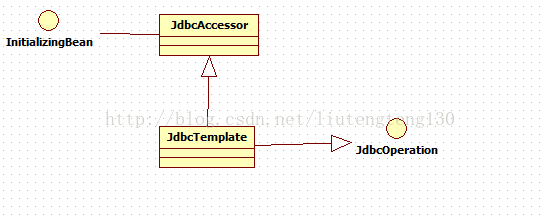
Spring封装好的模板，封装了数据库存取的基本过程，方便用户。

## 1、模板方法

Spring JdbcTemplate就是一个模板，它实现了设计模式中的模板模式。



JdbcTemplate继承了基类JdbcAccessor和接口类JdbcOperation。

* 在基类JdbcAccessor的设计中，对DataSource数据源进行管理和配置。
* 在JdbcTemplate接口中，定义了通过Jdbc操作数据库的基本方法，而JdbcTemplate提供这些接口方法的实现，比如execute方法、query方法、update方法等。

## 2、使用JdbcTemplate

//创建Jdbc模板对象，并传入数据源

JdbcTemplate temp = new JdbcTemplate(datasource);

class ExecuteStatementCallback implements StatementCallback<object>,Sqlprovider{

public Object doInStatement(Statement stmt) throws SQLException {

//spring封装数据库操作

stmt.execute();

return null;

}

public String getSql(){

return sql;

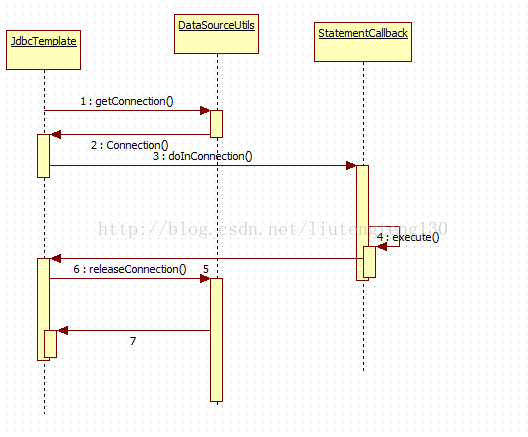
}

}

temp.sexecute(new ExecuteStatemnetCallback());

## 3、JdbcTemplate实现之Execute

以Execute为例：



Execute方法封装了对数据库的操作：

* 首先取得数据库连接Connection；
* 根据应用对数据库操作的需要创建数据库的Statement，对数据库操作进行回调；
* 处理数据库异常；
* 最后把数据库关闭。

**public void** execute(**final** String sql) **throws** DataAccessException {

**if** (**logger**.isDebugEnabled()) {  
 **logger**.debug(**"Executing SQL statement ["** + sql + **"]"**);  
 }  
  
 */\*\*  
 \* Callback to execute the statement.  
 \*/* **class** ExecuteStatementCallback **implements** StatementCallback<Object>, SqlProvider {  
 @Override  
 @Nullable  
 **public** Object doInStatement(Statement stmt) **throws** SQLException {  
 stmt.execute(sql);  
 **return null**;  
 }  
 @Override  
 **public** String getSql() {  
 **return** sql;  
 }  
 }  
  
 execute(**new** ExecuteStatementCallback());  
}

*// 使用java.sql.Statement处理静态SQL语句*

**public** <T> T execute(StatementCallback<T> action) **throws** DataAccessException {  
 Assert.*notNull*(action, **"Callback object must not be null"**);  
 *// 这里取得数据库的Connection*  
 Connection con = DataSourceUtils.*getConnection*(obtainDataSource());  
 Statement stmt = **null**;  
 **try** {

*//创建Statement*  
 stmt = con.createStatement();  
 applyStatementSettings(stmt);  
 *// 这里调用回调函数*

T result = action.doInStatement(stmt);  
 handleWarnings(stmt);  
 **return** result;  
 }  
 **catch** (SQLException ex) {  
 *// 如果捕捉到异常，把数据库连接释放掉，抛出一个经过String转换过的Spring数据库异常* String sql = *getSql*(action);  
 JdbcUtils.*closeStatement*(stmt);  
 stmt = **null**;  
 DataSourceUtils.*releaseConnection*(con, getDataSource());  
 con = **null**;  
 **throw** translateException(**"StatementCallback"**, sql, ex);  
 }  
 **finally** {

*// 释放数据库连接*  
 JdbcUtils.*closeStatement*(stmt);  
 DataSourceUtils.*releaseConnection*(con, getDataSource());  
 }  
}

## 4、使用Spring JdbcTemplate实现动态建表

private static ApplicationContext context = null;

//通过测试类测试

public static void main(String[] args) {

context = new ClassPathXmlApplicationContext("applicationContext.xml");

Users user = new Users();

user.setUserName("liutengteng");

user.setUserPass("liutengteng");

int re = insertObject("users",user);

System.out.println("================" + re + "====================");

}

/\*\*

\* 创建表，添加记录

\* @param tableName

\* @param obj

\* @return

\*/

public static int insertObject(String tableName,Object obj){

int re = 0;

try {

JdbcTemplate jt = (JdbcTemplate)context.getBean("jdbcTemplate");

SimpleDateFormat format = new SimpleDateFormat("yyyy\_MM");

String tname = tableName + "\_" + format.format(new Date());

// 判断数据库是否已经存在这个名称的表，如果有某表，则保存数据；否则动态创建表之后再保存数据

if(getAllTableName(jt,tname)){

re = saveObj(jt,tname,obj);

}else{

re = createTable(jt,tname,obj);

re = saveObj(jt,tname,obj);

}

} catch (Exception e) {

e.printStackTrace();

}

return re;

}

/\*\*

\* 根据表名称创建一张表

\* @param tableName

\*/

public static int createTable(JdbcTemplate jt,String tableName,Object obj){

StringBuffer sb = new StringBuffer("");

sb.append("CREATE TABLE `" + tableName + "` (");

sb.append(" `id` int(11) NOT NULL AUTO\_INCREMENT,");

Map<String,String> map = ObjectUtil.getProperty(obj);

Set<String> set = map.keySet();

for(String key : set){

sb.append("`" + key + "` varchar(255) DEFAULT '',");

}

sb.append(" `tableName` varchar(255) DEFAULT '',");

sb.append(" PRIMARY KEY (`id`)");

sb.append(") ENGINE=InnoDB DEFAULT CHARSET=utf8;");

try {

jt.update(sb.toString());

return 1;

} catch (Exception e) {

e.printStackTrace();

}

return 0;

}

/\*\*

\* 拼接语句，往表里面插入数据

\*/

public static int saveObj(JdbcTemplate jt,String tableName,Object obj){

int re = 0;

try{

String sql = " insert into " + tableName + " (";

Map<String,String> map = ObjectUtil.getProperty(obj);

Set<String> set = map.keySet();

for(String key : set){

sql += (key + ",");

}

sql += " tableName ) ";

sql += " values ( ";

for(String key : set){

sql += ("'" + map.get(key) + "',");

}

sql += ("'" + tableName + "' ) ");

re = jt.update(sql);

} catch (Exception e) {

e.printStackTrace();

}

return re;

}

/\*\*

\* 查询数据库是否有某表

\* @param cnn

\* @param tableName

\* @return

\* @throws Exception

\*/

@SuppressWarnings("unchecked")

public static boolean getAllTableName(JdbcTemplate jt,String tableName) throws Exception {

Connection conn = jt.getDataSource().getConnection();

ResultSet tabs = null;

try {

DatabaseMetaData dbMetaData = conn.getMetaData();

String[] types = { "TABLE" };

tabs = dbMetaData.getTables(null, null, tableName, types);

if (tabs.next()) {

return true;

}

} catch (Exception e) {

e.printStackTrace();

}finally{

tabs.close();

conn.close();

}

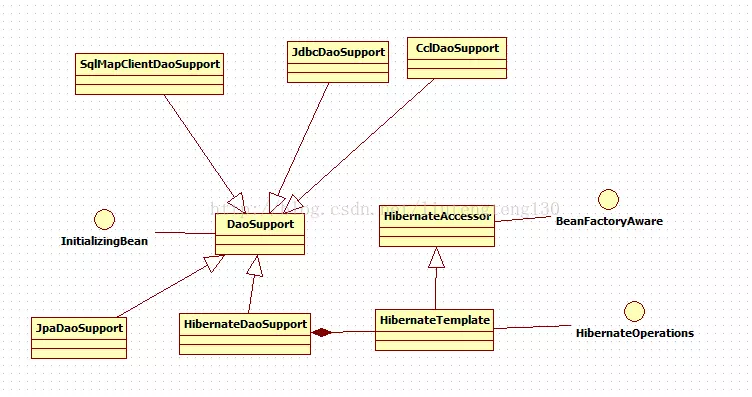
return false;

}

## 5、ORM设计与实现

通常情况下，ORM用的最多的是Hibernate。使用它，除了需要处理像Session、SessionFactory这些Hibernate类之外，还需要处理诸如事务处理，打开Session和关闭Session这样的问题，在某种程度上增加了使用Hibernate的难度。而Spring提供的Hibernate封装，如HibernateDaoSupport、HibernateTemplate等，简化了这些通用的过程。

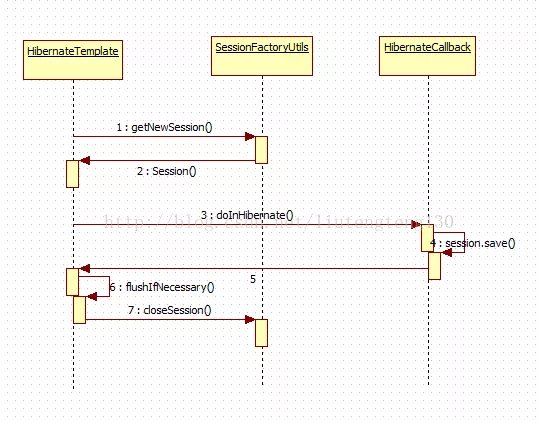
Spring的ORM包提供了对许多ORM产品的支持。通常使用Spring提供的Template类。在这些模板类里，封装了主要的数据操作的方法，比如query、update等，并且在Template封装中，已经包含了Hibernate中Session的处理，Connection的处理、事务的处理等。通过封装将Hibernate的持久化数据操作纳入到Spring统一的事务处理框架中，这部分是通过Spring的AOP来实现的。



DaoSupport是一个核心类，通过HibernateTemplate支持对Hibernate的操作。

Spring的ORM模块并不是重新开发的，通过IOC容器和AOP模块对Hibernate的使用进行封装。使用Hibernate，需要对Hibernate进行配置，这些配置通过SessionFactory来完成，在Spring的Hibernate模块中，提供了LocalSessionFactoryBean来封装SessionFactory的配置，通过这个LocalSessionFactory封装，可以将SessionFactory的配置信息通过Bean定义，注入到IOC容器中实例化好的SessionFactory单例对象中。这个LocalSessionFactoryBean设计为HIbernate的使用奠定了基础。

以hibernateTemplate为例与JdbcTemplate的使用类似，Spring使用相同的模式，通过execute回调来完成。



public <T> T execute(HibernateCallback<T> action) throws DataAccessException{

return doExecute(action,false,false);

}

protected <T> T doExecute(HIbernateCallback<T> action,boolean enforceNewSession,boolean enforceNativeSession) throws DataAccessException{

Assert.notNull(action,"Callback object must not be null");

//这里是取得HIbernate的Session，判断是否强制需要新的Session，

//如果需要，则直接通过SessionFactory打开一个新的session，否则需要结合配置和当前的Transaction的情况来使用Session

Session session = (enforceNewSession ? SessionFactoryUtils.getNewSession(getSessionFactory(),getEntityInterceptor()):getSession());

//判断Transaction是否已经存在，如果是，则使用的就是当前的Transaction的session

boolean existingTransaction = (!enforceNewSession &&

(!isAllowCreate()||SessionFactoryUtils.isSessionTransactional(session, getsessionFactory())));

if(existingTransaction){

logger.debug("Found thread-bound Session for HIbernateTemplate");

}

FlushMode previousFlushMode = null;

try {

previousFlushMode = applyFlushMOde(session,existingTransaction);

enableFilters(session);

Session sessionToExpose = (enforceNativeSession || isExposeNativeSession() ? session : createSessionProxy(session));

//这里是对HIbernateCallback中回调函数的调用，Session作为参数可以由回调函数使用

T result = action.doInHibernate(sessionToExpose);

flushIfNecessary(session,existingTransaction);

return result;

} catch (HibernateException ex) {

throw convertHibernateAccessException(ex);

}catch(SQLException ex){

throw convertJdbcAccessException(ex);

}catch(RuntimeException ex){

throw ex;

//如果存在Transaction，当前回调完成使用完session后，不关闭这个session

}finally{

if(existingTransaction){

logger.debug("Not closing pre-bound Hibernate Session after HibernateTemplate");

disableFilters(session);

if(previousFlushMode != null){

session.setFlushMode(previousFlushMode);

}

}

//如果不存在Transaction，那么关闭当前Session

else{

if(isAlwaysUseNewSession()){

SessionFactoryUtils.closeSession(session);

}else{

SessionFactoryUtils.closeSessionOrRegisterDeferredClose(session,getSessionFactory());

}

}

}

}

Spring封装了事务处理，以及通过HibernateTemplate封装了Session，不直接对Session进行操作。