Java私塾《深入浅出学Spring3》——系列精品教程

本节课程概览

- Spring对JDBC的支持

  包括：DAO支持，JdbcTemplate类的使用、NamedParameterJdbcTemplate类的使用等。

第四章：对JDBC和ORM的支持

DAO支持-1

- 简介

  Spring提供的DAO(数据访问对象)支持主要的目的是便于以标准的方式使用不同的数据访问技术，如JDBC，Hibernate或者JDO等。它不仅可以让你方便地在这些持久化技术间切换， 而且让你在编码的时候不用考虑处理各种技术中特定的异常。

- 一致的异常层次

  Spring提供了一种方便的方法，把特定于某种技术的异常，如SQLException， 转化为自己的异常，这种异常属于以DataAccessException 为根的异常层次。这些异常封装了原始异常对象，这样就不会有丢失任何错误信息的风险。

  如果使用拦截器方式，你在应用中 就得自己小心处理HibernateException、 JDOException等，最好是委托给 SessionFactoryUtils的 convertHibernateAccessException、 convertJdoAccessException等方法。这些方法可以把相应的异常转 化为与org.springframework.dao中定义的异常层次相兼容的异常。

DAO支持-2

- 一致的DAO支持抽象类

为了便于以一种一致的方式使用各种数据访问技术，如JDBC、JDO和Hibernate， Spring提供了一套抽象DAO类供你扩展。这些抽象类提供了一些方法，通过它们你可以 获得与你当前使用的数据访问技术相关的数据源和其他配置信息。

1：JdbcDaoSupport - JDBC数据访问对象的基类。 需要一个DataSource，同时为子类提供 JdbcTemplate。

2：HibernateDaoSupport - Hibernate数据访问对象的基类。 需要一个SessionFactory，同时为子类提供 HibernateTemplate。也可以选择直接通过提供一个HibernateTemplate来初始化。

3：JdoDaoSupport - JDO数据访问对象的基类。 需要设置一个PersistenceManagerFactory， 同时为子类提供JdoTemplate。

4：JpaDaoSupport - JPA数据访问对象的基类。 需要一个EntityManagerFactory，同时 为子类提供JpaTemplate。

对JDBC的支持-1

- 简介

Spring JDBC抽象框架所带来的价值将在以下几个方面得以体现：（注：使用了Spring JDBC抽象框架之后，应用开发人员只需要完成斜体加粗字部分的编码工作。）

1：指定数据库连接参数
2：打开数据库连接
3：声明SQL语句
4：预编译并执行SQL语句
5：遍历查询结果（如果需要的话）
6：处理每一次遍历操作
7：处理抛出的任何异常
8：处理事务
9：关闭数据库连接

Spring将替我们完成所有单调乏味的JDBC底层细节处理工作

对JDBC的支持-2

- Spring JDBC抽象框架由四个包构成：core、 dataSource、object以及support

1：core包由JdbcTemplate类以及相关的回调接口和类组成。

2：datasource包由一些用来简化DataSource访问的工具类，以及各种DataSource接口的简单实现(主要用于单元测试以及在J2EE容器之外使用JDBC)组成。

3：object包由封装了查询、更新以及存储过程的类组成，这些类的对象都是线程安全并且可重复使用的。它们类似于JDO，与JDO的不同之处在于查询结果与数据库是“断开连接”的。它们是在core包的基础上对JDBC更高层次的抽象。

4：support包提供了一些SQLException的转换类以及相关的工具类。

在JDBC处理过程中抛出的异常将被转换成org.springframework.dao包中定义的异常。因此使用Spring JDBC进行开发将不需要处理JDBC或者特定的RDBMS才会抛出的异常。所有的异常都是unchecked exception，这样我们就可以对传递到调用者的异常进行有选择的捕获。

JdbcTemplate类-1

- JdbcTemplate是core包的核心类。它替我们完成了资源的创建以及释放工作，从而简化了对JDBC的使用。它还可以帮助我们避免一些常见的错误，比如忘记关闭数据库连接。
- 定义接口如下：

```
public interface Api {
    public boolean create(UserModel um);
}
```

- 定义实现类如下：

```
public class Impl implements Api{
    private DataSource ds = null;
    public void setDs(DataSource ds){
          this.ds = ds;
    }
    public boolean create(UserModel um) {
          JdbcTemplate jt = new JdbcTemplate(ds);
jt.execute("insert into tbl_user (uuid,name) values('"+um.getUuid()+"','"+um.getName()+"')");
          return false;
    }
}
```

JdbcTemplate类-2

- 配置文件

```
<bean name="api" class="cn.javass.Spring3.jdbc.Impl">
    <property name="ds" ref="dataSource"></property>
</bean>
<bean name="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName"><value>oracle.jdbc.driver.OracleDriver</value></property>
<property name="url"><value>jdbc:oracle:thin:@localhost:1521:orcl</value></property>
    <property name="username"> <value>test</value> </property>
    <property name="password" value="test"/>
</bean>
```

- 客户端

```
public static void main(String[] args)throws Exception {
    ApplicationContext context = new ClassPathXmlApplicationContext(
            new String[] {"applicationContext-jdbc.xml"});
    Api api = (Api)context.getBean("api");
    UserModel um = new UserModel();
    um.setUuid("test1");
    um.setName("test1");
    api.create(um);
}
```

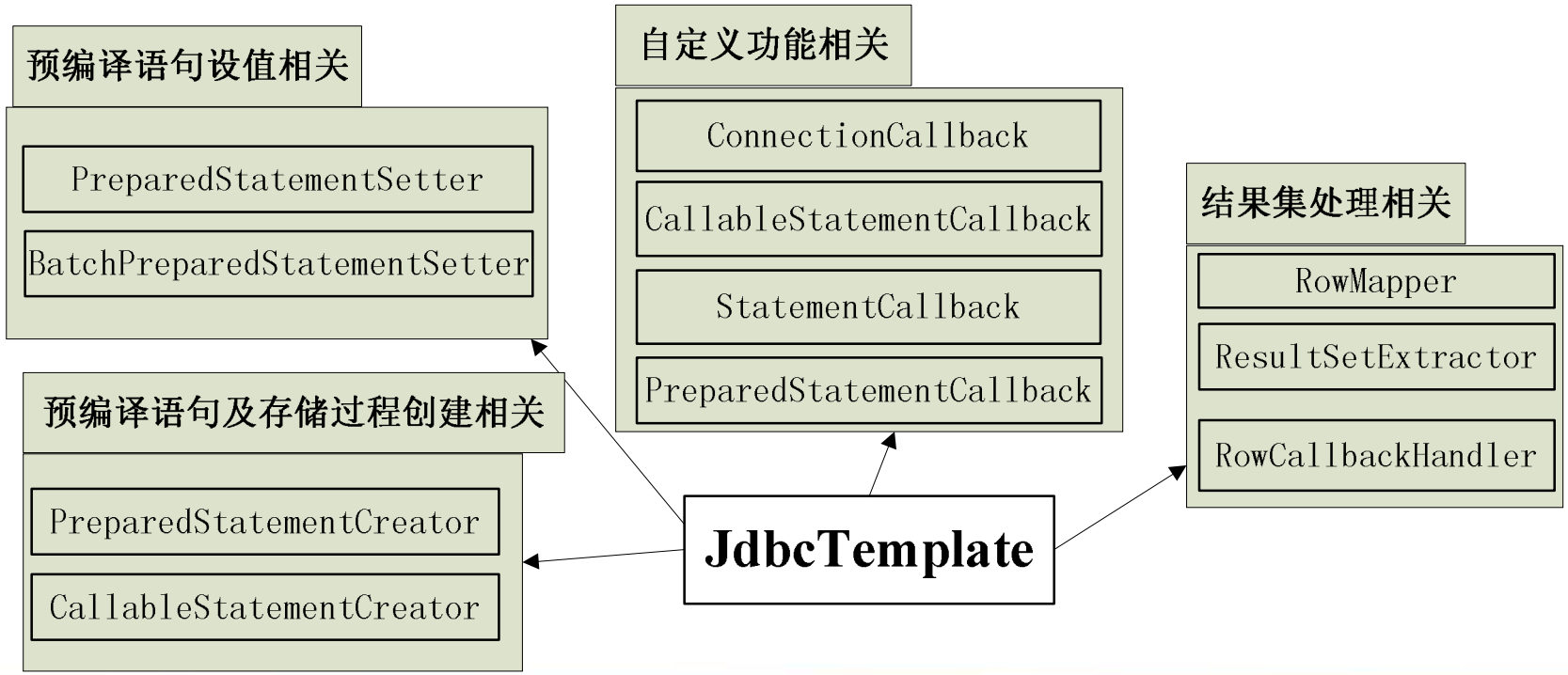
JdbcTemplate类-3

n 如果是需要向里面传递参数的，就需要回调接口，如下：

```
public boolean create(UserModel um1){
    JdbcTemplate jt = new JdbcTemplate(ds);
    final UserModel um = um1;
    class myCallBack implements PreparedStatementCallback{
          public Object doInPreparedStatement(PreparedStatement pstmt)
                              throws SQLException, DataAccessException {
                    pstmt.setString(1,um.getUuid());
                    pstmt.setString(2,um.getName());
                    System.out.println("ddddddddd");
                    return pstmt.executeUpdate();
          }
    }
jt.execute("insert into tbl_user (uuid,name) values(?,?)",new myCallBack());
    return false;
}
```

JdbcTemplate类-4

- JdbcTemplate类的基本实现方式：
  模板方法设计模式+回调方法，把变和不变的部分分离开来，不变化的部分，实现在模板里面，而变化的部分通过回调方法的方式，由外部传入。
- JdbcTemplate类的回调方法体系



NamedParameterJdbcTemplate类-1

- NamedParameterJdbcTemplate类在SQL语句中支持使用命名参数，比较适合做查询，如果做更新，同样需要使用回调方法，如下：

```
NamedParameterJdbcTemplate jt = new NamedParameterJdbcTemplate(ds);
Map paramMap = new HashMap();
paramMap.put("uuid",um.getUuid());
List list = jt.queryForList("select * from tbl_user where
    uuid=:uuid",paramMap);
Iterator it = list.iterator();
while(it.hasNext()){
    Map map = (Map)it.next();
System.out.println("uuid="+map.get("uuid")+",name="+map.get("name"));
}
```

- NamedParameterJdbcTemplate类是线程安全的，该类的最佳使用方式不是每次操作的时候实例化一个新的NamedParameterJdbcTemplate，而是针对每个DataSource只配置一个NamedParameterJdbcTemplate实例

NamedParameterJdbcTemplate类-2

n NamedParameterJdbcTemplate也可以自己做mapper，如下：

```
NamedParameterJdbcTemplate jt = new NamedParameterJdbcTemplate(ds);
Map paramMap = new HashMap();
paramMap.put("uuid",um1.getUuid());
RowMapper mapper = new RowMapper() {
      public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
          UserModel um = new UserModel();
          um.setName(rs.getString("name"));
          um.setUuid(rs.getString("uuid"));
          return um;
      }
  };
List list = jt.query("select * from tbl_user where uuid=:uuid",paramMap,mapper);
Iterator it = list.iterator();
while(it.hasNext()){
    UserModel tempUm = (UserModel)it.next();
    System.out.println("uuid="+tempUm.getUuid()+",name="+tempUm.getName());
}
```

其它JdbcTemplate类

- SimpleJdbcTemplate类

  SimpleJdbcTemplate类是JdbcTemplate类的一个包装器（wrapper），它利用了Java 5的一些语言特性，比如Varargs和Autoboxing。

- SQLExceptionTranslator接口

  SQLExceptionTranslator是一个接口，如果你需要在 SQLException和org.springframework.dao.DataAccessException之间作转换，那么必须实现该接口。

  转换器类的实现可以采用一般通用的做法(比如使用JDBC的SQLState code)，如果为了使转换更准确，也可以进行定制（比如使用Oracle的error code）。

  SQLErrorCodeSQLExceptionTranslator是SQLExceptionTranslator的默认实现。 该实现使用指定数据库厂商的error code，比采用SQLState更精确。

控制数据库连接

- DataSourceUtils类

  DataSourceUtils是一个帮助类提供易用且强大的数据库访问能力，们可以使用该类提供的静态方法从JNDI获取数据库连接以及在必要的时候关闭之。

  尤其在使用标准JDBC，但是又想要使用Spring的事务的时候，最好从DataSourceUtils类来获取JDBC的连接。

- SmartDataSource接口

  SmartDataSource是DataSource 接口的一个扩展，用来提供数据库连接。使用该接口的类在指定的操作之后可以检查是否需要关闭连接。

- AbstractDataSource类

  它实现了DataSource接口的 一些无关痛痒的方法，如果你需要实现自己的DataSource，那么继承 该类是个好主意 。

- SingleConnectionDataSource类

  是SmartDataSource接口 的一个实现，其内部包装了一个单连接。该连接在使用之后将不会关闭，很显然它不能在多线程 的环境下使用

本节课程小结

- Spring对JDBC的支持

  包括：DAO支持，JdbcTemplate类的使用、NamedParameterJdbcTemplate类的使用等。

- 作业：

1：在前面的项目练习中挑选一个模块，把DAO部分的实现，添加一个使用Spring对JDBC支持的实现

本节课程概览

- Spring对ORM的支持

  包括：DAO支持，HibernateTemplate类的使用、HibernateDaoSupport类的使用等。

第四章：对JDBC和ORM的支持

对ORM的支持

- 简介

  Spring在资源管理，DAO实现支持以及事务策略等方面提供了与 JDO、JPA、Hibernate、TopLink及iBATIS的集成。以Hibernate为例，Spring通过使用许多IoC的便捷特性对它提供了一流的支持，帮助你处理很多典型的Hibernate整合的问题。所有的这些支持，都遵循Spring通用的事务和DAO异常体系。通常来说有两种不同的整合风格：你可以使用Spring提供的DAO模板，或者直接使用Hibernate/JDO/TopLink等工具的原生API编写DAO。无论采取哪种风格，这些DAO都可以通过IoC进行配置，并参与到Spring的资源和事务管理中去。

- 使用Spring构建你的O/R Mapping DAO的好处包括：

1：测试简单
2：异常封装
3：通用的资源管理
4：综合的事务管理
5：避免绑定特定技术允许mix-and-match的实现策略

结合Hibernate

n 在Spring的application context中创建 SessionFactory ，如下：

```
<bean id="hbConfig"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource"><ref local="dataSource"/></property>
    <property name="mappingResources">
          <list>
                <value>com/lx/Parent.hbm.xml</value>
          </list>
    </property>
    <property name="hibernateProperties">
          <props>
                <prop key="hibernate.dialect">
                      org.hibernate.dialect.Oracle9Dialect
                </prop>
                <prop key="hibernate.show_sql">true</prop>
          </props>
    </property>
</bean>
```

结合Hibernate示例-1

- 定义接口如下：

```
public interface Api {
    public boolean create(UserModel um);
}
```

- 实现类如下：

```
public class Impl implements Api{
    private SessionFactory sf = null;
    public void setSf(SessionFactory sf){
        this.sf = sf;
    }
    public boolean create(UserModel um){
        HibernateTemplate ht = new HibernateTemplate(sf);
        ht.save(um);
        return false;
    }
}
```

结合Hibernate示例-2

- 配置文件如下：

```
<bean name="api" class="cn.javass.Spring3.h3.Impl">
   <property name="sf" ref="hbConfig"></property>
</bean>
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
   <property name="driverClassName">
        <value>oracle.jdbc.driver.OracleDriver</value>
   </property>
   <property name="url">
        <value>jdbc:oracle:thin:@localhost:1521:orcl</value>
   </property>
   <property name="username"> <value>test</value> </property>
   <property name="password" value="test"/>
</bean>
```

结合Hibernate示例-3

- 配置文件如下：

```
<bean id="hbConfig"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource"><ref local="dataSource"/></property>
    <property name="mappingResources">
          <list>
          <value>cn/javass/Spring3/h3/UserModel.hbm.xml</value>
          </list>
    </property>
    <property name="hibernateProperties">
          <props>
                    <prop key="hibernate.dialect">
                              org.hibernate.dialect.Oracle8iDialect
                    </prop>
                    <prop key="hibernate.show_sql">true</prop>
          </props>
    </property>
</bean>
```

结合Hibernate示例-4

- 客户端文件如下：

```
public static void main(String[] args)throws Exception {
    ApplicationContext context = new ClassPathXmlApplicationContext(
            new String[] {"applicationContext-h3.xml"});
    Api api = (Api)context.getBean("api");
    UserModel um = new UserModel();
    um.setUuid("test1");
    um.setName("test1");
    api.create(um);
}
```

- 前面的演示是更新操作，那么查询怎么做呢？如下：

```
List<UserModel> list = ht.find("select o from UserModel o");
```

- 要传入参数怎么做呢？如下：

```
Object params[] = new Object[1];
params[0] = "test1";
List<UserModel> list = ht.find("select o from UserModel o where
    o.uuid=?",params);
```

结合Hibernate示例-5

- 要传入命名参数怎么做呢？如下：

```
Object params[] = new Object[1];
params[0] = "test1";
String names[] = new String[1];
names[0] = "uuid";
List<UserModel> list = ht.findByNamedParam("select o from UserModel o where
    o.uuid=:uuid",names,params);
```

- 如果需要使用Hibernate的Query接口，该怎么做呢？如下：

```
HibernateCallback hcb =new HibernateCallback(){
public Object doInHibernate(Session session)throws HibernateException, SQLException {
    Query q = session.createQuery("select o from UserModel o where o.uuid=:uuid");
          q.setString("uuid", um.getUuid());
          return q.list();
    }
};
Collection<UserModel> list = (Collection<UserModel>)ht.execute(hcb);
```

- 一个回调实现能够有效地在任何Hibernate数据访问中使用。HibernateTemplate 会确保当前Hibernate的 Session 对象的正确打开和关闭，并直接参与到事务管理中去。 Template 实例不仅是线程安全的，同时它也是可重用的。因而他们可以作为外部对象的实例变量而被持有。

基于Spring的DAO实现

- 实现类改成如下：

```
public class Impl extends HibernateDaoSupport implements Api{
    public Collection testQuery(){
List<UserModel> list = this.getHibernateTemplate().find("select o from UserModel o");
            for(UserModel tempUm : list){
            System.out.println("uuid=="+tempUm.getUuid()+",name="+tempUm.getName());
            }
            return list;
    }
}
```

- 配置文件改成如下：

```
<bean name="api" class="cn.javass.Spring3.h3.Impl">
    <property name="sessionFactory" ref="hbConfig"></property>
</bean>
```

- 要想使用Session怎么办？如下：

```
Session session = getSession(getSessionFactory(), false);
```

通常将一个false作为参数（表示是否允许创建）传递到 getSession(..) 方法中进行调用。 此时，整个调用将在同一个事务内完成（它的整个生命周期由事务控制，避免了关闭返回的 Session 的需要）。

本节课程小结

- Spring对ORM的支持

  包括：DAO支持，HibernateTemplate类的使用、HibernateDaoSupport类的使用等。

- 作业：

1：在前面的项目练习中挑选一个模块，把DAO部分的实现，添加一个使用Spring对Hibernate支持的实现

本节课程概览

- Spring的事务

  包括：Spring对事务的解决方案、Spring中事务管理器体系、AOP的解决方案、基本的事务配置、声明性事务管理

第五章：Spring中的事务

Spring中事务简介-1

- 现有的问题：各种技术的事务处理方式不统一，导致编程的复杂性，如：

```
Connection conn =
    DataSourceUtils.getConnection();
 //开启事务
conn.setAutoCommit(false);
try {
    Object retVal =
        callback.doInConnection(conn);
    conn.commit(); //提交事务
    return retVal;
}catch (Exception e) {
    conn.rollback();//回滚事务
    throw e;
}finally {
    conn.close();
}
```

```
Session session = null;
Transaction transaction = null;
try {
    session = factory.openSession();
    //开启事务
    transaction = session.beginTransaction();
    transation.begin();
    session.save(user);
    transaction.commit();//提交事务
} catch (Exception e) {
    e.printStackTrace();
    transaction.rollback();//回滚事务
    return false;
}finally{
    session.close();
}
```

Spring中事务简介-2

- Spring框架引人注目的重要因素之一是它全面的事务支持。Spring框架提供了一致的事务管理抽象，这带来了以下好处：

1：为复杂的事务API提供了一致的编程模型，如JTA、JDBC、Hibernate、JPA和JDO

2：支持声明式事务管理

3：提供比复杂的事务API（诸如JTA）更简单的、更易于使用的编程式事务管理API

4：非常好地整合Spring的各种数据访问抽象

- Spring事务抽象的关键是事务策略的概念。这个概念由org.springframework.transaction.PlatformTransactionManager接口定义：

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

Spring中事务简介-3

- getTransaction(..)方法根据一个类型为 TransactionDefinition 的参数返回一个 TransactionStatus 对象。返回的 TransactionStatus 对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事务。如同J2EE事务环境，一个 TransactionStatus 也是和执行 线程 绑定的）
- TransactionDefinition接口指定

1：**事务隔离**：当前事务和其它事务的隔离的程度。例如，这个事务能否看到其他事务未提交的写数据？

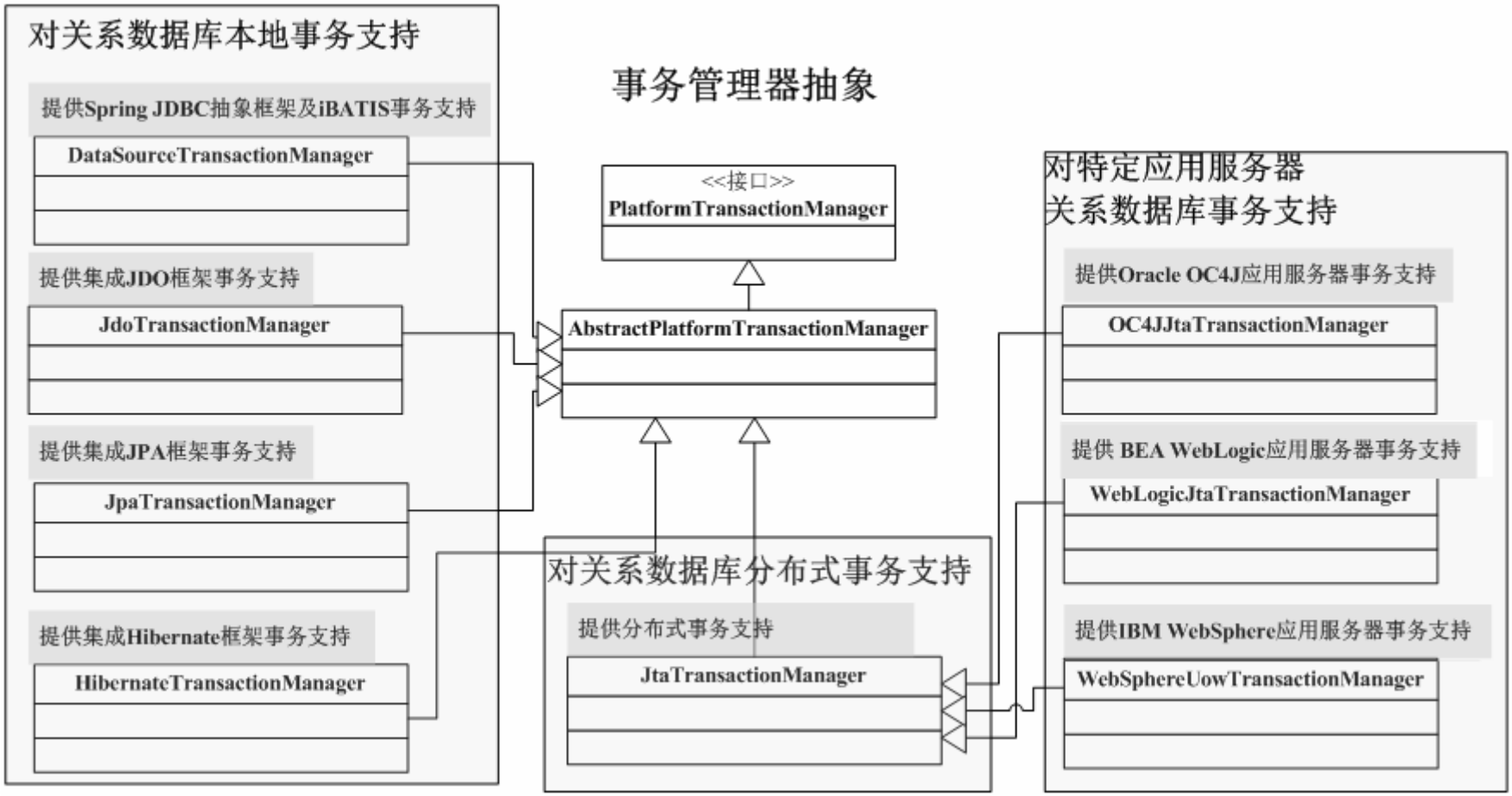
2：**事务传播**：通常在一个事务中执行的所有代码都会在这个事务中运行。但是，如果一个事务上下文已经存在，有几个选项可以指定一个事务性方法的执行行为：例如，简单地在现有的事务中继续运行（大多数情况）；或者挂起现有事务，创建一个新的事务。Spring提供EJB CMT中常见的事务传播选项。

3：**事务超时**：事务在超时前能运行多久（自动被底层的事务基础设施回滚）。

4：**只读状态**：只读事务不修改任何数据。只读事务在某些情况下（例如当使用Hibernate时），是一种非常有用的优化。

Spring中事务简介-4

- Spring的事务管理器体系



资源同步方案

- 高层次方案：

  首选的方法是使用Spring的高层持久化集成API。这种方式不会替换原始的API，而是在内部封装资源创建、复用、清理、事务同步以及异常映射等功能，这样用户的数据访问代码就不必关心这些，而集中精力于自己的持久化逻辑。

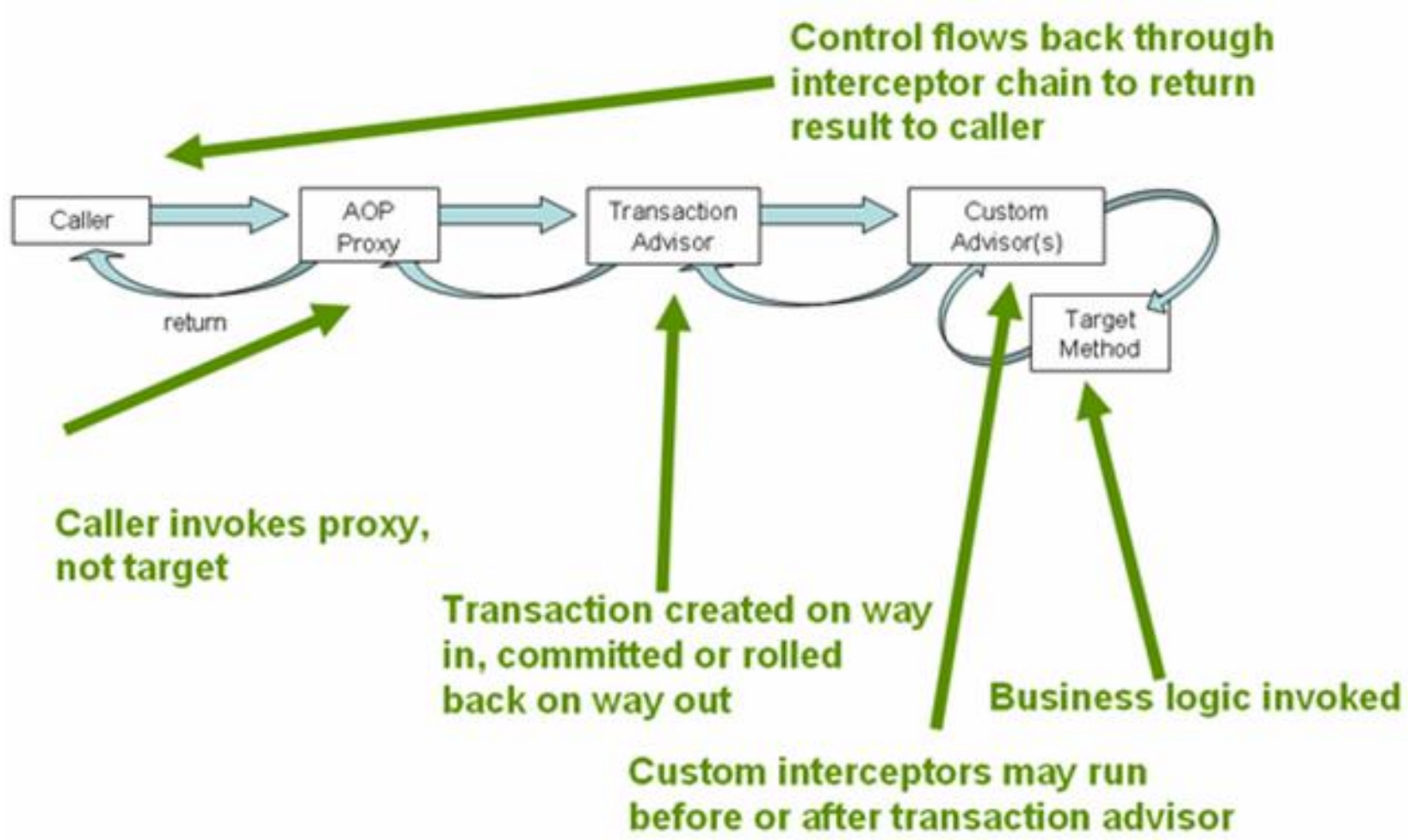
- 低层次方案 ：

  在较低层次上，有以下这些类：DataSourceUtils（针对JDBC），SessionFactoryUtils（针对Hibernate），PersistenceManagerFactoryUtils（针对JDO）等等。也就是说，在纯JDBC中，如果你需要使用Spring的事务，那么就需要使用上面的类来管理连接，如下：

  ```
  Connection conn=DataSourceUtils.getConnection(dataSource);
  ```

- 当然，一旦你用过Spring的JDBC支持或Hibernate支持，你一般就不再会选择DataSourceUtils 或是别的辅助类了，因为你会更乐意与Spring抽象一起工作，而不是直接使用相关的API。

AOP的解决方案-1


Control flows back through interceptor chain to return result to caller
Caller
AOP Proxy
Transaction Advisor
Custom Advisor(s)
Target Method
return
Caller invokes proxy, not target
Transaction created on way in, committed or rolled back on way out
Business logic invoked
Custom interceptors may run before or after transaction advisor

AOP的解决方案-2

- Spring框架提供了一致的事务管理抽象，通常实施事务的步骤是：

1：定义资源（DataSource/SessionFactory……）
2：定义事务管理器（管理资源的事务）
3：定义事务通知：定义了如何实施事务（实施事务的方法名和对应的事务属性），需要使用事务管理器管理事务，定义了如何选择目标对象的方法及实施的事务属性
4：定义advisor（切入点和事务通知）：切入点选择需要实施事务的目标对象（通常是业务逻辑层）
5：Spring织入事务通知到目标对象（AOP代理）

Spring中事务配置-1

- 要使用Spring的事务，首先要先定义一个与数据源连接的DataSource，然后使用Spring的DataSourceTransactionManager，并传入指向DataSource的引用，比如，JDBC的事务配置如下：

```
<bean name="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName">
   <value>oracle.jdbc.driver.OracleDriver</value></property>
<property name="url">
<value>jdbc:oracle:thin:@localhost:1521:orcl</value></property>
   <property name="username"> <value>test</value> </property>
   <property name="password" value="test"/>
</bean>
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource"/>
</bean>
```

Spring中事务配置-2

- 如果使用JTA，示例如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-
    3.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-3.0.xsd
">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>
  <bean id="txManager"
    class="org.springframework.transaction.jta.JtaTransactionManager" />
</beans>
```

Spring中事务配置-3

- Hibernate的事务配置，示例如下：

1：Spring对Hibernate的集成

2：然后配置事务如下：

```
<bean id="txManager "
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

- 可以简单地使用 JtaTransactionManager 来处理Hibernate事务和JTA事务，就像我们处理JDBC，或者任何其它的资源策略一样，如下：

```
<bean id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

- 注意任何资源的JTA配置都是这样的，因为它们都是全局事务，可以支持任何事务性资源。在所有这些情况下，应用程序代码根本不需要做任何改动。仅仅通过改变配置就可以改变事务管理方式，即使这些更改是在局部事务和全局事务间切换。

声明性事务管理-1

n Spring的声明式事务管理是通过AOP实现的，Xml风格的声明式事务配置如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-
    3.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
    <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
          <property name="dataSource" ref="dataSource"/>
    </bean>
    <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="get*" read-only="true"/>
        <tx:method name="*"/>
      </tx:attributes>
    </tx:advice>
    <aop:config>
      <aop:pointcut id="myPointCut" expression="execution(* cn.javass.spring3.jdbc.Impl.*(..))"/>
      <aop:advisor advice-ref="txAdvice" pointcut-ref="myPointCut"/>
    </aop:config>
```

声明性事务管理-2

```
<bean name="api" class="cn.javass.spring3.jdbc.Impl">
    <property name="ds" ref="dataSource"></property>
</bean>
<bean name="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName">
        <value>oracle.jdbc.driver.OracleDriver</value>
    </property>
    <property name="url">
        <value>jdbc:oracle:thin:@localhost:1521:orcl</value>
    </property>
    <property name="username"> <value>ztb</value> </property>
    <property name="password" value="ztb"/>
</bean>
</beans>
```

声明性事务管理-3

- 事务回滚：可以在配置中指定需要回滚的Exception，如下：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="false"
        rollback-for="NoProductInStockException"/>
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

- <tx:advice/> 有关的设置，默认的 <tx:advice/> 设置如下：

1：事务传播设置是 REQUIRED
2：隔离级别是 DEFAULT
3：事务是 读/写
4：事务超时默认是依赖于事务系统的，或者事务超时没有被支持。
5：任何 RuntimeException 将触发事务回滚，但是任何 checked Exception 将不触发事务回滚

声明性事务管理-4

- 〈tx:advice/〉有关的设置：

属性	是否需要？	默认值	描述
name	是		与事务属性关联的方法名。通配符（*）可以用来指定一批关联到相同的事务属性的方法。 如：'get*'、'handle*'、'on*Event'等等。
propagation	不	REQUIRED	事务传播行为
isolation	不	DEFAULT	事务隔离级别
timeout	不	-1	事务超时的时间（以秒为单位）
read-only	不	false	事务是否只读？
rollback-for	不		将被触发进行回滚的 Exception(s)；以逗号分开。 如：'com.foo.MyBusinessException,ServletException'
no-rollback-for	不		不被触发进行回滚的 Exception(s)；以逗号分开。 如：'com.foo.MyBusinessException,ServletException'

本节课程小结

- Spring的事务

  包括：Spring对事务的解决方案、Spring中事务管理器体系、AOP的解决方案、基本的事务配置、声明性事务管理

- 作业：

1：复习和掌握这些理论知识

2：动手配置和测试一下Spring的声明性事务，体会Spring事务的无侵入性

本节课程概览

- Spring的事务

  包括：使用@Transactional进行声明性事务管理、编程式事务管理，使用Spring的TransactionTemplate、或者使用Spring的PlatformTransactionManager进行事务管理

第五章：Spring中的事务

声明性事务管理-5

- 更简单的方式是使用@Transactional

1：在配置文件中开启对@Transactional的支持

```
<tx:annotation-driven transaction-manager="txManager"/>
```

2：在类或者方法上配置@Transactional

3：@Transactional后面可以跟属性配置，如下：

```
@Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
```

- 默认的@Transactional配置如下：

1：事务传播设置是 PROPAGATION_REQUIRED

2：事务隔离级别是 ISOLATION_DEFAULT

3：事务是 读/写

4：事务超时默认是依赖于事务系统的，或者事务超时没有被支持。

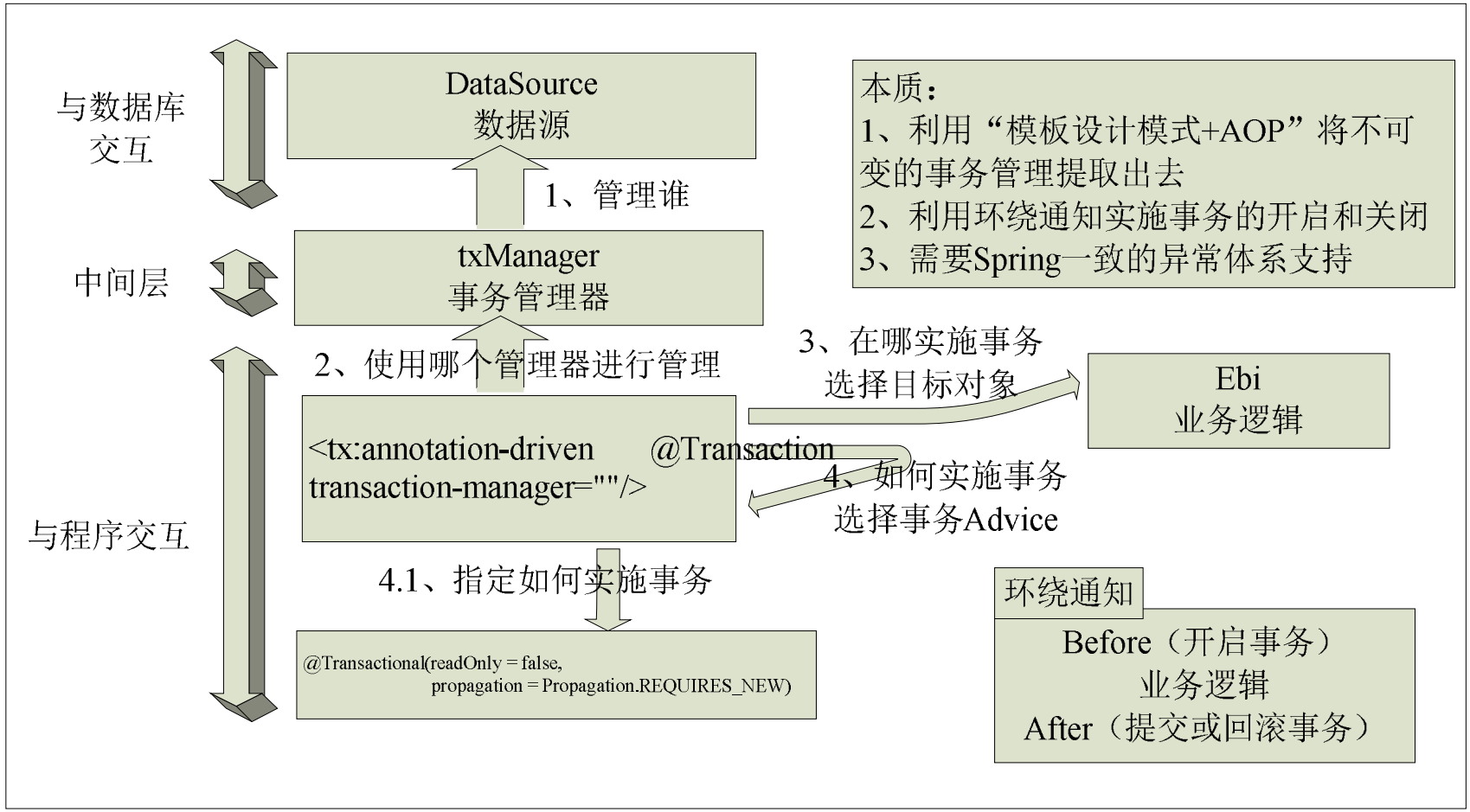
5：任何 RuntimeException 将触发事务回滚，但是任何 checked Exception 将不触发事务回滚

声明性事务管理-6

@Transactional 注解的属性

属性	类型	描述
传播性	枚举型：Propagation	可选的传播性设置
隔离性	枚举型：Isolation	可选的隔离性级别（默认值：ISOLATION_DEFAULT）
只读性	布尔型	读写型事务 vs. 只读型事务
超时	int型（以秒为单位）	事务超时
回滚异常类（rollbackFor）	一组 Class 类的实例，必须是 Throwable 的子类	一组异常类，遇到时 **必须** 进行回滚。默认情况下checked exceptions不进行回滚，仅unchecked exceptions（即RuntimeException的子类）才进行事务回滚。
回滚异常类名（rollbackForClassname）	一组 Class 类的名字，必须是 Throwable的子类	一组异常类名，遇到时 **必须** 进行回滚
不回滚异常类（noRollbackFor）	一组 Class 类的实例，必须是 Throwable 的子类	一组异常类，遇到时 **必须不** 回滚。
不回滚异常类名（noRollbackForClassname）	一组 Class 类的名字，必须是 Throwable 的子类	一组异常类，遇到时 **必须不** 回滚

声明性事务管理-7



编程式事务管理-1

- Spring提供两种方式的编程式事务管理：

1：使用 TransactionTemplate

2：直接使用一个 PlatformTransactionManager 实现

如果你选择编程式事务管理，Spring小组推荐你采用第一种方法（即使用TransactionTemplate）。第二种方法类似使用JTA的 UserTransaction API（除了异常处理简单点儿）。

- 使用 TransactionTemplate：TransactionTemplate 采用与Spring中别的摸板同样的方法，使用回调机制，将应用代码从样板式的资源获取和释放代码中解放出来，不再有大量的try/catch/finally/try/catch代码块。同样，和别的模板类一样，TransactionTemplate 类的实例是线程安全的。

- 使用TransactionTemplate示例如下：

编程式事务管理-2

- 直接使用TransactionTemplate

```
//获取事务管理器
PlatformTransactionManager txManager = (PlatformTransactionManager)
        ctx.getBean("txManager");
//定义事务管理的模板
TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
//定义事务属性
transactionTemplate.
    setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
//回调，执行真正的数据库操作，如果需要返回值需要在回调里返回
transactionTemplate.execute(new TransactionCallback() {
    @Override
    public Object doInTransaction(TransactionStatus status) {
    //执行数据库操作
    new JdbcTemplate(ds).execute("insert into tbl_user (uuid,name)
   values('"+um1.getUuid()+"','"+um1.getName()+"')");
    return null;
    }
});
```

编程式事务管理-3

- 使用 PlatformTransactionManager

  你也可以直接使用PlatformTransactionManager的实现来管理事务。只需通过bean引用简单地传入一个 PlatformTransactionManager 实现，然后使用 TransactionDefinition 和 TransactionStatus 对象，你就可以启动一个事务，提交或回滚。

- 使用 PlatformTransactionManager示例如下：

编程式事务管理-4

n 直接使用统一的PlatformTransactionManager

```
//获取事务管理器
PlatformTransactionManager txManager = (PlatformTransactionManager)
      ctx.getBean("txManager");
//定义事务属性
DefaultTransactionDefinition td = new DefaultTransactionDefinition();
td.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
//开启事务,得到事务状态
TransactionStatus status = txManager.getTransaction(td);
try {
    //执行数据库操作
new JdbcTemplate(ds).execute("insert into tbl_user (uuid,name)
    values('"+um1.getUuid()+"','"+um1.getName()+"')");
    //提交事务
    txManager.commit(status);
}catch (Exception e) {
    //回滚事务
    txManager.rollback(status);
}
```

本节课程小结

- Spring的事务

  包括：使用@Transactional进行声明性事务管理、编程式事务管理，使用Spring的TransactionTemplate、或者使用Spring的PlatformTransactionManager进行事务管理

- 作业：

1：把前面测试事务的实现，该成使用@Transactional来实现

2：编程体会一下Spring提供的两种编程式事务管理的方法

3：为前面的应用程序添加上事务管理的功能

本节课程概览

- Struts2+Spring3+Hibernate3整合

第六章：Struts2+Spring3+Hibernate3整合

Spring3整合Struts2-1

- 准备三个框架结合的lib包
- Spring3结合Struts2的步骤如下：

1：开启Struts2结合Spring3，在struts.xml中添加如下语句：

```
<constant name="struts.objectFactory" value="spring"/>
```

2：在web.xml中添加listener，如下：

```
<listener>
        <listener-class>
org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
```

3：在web.xml中指定需要初始读取的spring配置文件，如下：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext-
   *.xml,classpath*:applicationContext-*.xml</param-value>
</context-param>
```

当然别忘了加上Struts2自己的filter。

Spring3整合Struts2-2

4：在struts.xml中Action配置的时候，如下：

```
<action name="testAction" class="springBeanName">
    <result name="success">/index.jsp</result>
</action>
```

5：在Spring中正常定义Bean就可以了，把Action定义成为Bean，如下：

```
<bean id="testAction" class="cn.javass.test.web.TestActioin">
   <property name="ebi" ref="testEbi"/>
</bean>
```

6：在Struts的Action中，就可以通过依赖注入的方式来注入需要使用的接口了。

本节课程小结

- 掌握Struts2+Spring3+Hibernate3整合的方式

  分成两个部分，一个是Spring3和Hibernate3的整合；另外一个是Spring3和Struts2的整合。

  对于整合又要注意两个方面，一个是如何配置和部署；另外一个是整合过后，程序写法方面有什么变化。

- 作业：

1：按照课堂讲述，把Struts2+Spring3+Hibernate3整合起来

2：从前面的项目中找一个模块，把它修改成使用Struts2+Spring3+Hibernate3结合起来开发的方式。

本节课程概览

- Spring3的表达式语言

第七章：Spring3的表达式语言

概述

- 是什么

  Spring表达式语言全称为“Spring Expression Language”，缩写为“SpEL”，类似于Struts2x中使用的OGNL表达式语言，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等等，并能与Spring功能完美整合。

  表达式语言给静态Java语言增加了动态功能。

  SpEL是单独模块，只依赖于core模块，不依赖于其他模块，可单独使用。

- 能干什么

  表达式语言一般是用较为简单的形式完成主要的工作，减少开发的工作量。

  SpEL支持如下表达式：

一、**基本表达式：**字面量表达式、关系，逻辑与算数运算表达式、字符串连接及截取表达式、三目运算及Elivis表达式、正则表达式、括号优先级表达式；

二、**类相关表达式：**类类型表达式、类实例化、instanceof表达式、变量定义及引用、赋值表达式、自定义函数、对象属性存取及安全导航表达式、对象方法调用、Bean引用；

三、**集合相关表达式：**内联List、内联数组、集合，字典访问、列表，字典，数组修改、集合投影、集合选择；不支持多维内联数组初始化；不支持内联字典定义；

四、**其他表达式：**模板表达式。

HelloWorld

n 在Java中使用SpringEL是非常简单的，示例如下：

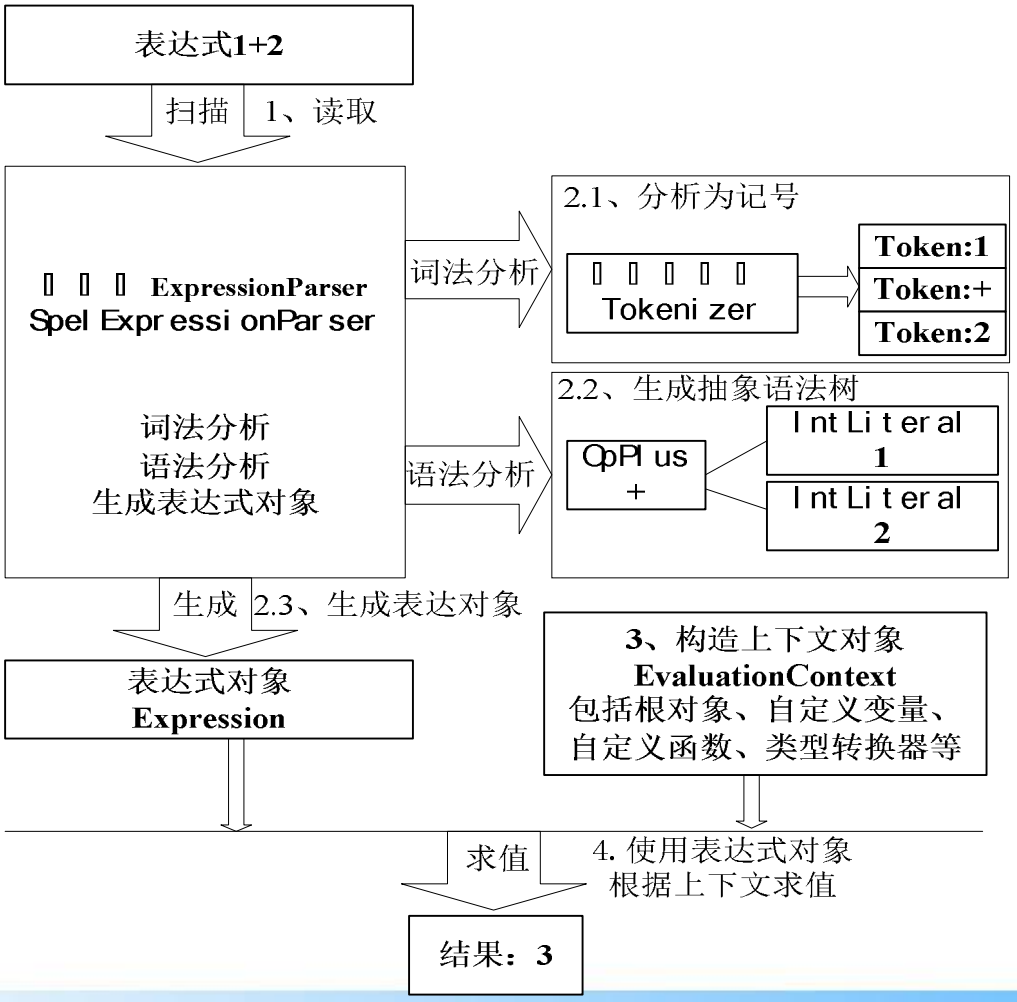
```
public class HelloWorld {
    public static void main(String[] args) {
        //构造上下文：准备比如变量定义等等表达式运行需要的上下文数据
        EvaluationContext context = new StandardEvaluationContext();
        //创建解析器：提供SpelExpressionParser默认实现
        ExpressionParser parser = new SpelExpressionParser();
        //解析表达式：使用ExpressionParser来解析表达式为相应的Expression对象
        Expression expression =
                parser.parseExpression("('Hello' + ' World').concat(#end)");

        //设置上下文中的变量的值
        context.setVariable("end", "!SpringEL");
        //执行表达式，获取运行结果
        String str = (String)expression.getValue(context);
        System.out.println("the str="+str);
    }
}
```

SpEL的基本概念

- **表达式**
  表达式是表达式语言的核心，从我们角度来看就是“要做什么”
- **解析器**
  用于将字符串表达式解析为表达式对象，从我们角度来看是“谁来做”
- **上下文**
  表达式对象执行的环境，该环境可能定义变量、定义自定义函数、提供类型转换等等，从我们角度看就是“在哪做”

SpEL的基本运行原理



SpEL的主要接口-1

- **ExpressionParser接口**

表示解析器，默认实现是org.springframework.expression.spel.standard包中的SpelExpressionParser类，使用parseExpression方法将字符串表达式转换为Expression对象。

配合**ExpressionParser** 接口使用的ParserContext接口用于定义字符串表达式是不是模板，及模板开始与结束字符，示例如下：

```
public static void main(String[] args) {
 ExpressionParser parser = new SpelExpressionParser();
 //自定义一个解析模板的规则
     ParserContext parserContext = new ParserContext() {
         public boolean isTemplate() {              return true;         }
         public String getExpressionPrefix() {      return "#{";         }
         public String getExpressionSuffix() {      return "}";        }
     };
     String template = "#{'Hello '}#{'World!'}";
     Expression expression = parser.parseExpression(template, parserContext);
     System.out.println( expression.getValue());
}
```

SpEL的主要接口-2

- 说明

1：在此演示的是使用ParserContext的情况，此处定义了ParserContext实现：定义表达式是模块，表达式前缀为“#{”，后缀为“}”；
2：使用parseExpression解析时传入的模板必须以“#{”开头，以“}”结尾，如“#{‘Hello ’}#{‘World!’}”。
3：请注意默认传入的字符串表达式不是模板形式，如之前演示的Hello World。

- Expression接口

表示表达式对象，默认实现是org.springframework.expression.spel.standard包中的SpelExpression，提供getValue方法用于获取表达式值，提供setValue方法用于设置对象值

- EvaluationContext接口

表示上下文环境，默认实现是org.springframework.expression.spel.support包中的StandardEvaluationContext类，使用setRootObject方法来设置根对象，使用setVariable方法来注册自定义变量，使用registerFunction来注册自定义函数等等。

SpEL的基本表达式-1

- **字面量表达式：**

SpEL支持的字面量包括：字符串、数字类型（int、long、float、double）、布尔类型、null类型 ，示例如下：

```
1: String str1 = parser.parseExpression("'Hello World!'").getValue(String.class);
2: String str2 = parser.parseExpression("\"Hello World!\"").getValue(String.class);
3: int int1 = parser.parseExpression("1").getValue(Integer.class);
4: float float1 = parser.parseExpression("1.1").getValue(Float.class);
5: boolean true1 = parser.parseExpression("true").getValue(boolean.class);
6: Object null1 = parser.parseExpression("null").getValue(Object.class);
```

- **算数运算表达式 ：**

SpEL支持加(+)、减(-)、乘(*)、除(/)、求余（%）、幂（^）运算，示例如下：

```
1: int result1 = parser.parseExpression("1+2-3*4/2").getValue(Integer.class);
2: int result2 = parser.parseExpression( “4%3” ).getValue(Integer.class) ;
3: int result3 = parser.parseExpression("2^3").getValue(Integer.class);
```

SpEL还提供求余（MOD）和除（DIV）运算符，与“%”和“/”等价，不区分大小写。

SpEL的基本表达式-2

- 关系表达式

  等于（==）、不等于(!=)、大于(>)、大于等于(>=)、小于(<)、小于等于(<=)，区间（between）运算，示例如下：

1：“parser.parseExpression(”1>2“).getValue(boolean.class);”将返回false；

2：“parser.parseExpression(”1 between {1, 2}“).getValue(boolean.class);”将返回true。

  SpEL同样提供了等价的“EQ”、“NE”、“GT”、“GE”、“LT”、“LE”来表示等于、不等于、大于、大于等于、小于、小于等于，不区分大小写

- 逻辑表达式：且（and）、或(or)、非(!或NOT)。 示例如下：

```
1: String expression1 = "2>1 and (!true or !false)";
boolean result1 = parser.parseExpression(expression1).getValue(boolean.class);
```

注意：逻辑运算符不支持 Java中的 && 和 ||

- 字符串连接及截取表达式

  使用“+”进行字符串连接，使用“‘String’ [index]”来获取一个字符，目前只支持获取一个字符，如“’Hello ’ + ’World!’”得到“Hello World!”；而“’Hello World!’[0]”将返回“H”

SpEL的基本表达式-3

- 三目运算及Elivis运算表达式

1：三目运算符 “表达式1?表达式2:表达式3”用于构造三目运算表达式，如“2>1?true:false”将返回true；

2：Elivis运算符“表达式1?:表达式2”从Groovy语言引入，用于简化三目运算符的，当表达式1为非null时则返回表达式1，当表达式1为null时则返回表达式2，如“null?:false”将返回false，而“true?:false”将返回true；

- 正则表达式

使用“str matches regex，如“ ‘123’ matches ‘\\d{3}’ ”将返回true；

- 括号优先级表达式

使用“(表达式)”构造，括号里的具有高优先级。

SpEL的类相关的表达式-1

- **类类型表达式**

使用“T(Type)”来表示java.lang.Class实例，“Type”必须是类全限定名，“java.lang”包除外，即该包下的类可以不指定包名；使用类类型表达式还可以进行访问类静态方法及类静态字段。 示例如下：

1：访问java.lang包的类

```
Class<String> result1 =
parser.parseExpression("T(String)").getValue(Class.class);
```

2：访问其他包下的类 ：

```
String expression2 = "T(cn.javass.spring.chapter5.SpELTest)";
 Class<String> result2 =
parser.parseExpression(expression2).getValue(Class.class);
```

3：访问类的静态字段

```
int
result3=parser.parseExpression("T(Integer).MAX_VALUE").getValue(int.class);
```

4：访问类的静态方法

```
int result4 =
parser.parseExpression("T(Integer).parseInt('1')").getValue(int.class);
```

SpEL的类相关的表达式-2

- 类实例化

类实例化同样使用java关键字“new”，类名必须是全限定名，但java.lang包内的类型除外，如String、Integer。示例如下：

```
1: String result1 = parser.parseExpression("new
   String('hello')").getValue(String.class);
2: Date result2 = parser.parseExpression("new
   java.util.Date()").getValue(Date.class);
```

- instanceof表达式

SpEL支持instanceof运算符，跟Java内使用同义；如“hello ‘ instanceof T(String)”将返回true。

- 变量定义及引用

1：变量通过EvaluationContext接口的setVariable(variableName, value)方法定义
2：在表达式中使用“#variableName”引用；
3：除了引用自定义变量，SpE还允许引用根对象及当前上下文对象，使用“#root”引用根对象，使用“#this”引用当前上下文对象；
示例如下：

SpEL的类相关的表达式-3

```
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = new StandardEvaluationContext();
context.setVariable("variable", "hello1");
context.setVariable("variable", "hello2");
String result1 = parser.parseExpression("#variable").getValue(context,
    String.class);
System.out.println("r1=="+result1);

context = new StandardEvaluationContext(12);
String result2 = parser.parseExpression("#root-1").getValue(context, String.class);
System.out.println("r2=="+result2);
String result3 = parser.parseExpression("#this").getValue(context, String.class);
System.out.println("r3=="+result3);
```

输出结果:

```
r1==hello2
r2==11
r3==12
```

SpEL的类相关的表达式-4

- **自定义函数**

目前只支持类静态方法注册为自定义函数；SpEL使用StandardEvaluationContext的registerFunction方法进行注册自定义函数，其实完全可以使用setVariable代替，两者其本质是一样的。示例如下：

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
Method parseInt =
        Integer.class.getDeclaredMethod("parseInt", String.class);
context.registerFunction("regParseInt", parseInt);
context.setVariable("parseInt2", parseInt);
String expression1 = "#regParseInt('3') == #parseInt2('3')";
boolean result =
    parser.parseExpression(expression1).getValue(context, boolean.class);
System.out.println("result="+result);
```

可以看出“registerFunction”和“setVariable”都可以注册自定义函数，但是两个方法的含义不一样，推荐使用“registerFunction”方法注册自定义函数。

SpEL的类相关的表达式-5

- 赋值表达式

  SpEL即允许给自定义变量赋值，也允许给跟对象赋值，直接使用“#variableName=value”即可赋值。示例如下：

```
1：parser.parseExpression("#root= ‘Hi’").getValue(context, String.class);
2：parser.parseExpression("#this= ‘Hi’").getValue(context, String.class);
3：context.setVariable("#variable", "variable");
```

- 对象属性存取及安全导航表达式

  对象属性获取非常简单，使用如“a.property.property”这种点缀式获取，SpEL对于属性名首字母是不区分大小写的；

  SpEL还引入了Groovy语言中的安全导航运算符“(对象|属性)?.属性”，用来避免当“?.”前边的表达式为null时抛出空指针异常，而是返回null；

  修改属性值可以通过赋值表达式或Expression接口的setValue方法修改。示例如下：

SpEL的类相关的表达式-6

```
UserModel um = new UserModel();
um.setUuid("User1");
um.setName("UserName1");
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setVariable("um",um);
//取值
Expression expression = parser.parseExpression("'uuid='+#um.uuid +
    ',name='+#um.name");
String v =  expression.getValue(context,String.class);
System.out.println("v=="+v);
//赋值
expression = parser.parseExpression("'uuid='+(#um.uuid='newUser') +
    ',name='+#um.name");
v =  expression.getValue(context,String.class);
System.out.println("v2=="+v);
```

输出结果：

```
v==uuid=User1,name=UserName1
v2==uuid=newUser,name=UserName1
```

SpEL的类相关的表达式-7

- **对象方法调用**

  对象方法调用更简单，跟Java语法一样；如“‘Hello’.substring(1,3)”将返回“el”；而对于根对象可以直接调用方法。

- **Bean引用**

  SpEL支持使用“@”符号来引用Bean，在引用Bean时需要使用BeanResolver接口实现来查找Bean，Spring提供BeanFactoryResolver实现，示例如下：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext.xml"});

ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setBeanResolver(new BeanFactoryResolver(ctx));
String result1 =
parser.parseExpression("@myBean.test()").getValue(context, String.class);
```

SpEL的集合相关的表达式-1

- **内联List**

从Spring3.0.4开始支持内联List，使用{表达式，……}定义内联List，如“{1,2,3}”将返回一个整型的ArrayList，而“{}”将返回空的List，对于字面量表达式列表，SpEL会使用java.util.Collections.unmodifiableList方法将列表设置为不可修改。示例如下：

```
1：//将返回不可修改的空List
List<Integer> result2 = parser.parseExpression("{}").getValue(List.class);
2：对于字面量列表也将返回不可修改的List
ExpressionParser parser = new SpelExpressionParser();
List<Integer> result1 = parser.parseExpression("{1,2,3}").getValue(List.class);
//result1.set(0, 2);//这句话会报错，因为list不可以修改
System.out.println(result1);
3：//对于列表中只要有一个不是字面量表达式，将只返回原始List，
//不会进行不可修改处理，也就是可以修改
String expression3 = "{{1+2,2+4},{3,4+4}}";
List<List<Integer>> result3 =
    parser.parseExpression(expression3).getValue(List.class);
result3.get(0).set(0, 1);
```

SpEL的集合相关的表达式-2

- **内联数组**

和Java 数组定义类似，只是在定义时进行多维数组初始化。 示例如下：

```
1：//定义一维数组并初始化
int[] result1 = parser.parseExpression("new
   int[1]").getValue(int[].class);
2：//声明二维数组并初始化
int[] result2 = parser.parseExpression("new
   int[2]{1,2}").getValue(int[].class);
3：//定义多维数组但不初始化
int[][][] result3 =
   parser.parseExpression(expression3).getValue(int[][][].class);
4：//错误的定义多维数组，多维数组不能初始化
String expression4 = "new int[1][2][3]{{1}{2}{3}}";
int[][][] result4 =
   parser.parseExpression(expression4).getValue(int[][][].class);
```

SpEL的集合相关的表达式-3

- **访问元素**

  SpEL目前支持所有集合类型和字典类型的元素访问，使用“集合[索引]”访问集合元素，使用“map[key]”访问字典元素 。示例如下；

1：//SpEL内联List访问

```
int result1 = parser.parseExpression("{1,2,3}[0]").getValue(int.class);
//相当于result1.get(0)
```

2：//SpEL目前支持所有集合类型的访问

```
Collection<Integer> collection = new HashSet<Integer>();
collection.add(1);
collection.add(2);
EvaluationContext context2 = new StandardEvaluationContext();
context2.setVariable("collection", collection);
int result2 = parser.parseExpression("#collection[1]").getValue(context2,
   int.class);
```

SpEL的集合相关的表达式-4

3：//SpEL对Map字典元素访问的支持

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 1);
EvaluationContext context3 = new StandardEvaluationContext();
context3.setVariable("map", map);
int result3 = parser.parseExpression("#map['a']").getValue(context3,
    int.class);
```

- 元素修改

  可以使用赋值表达式或Expression接口的setValue方法修改。示例如下：

1：//修改数组元素值

```
int[] array = new int[] {1, 2};
EvaluationContext context1 = new StandardEvaluationContext();
context1.setVariable("array", array);
int result1 = parser.parseExpression("#array[1] = 3").getValue(context1,
    int.class);
```

SpEL的集合相关的表达式-5

2：//修改集合值

```
Collection<Integer> collection = new ArrayList<Integer>();
collection.add(1);
collection.add(2);
EvaluationContext context2 = new StandardEvaluationContext();
context2.setVariable("collection", collection);
int result2 = parser.parseExpression("#collection[1] = 3").getValue(context2,
    int.class);
parser.parseExpression("#collection[1]").setValue(context2, 4);
result2 = parser.parseExpression("#collection[1]").getValue(context2,
    int.class);
```

3：//修改map元素值

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 1);
EvaluationContext context3 = new StandardEvaluationContext();
context3.setVariable("map", map);
int result3 = parser.parseExpression("#map['a'] = 2").getValue(context3,
    int.class);
```

SpEL的集合相关的表达式-6

- 集合投影

在SQL中投影指从表中选择出列，而在SpEL指从集合中的元素，通过选择来构造新的集合，该集合和原集合具有相同数量的元素，但可能属性不一样；SpEL使用“（list|map）.![投影表达式]”来进行投影运算。

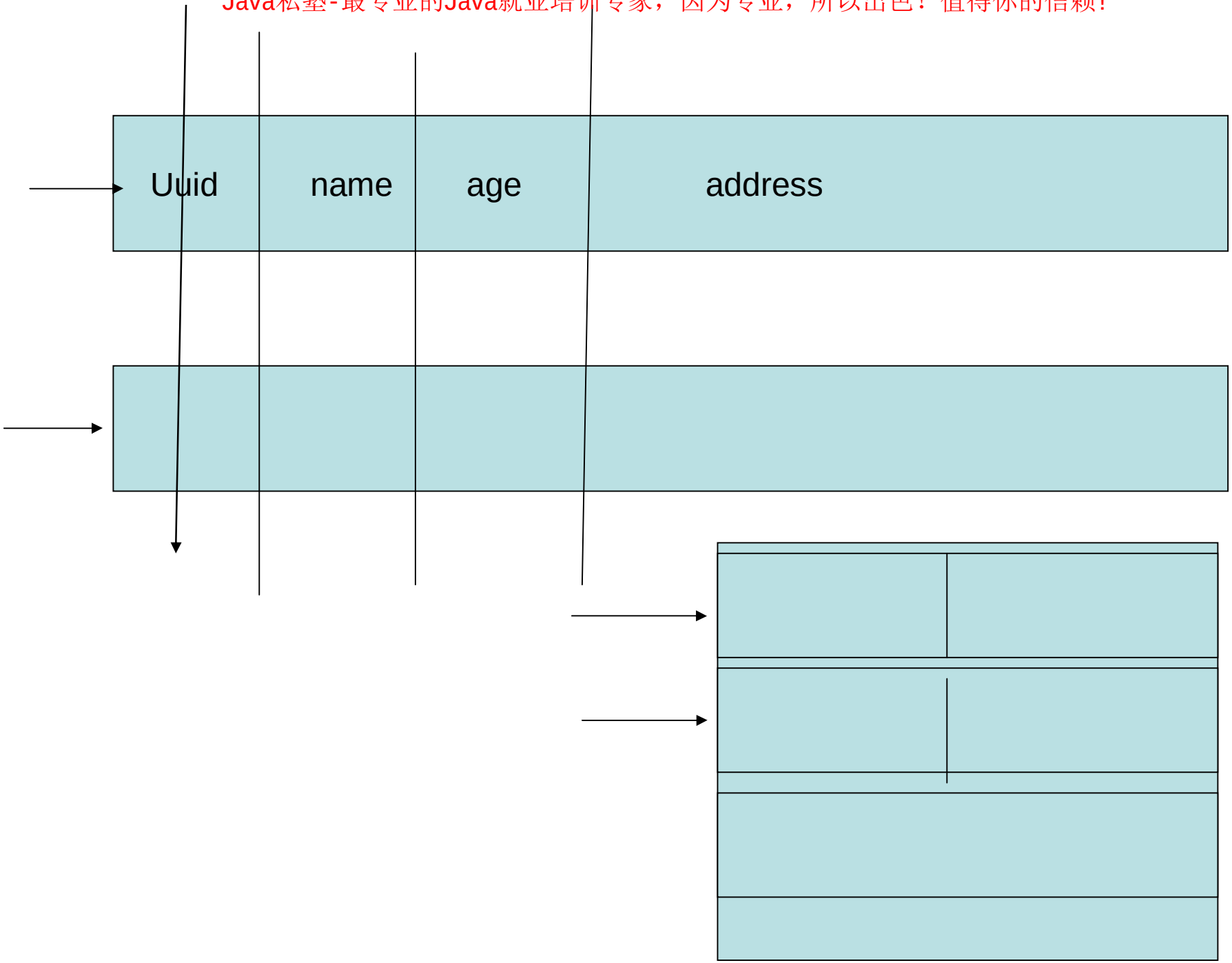
1：示例集合的投影

```
//1.首先准备测试数据
Collection<UserModel> collection = new ArrayList<UserModel>();
UserModel um1 = new UserModel();
um1.setUuid("u1");
um1.setName("u1Name");
collection.add(um1);
UserModel um2 = new UserModel();
um2.setUuid("u2");
um2.setName("u2Name");
collection.add(um2);
//2.测试集合或数组
EvaluationContext context1 = new StandardEvaluationContext();
ExpressionParser parser = new SpelExpressionParser();
context1.setVariable("collection", collection);
Collection<String> result1 =
parser.parseExpression("#collection.![#this.name]").getValue(context1, Collection.class);
```

SpEL的集合相关的表达式-7

SpEL投影运算还支持Map投影，但Map投影最终只能得到List结果，对于投影表达式中的“#this”将是Map.Entry，所以可以使用“value”来获取值，使用“key”来获取键。 示例如下：

```
//1.首先准备测试数据
UserModel um1 = new UserModel();
um1.setUuid("u1");
um1.setName("u1Name");
UserModel um2 = new UserModel();
um2.setUuid("u2");
um2.setName("u2Name");
Map<String, UserModel> map = new HashMap<String, UserModel>();
map.put(um1.getUuid(),um1);
map.put(um2.getUuid(),um2);
//2.测试Map投影
EvaluationContext context1 = new StandardEvaluationContext();
ExpressionParser parser = new SpelExpressionParser();
context1.setVariable("map", map);
Collection<String> result1 =
parser.parseExpression("#map.![#this.value.name]").getValue(context1, Collection.class);
```



SpEL的集合相关的表达式-8

- 集合筛选

在SpEL指根据原集合通过条件表达式选择出满足条件的元素并构造为新的集合，SpEL使用“(list|map).?[选择表达式]”，其中选择表达式结果必须是boolean类型，如果true则选择的元素将添加到新集合中，false将不添加到新集合中。示例如下：

```
//1：准备测试数据的过程跟上一个示例一样，就不重复了
//2.测试集合或数组的筛选
EvaluationContext context1 = new StandardEvaluationContext();
ExpressionParser parser = new SpelExpressionParser();
context1.setVariable("collection", collection);
Collection<String> result1 =
parser.parseExpression("#collection.?[#this.uuid.equals('u1')]").getValue(context1,
    Collection.class);

//测试Map筛选
EvaluationContext context1 = new StandardEvaluationContext();
ExpressionParser parser = new SpelExpressionParser();
context1.setVariable("map", map);
Map<String,UserModel> result1 =
parser.parseExpression("#map.?[#this.key=='u1']").getValue(context1, Map.class);
```

在Bean定义中使用SpEL-1

- Xml风格的配置

  SpEL支持在Bean定义时使用，默认使用“#{SpEL表达式}”表示，不允许嵌套。其中“#root”根对象默认可以认为是ApplicationContext，获取根对象属性其实是获取容器中的Bean。

- Xml风格的配置示例----通过SpEL表达式设置值

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
<property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>
</bean>
```

- Xml风格的配置示例----通过SpEL表达式参照其他的Bean

```
<bean id="t1" class="cn.javass.spring3.hello.T2">
    <property name="value" value="#{ T(java.lang.Math).random() * 100.0 }"/>
</bean>
<bean id="t2" class="cn.javass.spring3.hello.T2">
    <property name="value" value="#{ T(Double).parseDouble(t1.value) -
   1 }"></property>
</bean>
```

上面的t1就会被解析成为参照t1这个Bean，当然也可以使用@t1来表示。

在Bean定义中使用SpEL-2

- 注解风格的配置

使用@Value注解来指定SpEL表达式，该注解可以放到字段、方法及方法参数上。 但是要在配置文件中使用<context:annotation-config/> 来开启对注解的支持。示例如下：

```
public class SpELBean {
    @Value("#{ T(java.lang.Math).random() * 100.0 }")
    private String value;
    //setter和getter由于篇幅省略，自己写上
}
```

注意：如果同时使用了@Value和setter注入，那么setter注入将覆盖@Value的值。

本节课程小结

- 掌握Spring3的表达式语言

- 作业：

1：练习使用Spring3的表达式语言

2：使用Spring3的表达式语言，在一个自定义业务流程中，实现动态的条件判断。

整个课程小结

- 梳理已经学习到的Spring3的知识

1：Spring3的知识太多，这里只是抽取和讲述了它最核心、最重要、实际开发中必用的一些知识。

2：更多的知识，在今后会按照单项知识来讲述，比如：Spring的MVC，Spring的Security，Spring整合JavaEE的一些技术（如：EJB、JMS等）等等。

3：而关于零配置、插件等提高开发效率的知识，统一放到大项目之前，把所有涉及到的零配置放到一起来讲（比如：Struts2、Spring3、Hibernate4等），然后构建一个全部零配置的开发环境，插件也是一样，统一讲述和集成。