Project Part 6 Existing Test Cases: Realm-Java

Authors: Wen-Chia Yang, Junxian Chen, Zihua Weng

March 15, 2020

ABSTRACTION

Objectives

In this report, we explored existing test cases to understand more about the system.

Source Code

Repository: https://github.com/solution-accepted/realm-java

Branch: master

TEST CASES RESEARCH

What We Found

After reading existing test cases in realm-java, we found test cases in RealmInMemoryTest.java, RealmProcessorTest.java and IOSRealmTests.java are interesting as they provide us more detail information about each according feature in Realm. In addition, those details complement our understanding of realm-java from reading the implementation code.

Existing Test Cases

Test Case 1: RealmProcessorTest

```
@Test
public void compileAllTypesProxyFile() {
  ASSERT.about(javaSource())
        .that(allTypesModel)
        .compilesWithoutError();
}
@Test
public void compareProcessedAllTypesFile() {
  ASSERT.about(javaSource())
        .that(allTypesModel)
        .processedWith(new RealmProcessor())
        .compilesWithoutError()
        .and()
        .generatesSources(allTypesDefaultMediator, allTypesDefaultModule, allTypesDefaultMediator,
allTypesProxy);
}
```

These are the test cases related to our first pull request. In issue 1650 (https://github.com/realm/realm-java/issues/1650), we were asked to modify the toString method of RealmProxy, which was a dynamically generated Java class. With further investigation, RealmProxy was actually generated by a Kotlin class RealmProxyClassGenerator directly called by RealmProcessor. The RealmProcessor creates proxy classes for all classes marked with @RealmClass. To test RealmProcessor, there was a test case “compileAllTypesFile” in the Java test class RealmProcessorTest. It asserts that “allTypesModel” (some/test/AllTypes.java, or AllTypes) should be compiled without error. AllTypes is a RealmObject (RealmObject is annotated with @RealmClass) that will be processed by the RealmProcessor. It aims to verify the RealmProcessor is able to correctly produce proxy class for all data types. RealmProcessor will make some_test_AllTypesRealmProxy for AllTypes, and then RealmProcessorTest will compare the generated source code with pre-defined source code (some_test_AllTypesRealmProxy) to see if they match. If so, then the test case passes.

The reason why it is interesting is that it was the first time when we saw code that wrote code. To be exact, it was Kotlin writing Java code. The RealmProxyClassGenerator has a lot of “emit” methods to write different Java code, such as emitPackage, emitEmptyLine, emitImports. It is totally the mind process of the way humans write code. RealmProxyClassGenerator was not written in Kotlin at first but in Java. We learned that the Realm team had updated their technologies during these years, by introducing a new language.

Test Case 2: RealmInMemoryTest

```
@Test
public void multiThread() throws InterruptedException, ExecutionException {...}
```

Test cases in RealmInMemoryTest aim to test the interaction of in-memory Realm instances with each other. The development team tested Realm instances in the same thread and different threads along with some Realm instances operation (open, close, delete). The above mentioned test case tests the status of Realm instances in the main thread and work thread

under specific circumstances. It first creates an in-memory Realm instance in the main thread and then creates an in-memory Realm with the same name in another thread. Next, it closes the in-memory Realm instance in the main thread and the Realm data should not be released since another instance is still held by the other thread. When the in-memory Realm instance is closed, the Realm data should be released since no more instance with the specific name exists. From this test case, we understand that Realm instances with the same name in the main thread and work thread share the same data and the data will not be released if at least one thread holds the Realm instance.

Test Case 3: IOSRealmTests

```
@Test
public void iOSDataTypes() throws IOException { ... }
@Test
public void iOSDataTypesDefaultValues() throws IOException { ... }

@Test
public void iOSDataTypesNullValues() throws IOException { ... }
@Test
@SuppressWarnings("ConstantOverflow")
public void iOSDataTypesMinimumValues() throws IOException { ... }
@Test
@SuppressWarnings("ConstantOverflow")
public void iOSDataTypesMaximumValues() throws IOException { ... }
@Test
public void iOSEncryptedRealm() throws IOException { ... }
```

IOSRealmTests tests whether the realm database file(.realm) created on iOS is interoperable with Android devices or not. In this test class, it includes two directions. The one side is to test whether the code can execute files correctly with different iOS data types, with default/null/minimum/maximum values. The other side is to test loading data with the encrypted realm file(.realm).

SOLUTION-ACCEPTED

After figuring out the entire test class, we found it is intriguing since this is the first time that we saw the java project that includes iOS test cases. We assume that, in realm-java, developers tend to make the realm compatible with files that are generated from different mobile platforms(in this case: iOS). This interoperability makes realm-java more easy to use across different platforms.