2EE应用服务器上的资源适配器（resource adapter）。 在这种情况下，由服务器来管理连接池（connection pooling）、事务和安全（托管模式（managed mode））。 应用服务器还负责管理客户端应用程序之外所拥有的配置。连接器（connector）同样能在脱离应用服务器的情况下使用；在这种情况下，应用程序必须直接对它进行配置（非托管模式（non-managed mode））。   
  
CCI （Common Client Interface）是应用程序用来与连接器交互并与EIS通信的接口。同样还为本地事务划界提供了API。   
  
Spring对CCI的支持，目的是为了提供以典型的Spring方式来访问CCI连接器的类，并有效地使用Spring的通用资源和事务管理工具。   
  
Note   
连接器的客户端不必总是使用CCI。 某些连接器暴露它们自己的API，只提供JCA资源适配器（resource adapter） 以使用J2EE容器的某些系统契约（system contracts）（连接池（connection pooling），全局事务（global transactions），安全（security））。 Spring并没有为这类连接器特有（connector-specific）的API提供特殊的支持。   
  
  
21.2. 配置CCI   
21.2.1. 连接器配置   
使用JCA CCI的基础资源是 ConnectionFactory 接口。 所用的连接器必须提供这个接口的一个实现。   
  
为了使用连接器，可以把它部署到你的应用服务器上，并从服务器的JNDI环境（托管模式）取回 ConnectionFactory。 连接器必须打包为一个RAR文件（resource adapte archive）并包含一个部署描述符文件 ra.xml 。 部署时需要指定资源的实际名字。如果想通过Spring访问它，只要简单地使用Spring的 JndiObjectFactoryBean 来通过JNDI名字获取工厂。   
  
使用连接器的另一种方法是把它嵌入到应用程序中（非托管模式（non-managed mode）），而不用在应用服务器中部署并配置它。 Spring提供了通过已提供的 FactoryBean（LocalConnectionFactoryBean)）来将连接器配置成一个bean的可能。 在这种方式中，只需要把连接器类库放到classpath目录下（不需要RAR文件和 ra.xml 描述符）。 如果有必要的话，必须从连接器的RAR文件解压得到那个类库。   
  
一旦访问 ConnectionFactory 实例， 就可以将它注入到组件。这些组件既可以用简单的CCI API编码，也可以利用Spring支持CCI访问的类（比如，CciTemplate）。   
  
Note   
当在非托管模式（non-managed mode）下使用连接器时，将无法使用全局事务， 因为该资源从不会被加入或删除到当前线程的当前全局事务中。该资源根本不知道任何可能正在运行的全局的J2EE事务。   
  
  
21.2.2. 在Spring中配置ConnectionFactory   
为了创建到 EIS 的连接，如果处于托管模式（managed mode），你需要从应用服务器获取一个 ConnectionFactory ， 或者当你在非托管模式（non-managed mode）时直接从Spring去获取。   
  
在托管模式（managed mode）下，你可以从JNDI访问 ConnectionFactory， 它的属性将被配置在应用服务器中。   
  
<bean id="eciConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">   
<property name="jndiName" value="eis/cicseci"/>   
</bean>   
在非托管模式（non-managed mode）下，你必须在Spring配置中将你要用的 ConnectionFactory 配置为一个JavaBean。 LocalConnectionFactoryBean 类提供这种配置风格，把 ManagedConnectionFactory 传入到你的连接器的实现中， 暴露应用级的CCI ConnectionFactory。   
  
<bean id="eciManagedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">   
<property name="serverName" value="TXSERIES"/>   
<property name="connectionURL" value="tcp://localhost/"/>   
<property name="portNumber" value="2006"/>   
</bean>   
  
<bean id="eciConnectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">   
<property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>   
</bean>   
Note   
你不能直接实例化一个指定的 ConnectionFactory 接口。 你需要为连接器实现相应的 ManagedConnectionFactory 接口，这个接口是JCA SPI规范的一部分。   
  
  
21.2.3. 配置CCI连接   
JCA CCI允许开发者使用自己的连接器的 ConnectionSpec 接口实现来配置到 EIS 的连接。 为了配置该连接的属性，需要用一个指定的 ConnectionSpecConnectionFactoryAdapter 适配器来封装目标连接工厂。 因此，特定的 ConnectionSpec 接口可以用 connectionSpec 属性来配置（作为一个内部bean）。   
  
这个属性不是必需的，因为CCI ConnectionFactory 接口定义了两个不同的方法来获取 CCI 连接。 ConnectionSpec 的一些属性常常可以被配置在应用服务器中（托管模式（managed mode）） 或相关的本地 ManagedConnectionFactory 实现。   
  
public interface ConnectionFactory implements Serializable, Referenceable {   
...   
Connection getConnection() throws ResourceException;   
Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException;   
...   
}   
Spring提供了 ConnectionSpecConnectionFactoryAdapter 适配器， 允许你指定一个 ConnectionSpec 接口的实例, 供给定工厂的所有操作使用。 如果指定了适配器的 connectionSpec 属性，适配器就使用没有参数的 getConnection 变量， 而不是有 ConnectionSpec 参数的变量。   
  
<bean id="managedConnectionFactory"   
  class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">   
<property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>   
<property name="driverName" value="org.hsqldb.jdbcDriver"/>   
</bean>   
  
<bean id="targetConnectionFactory"   
  class="org.springframework.jca.support.LocalConnectionFactoryBean">   
<property name="managedConnectionFactory" ref="managedConnectionFactory"/>   
</bean>   
  
<bean id="connectionFactory"   
  class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">  
<property name="targetConnectionFactory" ref="targetConnectionFactory"/>   
<property name="connectionSpec">   
  <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">   
   <property name="user" value="sa"/>   
   <property name="password" value=""/>   
  </bean>   
</property>   
</bean>   
21.2.4. 使用一个 CCI 单连接   
如果想使用一个 CCI 单连接，Spring提供一个额外的 ConnectionFactory 适配器来管理它。 SingleConnectionFactory 适配器类将延迟打开一个单独的连接并在应用程序销毁这个bean的时候关闭它。 这个类将暴露出特殊 Connection 的相应代理，并共享同一个底层的物理连接。   
  
<bean id="eciManagedConnectionFactory"   
  class="com.ibm.connector2.cics.ECIManagedConnectionFactory">   
<property name="serverName" value="TEST"/>   
<property name="connectionURL" value="tcp://localhost/"/>   
<property name="portNumber" value="2006"/>   
</bean>   
  
<bean id="targetEciConnectionFactory"   
  class="org.springframework.jca.support.LocalConnectionFactoryBean">   
<property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>   
</bean>   
  
<bean id="eciConnectionFactory"   
  class="org.springframework.jca.cci.connection.SingleConnectionFactory">   
<property name="targetConnectionFactory" ref="targetEciConnectionFactory"/>   
</bean>   
Note   
这个 ConnectionFactory 适配器不能直接用 ConnectionSpec 配置。 如果需要特定 ConnectionSpec 的单一连接，那么可以用 SingleConnectionFactory 与之通信的中间 ConnectionSpecConnectionFactoryAdapter 适配器。   
  
  
21.3. 使用Spring的 CCI访问支持   
21.3.1. 记录转换   
对JCA CCI支持的一个目标是提供方便的机制来操作CCI记录。 开发人员可以通过使用Spring的 CciTemplate 来指定创建记录并从记录中提取数据的策略。 如果你不想在应用程序中直接操作记录，可以使用下面的接口来配置用于输入输出记录的策略。   
  
为了创建一个输入 Record，开发人员可以使用 RecordCreator 接口的一个特定实现。   
  
public interface RecordCreator {   
  
Record createRecord(RecordFactory recordFactory) throws ResourceException, DataAccessException;   
}   
正如你所看到的一样， createRecord(..) 方法接收一个 RecordFactory 实例作为参数， 该参数对应于所使用的 ConnectionFactory 的 RecordFactory 接口。 它能被用于创建 IndexedRecord 或者 MappedRecord 的实例。 下面的例子展示了如何使用 RecordFactory 接口和索引（indexed）/映射（mapped）记录。   
  
public class MyRecordCreator implements RecordCreator {   
  
public Record createRecord(RecordFactory recordFactory) throws ResourceException {   
  IndexedRecord input = recordFactory.createIndexedRecord("input");   
  input.add(new Integer(id));   
  return input;   
}   
}   
一个输出 Record 接口能被用于从EIS收回数据。 因此，一个 RecordExtractor 接口的特定实现可以被传给Spring的 CciTemplate ， 用来从输出 Record 接口中提取数据。   
  
public interface RecordExtractor {   
  
Object extractData(Record record) throws ResourceException, SQLException, DataAccessException;   
}   
下面的例子展示了如何使用 RecordExtractor 接口。   
  
public class MyRecordExtractor implements RecordExtractor {   
  
public Object extractData(Record record) throws ResourceException {   
  CommAreaRecord commAreaRecord = (CommAreaRecord) record;   
  String str = new String(commAreaRecord.toByteArray());   
  String field1 = string.substring(0,6);   
  String field2 = string.substring(6,1);   
  return new OutputObject(Long.parseLong(field1), field2);   
}   
}   
21.3.2. CciTemplate类   
CciTemplate 类是 CCI 核心支持包（org.springframework.jca.cci.core）中主要的类。 它简化了CCI的使用，因为它会处理资源的创建和释放。这有助于避免常见的错误，比如总是忘记关闭连接。 它关注连接和交互对象的生命周期，从而使应用程序的代码可以专注于处理从应用程序数据中生成输入记录和从输出记录中提取应用程序数据。   
  
JCA CCI规范定义了两个不同的方法来在EIS上调用操作。CCI Interaction 接口提供两个 execute 方法的签名：   
  
public interface javax.resource.cci.Interaction {   
...   
boolean execute(InteractionSpec spec, Record input, Record output) throws ResourceException;   
  
Record execute(InteractionSpec spec, Record input) throws ResourceException;   
...   
}   
依赖于模板方法的调用，CciTemplate 类可以知道 interaction上的哪个 execute 方法被调用。 在任何情况下，都必须有一个正确初始化过的 InteractionSpec 实例。   
  
CciTemplate.execute(..)可以以下列两种方式使用：   
  
在提供直接的 Record 参数的情况下，只需要传递输入记录给 CCI ， 而返回的对象就是对应的 CCI 输出记录。   
  
在提供使用记录映射的应用对象的情况下，你需要提供相应的 RecordCreator 和 RecordExtractor 实例。   
  
第一种方法将使用下面的模板方法。这些模板方法将直接对应到 Interaction 接口。   
  
public class CciTemplate implements CciOperations {   
  
public Record execute(InteractionSpec spec, Record inputRecord)   
   throws DataAccessException { ... }   
  
public void execute(InteractionSpec spec, Record inputRecord, Record outputRecord)   
   throws DataAccessException { ... }   
  
}   
第二种方法需要我们以参数的方式指定创建记录和记录提取的策略。 使用前面记录转化一节中描述的接口。对应的 CciTemplate 方法如下：   
  
public class CciTemplate implements CciOperations {   
  
public Record execute(InteractionSpec spec, RecordCreator inputCreator)   
   throws DataAccessException { ... }   
  
public Object execute(InteractionSpec spec, Record inputRecord, RecordExtractor outputExtractor)   
   throws DataAccessException { ... }   
  
public Object execute(InteractionSpec spec, RecordCreator creator, RecordExtractor extractor)   
   throws DataAccessException { ... }   
  
}   
除非在模板上设置 outputRecordCreator 属性（参见下一部分）， 不然每个方法都将调用CCI Interaction 中相应的含有两个参数： InteractionSpec 和输入 Record的 execute 方法， 并接收一个输出 Record 作为返回值。   
  
通过 createIndexRecord(..) 和 createMappedRecord(..) 方法， CciTemplate 在 RecordCreator 实现类外部也提供了创建 IndexRecord 和 MappedRecord。 还可以用来在DAO实现内创建记录实例并传入到相应的 CciTemplate.execute(..) 方法。   
  
public class CciTemplate implements CciOperations {   
  
public IndexedRecord createIndexedRecord(String name) throws DataAccessException { ... }   
  
public MappedRecord createMappedRecord(String name) throws DataAccessException { ... }   
  
}   
21.3.3. DAO支持   
Spring的 CCI 支持为 DAO 提供了一个抽象类，支持 ConnectionFactory 或 CciTemplate 实例的注入。这个类的名字是 CciDaoSupport： 它提供了简单的 setConnectionFactory 和 setCciTemplate 方法。 在内部，该类将为传入的 ConnectionFactory 创建一个 CciTemplate 实例， 并把它暴露给子类中具体的数据访问实现使用。   
  
public abstract class CciDaoSupport {   
  
public void setConnectionFactory(ConnectionFactory connectionFactory) { ... }   
public ConnectionFactory getConnectionFactory() { ... }   
  
public void setCciTemplate(CciTemplate cciTemplate) { ... }   
public CciTemplate getCciTemplate() { ... }   
  
}   
21.3.4. 自动输出记录生成   
如果所用的连接器只支持以输入输出记录作为参数的 Interaction.execute(..) 方法 （就是说，它要求传入期望的输出记录而不是返回适当的输出记录）， 你可以设定 CciTemplate 类的 outputRecordCreator 属性来自动生成一个输出记录， 当接收到响应时JCA连接器（JCA connector）将填充该记录并返回给模板的调用者。   
  
因为这个目的，这个属性只持有 RecordCreator 接口的一个实现。 RecordCreator 接口已经在 Section 21.3.1, “记录转换” 进行了讨论。 outputRecordCreator 属性必须直接在 CciTemplate 中指定，可以在应用代码中做到这一点。   
  
cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());   
或者如果 CciTemplate 被配置为一个专门的bean实例，那么outputRecordCreator还可以在Spring文件中配置（推荐的做法）：   
  
<bean id="eciOutputRecordCreator" class="eci.EciOutputRecordCreator"/>   
  
<bean id="cciTemplate" class="org.springframework.jca.cci.core.CciTemplate">   
<property name="connectionFactory" ref="eciConnectionFactory"/>   
<property name="outputRecordCreator" ref="eciOutputRecordCreator"/>   
</bean>   
Note   
因为 CciTemplate 类是线程安全的，所以它通常被配置为一个共享实例。   
  
  
21.3.5. 总结   
下表总结了 CciTemplate 类和在 CCI Interaction 接口上调用相应方法的机制：   
  
Table 21.1. Usage of Interaction execute methods   
  
CciTemplate method signature CciTemplate outputRecordCreator property execute method called on the CCI Interaction   
Record execute(InteractionSpec, Record) not set Record execute(InteractionSpec, Record)   
Record execute(InteractionSpec, Record) set boolean execute(InteractionSpec, Record, Record)   
void execute(InteractionSpec, Record, Record) not set void execute(InteractionSpec, Record, Record)   
void execute(InteractionSpec, Record, Record) set void execute(InteractionSpec, Record, Record)   
Record execute(InteractionSpec, RecordCreator) not set Record execute(InteractionSpec, Record)   
Record execute(InteractionSpec, RecordCreator) set void execute(InteractionSpec, Record, Record)   
Record execute(InteractionSpec, Record, RecordExtractor) not set Record execute(InteractionSpec, Record)   
Record execute(InteractionSpec, Record, RecordExtractor) set void execute(InteractionSpec, Record, Record)   
Record execute(InteractionSpec, RecordCreator, RecordExtractor) not set Record execute(InteractionSpec, Record)   
Record execute(InteractionSpec, RecordCreator, RecordExtractor) set void execute(InteractionSpec, Record, Record)   
  
21.3.6. 直接使用一个CCI Connection接口和Interaction接口   
类似于 JdbcTemplate 类和 JmsTemplate 类的操作方式， CciTemplate 类同样提供直接操作CCI 连接和交互的可能性。 比如说如果你想对一个CCI连接执行多种操作，这就会很有用。   
  
ConnectionCallback 接口提供以CCI Connection 作为参数， 为了在它上面执行自定义动作，添加了创建 Connection 的CCI ConnectionFactory。 后者在获取相关 RecordFactory 实例和创建indexed/mapped records时很有用。例如：   
  
public interface ConnectionCallback {   
  
Object doInConnection(Connection connection, ConnectionFactory connectionFactory)   
   throws ResourceException, SQLException, DataAccessException;   
}   
InteractionCallback 接口提供 CCI Interaction 接口， 为了在它上面执行自定义动作，请添加相应的CCI ConnectionFactory。   
  
public interface InteractionCallback {   
  
Object doInInteraction(Interaction interaction, ConnectionFactory connectionFactory)   
   throws ResourceException, SQLException, DataAccessException;   
}   
Note   
InteractionSpec 对象还可以在多个template调用之间被共享或者在每个回调方法内重新创建， 这完全取决于 DAO 的实现。   
  
  
21.3.7. CciTemplate 使用示例   
在本章节中，我们将展示如何使用 CciTemplate 和IBM CICS ECI连接器在ECI模式下访问一个CICS.   
  
首先，必须在CCI InteractionSpec 进行一些初始化以指定访问哪个CICS程序以及如何进行交互。   
  
ECIInteractionSpec interactionSpec = new ECIInteractionSpec();   
interactionSpec.setFunctionName("MYPROG");   
interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC\_SEND\_RECEIVE);   
然后，程序通过Spring的模板使用 CCI 并在自定义对象和 CCI Records 之间指定映射。   
  
public class MyDaoImpl extends CciDaoSupport implements MyDao {   
  
public OutputObject getData(InputObject input) {   
  ECIInteractionSpec interactionSpec = ...;   
  
  OutputObject output = (ObjectOutput) getCciTemplate().execute(interactionSpec,   
    new RecordCreator() {   
     public Record createRecord(RecordFactory recordFactory) throws ResourceException {   
      return new CommAreaRecord(input.toString().getBytes());   
     }   
    },   
    new RecordExtractor() {   
     public Object extractData(Record record) throws ResourceException {   
      CommAreaRecord commAreaRecord = (CommAreaRecord)record;   
      String str = new String(commAreaRecord.toByteArray());   
      String field1 = string.substring(0,6);   
      String field2 = string.substring(6,1);   
      return new OutputObject(Long.parseLong(field1), field2);   
     }   
    });   
  
  return output;   
}   
}   
正如之前讨论的那样，callbacks 可以被用来直接在 CCI 连接或交互上操作。   
  
public class MyDaoImpl extends CciDaoSupport implements MyDao {   
  
public OutputObject getData(InputObject input) {   
  ObjectOutput output = (ObjectOutput) getCciTemplate().execute(   
    new ConnectionCallback() {   
     public Object doInConnection(Connection connection, ConnectionFactory factory)   
       throws ResourceException {   
  
      // do something...   
     }   
    });   
  }   
  return output;   
}   
}   
Note   
当getCciTemplate().execute参数是 ConnectionCallback 时， 所用的 Connection 将被 CciTemplate 管理和关闭， 但是任何在连接上建立的交互都必须被callback实现类所管理。   
  
  
对于一个更特殊的callback，你可以实现一个 InteractionCallback 。 这样传入的 Interaction 将会被 CciTemplate 管理和关闭。   
  
public class MyDaoImpl extends CciDaoSupport implements MyDao {   
  
public String getData(String input) {   
  ECIInteractionSpec interactionSpec = ...;   
  
  String output = (String) getCciTemplate().execute(interactionSpec,   
    new InteractionCallback() {   
     public Object doInInteraction(Interaction interaction, ConnectionFactory factory)   
       throws ResourceException {   
      Record input = new CommAreaRecord(inputString.getBytes());   
      Record output = new CommAreaRecord();   
      interaction.execute(holder.getInteractionSpec(), input, output);   
      return new String(output.toByteArray());   
     }   
    });   
  
  return output;   
}   
}   
在上面的例子中，非托管模式（non-managed）下对应的spring beans的配置会像下面这样：   
  
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">   
<property name="serverName" value="TXSERIES"/>   
<property name="connectionURL" value="local:"/>   
<property name="userName" value="CICSUSER"/>   
<property name="password" value="CICS"/>   
</bean>   
  
<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">   
<property name="managedConnectionFactory" ref="managedConnectionFactory"/>   
</bean>   
  
<bean id="component" class="mypackage.MyDaoImpl">   
<property name="connectionFactory" ref="connectionFactory"/>   
</bean>   
在托管模式（managed mode）（也就是说，在一个J2EE环境）下，配置可能如下所示：   
  
<bean id="connectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">   
<property name="jndiName" value="eis/cicseci"/>   
</bean>   
  
<bean id="component" class="MyDaoImpl">   
<property name="connectionFactory" ref="connectionFactory"/>   
</bean>   
21.4. 建模CCI访问为操作对象   
org.springframework.jca.cci.object 包中包含的支持类允许你以另一种风格访问EIS： 通过可重用的操作对象，类似于Spring的JDBC操作对象（参见JDBC一章）。 它通常都封装了 CCI 的API：将应用级的输入对象传入到操作对象， 从而它能创建输入record然后转换接收到的record数据到一个应用级输出对象并返回它。   
  
注意: 这种方法内在地基于 CciTemplate 类和 RecordCreator / RecordExtractor 接口，重用了Spring核心CCI支持的机制。   
  
21.4.1. MappingRecordOperation   
MappingRecordOperation 本质上与 CciTemplate 做的事情是一样的， 但是它表达了一个明确的、预配置（pre-configured）的操作作为对象。它提供了两个模板方法来指明如何将一个输入对象转换为输入记录， 以及如何将一个输出记录转换为输出对象（记录映射）：   
  
createInputRecord(..) 指定了如何将一个输入对象转换为输入Record   
  
extractOutputData(..) 指定了如何从输出 Record 中提取输出对象   
  
下面是这些方法的签名：   
  
public abstract class MappingRecordOperation extends EisOperation {   
...   
protected abstract Record createInputRecord(RecordFactory recordFactory, Object inputObject)   
   throws ResourceException, DataAccessException { ... }   
  
protected abstract Object extractOutputData(Record outputRecord)   
   throws ResourceException, SQLException, DataAccessException { ... }   
...   
}   
此后，为了执行一个 EIS 操作，你需要使用一个单独的execute方法，传递一个应用级（application-level） 输入对象，并接收一个应用级输出对象作为结果：   
  
public abstract class MappingRecordOperation extends EisOperation {   
...   
public Object execute(Object inputObject) throws DataAccessException {   
...   
}   
正如你所看到的，与 CciTemplate 类相反， 这个execute方法并没有 InteractionSpec 参数， 然而，InteractionSpec 对操作是全局的。 下面的构造方法必须使用指定的 InteractionSpec 来初始化一个操作对象：   
  
InteractionSpec spec = ...;   
MyMappingRecordOperation eisOperation = new MyMappingRecordOperation(getConnectionFactory(), spec);   
...   
21.4.2. MappingCommAreaOperation   
一些连接器使用了基于COMMAREA的记录，该记录包含了发送给EIS的参数和返回数据的字节数组。 Spring提供了一个专门的操作类用于直接操作COMMAREA而不是操作记录。 MappingCommAreaOperation 类扩展了 MappingRecordOperation 类以提供这种专门的COMMAREA支持。 它隐含地使用了 CommAreaRecord类作为输入和输出record类型， 并提供了两个新的方法来转换输入对象到输入COMMAREA，以及转换输出COMMAREA到输出对象。   
  
public abstract class MappingCommAreaOperation extends MappingRecordOperation {   
...   
protected abstract byte[] objectToBytes(Object inObject)   
   throws IOException, DataAccessException;   
  
protected abstract Object bytesToObject(byte[] bytes)   
   throws IOException, DataAccessException;   
...   
}   
21.4.3. 自动生成输出记录   
由于每个 MappingRecordOperation 子类的内部都是基于 CciTemplate 的，所以 用 CciTemplate 以相同的方式自动生成输出record都是有效的。 每个操作对象提供一个相应的 setOutputRecordCreator(..) 方法。 更多的信息，请参见前面的Section 21.3.4, “自动输出记录生成”一节。   
  
21.4.4. 总结   
操作对象方法使用了跟 CciTemplate 相同的方式来使用记录。   
  
Table 21.2. Usage of Interaction execute methods   
  
MappingRecordOperation method signature MappingRecordOperationoutputRecordCreator property execute method called on the CCI Interaction   
Object execute(Object) not set Record execute(InteractionSpec, Record)   
Object execute(Object) set boolean execute(InteractionSpec, Record, Record)   
  
21.4.5. MappingRecordOperation 使用示例   
在本节中，将通过展示使用 Blackbox CCI 连接器访问一个数据库来说明 MappingRecordOperation 的用法。   
  
Note   
该连接器的最初版本是由SUN提供的J2EE SDK（1.3版本）。   
  
  
首先，必须在 CCI InteractionSpec 中进行一些初始化动作来指定执行哪些SQL请求。 在这个例子中，我们直接定义了将请求参数转换为CCI record以及将CCI结果记录转换为一个 Person 类的实例的方法。   
  
public class PersonMappingOperation extends MappingRecordOperation {   
  
public PersonMappingOperation(ConnectionFactory connectionFactory) {   
  setConnectionFactory(connectionFactory);   
  CciInteractionSpec interactionSpec = new CciConnectionSpec();   
  interactionSpec.setSql("select \* from person where person\_id=?");   
  setInteractionSpec(interactionSpec);   
}   
  
protected Record createInputRecord(RecordFactory recordFactory, Object inputObject)   
   throws ResourceException {   
  Integer id = (Integer) inputObject;   
  IndexedRecord input = recordFactory.createIndexedRecord("input");   
  input.add(new Integer(id));   
  return input;   
}   
  
protected Object extractOutputData(Record outputRecord)   
   throws ResourceException, SQLException {   
  ResultSet rs = (ResultSet) outputRecord;   
  Person person = null;   
  if (rs.next()) {   
   Person person = new Person();   
   person.setId(rs.getInt("person\_id"));   
   person.setLastName(rs.getString("person\_last\_name"));   
   person.setFirstName(rs.getString("person\_first\_name"));   
  }   
  return person;   
}   
}   
然后应用程序会以person标识符作为参数来得到操作对象。注意：操作对象可以被设为共享实例，因为它是线程安全的。   
  
public class MyDaoImpl extends CciDaoSupport implements MyDao {   
  
public Person getPerson(int id) {   
  PersonMappingOperation query = new PersonMappingOperation(getConnectionFactory());   
  Person person = (Person) query.execute(new Integer(id));   
  return person;   
}   
}   
对应的Spring beans的配置看起来类似于下面非托管模式（non-managed mode）的配置：   
  
<bean id="managedConnectionFactory"   
  class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">   
<property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>   
<property name="driverName" value="org.hsqldb.jdbcDriver"/>   
</bean>   
  
<bean id="targetConnectionFactory"   
  class="org.springframework.jca.support.LocalConnectionFactoryBean">   
<property name="managedConnectionFactory" ref="managedConnectionFactory"/>   
</bean>   
  
<bean id="connectionFactory"   
  class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">  
<property name="targetConnectionFactory" ref="targetConnectionFactory"/>   
<property name="connectionSpec">   
  <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">   
   <property name="user" value="sa"/>   
   <property name="password" value=""/>   
  </bean>   
</property>   
</bean>   
  
<bean id="component" class="MyDaoImpl">   
<property name="connectionFactory" ref="connectionFactory"/>   
</bean>   
在托管模式（managed mode）（也就是说，在一个J2EE环境中)），配置可能看起来像这样：   
  
<bean id="targetConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">   
<property name="jndiName" value="eis/blackbox"/>   
</bean>   
  
<bean id="connectionFactory"   
  class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">  
<property name="targetConnectionFactory" ref="targetConnectionFactory"/>   
<property name="connectionSpec">   
  <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">   
   <property name="user" value="sa"/>   
   <property name="password" value=""/>   
  </bean>   
</property>   
</bean>   
  
<bean id="component" class="MyDaoImpl">   
<property name="connectionFactory" ref="connectionFactory"/>   
</bean>   
21.4.6. MappingCommAreaOperation 使用示例   
在本节中，将展示 MappingCommAreaOperation类的用法：通过IBM ECI连接器以ECI的模式访问一个CICS。   
  
首先，CCI InteractionSpec 需要进行初始化以指定那个CICS程序去访问它以及如何与它交互。   
  
public abstract class EciMappingOperation extends MappingCommAreaOperation {   
  
public EciMappingOperation(ConnectionFactory connectionFactory, String programName) {   
  setConnectionFactory(connectionFactory);   
  ECIInteractionSpec interactionSpec = new ECIInteractionSpec(),   
  interactionSpec.setFunctionName(programName);   
  interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC\_SEND\_RECEIVE);   
  interactionSpec.setCommareaLength(30);   
  setInteractionSpec(interactionSpec);   
  setOutputRecordCreator(new EciOutputRecordCreator());   
}   
  
private static class EciOutputRecordCreator implements RecordCreator {   
  public Record createRecord(RecordFactory recordFactory) throws ResourceException {   
   return new CommAreaRecord();   
  }   
}   
}   
The abstract EciMappingOperation class can then be subclassed to specify mappings between custom objects and Records.   
  
EciMappingOperation 抽象类可以被子类化以指定自定义对象和 Records 之间的映射。   
  
public class MyDaoImpl extends CciDaoSupport implements MyDao {   
  
public OutputObject getData(Integer id) {   
  EciMappingOperation query = new EciMappingOperation(getConnectionFactory(), "MYPROG") {   
   protected abstract byte[] objectToBytes(Object inObject) throws IOException {   
    Integer id = (Integer) inObject;   
    return String.valueOf(id);   
   }   
   protected abstract Object bytesToObject(byte[] bytes) throws IOException;   
    String str = new String(bytes);   
    String field1 = str.substring(0,6);   
    String field2 = str.substring(6,1);   
    String field3 = str.substring(7,1);   
    return new OutputObject(field1, field2, field3);   
   }   
  });   
  
  return (OutputObject) query.execute(new Integer(id));   
}   
}   
对应的Spring beans的配置看起来类似于下面非托管模式（non-managed mode）的配置：   
  
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">   
<property name="serverName" value="TXSERIES"/>   
<property name="connectionURL" value="local:"/>   
<property name="userName" value="CICSUSER"/>   
<property name="password" value="CICS"/>   
</bean>   
  
<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">   
<property name="managedConnectionFactory" ref="managedConnectionFactory"/>   
</bean>   
  
<bean id="component" class="MyDaoImpl">   
<property name="connectionFactory" ref="connectionFactory"/>   
</bean>   
在托管模式（managed mode）（也就是说，在一个J2EE环境中），配置可能看起来像这样：   
  
<bean id="connectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">   
<property name="jndiName" value="eis/cicseci"/>   
</bean>   
  
<bean id="component" class="MyDaoImpl">   
<property name="connectionFactory" ref="connectionFactory"/>   
</bean>   
21.5. 事务   
JCA为资源适配器（resource adapters）指定了几个级别的事务支持。 你可以在 ra.xml 文件中指定你的资源适配器支持的事务类型。 它本质上有三个选项：none(例如CICS EPI 连接器)，本地事务（例如CICS ECI 连接器），全局事务（例如IMS 连接器）。   
  
<connector>   
  
<resourceadapter>   
  
  <!-- <transaction-support>NoTransaction</transaction-support> -->   
  <!-- <transaction-support>LocalTransaction</transaction-support> -->   
  <transaction-support>XATransaction</transaction-support>   
  
<resourceadapter>   
  
<connector>   
对于全局事务，可以使用Spring中常见的事务机制来划分事务， 并以 JtaTransactionManager 为后端（委托给后面的J2EE分布式事务协调程序）。   
  
对于单独CCI ConnectionFactory 上的本地事务， Spring为CCI提供了一个专门的事务管理策略， 类似于JDBC中的 DataSourceTransactionManager， CCI API定义了一个本地事务对象和相应的本地事务划分方法。 Spring的 CciLocalTransactionManager 执行这样的本地CCI事务， 完全依照Spring中常见的 PlatformTransactionManager 抽象。   
  
<bean id="eciConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">   
<property name="jndiName" value="eis/cicseci"/>   
</bean>   
  
<bean id="eciTransactionManager"   
  class="org.springframework.jca.cci.connection.CciLocalTransactionManager">   
<property name="connectionFactory" ref="eciConnectionFactory"/>   
</bean>   
声明式或编程式的事务策略都能被用于任意的Spring事务划分功能。 这是Spring通用的 PlatformTransactionManager 抽象的结果，它解耦了实际运行策略中的事务划分。 你可以保持现在的事务划分，只需要在 JtaTransactionManager 和 CciLocalTransactionManager 之间转换即可。