Introduction

Team: Solution Accepted;
Members: Junxian Chen, Wen Chia Yang, Zihua Weng
Realm is a mobile database that runs directly inside phones, tablets or wearables.

Feature 1: Customized Configurations

Background

Normally, to apply Realm with default configuration in Android apps, we can first initialize it using the following codes:

```
public class MainApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        Realm.init(this);
    }
}
```

To apply Realm with customized configurations, we can use codes like this:

```
private static final String REALM_PATH
=Environment.getExternalStorageDirectory().getAbsolutePath() + "/realm_dir/";

    @Override
    public void onCreate() {
        super.onCreate();
        Realm.init(this);
        RealmConfiguration.Builder builder = new RealmConfiguration.Builder()
                .name("custom.realm")
                .directory(new File(REALM_PATH))
                .schemaVersion(1)
                .modules(Realm.getDefaultModule())
                .deleteRealmIfMigrationNeeded()
                .encryptionKey(getKey());
        Realm.getInstance(builder.build());
    }
```

We want to know more about how it is done behind the scenes. So we choose to do some research on how the Realm database gets initialized.

Research

First, we located the Realm core code inside /realm/realm-library/src/main/java/io/realm. Then we search with the keyword “init”, which leads us to the “Realm.java” file. There we found 2 functions named “init” and here are their signatures:

```
public static synchronized void init(Context context, String userAgent) {}
public static synchronized void init(Context context) {}
```

They initialize the Realm library and create a default configuration that is ready to use. It is required to call this method before interacting with any other of the Realm API's. Moreover, they are both synchronized methods, so they are all thread-safe. Similarly they call the following function inside their bodies:

```
private static void initializeRealm(Context context, String userAgent) {
  if (BaseRealm.applicationContext == null) {
    //noinspection ConstantConditions

    if (context == null) {
      throw new IllegalArgumentException("Non-null context required.");
    }
      // In some cases, Context.getFilesDir() is not available
      // when the app launches the first time.
    checkFilesDirAvailable(context);

      // load librealm-jni.so
    RealmCore.loadLibrary(context);

      // ! set the default configuration of Realm
      // and we are interested in digging deeper
    setDefaultConfiguration(new RealmConfiguration.Builder(context).build());

    ObjectServerFacade.getSyncFacadeIfPossible().initialize(context, userAgent);

    if (context.getApplicationContext() != null) {
      BaseRealm.applicationContext = context.getApplicationContext();
    } else {
      BaseRealm.applicationContext = context;
    }
```

```
            OsSharedRealm.initialize(new File(context.getFilesDir(), ".realm.temp"));
        }
    }
```

The “new RealmConfiguration.Builder(context)” creates an instance of the Builder for the RealmConfiguration, which has a method called build(). The method build() creates the RealmConfiguration based on the builder parameters. Here the Builder receives only one parameter “context” so it builds RealmConfiguration according to the context.

```
Builder(Context context) {
    //noinspection ConstantConditions
    if (context == null) {
        throw new IllegalStateException("Call `Realm.init(Context)` before creating a
RealmConfiguration");
    }
    RealmCore.loadLibrary(context);
    initializeBuilder(context);
}
.

// Setups builder in its initial state.
private void initializeBuilder(Context context) {
    this.directory = context.getFilesDir();
    this.fileName = Realm.DEFAULT_REALM_NAME;
    this.key = null;
    this.schemaVersion = 0;
    this.migration = null;
    this.deleteRealmIfMigrationNeeded = false;
    this.durability = OsRealmConfig.Durability.FULL;
    this.readOnly = false;
    this.compactOnLaunch = null;
    if (DEFAULT_MODULE != null) {
            // !important
        this.modules.add(DEFAULT_MODULE);
    }
}
```

Now we know that from the context, the fields of Builder are set to their default values. Among them, the only field that needs to access context is “directory”, which is set to the “FilesDir” of context, and the other properties are independent of context. Finally, the property “modules” is modified. Let’s see what DEFAULT_MODULE is.

```
private static final Object DEFAULT_MODULE;
```

```
protected static final RealmProxyMediator DEFAULT_MODULE_MEDIATOR;

static {
  DEFAULT_MODULE = Realm.getDefaultModule(); // should NOT be null by default
  if (DEFAULT_MODULE != null) {
     final RealmProxyMediator mediator =
getModuleMediator(DEFAULT_MODULE.getClass().getCanonicalName());
     if (!mediator.transformerApplied()) {
        throw new ExceptionInInitializerError("RealmTransformer doesn't seem to be
applied." +
              " Please update the project configuration to use the Realm Gradle plugin." +
              " See https://realm.io/news/android-installation-change/");
     }
     DEFAULT_MODULE_MEDIATOR = mediator;
  } else {
     DEFAULT_MODULE_MEDIATOR = null;
  }
}
```

DEFAULT_MODULE and DEFAULT_MODULE_MEDIATOR are initialized by a static code block. DEFAULT_MODULE is set by “Realm.getDefaultModule()”. Let’s take a look inside.

```
/*
* Returns the default Realm module. This module contains all Realm classes in the
current project, but not those
* from library or project dependencies. Realm classes in these should be exposed using
their own module.
*
* @return the default Realm module or {@code null} if no default module exists.
* @throws RealmException if unable to create an instance of the module.
* @see io.realm.RealmConfiguration.Builder#modules(Object, Object...)
*/
@Nullable
public static Object getDefaultModule() {
  String moduleName = "io.realm.DefaultRealmModule";
  Class<?> clazz;
  //noinspection TryWithIdenticalCatches
  try {
     clazz = Class.forName(moduleName);
     Constructor<?> constructor = clazz.getDeclaredConstructors()[0];
     constructor.setAccessible(true);
     return constructor.newInstance();
  } catch (ClassNotFoundException e) {
```

```
            return null;
        } catch (InvocationTargetException e) {
            throw new RealmException("Could not create an instance of " + moduleName, e);
        } catch (InstantiationException e) {
            throw new RealmException("Could not create an instance of " + moduleName, e);
        } catch (IllegalAccessException e) {
            throw new RealmException("Could not create an instance of " + moduleName, e);
        }
    }
```

So here the code tries to dynamically load a class named "io.realm.DefaultRealmModule" (Java Reflection). However, such a class cannot be found in our current project because we are not yet ready to build it. But if we assume the default module will NOT be null when the code executes, then DEFAULT_MODULE should NOT be null by default. Next it is going to build DEFAULT_MODULE_MEDIATOR.

```
    // Finds the mediator associated with a given module.
    private static RealmProxyMediator getModuleMediator(String
    fullyQualifiedModuleClassName) {
        String[] moduleNameParts = fullyQualifiedModuleClassName.split("\\.");
        String moduleSimpleName = moduleNameParts[moduleNameParts.length - 1];
        String mediatorName = String.format(Locale.US, "io.realm.%s%s",
    moduleSimpleName, "Mediator");
        Class<?> clazz;
        //noinspection TryWithIdenticalCatches
        try {
            clazz = Class.forName(mediatorName);
            Constructor<?> constructor = clazz.getDeclaredConstructors()[0];
            constructor.setAccessible(true);
            return (RealmProxyMediator) constructor.newInstance();
        } catch (ClassNotFoundException e) {
            throw new RealmException("Could not find " + mediatorName, e);
        } catch (InvocationTargetException e) {
            throw new RealmException("Could not create an instance of " + mediatorName, e);
        } catch (InstantiationException e) {
            throw new RealmException("Could not create an instance of " + mediatorName, e);
        } catch (IllegalAccessException e) {
            throw new RealmException("Could not create an instance of " + mediatorName, e);
        }
    }
```

We may see getModuleMediator() is very similar to getDefaultModule(). They are both using Java Reflection. Now we have finished analysis of what “new RealmConfiguration.Builder(context)” does. Then we investigate what “build()” does.

```
/**
* Creates the RealmConfiguration based on the builder parameters.
*
* @return the created {@link RealmConfiguration}.
*/
public RealmConfiguration build() {
  // Check that readOnly() was applied to legal configuration. Right now it should only be allowed if
  // an assetFile is configured
  if (readOnly) {
      if (initialDataTransaction != null) {
          throw new IllegalStateException("This Realm is marked as read-only. Read-only Realms cannot use initialData(Realm.Transaction).");
      }
      if (assetFilePath == null) {
          throw new IllegalStateException("Only Realms provided using 'assetFile(path)' can be marked read-only. No such Realm was provided.");
      }
      if (deleteRealmIfMigrationNeeded) {
          throw new IllegalStateException("'deleteRealmIfMigrationNeeded()' and read-only Realms cannot be combined");
      }
      if (compactOnLaunch != null) {
          throw new IllegalStateException("'compactOnLaunch()' and read-only Realms cannot be combined");
      }
  }

  if (rxFactory == null && isRxJavaAvailable()) {
      rxFactory = new RealmObservableFactory();
  }


  return new RealmConfiguration(directory,
          fileName,
          getCanonicalPath(new File(directory, fileName)),
          assetFilePath,
          key,
          schemaVersion,
```

```
            migration,
            deleteRealmIfMigrationNeeded,
            durability,
            createSchemaMediator(modules, debugSchema),
            rxFactory,
            initialDataTransaction,
            readOnly,
            compactOnLaunch,
            false
    );
}
```

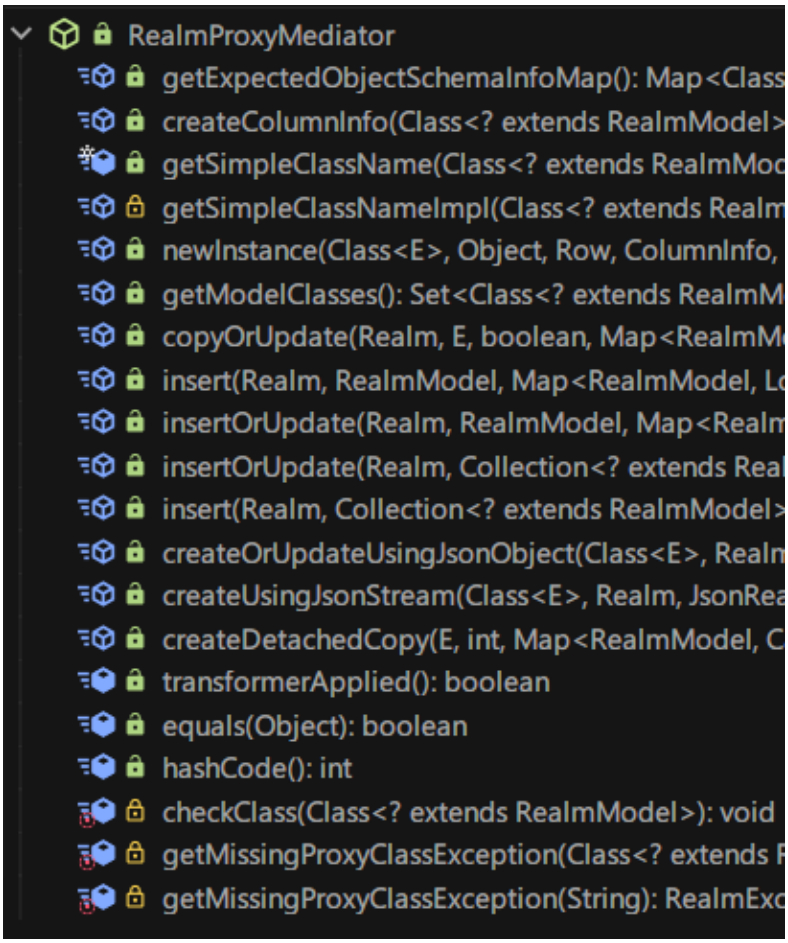
So basically “build()” creates a new RealmConfiguration object. Firstly the function does an input validation. Then it constructs and returns a RealmConfiguration. Here we encountered a new function “createSchemaMediator(modules, debugSchema)”.

```
// Creates the mediator that defines the current schema.
protected static RealmProxyMediator createSchemaMediator(Set<Object> modules,
        Set<Class<? extends RealmModel>> debugSchema) {

    // If using debug schema, uses special mediator.
    if (debugSchema.size() > 0) {
        return new FilterableMediator(DEFAULT_MODULE_MEDIATOR, debugSchema);
    }

    // If only one module, uses that mediator directly.
    if (modules.size() == 1) {
        return
getModuleMediator(modules.iterator().next().getClass().getCanonicalName());
    }

    // Otherwise combines all mediators.
    RealmProxyMediator[] mediators = new RealmProxyMediator[modules.size()];
    int i = 0;
    for (Object module : modules) {
        mediators[i] = getModuleMediator(module.getClass().getCanonicalName());
        i++;
    }
    return new CompositeMediator(mediators);
}
```

“createSchemaMediator()” creates a RealmProxyMediator object. In this case, we have only one module which is DEFAULT_MODULE (DefaultRealmModule). So we will get a module mediator for the DEFAULT_MODULE (DefaultRealmModuleMediator).

Limited by time we have no access to DefaultRealmModuleMediator. Navigate to its superclass realm\realm-library\src\main\java\io\realm\internal\RealmProxyMediator.java, and look into its structure. There we have found it defines some basic operations for the database, like insert(), insertOrUpdate(). So we assume DefaultRealmModuleMediator must have realized these operations.

Now we have gone through the whole process of the initialization.



Mental Model

Folder	File	Method	Relevant?	Relevant How?	Confidence	Note

realm/realm-library/src/main/java/io/realm	Realm.java	init	Yes	Name match	100%	
realm/realm-library/src/main/java/io/realm	Realm.java	initializeRealm	Yes	The only function called init.	100%	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	Builder(context)	Yes	We need to set a default configuration for a new realm	100%	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	initializeBuilder	Yes	The name indicates that this method does initiate the builder	100%	This method configure the all default properties for the builder.
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	DEFAULT_MODULE	Yes	initializeBuilder needs to check DEFAULT_MODULE and we need to know what is a DEFAULT_MODULE.	80%	
realm/realm-library/src/main/java/io/realm	Realm.java	getDefaultModule	Yes	DEFAULT_MODULE is created by getDefaultModule.	100%	It returns the default Realm module.
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	getModuleMediator	Yes	A mediator is created by calling getModuleMediator method.	80%	It returns the default Realm mediator.
realm/realm-library/src/main/java/io/realm/internal	RealmProxyMediator.java	RealmProxyMediator	Yes	RealmProxyMediator is the father class of getModuleMediator, so we might need to check out how it works.	80%	Superclass for the RealmProxy Mediator class. This class contains all static methods introduced by the annotation, processor as part of the RealmProxy classes.
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	build	Yes	This is called after Builder()	100%	It returns a RealmConfiguration instance.

realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	createSchemaMediator	Yes	RealmConfiguration is one of the parameters of RealmConfiguration.	80%	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	RealmConfiguration	Yes	It is returned by build() and it is a configuration we need.	100%	RealmConfiguration creates the RealmConfiguration based on the builder parameters.

Folder	File	Method	Why?	Priority	Notes
realm/realm-library/src/main/java/io/realm	Realm.java	initializeRealm	The only function called by init.	1	
realm/realm-library/src/main/java/io/realm	Realm.java	setDefaultConfiguration	The method helps set the default configuration for the Builder.	2	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	Builder(context)	It creates a Builder and pass it to set default configuration.	1	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	RealmCore.loadLibrary(context);	It is called in the Builder method.	3	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	initializeBuilder	This method setup builder in its initial state.	1	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	DEFAULT_MODULE	The method name includes default which is what we are looking for.	1	
realm/realm-library/src/main/java/io/realm	Realm.java	getDefaultModule	It returns the default Realm module.	1	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	getModuleMediator	getModuleMediator finds the mediator associated with a given module.	2	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	RealmConfiguration	RealmConfiguration creates the RealmConfiguration based on the builder parameters.	1	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	createSchemaMediator	It creates the mediator that defines the current schema.	1	
realm/realm-library/src/main/java/io/realm	RealmConfiguration.java	getModuleMediator	It finds the mediator associated with a given module.	1	

Junxian Chen, Wen Chia Yang, Zihua Weng

Feature 2: Creating Tables

Background

In the official sample code, it begins with creating RealmObject subclasses:

```
// Define your model class by extending RealmObject
public class Dog extends RealmObject {
    private String name;
    private int age;

    // ... Generated getters and setters ...
}

// Use them like regular java objects
Dog dog = new Dog();
dog.setName("Rex");
dog.setAge(1);

// … realm init ...

final Dog managedDog = realm.copyToRealm(dog); // Persist unmanaged objects

public class Person extends RealmObject {
    @PrimaryKey
    private long id;
    private String name;
    private RealmList<Dog> dogs; // Declare one-to-many relationships

    // ... Generated getters and setters ...
}

Person person = realm.createObject(Person.class); // Create managed objects directly
```

So a RealmObject can be seen as a model here, where its fields represent the columns of the table that will be created later. Then by calling “Dog dog = new Dog()” we create a record for the Dog table, and we can set attributes for the record. What gets us interested is how Realm makes RealmObject persist, in other words, how Realm creates a table.

Research

Let's first take a look at RealmObject. RealmObject is annotated with "@RealmClass" which may be used during the process of code generation. Furthermore, RealmObject implements two interfaces, RealmModel and ManagableObject.

```
/*
In Realm you define your RealmObject classes by sub-classing RealmObject and adding
fields to be persisted. You then create your objects within a Realm, and use your custom
subclasses instead of using the RealmObject class directly.

An annotation processor will create a proxy class for your RealmObject subclass.
...
*/

@RealmClass
public abstract class RealmObject implements RealmModel, ManagableObject { … }
```

Next, clients can create a RealmObject and make it be managed by Realm through Realm.createObject. In this step, the custom RealmObject class would be bound to Realm and also the related table would be created.

The following is the method 'createObject'. Basically, it will check the running Realm environment first, and then just call another method createObjectInternal to actually initialize tables and bound object(RealObject).

```
public <E extends RealmModel> E createObject(Class<E> clazz) {
  checkIfValid();
  return createObjectInternal(clazz, true, Collections.<String>emptyList());
}

<E extends RealmModel> E createObjectInternal(
      Class<E> clazz,
      boolean acceptDefaultValue,
      List<String> excludeFields) {
            Table table = schema.getTable(clazz);
            // Checks and throws the exception earlier for a better exception message.
            if (OsObjectStore.getPrimaryKeyForObject(
                  sharedRealm,
          configuration.getSchemaMediator().getSimpleClassName(clazz)) != null) {
                throw new RealmException(String.format(Locale.US, "'%s' has a primary
          key, use" + " 'createObject(Class<E>, Object)' instead.", table.getClassName()));
```

```
        }
        return configuration.getSchemaMediator().newInstance(clazz, this,
                OsObject.create(table),
                schema.getColumnInfo(clazz),
                acceptDefaultValue, excludeFields);
    }
```

The called method createObjectInternal will then called from the proxy class as mentioned in the comments. createObjectInternal does the following two things: first, it creates a table by calling getTable and then returns a new mediator object using the table object created.

Next, we need to keep track of the object Table which is returned from getTable. It turns out that Table is a base class for all Realm tables and it supports all low level methods a table has. That means by returning a Table instance, we could create a table as required.

Going back to the getTable method, we found out that, in cases where we need to create the first table, it will return a Table object created by the other method also called getTable from OsSharedRealm.java. And for cases where we already have table in cache, then the method returns the existing table by matching table name.

```
    Table getTable(String className) {
        String tableName = Table.getTableNameForClass(className);
        Table table = dynamicClassToTable.get(tableName);
        if (table != null) { return table; }

        table = realm.getSharedRealm().getTable(tableName);
        dynamicClassToTable.put(tableName, table);

        return table;
    }
```

Our goal is to find out how realm creates new tables, so we need to check out how getTable from OsSharedRealm.java works.

```
    public Table getTable(String name) {
        long tablePtr = nativeGetTable(nativePtr, name);
        return new Table(this, tablePtr);
    }
```

The code indicates that it returns a new Table instance and names the table with a given name (String). And the table is linked with realm native database by calling nativeGetTable(). So far, we have found the essential code how realm creates a table for the first time.

Mental Model

Folder	File	Method	Relevant?	Relevant How?	Confidence	Note
realm/realm-library/src/main/java/io/realm	Realm.java	createObject	yes	keyword create	80%	not sure what object is
realm/realm-library/src/main/java/io/realm	Realm.java	createObjectInternal	yes	function inside the createObject method	100%	
realm/realm-library/src/main/java/io/realm	RealmObject.java	n/a	yes	When creating objects, the function needs to take a RealmObject class as an input	100%	every RealmObject acts as a model in Realm
realm/realm-library/src/main/java/io/realm	BaseRealm.java	checkIfValid	no	not sure, but was called in createObject	60%	Checks if a Realm's underlying resources are still available or not getting accessed from the wrong thread.
realm/realm-library/src/main/java/io/realm/internal	Table.java	n/a	yes	An object for representing database table	100%	table object
realm/realm-library/src/main/java/io/realm	RealmSchema.java	getTable	yes	getTable is called in createObjectInternal	100%	init table if not exist or return existed table
realm/realm-library/src/main/java/io/realm	BaseRealm.java	getSharedRealm	yes	schema is used in createObjectInternal	100%	get sharedRealm object

Junxian Chen, Wen Chia Yang, Zihua Weng

Features of realm-java

realm/realm-library/src/main/java/io/realm/internal	OsSharedRealm.java	getTable	yes	getTable is called in getTable(RealmSchema.java)	100%	link table and realm native database

Folder	File	Method	Why?	Priority	Notes
realm/realm-library/src/main/java/io/realm	Realm.java	createObjectInternal	one of two methods call in createObject()	1	Need to find how operations in this method
realm/realm-library/src/main/java/io/realm	BaseRealm.java	checkIfValid	one of two methods call in createObject()	3	Just a check method that is irrelevant to actual creation of tables
realm/realm-library/src/main/java/io/realm/internal	Table.java	n/a	An object for representing database table	1	an important info
realm/realm-library/src/main/java/io/realm	RealmSchema.java	getTable	getTable is called in createObjectInternal	1	called by layers
realm/realm-library/src/main/java/io/realm	BaseRealm.java	getSharedRealm	schema is used in createObjectInternal	1	called by layers
realm/realm-library/src/main/java/io/realm/internal	OsSharedRealm.java	getTable	getTable is called in getTable(RealmSchema.java)	1	called by layers

Junxian Chen, Wen Chia Yang, Zihua Weng
Jan. 29, 2020