++++++++++++++++++++++++++++++

+Chapter 08 Database 연동 지원과 JDBC 지원+

++++++++++++++++++++++++++++++

: 거의 모든 WEB Application은 Database 연동을 필요로 한다. JDBC API를 이용해서 Database에 접근할 수 있으며, iBatis나 Hibernate, JPA와 같은 ORM

Framework를 이용해서 Database를 연동할 수도 있다. SPRING은 JDBC를 비롯하여 ORM Framework를 직접적으로 지원하고 있기 때문에 약간의 노력만 들이면

JDBC뿐만 아니라 다른 ORM Framework를 SPRING과 간단하게 연동할 수 있다.

본 장에서는 SPRING이 어떻게 Database 연동을 지원하는지 살펴볼 것이다. 그리고, DataSource의 설정 방법과 JdbcTemplate과 같은 Class를 이용해서

JDBC 기반의 Database 연동 구현 방법을 살펴보도록 하자.

===================================================================================================

1. SPRING의 Database 연동 지원

: SPRING은 JDBC, Hibernate, iBatis 등의 다양한 기술을 이용해서 손쉽게 DAO Class를 구현할 수 있도록 지원하고 있는데, 지원하는 내용은 다음과 같다.

- 템플릿 Class를 통한 데이터 접근 지원

- 의미 있는 예외 Class 제공

- Transaction 처리

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

1.1 Database 연동을 위한 템플릿 Class

: 데이터에 접근하는 Code는 거의 동일한 Code 구성을 갖는다. 예를 들어, JDBC를 사용할 경우 특정 테이블에서 데이터를 로딩하는 Code는 다음과 같은 형식을

취하게 된다.

Connection conn = null;

PreparedStatement pstmt = null;

ResultSet rs = null;

try{

conn = getConnection();

pstmt = conn.prepareStatement("select \* from message where guestBookId = ?");

pstmt.setInt(1,guestBookId);

rs = pstmt.executeQuery();

if(rs.next()){

do{

Message message = new Message();

message.setContent(rs.getString("content"));

...

}while(rs.next());

}

}catch(SQLException ex){

//알맞은 예외처리

}finally{

if(rs != null) rs.close();

if(pstmt != null) pstmt.close();

if(conn != null) conn.close();

}

위 Code에서 Connection을 생성하고, PreparedStatement, ResultSet, 그리고 Connection 등의 Resource을 반환하는 Code는 거의 모든 JDBC Code에서 중복되는 Code이다.

Connection conn = null;

...

try{

conn = getConnection();

...

}finally{

...

if(conn != null) conn.close();

}

뿐만 아니라, PreparedStatement를 구하고 ResultSet으로 부터 데이터를 읽어와 Java 빈 객체에 저장하는 Code 역시 동일한 형식을 취한다.

SPRING은 Database 연동을 위한 템플릿 Class를 제공함으로써, 개발자가 중복된 Code를 입력해야하는 성가신 작업을 줄일 수 있도록 돕고 있다.

JDBC뿐만 아니라 iBatis, JMS와 같은 다양한 기술에 대해 템플릿 Class를 제공하고 있다.

예를 들어, JDBC의 경우 JdbcTemplate Class를 제공하고 있으며, JdbcTemplate Class를 사용함으로써 다음과 같은 try~catch~finally 블록 및 커넥션 관리를

위한 중복되는 Code를 줄이거나 없앨 수 있다.

JdbcTemplate jdbcTemplate = getJdbcTemplate();

List<Message> list = jdbcTemplate.query("select \* from GUESTBOOK\_MESSAGE order by GUESTBOOK\_MESSAGE\_ID desc limit ?, ?",

new Object[]{startRow, pageSize}, new RowMapper(){

@Override

public Object mapRow(ResultSet rs, int rowNum) throws SQLException{

Message message = new Message();

message.setId(rs.getInt("GUESTBOOK\_MESSAGE\_ID"));

...

return message;

}

});

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

1.2 SPRING의 예외 지원

: JDBC 프로그래밍을 할 때 아쉬운 부분 분의 하나는 Database 처리 과정에서 발생하는 에러는 항상 SQLException 이라는 점이다. 예를 들어, 아래 Code의 경우

SQLException을 catch하는 시점에서는 Connection을 구하는 과정에서 예외가 발생했는지, Statement를 생성하는 과정에서 예외가 발생했는지, 아니면 SQL 쿼리를

실행하는 과정에서 예외가 발생했는지 확인할 수 없다.

try{

conn = getConnection();

stmt = conn.createStatement("select \* from GUESTBOOK\_MESSAGE");

...

}catch(SQLException ex){

// 왜 예외가 발생한 걸까?

}

왜 예외가 발생했는지 확인하려면 SQLException의 실제 타입이 뭔지 확인해야 하고, 에러 Code를 확인해야 한다. 하지만, 예외 발생 원인을 찾기 위한 Code를 작성하는

일은 꽤 성가신 일이다.

SPRING은 Database 처리과정에서 발생한 예외가 왜 발생했는지를 좀 더 구체적으로 확인할 수 있도록 하기 위해, Database 처리와 관련된 예외 Class를

제공하고 있다. 예를 들어, OptimisticLockingFailureException이나 DataRetrievalFailureException과 같이 조금 더 구체적으로 실패 원인을 설명해 주는 예외 Class를

제공하고 있다.

JdbcTemplate Class는 처리과정에서 SQLException이 발생하면 SPRING이 제공하는 예외 Class 중 알맞은 예외 Class로 변환해서 발생시킨다. 예를 들어, 아래 Code와

같이 올바르지 않은 SQL 쿼리를 실행하는 경우 JdbcTemplate은 BadSqlGrammerException 예외를 발생시킨다.

JdbcTemplate jdbcTemplate = getJdbcTemplate();

List<Message> list = jdbcTemplate.query("select \* from GUESTBOOK\_MESSAGE order by GUESTBOOK\_MESSAGE\_ID desc limit ?, ?", //잘못된 SQL 입력시

...

);

SPRING이 제공하는 Database 관련 예외 Class들은 모두 DataAccessException Class를 상속받고 있는데, DataAccessException은 RuntimeException이다.

따라서, 필요한 경우에만 try~catch 블럭을 이용해서 예외를 처리하면 된다.

JdbcTemplate 뿐만 아니라 SqlMapClientTemplate과 같이 SPRING이 제공하는 템플릿 Class는 내부적으로 발생하는 예외 Class를 SPRING이 제공하는 예외 Class로

알맞게 변환해서 예외를 발생시킨다. 따라서, SPRING이 제공하는 템플릿 Class를 사용하면 Database 연동을 위해 사용하는 기술에 상관 없이 동일한 방식으로

예외처리를 할 수 있게 된다.

===================================================================================================

2. DataSource 설정

: DataSupport Class나 템플릿 Class 그리고 Hibernate나 iBatis와 같은 Framework와의 연동을 위해 제공되는 Class를 사용할 경우 SPRING은 DataSource를 통해서

Connection을 제공할 수 있다.

- 커넥션 풀을 이용한 DataSource 설정

- JNDI를 이용한 DataSource 설정

- DriverManager를 이용한 DataSource 설정

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.1 커넥션 풀을 이용한 DataSource 설정

: SPRING이 직접적으로 커넥션 풀 구현 Class를 제공하진 않지만, DBCP(Jakarta Commons Database Connection Pool) API와 같이 커넥션 풀 Library를 이용해서

커넥션 풀 기반의 DataSource를 설정할 수 있다. 아래 Code는 DBCP가 제공하는 BasicDataSource Class를 이용해서 DataSource를 설정하는 예이다.

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"

p:driverClassName="oracle.jdbc.driver.OracleDriver"

p:url="jdbc:oracle:thin:@192.168.107.100:1521:ORCL"

p:username="spring"

p:password="spring"/>

BasicDataSource Class는 커넥션 풀을 관리하기 위한 다양한 프로퍼티를 제공하는데, 이들 프로퍼티는 아래와 같다.

프로퍼티 설명

=========================================================================

initialSize 초기에 풀에 생성되는 커넥션의 개수

=========================================================================

maxActive 커넥션 풀이 제공할 최대 커넥션 개수

=========================================================================

maxIdl 사용되지 않고 풀에 저장될 수 있는 최대 커넥션 개수. 음수일 경우 제한이 없다.

=========================================================================

minIdl 사용되지 않고 풀에 저장될 수 있는 최소 커넥션 개수

=========================================================================

maxWait 풀에 커넥션이 존재하지 않을 때, 커넥션이 다시 풀에 리턴될때 까지 대기하는 시간.

단위는 1/1000초이며, -1일 경우 무한히 대기한다.

=========================================================================

minEvictableIdleTimeMillis 사용되지 않는 커넥션을 추출할 때 이 속성에서 지정한 시간 이상 비활성화 상태인

커넥션만 추출한다. 양수가 아닌 경우 비활성화된 시간으로는 풀에서 제거되지

않는다. 시간단위는 1/1000초이다.

=========================================================================

timeBetweenEvictionRunsMillis 사용되지 않은 커넥션을 추출하는 쓰레드의 실행주기를 지정한다.

양수가 아닐 경우 실행되지 않는다. 단위는 1/1000초이다.

=========================================================================

numTestsPerEvictionRun 사용되지 않는 커넥션을 몇 개 검사할지 지정한다.

=========================================================================

testOnBorrow true일 경우 커넥션 풀에서 커넥션을 가져올 때 커넥션이 유효한지의 여부를 검사한다.

=========================================================================

testOnReturn true일 경우 커넥션 풀에서 커넥션을 반환할 때 커넥션이 유효한지의 여부를 검사한다.

=========================================================================

testWhileIdle true일 경우 비활성화 커넥션을 추출할 때 커넥션이 유효한지의 여부를 검사해서

유효하지 않은 커넥션은 풀에서 제거한다.

=========================================================================

DBCP와 함께 널리 사용되는 커넥션 풀 API인 c3p0 Library를 사용해서 DataSource를 설정할 수도 있다. 아래 Code는 c3p0가 제공하는

ComboPooledDataSource를 이용해서 DataSource를 설정하는 예제 Code이다.

<bean id="dataSource" class="com.mchange.v2.c3p0.comboPooledDataSource" destroy-method="close"

p:driverClassName="oracle.jdbc.driver.OracleDriver"

p:url="jdbc:oracle:thin:@localhost:1521:ORCL"

p:username="nals"

p:password="dkagh1234."/>

ComboPooledDataSource Class는 커넥션 풀을 관리하기 위한 다양한 프로퍼티를 제공하는데, 이들 프로퍼티는 다음과 같다.

프로퍼티 설 명

========================================================================================

aquireIncrement 풀에 커넥션이 없을 때 증가시킬 커넥션의 개수. 기본값은 3이다.

========================================================================================

initialPoolSize 초기의 커넥션 풀의 크기, 기본값은 3이다.

========================================================================================

maxPoolSize 커넥션 풀의 최대 크기. 기본값은 15이다.

========================================================================================

minPoolSize 커넥션 풀의 최소 크기. 기본값은 3이다.

========================================================================================

maxConnectionAge 커넥션의 유효 시간. 단위는 초. 지정한 시간이 지나면 자동으로 풀에서 제거된다. 값이 0일 경우

제거하지 않는다. 기본값은 0이다.

========================================================================================

maxIdleTime 지정한 시간동안 사용되지 않는 커넥션을 제거한다. 단위는 초이다. 값이 0일 경우

제거하지 않는다. 기본값은 0이다.

========================================================================================

checkoutTimeout 풀에서 커넥션을 가져올 때 대기 시간. 단위는 1/1000초. 0은 무한히 기다리는 것을 의미한다.

지정한 시간동안 풀에서 커넥션을 가져오지 못할 경우 SQLException을 발생시킨다. 기본값은 0이다.

========================================================================================

automaticTestTable 값이 존재할 경우 지정한 이름의 테이블을 생성한 뒤, 해당 테이블을 이용해서 커넥션이 유효한 지의

여부를 검사한다. 기본값은 null이다. 이 값을 제공하면 preferredTestQuery는 무시된다.

========================================================================================

idleConnectionTestPeriod 풀 속에 있는 커넥션의 Test 주기, 단위는 초이며, 0인 경우 검사하지 않는다. 기본값은 0이다.

========================================================================================

preferredTestQuery 커넥션을 Test 할 때 사용할 쿼리. 기본값은 null이다.

========================================================================================

testConnectionOnCheckIn true인 경우 커넥션을 풀에 반환할 때 커넥션이 유효한지의 여부를 비동기로 검사한다. 기본값은 false이다.

========================================================================================

testConnectionOnCheckOut true인 경우 커넥션을 풀에서 가져올 때 유효한지의 여부를 겁사한다. 기본값은 false이다. 추가적인 검사로 인한

성능 저하가 발생할 수 있기 때문에, 이 값을 true로 하기 보다는 idleConnectionTestPeriod를 이용해서

주기적으로 검사하는 방법이 더 나은 선택이다.

========================================================================================

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.2 JNDI를 이용한 DataSource 설정

: Weblogic이나 JBoss와 같은 JEE Application 서버를 이용할 경우, JNDI를 이용해서 DataSource를 구하곤 한다. 심지어 톰캣이나 Resin등의

WEB Container를 사용하는 경우에도 JNDI로부터 DataSource를 구하도록 설정이 가능하다.

JNDI로부터 DataSource를 가져오고 싶다면, 다음과 같이 <jee:jndi-lookup> 태그를 이용해서 JNDI에 등록된 객체의 이름을 명시하면 된다.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:jee="http://www.springframework.org/schema/jee"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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">

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/guestbook" resource-ref="true"/>

<bean id="messageDao" class="kame.spring.guestbook.dao.jdbc.JdbcMessageDao" p:dataSource-ref="dataSource"/>

</beans>

<jee:jndi-lookup> 태그는 SPRING 2.0버전부터 지원하며, 이 태그를 사용하기 위해서는 jee 네임스페이스 및 관련 XML 스키마로 등록해 주어야 한다.

<jee:jndi-lookup> 태그의 jndi-name 속성은 JNDI에서 객체를 검색할 때 사용할 이름을 입력한다. resource-ref 속성의 값이 true일 경우 검색할 이름

앞에 "java:comp/env"가 붙는다. 따라서, 위 설정은 "java:comp/env/jdbc/guestbook"을 사용해서 JNDI에서 객체를 검색하게 된다.

<jee:jndi-lookup> 태그를 사용하지 않고 다음과 같이 JndiObjectFactoryBean Class를 이용해서 JNDI로 부터 DataSource를 구하도록 설정할 수 있다.

<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName" value="jdbc/guestbook"/>

<property name="resourceRef" value="true"/>

</bean>

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.3 DriverManager를 이용한 DataSource 설정

: 커넥션 풀이나 JNDI를 사용할 수 없는 경우에는 DriverManager를 이용해서 커넥션을 제공하는 DriverManagerDataSource Class를 사용할 수 있다.

설정 방법은 다음과 같다.

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"

p:driverClassName="oracle.jdbc.driver.OracleDriver"

p:url="jdbc:oracle:thin:@localhost:1521:ORCL"

p:username="nals"

p:password="dkagh1234." />

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.4 DataSource로부터 Connection 구하기

: SPRING은 JDBC를 위한 템플릿 Class와 DAO 지원 Class를 제공하고 있기 때문에, SPRING을 사용하면서 직접적으로 Connection을 이용해서 JDBC

프로그래밍을 해야하는 경우가 많지는 않다. 하지만, 외부에서 제공한 Module이 Connection을 파라미터로 전달받는다면, 어쩔 수 없이 DataSource로부터

Connection을 구해서 해당 Module에 제공해야 할 것이다.

DataSource로부터 Connection을 직접 구해야 하는 경우, 다음과 같이 앞서 설정한 DataSource의 getConnection() 메서드를 사용해서 Connection을

구할 수 있을 것이다.

public class JdbcMessageDao{

private DataSource dataSource;

public void setDataSource(DataSource dataSource){

this.dataSource=dataSource;

}

@Override

public int selectCount(){

Connection conn = null;

...

try{

conn = dataSource.getConnection();

...

}finally{

JdbcUtils.closeConnection(conn);

}

}

}

하지만, DataSource에서 직접 Connection을 구하게 되면, SPRING이 제공하는 Transaction 관리 기능을 완전히 활용할 수 없게 된다. 만약, SPRING이 제공하는

Transaction 기능을 올바르게 적용하고 싶다면 SPRING이 제공하는 DataSourceUtils Class를 이용해서 Connection을 구하고 반환해야 한다.

try{

conn = DataSourceUtils.getConnection(dataSource);

...

}finally{

DataSourceUtils.releaseConnection(conn,dataSource);

}

===================================================================================================

3. SPRING JDBC 지원

: JDBC를 이용해서 프로그래밍을 할때 성가신 작업중의 하나는 늘 동일한 형태의 try~catch~finally 블록을 사용해야 한다는 점이다. 예를 들어, 데이터를 삽입하는

INSERT 쿼리를 실행하는 Code를 보자.

private static final String INSERT\_SQL = "insert into GUESTBOOK\_MESSAGE(GUEST\_NAME,CONTENT) values(?,?);

public void insert(Message message){

Connection conn = null;

PreparedStatement pstmt = null;

try{

conn = DataSourceUtils.getConnection(dataSource);

pstmt = conn.prepareStatement(INSERT\_SQL);

pstmt.setString(1, message.getGuestName());

pstmt.setString(2, message.getContent());

pstmt.executeUpdate();

}catch(SQLException e){

throw exceptionTranslator.translate("insert", INSERT\_SQL, e);

}finally{

JdbcUtils.closeStatement(pstmt);

DataSourceUtils.releaseConnection(conn, dataSource);

}

}

메서드의 절반 이상이 Connection을 구하고 예외 처리를 하고, Resource을 반환하는 Code이다. 실제로 INSERT 자체와 관련된 Code는 단 네줄에 불과하다.

이번에는 테이블에 저장된 행의 모든 개수를 구하는 SELECT 쿼리를 실행하는 Code를 보자.

private String SELECT\_COUNT\_SQL = "select count(\*) from GUESTBOOK\_MESSAGE";

public int selectCount(){

Connection conn = null;

Statement stmt = null;

ResultSet rs = null;

try{

conn = dataSource.getConnection();

stmt = conn.createStatement();

rs = stmt.executeQuery(SELECT\_COUNT\_SQL);

int count = 0;

if(rs.next())

count = rs.getInt(1);

return count;

}catch(SQLExcetion e){

throw exceptionTranslator.translate("selectCount", SELECT\_COUNT\_SQL, e);

}finally{

JdbcUtils.closeStatement(pstmt);

DataSourceUtils.releaseConnection(conn, dataSource);

}

}

앞서 INSERT의 경우와 마찬가지로 Connection과 관련된 Code와 예외처리, Resource을 반환하기 위한 finally 블록의 Code가 절반 이상을 차지하고 있다.

Connection을 구하고, try~catch~finally로 Resource을 관리하는 등의 중복된 Code를 매번 입력하는 것은 꽤 성가신 일이다. SPRING은 이런 중복되는

Code를 제거할 수 있도록 해 주는 템플릿 Class를 제공하고 있으며, JDBC 지원을 위해 다음과 같은 Class를 제공하고 있다.

- JdbcTemplate : 기본적인 JDBC 템플릿 Class로서 JDBC를 이용해서 데이터에 대한 접근을 제공한다.

- NamedParameterJdbcTemplate : PreparedStatement에서 인덱스 기반의 파라미터가 아닌 이름을 가진 파라미터를 사용할 수 있도록 지원하는 템플릿 Class.

- SimpleJdbcTemplate : Java5의 가변 인자를 이용해서 쿼리를 실행할 때 사용되는 데이터를 전달할 수 있는 템플릿 Class.

- SimpleJdbcInsert : 데이터 삽입을 위한 Interface를 제공해주는 Class.

- SimpleJdbcCall : 프로시저 호출을 위한 Interface를 제공해 주는 Class.

각 Class의 사용방법을 차례대로 살펴보도록 하자.

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------s

3.1 JdbcTemplate Class를 이용한 JDBC 프로그래밍

: JdbcTemplate Class는 SQL 실행을 위한 메서드를 제공하고 있다. 이들 메서드를 사용하면 데이터 조회, 삽입, 수정, 삭제를 위한 SQL 쿼리를 실행할 수 있다.

JdbcTemplate Class를 사용하려면 다음과 같이 JdbcTemplate 객체를 생성할 때 DataSource를 전달해주면 된다.

public class JdbcTemplateGuestMessageDao implements GuestMessageDao {

private JdbcTemplate jdbcTemplate;

public JdbcTemplateGuestMessageDao(DataSource dataSource) {

jdbcTemplate = new JdbcTemplate(dataSource);

}

...

}

위 Class에 대한 SPRING 설정 파일은 다음과 같을 것이다.

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"

p:driverClassName="oracle.jdbc.driver.OracleDriver"

p:url="jdbc:oracle:thin:@localhost:1521:ORCL"

p:username="nals"

p:password="dkagh1234." />

<bean id="guestMessageDao" class="madvirus.spring.chap08.dao.JdbcTemplateGuestMessageDao">

<constructor-arg>

<ref bean="dataSource" />

</constructor-arg>

</bean>

다음과 같이 JdbcTemplate Class를 전달받도록 구현할 수도 있다.

public class JdbcTemplateGuestMessageDao implements GuestMessageDao {

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate){

this.jdbcTemplate = jdbcTemplate;

}

...

}

DAO Class에서 JdbcTemplate을 프로퍼티나 생성자에서 전달받을 경우, SPRING 설정파일에서는 다음과 같이 설정해주면 된다.

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" .../>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate" p:dataSource-ref="dataSource"/>

<bean id="guestMessageDao" class="madvirus.spring.chap08.dao.JdbcTemplateGuestMessageDao" p:jdbcTemplate-ref="jdbcTemplate"/>

DAO 객체가 내부적으로 JdbcTemplate 객체를 생성하거나 JdbcTemplate 객체를 전달받았다면,

JdbcTemplate 객체를 이용해서 SQL을 실행할 수 있다.

[JdbcTemplateGuestMessageDao.java]

package madvirus.spring.chap08.dao;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import java.sql.Timestamp;

import java.sql.Types;

import java.util.List;

import javax.sql.DataSource;

import madvirus.spring.chap08.model.GuestMessage;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.PreparedStatementCreator;

import org.springframework.jdbc.support.GeneratedKeyHolder;

import org.springframework.jdbc.support.KeyHolder;

public class JdbcTemplateGuestMessageDao implements GuestMessageDao {

private JdbcTemplate jdbcTemplate;

public JdbcTemplateGuestMessageDao(DataSource dataSource) {

jdbcTemplate = new JdbcTemplate(dataSource);

}

@Override

public int count() {

return jdbcTemplate.queryForInt("select count(\*) from GUESTBOOK\_MESSAGE");

}

@Override

public int delete(int id) {

return jdbcTemplate.update(

"delete from GUESTBOOK\_MESSAGE where MESSAGE\_ID = ?", id);

}

@Override

public int insert(final GuestMessage message) {

KeyHolder keyHolder = new GeneratedKeyHolder();

int insertedCount = jdbcTemplate.update(new PreparedStatementCreator() {

@Override

public PreparedStatement createPreparedStatement(Connection con)

throws SQLException {

PreparedStatement pstmt = con

.prepareStatement(

"insert into GUESTBOOK\_MESSAGE (MESSAGE\_ID, GUEST\_NAME, MESSAGE, REGISTRY\_DATE) values (MESSAGE\_ID\_SEQ.NEXTVAL, ?, ?, ?)",

new String[] { "MESSAGE\_ID" });

pstmt.setString(1, message.getGuestName());

pstmt.setString(2, message.getMessage());

pstmt.setTimestamp(3, new Timestamp(message.getRegistryDate()

.getTime()));

return pstmt;

}

}, keyHolder);

Number keyNumber = keyHolder.getKey();

message.setId(keyNumber.intValue());

return insertedCount;

// int insertedCount = jdbcTemplate

// .update(

// "insert into GUESTBOOK\_MESSAGE (GUEST\_NAME, MESSAGE, REGISTRY\_DATE) values (?, ?, ?)",

// message.getGuestName(), message.getMessage(), message

// .getRegistryDate());

// if (insertedCount > 0) {

// int id = jdbcTemplate.queryForInt("select last\_insert\_id() ");

// message.setId(id);

// }

// return insertedCount;

}

@Override

public List<GuestMessage> select(int begin, int end) {

int startRowNum = begin - 1;

int count = end - begin + 1;

return jdbcTemplate

.query(

"select \* from GUESTBOOK\_MESSAGE order by MESSAGE\_ID desc limit ?, ?",

new Object[] { startRowNum, count },

new GuestMessageRowMapper());

}

@Override

public int update(GuestMessage message) {

return jdbcTemplate

.update(

"update GUESTBOOK\_MESSAGE set MESSAGE = ? where MESSAGE\_ID = ?",

new Object[] { message.getMessage(), message.getId() },

new int[] { Types.VARCHAR, Types.INTEGER });

}

}

(1) 조회를 위한 메서드 : query(), queryForList(), queryForObject(), queryForInt(), queryForLong()

: 쿼리 실행 결과를 객체 목록으로 가져올 때에는 RowMapper를 이용하는 query() 메서드를 이용하면 된다. JdbcTemplate Class는 다음과 같은 query()

메서드를 제공하고 있다.

- query(String sql, RowMapper<?> rowMapper)

- List<T> query(String sql, Object[] args, RowMapper<T> rowMapper)

- List<T> query(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)

위 Code에서 sql 파라미터는 실행할 쿼리를, RowMapper는 실행 결과를 Java 객체로 변환해주는 매퍼를, args 파라미터는 PreparedStatement를 실행할 때

사용할 파라미터 바인딩 값 목록을, argTypes는 파라미터 바인딩을 할 때 사용할 SQL 타입 목록을 의미한다. argTypes에 사용되는 값은 java.sql.Types

Class에 정의된 값을 사용한다.

RowMapper는 ResultSet에서 값을 가져와 원하는 타입으로 Mapping할 때 사용되며, 다음과 같이 정의되어 있다.

public interface RowMapper<T>{

T mapRow(ResultSet rs, int rowNum) throws SQLException;

}

RowMapper의 mapRow() 메서드는 ResultSet에서 읽어온 값을 이용해서 원하는 타입의 객체를 생성한 뒤 리턴한다.

rowNum은 행번호를 의미하며 0부터 시작한다.

아래 Code는 query() 메서드의 사용 예이다.

@Override

public List<GuestMessage> select(int begin, int end) {

//int startRowNum = begin - 1;

//int count = end - begin + 1;

return jdbcTemplate

.query(

"select MESSAGE\_ID, GUEST\_NAME, MESSAGE, REGISTRY\_DATE from ( "

+ " select rownum rnum, MESSAGE\_ID, GUEST\_NAME, MESSAGE, REGISTRY\_DATE from ( "

+ " select \* from GUESTBOOK\_MESSAGE order by MESSAGE\_ID desc "

+ " ) where rownum <= ? "

+ " ) where rnum >= ?" ,

new Object[] { end, begin },

new RowMapper<GuestMessage>(){

@Override

public GuestMessage mapRow(ResultSet rs, int rowNum) throws SQLException {

GuestMessage message = new GuestMessage();

message.setId(rs.getInt("MESSAGE\_ID"));

message.setGuestName(rs.getString("GUEST\_NAME"));

message.setMessage(rs.getString("MESSAGE"));

message.setRegistryDate(rs.getDate("REGISTRY\_DATE"));

return message;

}

});

}

query() 메서드에 RowMapper의 구현 객체를 전달할 때에는 위와 같이 임의 Class(Anonymous Class)를 주로 사용한다. 하지만, 여러 메서드에서 공통으로 사용되는

Code가 있다면, 다음과 같이 RowMapper 구현 Class를 별도로 구현해서 Code 중복을 제거할 수 있다.

public class GuestMessageRowMapper implements RowMapper<GuestMessage> {

@Override

public GuestMessage mapRow(ResultSet rs, int rowNum) throws SQLException {

GuestMessage message = new GuestMessage();

message.setId(rs.getInt("MESSAGE\_ID"));

message.setGuestName(rs.getString("GUEST\_NAME"));

message.setMessage(rs.getString("MESSAGE"));

message.setRegistryDate(rs.getDate("REGISTRY\_DATE"));

return message;

}

}

GuestMessageRowMapper Class를 개발하면 앞서 예로 든 select() 메서드를 다음과 같이 구현할 수 있을 것이다.

@Override

public List<GuestMessage> select(int begin, int end) {

//int startRowNum = begin - 1;

//int count = end - begin + 1;

return jdbcTemplate

.query(

"select MESSAGE\_ID, GUEST\_NAME, MESSAGE, REGISTRY\_DATE from ( "

+ " select rownum rnum, MESSAGE\_ID, GUEST\_NAME, MESSAGE, REGISTRY\_DATE from ( "

+ " select \* from GUESTBOOK\_MESSAGE order by MESSAGE\_ID desc "

+ " ) where rownum <= ? "

+ " ) where rnum >= ?" ,

new Object[] { end, begin },

new GuestMessageRowMapper());

}

쿼리 실행 결과로 읽어온 컬럼 개수가 한 개라면 다음의 queryForList() 메서드를 이용해서 데이터를 조회할 수 있다.

- List<T> queryForList(String sql, Class<T> elementType)

- List<T> queryForList(String sql, Object[] args, Class<T> elementType)

- List<T> queryForList(String sql, Object[] args, int[] argTypes, Class<T> elementTypes)

다음은 컬럼 개수가 한 개인 경우에 queryForList() 메서드를 사용하는 Code의 작성 예이다.

List<String> names = jdbcTemplate.queryForList(

"select name from MEMBER where id=?", new Object[]("madvirus"), String class);

쿼리 실행 결과 행의 개수가 한 개인 경우에는 queryForObject() 메서드를 사용해서 쿼리 실행 결과를 가져올 수 있다. 다음은 queryForObject() 메서드를

보여주고 있다.

- T queryForObject(String sql, RowMapper<T> rowMapper)

- T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)

- T queryForObject(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)

- T queryForObject(String sql, Class<T> requiredType)

- T queryForObject(String sql, Object[] args, Class<T> requiredType)

- T queryForObject(String sql, Object[] args, int[] argTypes, Class<T> requiredType)

queryForObject() 메서드에 전달되는 각각의 파라미터는 query() 메서드와 동일하며, 차이점이 있다면 List 대신 한 개의 객체를 리턴한다는 점이다. 단, 쿼리 실행

결과의 행 개수가 한 개가 아닌 경우에는, 즉 행의 개수가 0이거나 2개 이상인 경우에는, IncorrectResultSizeDataAccessException 예외를 발생시킨다.

(이 예외는 DataAccessException의 하위 Class이다.)

queryForObject() 메서드에서 쿼리 실행 결과가 한 개 이상의 컬럼을 조회하는 경우에는 RowMapper를 사용해서 데이터를 가져오면 되고, 한 개 컬럼만 조회하는

경우에는 Class를 인자로 받는 queryForObject() 메서드를 사용하면 된다. Class를 인자로 전달받는 경우, 조회하는 컬럼의 개수가 두 개 이상인 경우 예외를

발생시킨다. 아래 Code는 queryForObject() Code의 작성 예이다.

public int count(){

return jdbcTemplate.queryForObject("select count(\*) from GUESTBOOK\_MESSAGE", Integer.class);

}

JdbcTemplate은 Object가 아닌 int나 long 타입의 결과를 구할때 사용할 수 있는 메서드를 추가로 제공하고 있다. 이들 메서드는 다음과 같다.

- int queryForInt(String sql)

- int queryForInt(String sql, Object ... args)

- int queryForInt(String sql, Object[] args, int[] argTypes)

- long queryForLong(String sql)

- long queryForLong(String sql, Object ... args)

- long queryForLong(String sql, Object[] args, int[] argTypes)

이들 메서드는 queryForObject()와 같이 결과 행 개수가 한 개가 아닌 경우 예외를 발생시킨다. 아래 Code는 queryForInt() 메서드의 사용 예를 보여주고 있다.

public int count(){

return jdbcTemplate.queryForInt("select count(\*) from GUESTBOOK\_MESSAGE");

}

(2) 삽입/수정/삭제를 위한 메서드 : update()

: INSERT, UPDATE, DELETE 쿼리를 실행할 때에는 update() 메서드를 사용하면 된다. update() 메서드도 query() 메서드와 마찬가지로 인덱스 파라미터를 위한

값을 전달받는 메서드와 그렇지 않은 메서드로 구분된다.

- int update(String sql)

- int update(String sql, Object ... args)

- int update(String sql, Object[] args, int[] argTypes)

update() 메서드는 쿼리 실행 결과 변경된 행의 개수를 리턴한다. 아래 Code는 update() 메서드의 사용 예이다.

@Override

public int update(GuestMessage message) {

return jdbcTemplate

.update(

"update GUESTBOOK\_MESSAGE set MESSAGE = ? where MESSAGE\_ID = ?",

new Object[] { message.getMessage(), message.getId() },

new int[] { Types.VARCHAR, Types.INTEGER });

}

@Override

public int insert(GuestMessage message) {

int insertedCount = jdbcTemplate

.update(

"insert into GUESTBOOK\_MESSAGE (MESSAGE\_ID, GUEST\_NAME, MESSAGE, REGISTRY\_DATE)

values (MESSAGE\_ID\_SEQ.NEXTVAL, ?, ?, ?)");

...

return insertedCount;

}

(3) KeyHolder를 이용한 자동 생성 키 구하기

: MySQL의 auto\_increment 컬럼과 같이 데이터를 삽입할 때 자동으로 생성되는 키 컬럼 값을 구하고 싶은 경우에는 PreparedStatementCreator와

KeyHolder를 함께 사용하는 update() 메서드를 호출하면 된다.

@Override

public int insert(final GuestMessage message) {

KeyHolder keyHolder = new GeneratedKeyHolder();

int insertedCount = jdbcTemplate.update(new PreparedStatementCreator() {

@Override

public PreparedStatement createPreparedStatement(Connection con)

throws SQLException {

PreparedStatement pstmt = con

.prepareStatement(

"insert into GUESTBOOK\_MESSAGE (GUEST\_NAME, MESSAGE, REGISTRY\_DATE) values (?, ?, ?)",

new String[] { "MESSAGE\_ID" });

pstmt.setString(1, message.getGuestName());

pstmt.setString(2, message.getMessage());

pstmt.setTimestamp(3, new Timestamp(message.getRegistryDate()

.getTime()));

return pstmt;

}

}, keyHolder);

Number keyNumber = keyHolder.getKey();

message.setId(keyNumber.intValue());

return insertedCount;

}

org.springframework.jdbc.support.KeyHolder Interface는 생성된 키 값을 구해주는 getKey() 메서드를 제공하고 있다. 이 메서드는 java.lang.Number

타입을 리턴하므로 Number.intValue()나 Number.longValue() 메서드를 이용해서 생성된 키 값을 구할 수 있다. SPRING은 KeyHolder Interface의

구현 Class로 GeneratedKeyHolder Class를 제공하고 있다.

(4) ConnectionCallback을 이용한 Connection 사용

: Connection을 직접 사용해야 한다면 execute() 메서드를 사용하면 된다. execute() 메서드는 파라미터로 전달받은 ConnectionCallback Interface

구현 객체의 doInConnection() 메서드를 호출하는데 이 때 Connection을 doInConnection()에 인자로 전달한다.

public int count(){

return jdbcTemplate.execute(new ConnectionCallback<Integer>(){

@Override

public Integer doInConnection(Connection conn) throws SQLException, DataAccessException{

Statement stmt = null;

ResultSet rs = null;

try{

stmt = conn.createStatement();

rs = stmt.executeQuery("select count(\*) from GUESTBOOK\_MESSAGE");

rs.next();

return rs.getInt(1);

}finally{

JdbcUtils.closeResultSet(rs);

JdbcUtils.closeStatement(stmt);

}

}

});

}

ConnectionCallback 구현 객체의 doInConnection() 메서드는 파라미터로 전달 받은 Connection을 이용해서 알맞은 작업을 수행하면 된다. 커넥션 생성과

종료는 JdbcTemplate이 처리하므로 doInConnection() 메서드에서는 Connection을 종료할 필요가 없다.

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.2 NamedParameterJdbcTemplate Class를 이용한 JDBC 프로그래밍

: NamedParameterJdbcTemplate Class는 JdbcTemplate Class와 비슷한 기능을 제공하는데, 차이점이 있다면 인덱스 기반의 파라미터가 아니라 이름 기반의

파라미터를 설정할 수 있도록 해준다는 점이다. 예를 들어, 인덱스 기반의 파라미터를 전달받는 물음표를 사용하지 않고 다음과 같이 이름 기반의 파라미터를 쿼리에서

사용할 수 있도록 지원한다.

select \* from GUESTBOOK\_MESSAGE order by MESSAGE\_ID desc limit :startRow, :fetchSize

NamedParameterJdbcTemplate Class는 다음과 같이 생성자를 이용해서 DataSource를 전달받는다.

public class NamedParamGuestMessageDao implements GuestMessageDao {

private NamedParameterJdbcTemplate template;

public NamedParamGuestMessageDao(DataSource dataSource) {

template = new NamedParameterJdbcTemplate(dataSource);

}

...

}

NamedParameterJdbcTemplate Class는 JdbcTemplate Class와 동일한 이름의 메서드를 제공한다. 차이점이 있다면 인덱스 기반의 파라미터가 아니라 이름 기반의

파라미터 값을 설정하기 위해 Map이나 SqlParameterSource을 전달받는다는 것이다. Map을 전달받는 메서드와 SqlParameterSource를 전달받는 메서드를

차례대로 살펴보도록 하자.

(1) Map을 이용한 파라미터 값 설정 메서드

: Map 기반의 메서드는 Object 배열이 아닌 Map을 이용해서 이름을 가진 파라미터 값을 설정한다. 아래 Code는 Map 기반 메서드와 이름 기반의 파라미터를 갖는

SQL 쿼리의 사용 예이다.

@Override

public int delete(int id) {

Map<String, Object> paramMap = new HashMap<String, Object>();

paramMap.put("id", id);

return template.update(

"delete from GUESTBOOK\_MESSAGE where MESSAGE\_ID = :id",

paramMap);

}

위 Code에서 보듯이 SQL 쿼리는 인덱스 기반 파라미터 대신에 이름 기반의 파라미터를 사용하고 있으며, Map에서 동일한 이름을 갖는 값이 파라미터의

값으로 설정된다.

NamedParameterJdbcTemplate Class가 제공하는 Map 기반 메서드는 다음과 같다.

- List<T> query(String sql, Map<String, ?> paramMap, RowMapper<T> rowMapper)

- List<T> queryForList(String sql, Map<String, ?> paramMap, Class<T> elementType)

- T queryForObject(String sql, Map<String, ?> paramMap, RowMapper<T> rowMapper)

- T queryForObject(String sql, Map<String, ?> paramMap, Class<T> requiredType)

- int queryForInt(String sql, Map<String, ?> paramMap)

- long queryForLong(String sql, Map<String, ?> paramMap)

- int update(String sql, Map<String, ?> paramMap)

아래 Code는 query() 메서드의 사용 예를 보여주고 있다.

@Override

public List<GuestMessage> select(int begin, int end) {

Map<String, Object> paramMap = new HashMap<String, Object>();

paramMap.put("startRowNum", begin - 1);

paramMap.put("count", end - begin + 1);

return template

.query(

"select \* from GUESTBOOK\_MESSAGE order by MESSAGE\_ID desc limit :startRowNum, :count",

paramMap, new RowMapper<GuestMessage>() {

@Override

public GuestMessage mapRow(ResultSet rs, int rowNum)

throws SQLException {

GuestMessage message = new GuestMessage();

message.setId(rs.getInt("MESSAGE\_ID"));

message.setGuestName(rs.getString("GUEST\_NAME"));

message.setMessage(rs.getString("MESSAGE"));

message.setRegistryDate(rs.getDate("REGISTRY\_DATE"));

return message;

}

});

}

이름 기반의 파라미터를 갖지 않는 쿼리를 실행하는 경우에는 아무 값도 갖지 않는 Map 객체를 사용하면 된다.

int id = tempate.queryForInt("select last\_insert\_id() ", Collections.<String, Object> emptyMap());

(2) SqlParameterSource를 이용한 파라미터 값 설정 메서드

: Map 대신에 SqlParameterSource를 이용해서 파라미터 값을 설정할 수도 있다. 다음 메서드는 SqlParameterSource를 인자로 전달받는 메서드의 목록이다.

- List<T> query(String sql, SqlParameterSource paramSource, RowMapper<T> rowMapper)

- List<T> queryForList(String sql, SqlParameterSource paramSource, Class<T> elementType)

- T queryForObject(String sql, SqlParameterSource paramSource, RowMapper<T> rowMapper)

- T queryForObject(String sql, SqlParameterSource paramSource, Class<T> requiredType)

- int queryForInt(String sql, SqlParameterSource paramSource)

- long queryForLong(String sql, SqlParameterSource paramSource)

- int update(String sql, SqlParameterSource paramSource)

SqlParameterSource는 Interface이기 때문에 실제로 사용할 때에는 SqlParameterSource Interface를 구현한 Class를 사용해서 파라미터 값을 전달해

주어야 한다. SPRING은 다음과 같은 두 개의 SqlParameterSource 구현 Class를 제공하고 있다.

- org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource

- org.springframework.jdbc.core.namedparam.MapSqlParameterSource

BeanPropertySqlParameterSource Class는 동일한 이름을 갖는 Java 객체의 프로퍼티 값을 이용해서 파라미터 값을 설정한다. 아래 Code는

BeanPropertySqlParameterSource Class의 사용 예를 보여주고 있다.

@Override

public int insert(GuestMessage message) {

BeanPropertySqlParameterSource paramSource = new BeanPropertySqlParameterSource(

message);

int insertedCount = template.update(

"insert into GUESTBOOK\_MESSAGE (GUEST\_NAME, MESSAGE, REGISTRY\_DATE) values "

+ "(:guestName, :message, :registryDate)", paramSource);

...

return insertedCount;

}

위 Code에서 쿼리에 포함된 guestName, message, registryDate 파라미터는 각각 message 객체의 guestName 프로퍼티, message 프로퍼티, 그리고

registryDate 프로퍼티 값을 이용해서 설정된다.

MapSqlParameterSource Class는 Map과 비슷하게 <이름,값> 쌍을 이용해서 파라미터의 값을 설정한다. MapSqlParameterSource 객체를 생성한 뒤,

addValue() 메서드를 이용해서 파라미터 이름과 값을 설정해주면 된다. 아래 Code는 MapSqlParameterSource Class를 사용해서 파라미터 값을 출력하는

예를 보여주고 있다.

@Override

public int update(GuestMessage message) {

MapSqlParameterSource paramSource = new MapSqlParameterSource();

paramSource.addValue("message", message.getMessage());

paramSource.addValue("id", message.getId(), Types.INTEGER);

return template

.update(

"update GUESTBOOK\_MESSAGE set MESSAGE = :message where MESSAGE\_ID = :id",

paramSource);

}

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.3 SimpleJdbcTemplate Class를 이용한 JDBC 프로그래밍

: SimpleJdbcTemplate Class는 JdbcTemplate과 NamedParameterJdbcTemplate을 합쳐 놓은 템플릿 Class로서 이름 기반의 파라미터 설정과 인덱스 기반의

파라미터 설정을 모두 지원한다. 이름 기반의 파라미터를 설정할 때에는 Map과 SqlParameterSource 두 가지 방법을 모두 사용할 수 있다.

SimpleJdbcTemplate Class가 제공하는 주요 메서드는 다음과 같다.

- int queryForInt(String sql, Map<String,?> args)

- int queryForInt(String sql, SqlParameterSource args)

- int queryForInt(String sql, Object ... args)

- long queryForLong(String sql, Map<String,?> args)

- long queryForLong(String sql, SqlParameterSource args)

- long queryForLong(String sql, Object ... args)

- T queryForObject(String sql, Class<T> requiredType, Map<String, ?> args)

- T queryForObject(String sql, Class<T> requiredType, SqlParameterSource args)

- T queryForObject(String sql, Class<T> requiredType, Object ... args)

- T queryForObject(String sql, RowMapper<T> rm, Map<String,?> args)

- T queryForObject(String sql, RowMapper<T> rm, SqlParameterSource args)

- T queryForObject(String sql, RowMapper<T> rm, Object ... args)

- List<T> query(String sql, RowMapper<T> rm, Map<String,?> args)

- List<T> query(String sql, RowMapper<T> rm, SqlParameterSource args)

- List<T> query(String sql, RowMapper<T> rm, Object ... args)

- Map<String, Object> queryForMap(String sql, Map<String,?> args)

- Map<String, Object> queryForMap(String sql, SqlParameterSource args)

- Map<String, Object> queryForMap(String sql, Object ... args)

- List<Map<String, Object>> queryForList(String sql, Map<String,?> args)

- List<Map<String, Object>> queryForList(String sql, SqlParameterSource args)

- List<Map<String, Object>> queryForList(String sql, Object ... args)

- int update(String sql, Map<String,?> args)

- int update(String sql, SqlParameterSource args)

- int update(String sql, Object ... args)

아래 Code는 SimpleJdbcTemplate Class의 사용 예를 보여주고 있다.

public class SimpleJdbcGuestMessageDao implements GuestMessageDao {

private SimpleJdbcTemplate template;

public SimpleJdbcGuestMessageDao(DataSource dataSource) {

template = new SimpleJdbcTemplate(dataSource);

}

@Override

public int delete(int id) {

return template.update(

"delete from GUESTBOOK\_MESSAGE where MESSAGE\_ID = ?", id);

}

@Override

public int insert(GuestMessage message) {

BeanPropertySqlParameterSource paramSource = new BeanPropertySqlParameterSource(

message);

int insertedCount = template.update(

"insert into GUESTBOOK\_MESSAGE " +

"(GUEST\_NAME, MESSAGE, REGISTRY\_DATE) " +

"values(:guestName, :message, :registryDate)", paramSource);

if(insertedCount > 0){

int id = template.queryForInt("select last\_insert\_id()");

message.setId(id);

}

return insertedCount;

}

...

}

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.4 SimpleJdbcInsert Class를 이용한 데이터 삽입

: SimpleJdbcInsert Class는 쿼리를 사용하지 않고 데이터를 삽입할 수 있도록 해주는 Class이다. SimpleJdbcInsert Class를 이용하는

가장 간단한 방법은 다음과 같다.

public class SimpleJdbcGuestMessageDao implements GuestMessageDao {

private SimpleJdbcInsert insertMessage;

public SimpleJdbcGuestMessageDao(DataSource dataSource) {

insertMessage = new SimpleJdbcInsert(dataSource);

insertMessage.withTableName("GUESTBOOK\_MESSAGE");

}

@Override

public int insert(GuestMessage message) {

Map<String, Object> paramValueMap = HashMap<String, Object>();

paramValueMap.put("GUEST\_NAME", message.getGuestName());

paramValueMap.put("MESSAGE", message.getMessage());

paramValueMap.put("REGISTRY\_DATE", message.getRegistryDate());

return insertMessage.execute(paramValueMap);

}

위 Code에서 SimpleJdbcInsert Class의 withTableName() 메서드는 데이터를 삽입할 테이블의 이름을 지정한다. execute(Map<String, Object>) 메서드는

Map의 키를 컬럼명으로 사용하고, 값을 컬럼에 삽입할 데이터로 사용하는 SQL 쿼리를 실행한다. 즉, 위 Code가 실행하는 SQL 쿼리는 다음과 같다.

insert into GUESTBOOK\_MESSAGE (GUEST\_NAME, MESSAGE, REGISTRY\_DATE) values(?,?,?)

usingColumns() 메서드를 사용하면 쿼리를 생성할 때 사용할 컬럼을 직접 지정할 수도 있다. usingColumns() 메서드는 가변 인자를 파라미터로 갖기 때문에

알맞은 개수만큼 컬럼명을 지정해주면 된다.

public SimpleJdbcGuestMessageDao(DataSource dataSource) {

insertMessage = new SimpleJdbcInsert(dataSource);

insertMessage.withTableName("GUESTBOOK\_MESSAGE");

insertMessage.usingColumns("GUEST\_NAME","MESSAGE","REGISTRY\_DATE");

}

usingColumns() 메서드를 사용하면, 지정한 컬럼에 대해서만 값을 삽입하게 된다. 따라서, usingColumns() 메서드에서 지정하지 않은 컬럼에 대해서는

값이 삽입되지 않는다. 예를 들어, 아래 Code의 경우 REGISTRY\_DATE 컬럼에 대해서는 값이 삽입되지 않는다.

public SimpleJdbcGuestMessageDao(DataSource dataSource) {

insertMessage = new SimpleJdbcInsert(dataSource);

insertMessage.withTableName("GUESTBOOK\_MESSAGE");

insertMessage.usingColumns("GUEST\_NAME","MESSAGE");

}

@Override

public int insert(GuestMessage message) {

Map<String, Object> paramValueMap = HashMap<String, Object>();

paramValueMap.put("GUEST\_NAME", message.getGuestName());

paramValueMap.put("MESSAGE", message.getMessage());

paramValueMap.put("REGISTRY\_DATE", message.getRegistryDate());

return insertMessage.execute(paramValueMap);

}

SimpleJdbcTemplate Class가 제공하는 설정 메서드는 메서드 체이닝(method chainning)을 지원하기 때문에, 다음과 같이 메서드 체이닝을 이용해서 설정 Code를

작성할 수도 있다.

public SimpleJdbcGuestMessageDao(DataSource dataSource) {

insertMessage.withTableName("GUESTBOOK\_MESSAGE")

.usingGeneratedKeyColumns("MESSAGE\_ID")

.usingColumns("GUEST\_NAME", "MESSAGE", "REGISTRY\_DATE");

}

(1) execute() 메서드를 이용한 데이터 삽입

: SimpleJdbcTemplate Class를 이용해서 데이터를 삽입할 때에는 execute() 메서드를 사용하면 된다. SimpleJdbcTemplate Class는 다음과 같은 execute() 메서드를

제공하고 있다. 두 execute() 메서드는 쿼리 실행 결과로 영향을 받는 행의 개수를 리턴한다.

- int execute(Map<String, Object> args)

- int execute(SqlParameterSource parameterSource)

Map을 전달하는 경우 대소문자를 구분하지 않고 컬럼명과 Map의 키 값이 일치하는지 여부를 검사한다. 예를 들어, 아래 Code와 같이 SimpleJdbcTemplate.usingColumns()

메서드에서 지정한 컬럼명과 Map의 키는 소문자로 변환해서 일치할 경우 매칭된다.

public SimpleJdbcGuestMessageDao(DataSource dataSource) {

insertMessage = new SimpleJdbcInsert(dataSource);

insertMessage.withTableName("GUESTBOOK\_MESSAGE");

insertMessage.usingColumns("GUEST\_NAME","MESSAGE","REGISTRY\_DATE);

}

@Override

public int insert(GuestMessage message) {

Map<String, Object> paramValueMap = HashMap<String, Object>();

paramValueMap.put("GUEST\_NAME", message.getGuestName());

paramValueMap.put("message", message.getMessage());

paramValueMap.put("registry\_DATE", message.getRegistryDate());

return insertMessage.execute(paramValueMap);

}

SqlParameterSource를 사용해서 execute() 메서드를 실행하는 경우, 다음 규칙에 따라서 컬럼명의 일치 여부를 검사한다.

- 지정한 컬럼명과 동일한 이름을 갖는 파라미터 값이 설정되어 있는지 검사한다.

- '\_'이 포함된 경우 '\_'를 제외한 나머지 문자열과 일치하는 파라미터 값이 설정되어 있는지 검사한다.

아래 Code는 SqlParameterSource를 사용한 Code의 예이다.

public SimpleJdbcGuestMessageDao(DataSource dataSource) {

insertMessage = new SimpleJdbcInsert(dataSource);

insertMessage.withTableName("GUESTBOOK\_MESSAGE")

.usingColumns("GUEST\_NAME","MESSAGE","REGISTRY\_DATE);

}

@Override

public int insert(GuestMessage message) {

BeanPropertySqlParameterSource paramSource = new BeanPropertySqlParameterSource(message);

return insertMessage.execute(paramSource);

}

(2) executeAndReturnKey() 메서드를 이용한 데이터 삽입 및 자동 생성 키 조회

: MySQL의 auto\_increment 컬럼은 데이터가 삽입될 때마다 자동으로 증가된 값이 생성되는 컬럼이다. 이렇게 데이터 삽입시 자동으로 생성되는 키 컬럼을

구하고 싶을 때에는 executeAndReturnKey() 메서드를 사용하면 된다. executeAndReturnKey() 메서드는 다음과 같이 네 개가 존재한다.

- Number executeAndReturnKey(Map<String,Object> args)

- Number executeAndReturnKey(SqlParameterSource paramSource)

- KeyHolder executeAndReturnKeyHolder(Map<String,Object> args)

- KeyHolder executeAndReturnKeyHolder(SqlParameterSource paramSource)

executeAndReturnKey() 메서드를 사용하려면 usingGeneratedKeyColumns() 메서드를 이용해서 자동 생성되는 키 컬럼을 지정해주어야 한다.

insertMessage.withTableName("GUESTBOOK\_MESSAGE")

.usingGeneratedKeyColumns("MESSAGE\_ID")

.usingColumns("GUEST\_NAME","MESSAGE","REGISTRY\_DATE);

usingGeneratedKeyColumns() 메서드로 자동 생성 키 컬럼을 지정했다면, executeAndReturnKey() 메서드를 이용해서 생성되는 키 값을 구할 수 있다.

public SimpleJdbcGuestMessageDao(DataSource dataSource) {

insertMessage = new SimpleJdbcInsert(dataSource);

insertMessage.withTableName("GUESTBOOK\_MESSAGE")

.usingGeneratedKeyColumns("MESSAGE\_ID")

.usingColumns("GUEST\_NAME","MESSAGE","REGISTRY\_DATE);

}

@Override

public int insert(GuestMessage message) {

BeanPropertySqlParameterSource paramSource = new BeanPropertySqlParameterSource(message);

Number keyValue = insertMessage.executeAndReturnKey(paramSource);

message.setId(keyValue.intValue());

return 1;

}

Number가 아니라 KeyHolder를 리턴하는 executeAndReturnKeyHolder() 메서드를 사용하는 경우에는 다음과 같이 KeyHolder로부터 키 값을 구하면 된다.

KeyHolder keyHolder = insertMessage.executeAndReturnKeyHolder(paramSource);

message.setId(keyHolder.getKey().intValue());