공학
트렌드

소프트웨어공학센터 경영지원TF팀

Cassandra(Nosql)를 이용한 유닛 테스트 방법
Part 2 : 카산드라 테스트

2014. 7. 15. [제99호]

Contents

정보통신산업진흥원 부설
nipa 소프트웨어공학센터

Ⅰ. Cassandra 테스트 방법

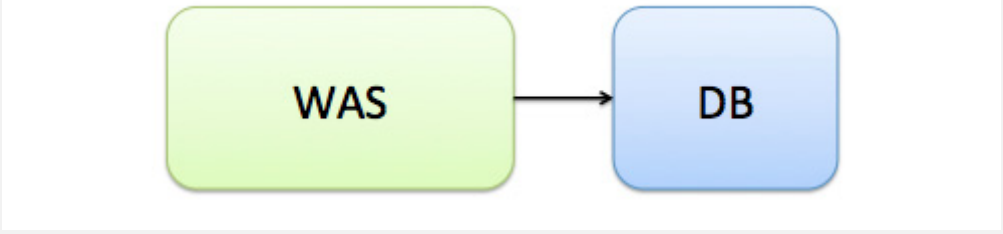
Cassandra와 연동하는 DAO(Data Access Object) 테스트를 어떻게 해야 할 지를 고민했다. 기존의 WAS-DB 형태의 모델에서도 DAO 테스트를 진행해야 하는 고민이 늘 있었고, 개인적으로 처음 쓰는 Cassandra를 어떻게 테스트 할 지가 가장 중요한 키 포인트였다. 이 문제를 해결했고 이 문제를 해결하는 과정을 소개한다.

1. 일반적인 DB DAO 테스트

일반적인 웹 서비스의 경우 DB에 SQL 쿼리를 실행하는 DAO 테스트는 세 가지로 진행한다. 첫 번째는 DAO 테스트를 하지 않는 것이다. 두 번째는 DB를 DAO 테스트용으로 설치하여 DAO 테스트를 진행한다. 세 번째는 HSQL DB(http://hsqldb.org)를 이용하여 자바 프로세스 데몬(Demon)을 실행하여 테스트하는 방식이다. 테스트할 때만 임베디드(Embedded) 되어 실행되기 때문에 편한 부분이 있으나 ANSI SQL이 아닌 특정 DB에 특화된 SQL을 실행 시에는 DAO 테스트가 실패할 수 있다.

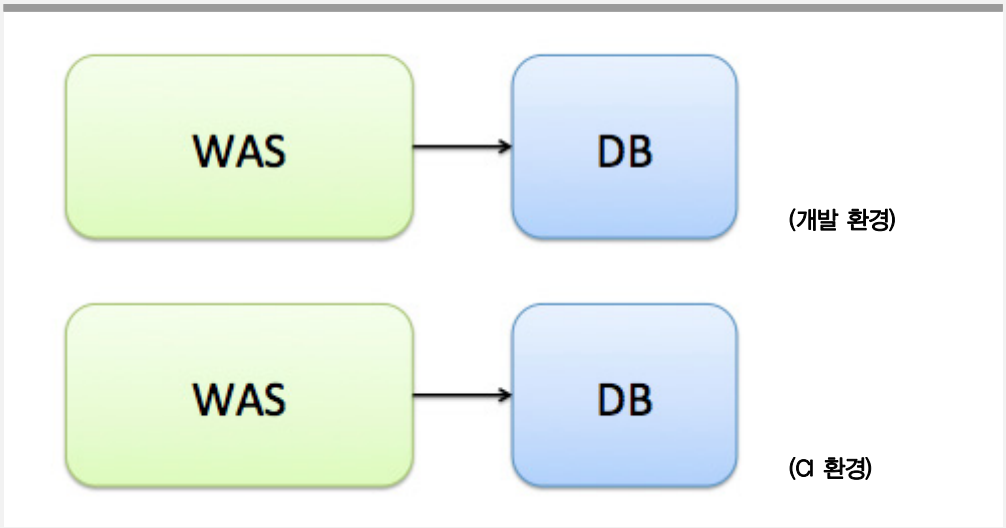
DAO 테스트를 하지 않는 경우는 Business Logic(Service Layer)이나 MVC만 테스트한다. DAO 부분은 Mocking 라이브러리를 이용해서 예상되는 값만을 리턴하게 해서 테스트하는 경우가 많다. DB 기반 위에 DAO 테스트를 진행한다는 것은 Transaction을 SQL보다 먼저 진행하고 DAO 테스트를 진행 완료 후 Rollback을 할 수 있고, Transaction 없이 DAO 테스트를 완료 후, DB 데이터를 모두 삭제하는 작업하는 부분이 있을 수 있다. 관련해서는 DB 환경에 맞게 DAO 테스트 환경을 구현한다. 일반적으로 DAO 테스트를 진행한다면 〈그림 1〉과 같을 것이다. DB 와 WAS를 설치해서 테스트를 진행한다.

그림 1_일반적인 DAO 테스트 환경



이슈는 Unit Test 할 때 개발 환경 DB와 Unit Test DB를 하나로 사용할 때 조심해야 한다. 하나의 DB를 사용하다 보니, Unit Test에서 돌아가는 DAO 테스트와 개발 환경에서 개발자가 테스트하는 DAO 테스트가 중복으로 실행이 되어 데이터가 꼬이는 문제가 발생할 수 있다. 따라서 이런 문제를 원칙적으로 해결하기 위해서는 DB를 두개로 설치해야 한다. 〈그림 2〉처럼 하나는 개발환경용, 또 다른 하나는 CI(Continuous Integration) 용으로 설치해야 한다.

그림 1_DB 데이터의 훼손을 방지하기 위해서 개발 환경/CI 별로 DB를 설치하는 환경구성

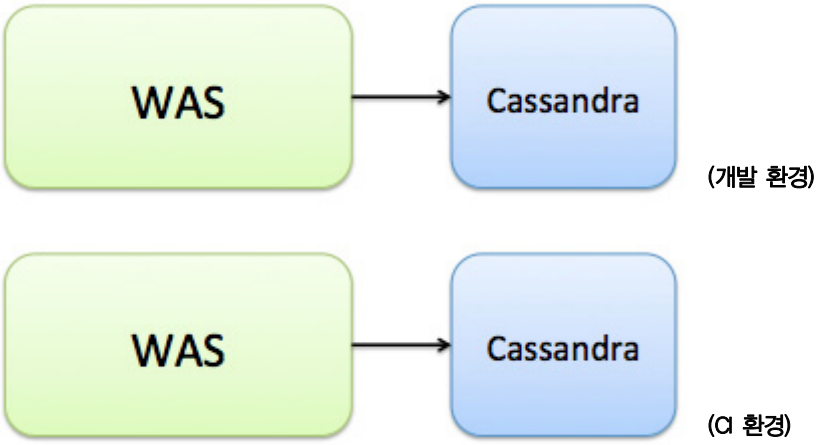


DAO 테스트를 진행하기 위해서 인프라적 요소를 신경 써야 하는 부분이 생긴다. 코드도 일부 수정해야 하는 부분이 있어서 DAO 테스트를 적잖이 번거롭게 하는 경우가 발생한다.

2. Cassandra 테스트 환경

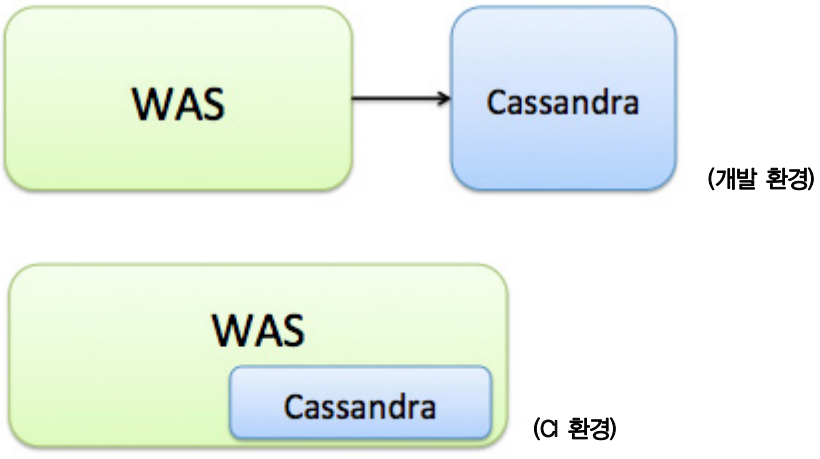
Cassandra를 사용하는 경우에도 동일한 경우가 발생한다. 만약 Cassandra를 처음 쓰는 경우라면 Cassandra DAO 테스트는 더욱 중요하다. 〈그림 2〉처럼 기존 DB의 DAO 테스트하는 방식처럼 Cassandra를 개발용도와 CI 용으로 나누는 경우인 〈그림 3〉과 같이 Cassandra를 설치해 테스트를 진행할 수 있다.

그림 3_Cassandra 데이터의 훼손을 방지하기 위해서 개발 환경/CI 별로 DB를 설치하는 환경구성



이와 같이 Cassandra DAO 테스트를 진행한다면 Cassandra 역시 불편해질 수 있다. 그러나 다행히 Cassandra는 Java로 개발된 서버이기 때문에 DAO 테스트할 때는 Cassandra를 Thread 데몬형태로 띄워 테스트를 진행할 수 있다. 따라서 〈그림 4〉와 같이 개발 환경에서는 그대로 물리 서버에 Cassandra를 설치해서 사용할 수 있고, DAO 테스트할 때는 특별히 Cassandra를 설치할 필요가 없다.

그림 4_Cassandra 데이터의 훼손을 방지하기 위해서 개발 환경/CI 별로 DB를 설치하는 환경구성



이 방법을 이용하려면 Cassandra-unit[1] 프로젝트에 접속하여 필요한 라이브러리를 사용하면 된다.

1) https://github.com/jsevellec/cassandra-unit)

예제는 cassandra-unit-examples[2]에 접속하여 소스를 다운로드해 사용하면 이해가 쉽다. 이 예제를 바탕으로 테스트하면 된다.

Local 환경과 CI 환경에서는 내장된 Cassandra를 사용하여 테스트를 진행할 수 있고, 물리적인 서버 기반인 Alpha 환경에서는 Cassandra를 설치하여 테스트할 수 있도록 환경을 셋팅했다. 생산성과 유지보수성을 높일 수 있는 방법이었다고 생각한다.

II. Cassandra Unit Test

Cassandra-unit-examples에 좋은 예제가 많다. 그 중 사용할 만한 예제들의 일부를 보기 쉽게 수정한 예제를 공유하고 저자가 사용한 방식의 테스트 방식을 소개한다.

1. Hector-ColumnFamily 연동 테스트

dataSetDefaultValues.json은 'beautifulKeyspaceName' keyspace의 'columnFamily1' colum family에 '01' key와 column 데이터를 추가하는 내용이다.

〈dataSetDefaultValues.json〉

```
{
   "name" : "beautifulKeyspaceName",
   "columnFamilies" : [{
        "name" : "columnFamily1",
        "rows" : [{
            "key" : "01",
            "columns" : [{
                "name" : "02",
                "value" : "03"
            }]
        }]
   }]
}
```

2) https://github.com/jsevellec/cassandra-unit-examples

LoadDataSetDefaultValueTest 클래스는 EmbeddedCassandraServerHelper.startEmbedded Cassandra() 메소드를 호출하여 Cassandra를 실행하고, dataSetDefaultValues.json 파일을 읽어 column family를 저장한다. Hector API인 SliceQuery.execute() 메소드 호출을 하여 'beautifulKeyspaceName' keyspace의 'columnFamily1' colum family에 '01' key의 '02' name의 value 값을 읽어오고 출력한다.

〈LoadDataSetDefaultValueTest.java〉

```
public class LoadDataSetDefaultValueTest {

 @Test
 public void shouldHaveLoadASimpleDataSet() throws Exception {

    String clusterName = "TestCluster";
    String host = "localhost:9171";
    EmbeddedCassandraServerHelper.startEmbeddedCassandra();

    DataSet dataSet = new ClassPathJsonDataSet("dataSetDefaultValues.json");
    DataLoader dataLoader = new DataLoader(clusterName, host);
    dataLoader.load(dataSet);

    Cluster cluster = HFactory.getOrCreateCluster(clusterName, host);
    Keyspace keyspace = HFactory.createKeyspace("beautifulKeyspaceName", cluster);

    SliceQuery<byte[], byte[], byte[]> sq = HFactory.createSliceQuery(keyspace, BytesArraySeri
alizer.get(),
          BytesArraySerializer.get(), BytesArraySerializer.get());

    sq.setKey(Hex.decodeHex("01".toCharArray()));
    sq.setColumnFamily("columnFamily1");
    sq.setColumnNames(Hex.decodeHex("02".toCharArray()));

    QueryResult<ColumnSlice<byte[], byte[]>> result = sq.execute();
    List<HColumn<byte[], byte[]>> columns = result.get().getColumns();
    System.out.println("slice:" + result.get() + " slice.length:" + columns.size());
 }
}
```

Hector API가 thrift 기반이기 때문에 low-level type의 특성을 가지고 있어서 코드 관리가 쉽지 않은 부분이 있다. byte 단위의 데이터 통신구조라 작은 실수가 발생하면 기대치 값을 얻지 못할 수 있다.

2. CQL-Cassandra Driver 연동 테스트

Cassandra Driver를 이용하여 CQL 기반 데이터를 저장하고 읽어오는 예제이다. 아래 'simple.cql'은 myTable 테이블을 생성하고 'myKey01' Key를 추가한다.

〈simple.cql〉

```
CREATE TABLE myTable(
   id varchar,
   value varchar,
   PRIMARY KEY(id));

INSERT INTO myTable(id, value) values('myKey01','myValue01');
```

CQLScriptLoadTest.java 는 CassandraCQLUnit을 이용해서 simple.cql 파일을 읽어 table생성과 데이터를 추가한다. 그리고 select 문을 실행하여 'myValue01' 데이터를 읽어온 후, 그 값이 정상적으로 들어왔는지 확인한다.

〈CQLScriptLoadTest.java〉

```
public class CQLScriptLoadTest {

 @Rule
 public CassandraCQLUnit cassandraCQLUnit = new CassandraCQLUnit(new ClassPathCQLDataSet("simple.cql","keyspaceNameToCreate"));

 @Test
 public void test() throws Exception {
   ResultSet result = cassandraCQLUnit.session.execute("select * from mytable WHERE id='myKey01'");
   MatcherAssert.assertThat(result.iterator().next().getString("value"), Matchers
.is("myValue01"));
 }
}
```

실제 업무에서 CassnadraCQLUnit 또는 Driver를 이용해서 개발하는 것은 쉽지 않다. 보통은 Framework 기반에서 개발하기 때문에 이를 지원하는 형태도 필요하다.

3. Spring Test-CQL 연동 테스트

Spring MVC를 사용하는 WAS에서 테스트할 수 있다. 특히 미리 CQL을 실행시켜 테스트 환경에서는 Select를 이용하면 된다. 아래 예제는 simple.cql 파일과 Spring JUnit 테스트 환경을 보여준다. id와 value라는 column을 갖는 myTable 이라는 테이블을 생성하고 두 개의 값을 저장한다.

〈simple.cql〉

```
CREATE TABLE myTable(
   id varchar,
   value varchar,
   PRIMARY KEY(id));

INSERT INTO myTable(id, value) values('myKey01','myValue01');

INSERT INTO myTable(id, value) values('myKey02','myValue02');
```

SpringCQLScriptLoadTest.java 는 SpringJUnit4ClassRunner.class Unit Test로 실행된다. @EmbeddedCassandra 어노테이션(Annotation)을 두어 테스트 실행 시 자동으로 Cassandra 서버가 실행한다. connect() 메소드로 Cassandra 서버에 통신 연결한다. 그리고 testSelect() 메소드에서는 'simple.cql' 에서 insert했던 'myKey01' Key 값으로 저장된 value를 확인한다. testInsert() 메소드는 새로운 insert문으로 table에 'myKey03' Key 값을 추가하고 select 문으로 정상적으로 데이터를 가져오는지 확인한다. 마지막으로 testInsertTTL() 메소드는 TTL(Time To Live) 5초로 'myKey04' 값을 저장하고 6초 뒤에 'myKey04' Key로 select 문으로 찾고 null인지 확인한다.

〈SpringCQLScriptLoadTest.java〉

```
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners({ CassandraUnitTestExecutionListener.class })
@CassandraDataSet(value = { "simple.cql" })
@EmbeddedCassandra
public class SpringCQLScriptLoadTest {
 Session session ;

 @Before
```

```
public void connect() throws Exception {
  Cluster cluster = Cluster.builder()
    .addContactPoints("127.0.0.1")
    .withPort(9142)
    .build();
  session = cluster.connect("cassandra_unit_keyspace");
}

@Test
public void testSelect() {
  ResultSet result = session.execute("select * from mytable WHERE id='myKey01'");
  MatcherAssert.assertThat(result.iterator().next().getString("value"), Matchers.is("myValue01"));
}

@Test
public void testInsert() {
  session.execute("INSERT INTO myTable(id, value) values('myKey03','myValue03')");
  ResultSet result = session.execute("select * from mytable WHERE id='myKey03'");
  String value = result.iterator().next().getString("value");
  MatcherAssert.assertThat(value, Matchers.is("myValue03"));
}

@Test
public void testInsertTTL() throws InterruptedException {
  session.execute("INSERT INTO myTable(id, value) values('myKey04','myValue04') using TTL
5");
  Thread.sleep(6000);
  ResultSet result = session.execute("select * from mytable WHERE id='myKey04'");
  Object value = result.iterator().next();
  MatcherAssert.assertThat(value, Matchers.nullValue());
}
}
```

4. YAML-자동 실행 연동 테스트

Cassandra의 yaml 파일을 읽어 테스트를 진행할 수 있다. yaml 파일은 json과 비슷하며 사람이 쉽게 읽을 수 있는 직렬파일이다[3]. simpleDataSet.yaml 파일은 그 예를 보여

주고 있다. json과 비슷하고 key, value로 구성되어 있다.

〈simpleDataSet.yaml〉

```
name: beautifulKeyspaceName
columnFamilies:
- name: beautifulColumnFamilyName
 rows:
 - key: 10
   columns:
   - {name: 11, value: 11}
   - {name: 12, value: 12}
 - key: 20
   columns:
   - {name: 21, value: 21}
```

AutomaticallyStartAndLoadSimpleYamlDataSetTest 클래스는 AbstractCassandraUnit4TestCase 클래스를 상속받는다. Cassandra가 실행하면서 'simpleDataSet.yaml' 파일을 읽고 체크할 수 있다. 특별히 Cassandra를 구동하는 코드가 없으며, AbstractCassandraUnit4TestCase 클래스의 Hector API인 Keyspace 객체를 이용하여 테스트를 할 수 있다.

〈AutomaticallyStartAndLoadSimpleYamlDataSetTest.java〉

```
public class AutomaticallyStartAndLoadSimpleYamlDataSetTest extends AbstractCassandraUnit4TestCase {

 @Override
 public DataSet getDataSet() {
   return new ClassPathYamlDataSet("simpleDataSet.yaml");
 }

 @Test
 public void shouldHaveLoadASimpleDataSet() throws Exception {
   MatcherAssert.assertThat(getKeyspace(), notNullValue());
   MatcherAssert.assertThat(getKeyspace().getKeyspaceName(), is("beautifulKeyspaceName"));

 }
}
```

3) 참조 : http://ko.wikipedia.org/wiki/YAML

5. Spring Test-Astyanax 연동 테스트

저자는 Cassandra를 서비스 모니터링 용 데이터 저장소로, 자바 클라이언트는 개발/테스트를 하기 쉽고 이해가 빠른 Netflix의 Astyanax를 사용하였다.

cassandra-unit에서는 Astyanax 테스트가 어려워서 Astyanax에서 제공하는 클래스를 이용하여 테스트 코드를 개발하였다. Spring Framework에서 테스트를 가능하고 유지보수가 편하다. 게다가 테스트 시에도 상용 환경의 설정을 동일하게 작동할 수 있게 있어 코드의 일관성을 유지할 수 있다.

Spring 설정(applicationContext-cassandra-astyanax.xml, cassandra.properties)으로 연결정보를 따로 두어 개발환경에 맞는 properties를 사용하면 테스트가 가능하다. 원래 코드는 더 복잡하나 핵심적인 기능은 동일하다.

〈applicationContext-cassandra-astyanax.xml〉

```
<beans >
 <context:component-scan base-package="org.cassandraunit.test.astyanax"/>

 <context:property-placeholder order="1"
   ignore-unresolvable="true" location="classpath:cassandra.properties" />

 <bean id="storageDao" class="org.cassandraunit.test.astyanax.StorageDao"/>

 <bean id="osClient" class="org.cassandraunit.test.astyanax.AstyanaxClient">
   <property name="clusterName" value="${system.cassandra.cluster.name}" />
   <property name="discoveryType" value="${system.cassandra.discovery.type.token}" />
   <property name="cqlVersion" value="${system.cassandra.cql.version}" />
   <property name="targetCassandraVersion" value="${system.cassandra.target.version}" />
   <property name="port" value="${system.cassandra.port}" />
   <property name="maxConnsPerHost" value="${system.cassandra.max.host}" />
   <property name="seeds" value="${system.cassandra.seeds}" />
   <property  name="connectionPoolName"  value="${system.cassandra.connection.pool.name}"
/>
   <property name="keyspaceName" value="${system.cassandra.os.keyspace.name}" />
 </bean>
</beans>
```

〈cassandra.properties〉

```
system.cassandra.connection.pool.name=cassandraPool
system.cassandra.port=9160
system.cassandra.max.host=1
system.cassandra.seeds=127.0.0.1:9160
system.cassandra.cql.version=3.0.0
system.cassandra.target.version=2.0.2
system.cassandra.cluster.name=Test Cluster

system.cassandra.discovery.type.ring=RING_DESCRIBE
system.cassandra.discovery.type.discovery=DISCOVERY_SERVICE
system.cassandra.discovery.type.token=TOKEN_AWARE
system.cassandra.discovery.type.none=NONE

system.cassandra.os.keyspace.name=os
```

AstyanaxCassandraSpringTest 클래스는 SingletonEmbeddedCassandra 클래스 인스턴스를 이용하여 Cassandra를 standalone으로 실행한다. 'os' keyspace를 생성하고, 'cpu_util_pct' table을 생성한다. 그 후, Spring 설정 파일(applicationContext-cassandra-astyanax.xml)을 읽어 Standalone으로 실행하고 있는 Cassandra에 접속한다. 'Test Cluaster' Cluster의 'os' keyspace에 연결하고 table을 접속 후, DAO 를 테스트한다. insert 한 값이 그대로 저장되었는지 select 하여 확인할 수 있다.

〈AstyanaxCassandraSpringTest.java〉

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:applicationContext-cassandra-astyanax.xml" })
public class AstyanaxCassandraSpringTest {

 private static Keyspace                      keyspace;
 private static AstyanaxContext<Keyspace> keyspaceContext;

 private static String TEST_CLUSTER_NAME = "Test Cluster";
 private static String TEST_KEYSPACE_NAME = "os";
 private static final String SEEDS = "localhost:9160";

 private static ColumnFamily<String, String> TABLE =
          ColumnFamily.newColumnFamily("cpu_util_pct", StringSerializer.get(), StringSerializer.g
```

```
et(), LongSerializer.get());

    @Autowired
 StorageDao storageDao;

    @Before
    public void before() throws Exception {
    SingletonEmbeddedCassandra.getInstance();
    Thread.sleep(1000 * 3);
    createKeyspace();
    createTable();
    Thread.sleep(1000 * 3);
    }

 private void createTable() throws Exception {
    OperationResult<CqlResult<String, String>> result;
    result = keyspace
          .prepareQuery(TABLE)
          .withCql("create table cpu_util_pct (guid text PRIMARY KEY, pct double) ")
          .execute();
 }

 @AfterClass
 public static void teardown() throws Exception {
    if (keyspaceContext != null)
       keyspaceContext.shutdown();

    Thread.sleep(1000 * 10);
 }

 private  void createKeyspace() throws Exception {
    keyspaceContext = new AstyanaxContext.Builder()
    .forCluster(TEST_CLUSTER_NAME)
    .forKeyspace(TEST_KEYSPACE_NAME)
    .withAstyanaxConfiguration(
          new AstyanaxConfigurationImpl()
          .setDiscoveryType(NodeDiscoveryType.RING_DESCRIBE)
          .setConnectionPoolType(ConnectionPoolType.TOKEN_AWARE))
```

```
        .withConnectionPoolConfiguration(
            11new ConnectionPoolConfigurationImpl(TEST_CLUSTER_NAME
                + "_" + TEST_KEYSPACE_NAME)
            .setSocketTimeout(30000)
            .setMaxTimeoutWhenExhausted(2000)
            .setMaxConnsPerHost(20)
            .setInitConnsPerHost(10)
            .setSeeds(SEEDS))
            .withConnectionPoolMonitor(new CountingConnectionPoolMonitor())
            .buildKeyspace(ThriftFamilyFactory.getInstance());

    keyspaceContext.start();

    keyspace = keyspaceContext.getEntity();

    keyspace.createKeyspace(ImmutableMap.<String, Object>builder()
        .put("strategy_options", ImmutableMap.<String, Object>builder()
            .put("replication_factor", "1")
            .build())
            .put("strategy_class",      "SimpleStrategy")
            .build()
        );
  }
  @org.junit.Test
  public void insertTest() {
    storageDao.insertCpu("a", 0);
    double d1 = storageDao.getCpuUtil("a");
    MatcherAssert.assertThat(d1, Matchers.is((double) 0));

    storageDao.insertCpu("a", 2);
    double d2 = storageDao.getCpuUtil("a");
    MatcherAssert.assertThat(d2, Matchers.is((double) 2));

    storageDao.insertCpu("b", 10);
    double d3 = storageDao.getCpuUtil("b");
    MatcherAssert.assertThat(d3, Matchers.is((double) 10));
  }
}
```

StorageDao 클래스는 astyanax의 cql 쿼리를 사용하여 insert, select문을 호출한다. 연결에 대한 모든 책임은 AstyanaxClient 클래스에 위임하고 StorageDao 클래스는 쿼리에 집중한다.

〈StroageDao.java〉

```
@Component
public class StorageDao {
 private static final Logger log = LoggerFactory.getLogger(StorageDao.class);

 @Autowired
 @Qualifier("osClient")
 private AstyanaxClient osClient;

 public void insertCpu(String guid, double pct) {
   Keyspace keyspace = osClient.getKeyspace();
   ColumnFamily<String, String> COLUMN_FAMILY =
       ColumnFamily.newColumnFamily("cpu_util_pct",  StringSerializer.get(),  StringSerializer.get(
));

   try {
     String statement = String.format("insert into cpu_util_pct (guid, pct) values (?, ?);");
     log.debug(statement);

     keyspace.prepareQuery(COLUMN_FAMILY)
         .withCql(statement)
         .asPreparedStatement()
         .withStringValue(guid)
         .withDoubleValue(pct)
         .execute();
   } catch (Exception e) {
     e.printStackTrace();
     throw new RuntimeException("failed to insert.", e);
   }
 }

 public double getCpuUtil(String guid) {
   Keyspace keyspace = osClient.getKeyspace();
   ColumnFamily<Integer, String> COLUMN_FAMILY = ColumnFamily.newColumnFamily("cpu_util
```

```
_pct", IntegerSerializer.get(), StringSerializer.get());


    double cpuPct = 0;
    try {
      OperationResult<CqlResult<Integer, String>> result = keyspace.prepareQuery(COLUMN_FAMILY)
                                        .withCql("SELECT pct FROM cpu_util_pct WHERE guid = '" +
guid + "'")
                                        .execute();
      for (Row<Integer, String> row : result.getResult().getRows()) {
        ColumnList<String> cols = row.getColumns();
        cpuPct = cols.getDoubleValue("pct", 0.0D);
      }
    } catch (ConnectionException e) {
      e.printStackTrace();
      throw new RuntimeException("failed to read from cql", e);
    }
    return cpuPct;
  }

  class IntTreeMapComparator implements Comparator<Long> {
    @Override
     public int compare(Long o1, Long o2) {
           return o1.compareTo(o2);
      }
  }
}
```

AstyanaxClient 클래스는 Cassandra와 통신을 책임진다. cassandra.properties로 읽은 설정 값들을 파라미터로 넘겨 AstyanaxContext 를 생성하도록 한다. 서버에 접속한 후 Keyspace 객체를 생성하여 StorageDao 클래스에서 사용할 수 있도록 한다.

〈AstyanaxClient.java〉

```
public class AstyanaxClient {


  private static final Logger log = LoggerFactory.getLogger(AstyanaxClient.class);


  private AstyanaxContext<Keyspace> context;
  private Keyspace keyspace;
```

```
private String clusterName;
private String keyspaceName;
private String cqlVersion;
private String targetCassandraVersion;
private String connectionPoolName;
private int port;
private int maxConnsPerHost;
private String seeds;
private NodeDiscoveryType discoveryType;

@PostConstruct
public void init() {
  context = new AstyanaxContext.Builder()
        .forCluster(clusterName)
        .forKeyspace(keyspaceName)
        .withAstyanaxConfiguration(
              new AstyanaxConfigurationImpl()
                    .setDiscoveryType(discoveryType)
                    .setCqlVersion(cqlVersion)
                    .setTargetCassandraVersion(
                          targetCassandraVersion))
        .withConnectionPoolConfiguration(
              new ConnectionPoolConfigurationImpl(connectionPoolName)
                    .setPort(port)
                    .setMaxConnsPerHost(maxConnsPerHost)
                    .setSeeds(seeds)
                    .setTimeoutWindow(3000)
                    .setConnectTimeout(3000)
                    )
        .withConnectionPoolMonitor(new CountingConnectionPoolMonitor())
        .buildKeyspace(ThriftFamilyFactory.getInstance());

  context.start();
  keyspace = context.getClient();
}

@PreDestroy
public void destroy() {
```

```
    context.shutdown();
  }

// getter, setter
...
```

이 설정 이외에도 cassandra-template.yaml과 같은 서버 설정 정보와 dependency가 pom.xml 설정은 http://knight76.tistory.com/entry/cassandra-astyanax-cql3-unit-test-code-in-Spring-test 를 참조한다.

Ⅲ. 결론

Cassandra는 Dynamo 분산 모델의 큰 특징인 event consistency 와 BigTable Column 기반의 큰 특징인 Column 기반의 key-value 체계를 큰 틀로 하고 있다. 여기에 분산 처리(Distributed)가 되게 하여 SPOF(Single Point Of Failure)을 줄여주고 데이터 손실이 없도록 한다. 어떤 노드에 장애가 발생해도 전체 시스템은 멈추지 않도록 한다. 장비를 추가하고 제거하는 과정이 단순해서 새로운 장비를 추가하고 설정을 바꾼 후 Cassandra를 재 시작하면 되는 편리한 도구(Tool)이다.

특히 자료 저장을 위해서 ColumnFamily를 사용할 수 있으며, SQL과 비슷한 CQL를 제공하여 쿼리에 익숙한 개발자가 사용하기에 큰 불편함이 없도록 지원하고 있다. Cassandra를 특징을 살펴보고, java client 종류 중 많이 사용하고 있는 Thrift, Hector, Astyanax, CQL를 확인하였다. DAO 테스트할 때 Cassandra는 java daemon으로 실행시켜 테스트를 상용과 비슷한 환경으로 테스트할 수 있는 환경을 살펴보았다.

Cassandra 를 이용한 테스트 환경 구축 시 유용한 도움이 되길 바란다.

참고 자료

1. https://github.com/jsevellec/cassandra-unit
2. https://github.com/jsevellec/cassandra-unit-examples