Appendix

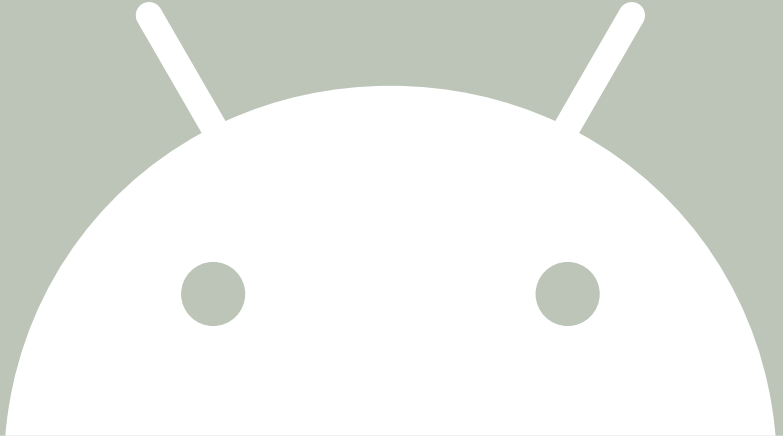
1

追加Step

本記事は『作って学ぶAndroidアプリ開発［Kotlin対応］』（有山圭二 著、技術評論社 刊、ISBN 978-4-297-11343-8）の付属記事です。

本書に掲載できなかったStepを公開します。本書のStep 46以降として、Step 47（選択した画像のプレビューを表示する）からStep 53（Activity/Fragmentの再生成に対応する）までになります。

ぜひ、本書をさらに活用するために挑戦してみてください。



Step 47　選択した画像のプレビューを表示する

画像の添付ができるようになりましたが、今の実装では表示が変化しないので、画像が添付されていることをユーザーがわからない状態です。そこで、投稿画面に選択した画像のプレビューを表示します。

画像のプレビュー表示は、次の手順で行います。

・BindingAdapter を追加
・プレビュー表示用のレイアウトを追加
・投稿画面のレイアウトにプレビュー表示領域を追加
・プレビュー表示用の Adapter を作成
・プレビューの表示

BindingAdapterを追加

.ui.DataBindingAdapter.kt を**リスト 47.1** のように変更します。

○リスト 47.1：.ui.DataBindingAdapter.kt

```
     import com.bumptech.glide.Glide
+    import io.keiji.sample.mastodonclient.entity.LocalMedia
     import io.keiji.sample.mastodonclient.entity.Media

     @BindingAdapter("media")
     fun ImageView.setMedia(media: Media?) {
         // 省略
     }

+    @BindingAdapter("localMedia")
+    fun ImageView.setLocalMedia(localMedia: LocalMedia?) {
+        if (localMedia == null) {
+            setImageDrawable(null)
+            return
+        }
+        Glide.with(this)
+            .load(localMedia.file)
+            .into(this)
+    }

     @BindingAdapter("spannedContent")
     fun TextView.setSpannedString(content: String) {
         text = HtmlCompat.fromHtml(content, HtmlCompat.FROM_HTML_MODE_COMPACT)
     }
```

プレビュー表示用レイアウトを追加

レイアウトリソース list_item_media_preview.xml を作成して、**リスト 47.2** のようにします。

○リスト47.2：res/layout/list_item_media_preview.xml

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">
    <data>
        <variable
                name="localMedia"
                type="io.keiji.sample.mastodonclient.entity.LocalMedia" />
    </data>

    <androidx.appcompat.widget.AppCompatImageView
            android:id="@+id/image"
            android:layout_width="wrap_content"
            android:layout_height="128dp"
            android:layout_marginStart="8dp"
            android:adjustViewBounds="true"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="parent"
            app:localMedia="@{localMedia}" />

</layout>
```

投稿画面のレイアウトにプレビュー表示領域を追加

レイアウトリソース`fragment_toot_edit.xml`を作成して、**リスト47.3**のようにします。

○リスト47.3：res/layout/fragment_toot_edit.xml

```
<?xml version="1.0" encoding="utf−8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res−auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>

        <variable
            name="viewModel"
            type="io.keiji.sample.mastodonclient.ui.toot_edit.TootEditViewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <com.google.android.material.textfield.TextInputEditText
            android:id="@+id/status"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:hint="いまなにをしていますか"
            android:text="@={viewModel.status}"
−           app:layout_constraintBottom_toTopOf="@+id/add_media"
+           app:layout_constraintBottom_toTopOf="@+id/media_preview"
```

```
+            app:layout_constraintTop_toTopOf="parent" />

+        <androidx.recyclerview.widget.RecyclerView
+            android:id="@+id/media_preview"
+            android:layout_width="match_parent"
+            android:layout_height="100dp"
+            app:layout_constraintBottom_toTopOf="@+id/add_media"
+            tools:listitem="@layout/list_item_media" />

        <androidx.appcompat.widget.AppCompatImageButton
            style="@style/Widget.AppCompat.Button.Borderless"
            android:id="@+id/add_media"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/baseline_add_to_photos_black_24"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintStart_toStartOf="parent" />

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

このレイアウトをデザインビューで見ると**図47.1**のようになります。

○**図47.1**：

プレビュー表示用のAdapterを作成

パッケージ.ui.toot_editにクラスMediaPreviewAdapterを作成して**リスト47.4**のようにします。このクラスは.ui.toot_list.TootListAdapterに似ています。TootListAdapterはTootを取り扱いますが、MediaPreviewAdapterはLocalMediaを取り扱う点で異なります。

○リスト47.4：.ui.toot_edit.MediaPreviewAdapter

```
package io.keiji.sample.mastodonclient.ui.toot_edit

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.databinding.DataBindingUtil
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.RecyclerView
import io.keiji.sample.mastodonclient.R
import io.keiji.sample.mastodonclient.databinding.ListItemMediaPreviewBinding
import io.keiji.sample.mastodonclient.entity.LocalMedia
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext

class MediaPreviewAdapter(
    private val inflater: LayoutInflater,
    private val coroutineScope: CoroutineScope
) : RecyclerView.Adapter<MediaPreviewAdapter.ViewHolder>() {

    private class RecyclerDiffCallback(
        private val oldList: List<LocalMedia>,
        private val newList: List<LocalMedia>
    ) : DiffUtil.Callback() {

        override fun getOldListSize() = oldList.size
        override fun getNewListSize() = newList.size

        override fun areItemsTheSame(
            oldItemPosition: Int,
            newItemPosition: Int
        ) = oldList[oldItemPosition] == newList[newItemPosition]

        override fun areContentsTheSame(
            oldItemPosition: Int,
            newItemPosition: Int
        ): Boolean {
            val oFilePath = oldList[oldItemPosition].file.absolutePath
            val nFilePath = newList[newItemPosition].file.absolutePath
            return oFilePath == nFilePath
        }
    }

    var mediaAttachments = ArrayList<LocalMedia>()
        set(value) {
            coroutineScope.launch(Dispatchers.Main) {
                val diffResult = withContext(Dispatchers.Default) {
                    DiffUtil.calculateDiff(
                        RecyclerDiffCallback(field, value)
                    )
                }

                field = value

                diffResult.dispatchUpdatesTo(this@MediaPreviewAdapter)
            }
        }
```

①

```
    override fun getItemCount() = mediaAttachments.size

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): ViewHolder {
        val binding = DataBindingUtil.inflate<ListItemMediaPreviewBinding>(
            inflater,
            R.layout.list_item_media_preview,
            parent,
            false
        )
        return ViewHolder(binding)
    }

    override fun onBindViewHolder(
        holder: ViewHolder,
        position: Int
    ) {
        holder.bind(mediaAttachments[position])
    }

    class ViewHolder(
        private val binding: ListItemMediaPreviewBinding
    ) : RecyclerView.ViewHolder(binding.root) {

        fun bind(localMedia: LocalMedia) {
            binding.localMedia = localMedia
        }
    }
}
```

① 要素の「内容」が同じか判定する基準は、指し示すファイルの絶対パス

プレビューの表示

.ui.toot_edit.TootEditFragment を**リスト 47.5** のように変更します。

○リスト 47.5：.ui.toot_edit.TootEditFragment

```
    import androidx.lifecycle.lifecycleScope
+   import androidx.recyclerview.widget.LinearLayoutManager
    import com.google.android.material.snackbar.Snackbar
    // 省略

    class TootEditFragment : Fragment(R.layout.fragment_toot_edit) {

        // 省略

        override fun onAttach(context: Context) {
            // 省略
        }

+       private lateinit var adapter: MediaPreviewAdapter
```

```
+       private lateinit var layoutManager: LinearLayoutManager

        override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
            // 省略

            bindingData.lifecycleOwner = viewLifecycleOwner
            bindingData.viewModel = viewModel

+           adapter = MediaPreviewAdapter(layoutInflater, lifecycleScope)
+           layoutManager = LinearLayoutManager(requireContext(),
+               LinearLayoutManager.HORIZONTAL,
+               false
+           )

+           bindingData.mediaPreview.also {
+               it.layoutManager = layoutManager
+               it.adapter = adapter
+           }
            bindingData.addMedia.setOnClickListener {
                openMediaChooser()
            }

            viewModel.loginRequired.observe(viewLifecycleOwner, Observer {
                // 省略
            })
+           viewModel.mediaAttachments.observe(viewLifecycleOwner, Observer {
+               adapter.mediaAttachments = it
+           })
            viewModel.postComplete.observe(viewLifecycleOwner, Observer {
                Toast.makeText(requireContext(), "投稿完了しました", Toast.LENGTH_LONG).
                show()
                callback?.onPostComplete()
            })

            // 省略
        }
```

① 添付画像が変更されると、差分計算と更新を実行

Step 48 バックグラウンドで投稿する

メディアを添付して投稿できるようになったことで、1つの問題が生まれます。文章（テキスト）と比べて画像はデータ量が大きいので、投稿完了までの時間がかかります。

また、投稿完了前に画面を終了すると投稿処理がキャンセルされるため、ユーザーは投稿完了まで待つ必要があります。操作ができない待ち時間はユーザーにストレスを与えることになります。

投稿編集画面では添付するメディアと投稿内容を作成するだけに留めます。メディアと投稿内容をストレージに保存して、実際の投稿処理はバックグラウンドで実行します。

バックグラウンドでの投稿は、次の手順で行います。

・ライブラリの追加（Room）
・Entity クラスの定義
・Dao クラスの定義
・RoomDatabase の定義
・データベースの初期化
・Repository の作成
・サービス（Service）によるバックグラウンド処理
・投稿キューの追加
・サービスの起動

ライブラリの追加（room）

投稿内容の保存にはデータベースを使います。前に紹介したSharedPreferencesは、複数のデータ（レコード）を記録することに適していないためです。

Androidには、軽量型データベースのSQLiteが組み込まれています。近年はSQLite向けにさまざまなORM（Object-relational mapping）の仕組みが登場し、直接SQLiteを使う機会は減少しています。

本書では米Google社がオープンソースで開発しているORM「Room」を使ってデータベースを構築します。build.gradleを開いて、dependenciesにRoomを追加します（**リスト48.1**）。

○リスト48.1：app/build.gradle

```
    dependencies {
        // 省略
        implementation "com.google.android.material:material:1.1.0"
+       implementation "androidx.room:room-runtime:2.2.5"
+       implementation "androidx.room:room-ktx:2.2.5"          ①
+       kapt "androidx.room:room-compiler:2.2.5"
    }
```

① roomと拡張ライブラリ(room-ktx)を追加する。バージョンは本書執筆時点で最新の2.2.5を指定する

Entityクラスの作成

次にデータベースに記録するデータをEntityクラスとして定義します。「投稿内容」と「添付するメディア情報」をそれぞれキュー（Queue）としてデータベースに記録します。

パッケージ.db.entityに、クラスPostTootQueueを作成して、**リスト48.2**のようにします。

○リスト48.2：.db.entity.PostTootQueue

```
package io.keiji.sample.mastodonclient.db.entity

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity    ①
data class PostTootQueue(
    val instanceUrl: String,
    val username: String,                    ②
    val status: String
) {
    @PrimaryKey(autoGenerate=true)           ③
    var id: Long = 0
}
```

① データベースのテーブルであることを示す
② プロパティはテーブルのカラムに対応
③ 自動的に採番されるID

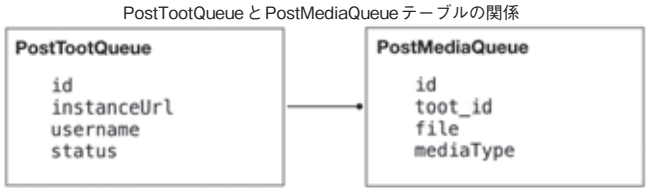
次に、パッケージ.db.entityにクラスPostMediaQueueを作成して、**リスト48.3**のようにします。

○リスト 48.3：.db.entity.PostMediaQueue

```
package io.keiji.sample.mastodonclient.db.entity

import androidx.room.Entity
import androidx.room.ForeignKey
import androidx.room.PrimaryKey

@Entity(
    foreignKeys = [ForeignKey(
        entity = PostTootQueue::class,
        parentColumns = arrayOf("id"),
        childColumns = arrayOf("tootId"),
        onDelete = ForeignKey.CASCADE
    )]
)
data class PostMediaQueue(
    val tootId: Long,
    val file: String,
    val mediaType: String
) {
    @PrimaryKey(autoGenerate = true)
    var id: Long = 0

    var mediaId: String? = null
}
```

1つの「投稿内容」に複数の「添付するメディア情報」が存在する可能性があるため、2つのテーブルを関連付けます（**図 48.1**）。

○図 48.1：



Room ではキーの参照関係を foreignKeys（外部キー）として設定します。対象の Entity の parentColumns で示すカラムと、childColumns で示すカラムを紐付けます。

今回の場合 foreignKeys を設定することで、PostMediaQueue に設定する tootId と、同じ値の id を持つ PostTootQueue が存在するかをチェックできます。また、PostTootQueue を削除すると（onDelete）、そのレコードと紐付いているすべての PostMediaQueue が連鎖（CASCADE）して削除されます。

DAO クラスの定義

DAO（Data Access Object）は、テーブルのデータにアクセスする手段を提供します。

パッケージ.db.daoにクラスPostTootQueueDaoを作成して、**リスト48.4**のようにします。

○リスト48.4：.db.dao.PostTootQueueDao

```
package io.keiji.sample.mastodonclient.db.dao

import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import io.keiji.sample.mastodonclient.db.entity.PostTootQueue

@Dao
interface PostTootQueueDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun upsert(data: PostTootQueue): Long

    @Query("SELECT * FROM PostTootQueue")
    suspend fun all(): List<PostTootQueue>

    @Delete
    suspend fun delete(data: PostTootQueue)
}
```

① PostTootQueueを取り扱うDAOの定義
② upsertは、UpdateとInsertを組み合わせた造語。同じPrimaryKeyのレコードが存在すれば更新（Update）、なければ新規追加（Insert）
③ すべてのレコードを取得
④ レコードを1つ削除

パッケージ.db.daoにクラスPostMediaQueueDaoを作成して、**リスト48.5**のようにします。

○リスト 48.5：.db.dao.PostMediaQueueDao

```
package io.keiji.sample.mastodonclient.db.dao

import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import io.keiji.sample.mastodonclient.db.entity.PostMediaQueue

@Dao
interface PostMediaQueueDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun upsert(data: PostMediaQueue): Long

    @Query("SELECT * FROM PostMediaQueue WHERE tootId = :tootId")
    suspend fun findByTootId(tootId: Long): List<PostMediaQueue>

    @Delete
    suspend fun delete(data: PostMediaQueue)
}
```

① 引数の tootId と一致する値をもつレコード（投稿に添付された画像）を検索

RoomDatabase の定義

パッケージ .db にクラス AppDatabase を作成して、**リスト 48.6** のようにします。

○リスト48.6：.db.AppDatabase

```
package io.keiji.sample.mastodonclient.db

import androidx.room.Database
import androidx.room.RoomDatabase
import io.keiji.sample.mastodonclient.db.dao.PostMediaQueueDao
import io.keiji.sample.mastodonclient.db.dao.PostTootQueueDao
import io.keiji.sample.mastodonclient.db.entity.PostMediaQueue
import io.keiji.sample.mastodonclient.db.entity.PostTootQueue

@Database(
    version = 1,
    entities = [                    ┐
        PostTootQueue::class,       ├ ①
        PostMediaQueue::class       ┘
    ])
abstract class AppDatabase : RoomDatabase() {

    abstract fun postTootQueue(): PostTootQueueDao     ②

    abstract fun postMediaQueue(): PostMediaQueueDao     ③
}
```

① Roomで管理対象にするクラスのエントリー。Entityアノテーションを付けていても、ここに設定していなければ利用できない

② PostTootQueueDaoの取得

③ PostMediaQueueDaoの取得

データベースの初期化

アプリの起動時にデータベースの初期化処理を行います。アプリの起動タイミングで処理を行うには、Applicationを継承したクラスを作成します。

パッケージのトップ（io.keiji.sample.mastodonclient）にクラス`MyApplication`を作成して、**リスト48.7**のようにします。

○リスト48.7：.MyApplication

```
package io.keiji.sample.mastodonclient

import android.app.Application
import android.util.Log
import androidx.room.Room
import io.keiji.sample.mastodonclient.db.AppDatabase

class MyApplication : Application() {    ①

    companion object {
                                                                    ↗
```

```
        private val TAG = MyApplication::class.java.simpleName

        private const val DB_FILE_NAME = "room.db"    ②

        private var appDatabase: AppDatabase? = null    ③

        @JvmStatic
        fun getAppDatabase(application: Application): AppDatabase {    ④
            appDatabase?.also {
                return it                                              ⑤
            }

            return initDatabase(application).also {
                appDatabase = it                                       ⑥
            }
        }

        @JvmStatic
        private fun initDatabase(application: Application): AppDatabase {
            return Room.databaseBuilder(
                application,
                AppDatabase::class.java,
                DB_FILE_NAME
            ).build()
        }
    }

    override fun onCreate() {
        super.onCreate()
        Log.d(TAG, "Application is created.")                          ⑦

        initDatabase(this)
    }
}
```

① Applicationクラスを継承
② Roomが作成するデータベースのファイル名
③ Companion Objectが保持するデータベースのインスタンス
④ データベースのインスタンスを取得（他のクラスに渡す）
⑤ 既存のインスタンスがあれば返却
⑥ データベースのインスタンスを生成してCompanion Objectに保持
⑦ アプリケーションの起動時に一度だけ実行

アプリの起動時にApplicationクラスを呼び出すために、AndroidManifest.xmlをリスト48.8のように変更します。

○リスト 48.8：app/src/main/AndroidManifest.xml

```
    <application
+       android:name=".MyApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
```

変更後アプリを起動するとLogcatにログが出力されます（**リスト 48.9**）。MyApplicationクラスのonCreateが実行されていることが確認できます。

○リスト 48.9：アプリ起動時のLogcat

```
io.keiji.sample.mastodonclient D/MyApplication: Application is created.
```

Repositoryの作成

PostTootQueueとPostMediaQueueを取り扱うRepotitoryクラスを作成します。それぞれにRepositoryを作成せず、2つのエンティティをひとまとめに取り扱います。

パッケージ.repositoryにクラスPostQueueRepositoryを作成して、**リスト 48.10**のようにします。

○リスト 48.10：.repository.PostQueueRepository

```
package io.keiji.sample.mastodonclient.repository

import android.app.Application
import io.keiji.sample.mastodonclient.MyApplication
import io.keiji.sample.mastodonclient.db.entity.PostMediaQueue
import io.keiji.sample.mastodonclient.db.entity.PostTootQueue
import io.keiji.sample.mastodonclient.entity.LocalMedia
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext

class PostTootQueueRepository(application: Application) {

    private val appDatabase = MyApplication.getAppDatabase(application)     ①

    suspend fun addQueue(
        instanceUrl: String,
        username: String,
        status: String,
        mediaAttachments: ArrayList<LocalMedia>?
```

```
    ) = withContext(Dispatchers.IO) {
        val tootQueue = PostTootQueue(
            instanceUrl,
            username,
            status
        )

        val tootId = appDatabase
            .postTootQueue()
            .upsert(tootQueue)

        val postMediaQueueDao = appDatabase.postMediaQueue()
        mediaAttachments?.forEach {
            val mediaQueue = PostMediaQueue(
                tootId,
                it.file.absolutePath,
                it.mediaType
            )
            postMediaQueueDao.upsert(mediaQueue)
        }

        return@withContext tootQueue
    }

    suspend fun delete(
        postTootQueue: PostTootQueue
    ) = withContext(Dispatchers.IO) {
        appDatabase
            .postTootQueue()
            .delete(postTootQueue)
    }

    suspend fun upsert(
        postMediaQueue: PostMediaQueue
    ) = withContext(Dispatchers.IO) {
        appDatabase
            .postMediaQueue()
            .upsert(postMediaQueue)
    }

    suspend fun allTootQueues(
    ): List<PostTootQueue> = withContext(Dispatchers.IO) {
        return@withContext appDatabase
            .postTootQueue()
            .all()
    }

    suspend fun findMediasByTootId(
        tootId: Long
    ): List<PostMediaQueue> = withContext(Dispatchers.IO) {
        return@withContext appDatabase
            .postMediaQueue()
            .findByTootId(tootId)
    }
}
```

① Roomのデータベースインスタンスを取得
② 投稿と投稿に添付されているメディアをキューに追加
③ 投稿を投稿キューに追加して投稿キューのIDを取得
④ 添付メディアに投稿キューのIDを設定して追加
⑤ 投稿キューを削除
⑥ 添付メディアを追加。すでに存在すれば更新
⑦ 保存されている投稿キューをすべて取得
⑧ 投稿に添付されたメディアのリストを取得行

サービス（Service）によるバックグラウンド処理

バックグラウンド処理を実行するServiceを追加します。

Serviceは、Androidのシステムコンポーネントの1つで、バックグランドでの処理を担当します。画面が終了して、アプリが表示されなくなっても、Serviceは継続して処理を実行できます。

Serviceには、常駐して処理を実行する「Service」と、単発でバックグラウンド処理を実行する（実行後に終了する）「IntentService」があります。今回はIntentServiceで投稿を実行します。

パッケージ.serviceにクラスPostTootServiceを作成して、**リスト48.11**のようにします。

○リスト48.11：.service.PostTootService

```
package io.keiji.sample.mastodonclient.service

import android.app.IntentService
import android.content.Intent
import android.util.Log
import io.keiji.sample.mastodonclient.db.entity.PostMediaQueue
import io.keiji.sample.mastodonclient.db.entity.PostTootQueue
import io.keiji.sample.mastodonclient.repository.PostTootQueueRepository
import io.keiji.sample.mastodonclient.repository.TootRepository
import io.keiji.sample.mastodonclient.repository.UserCredentialRepository
import kotlinx.coroutines.runBlocking
import retrofit2.HttpException
import java.io.File
import java.io.IOException
import java.net.HttpURLConnection

class PostTootService : IntentService(PostTootService::class.java.simpleName) {     ①
    companion object {
        private val TAG = PostTootService::class.java.simpleName
    }
```

```
override fun onHandleIntent(intent: Intent?) {
    val postTootQueueRepository = PostTootQueueRepository(application)
    val userCredentialRepository = UserCredentialRepository(application)

    runBlocking {
        postTootQueueRepository
            .allTootQueues()
            .forEach { queue ->
                try {
                    postToot(
                        queue,
                        postTootQueueRepository,
                        userCredentialRepository
                    )
                } catch (e: HttpException) {
                    handleException(e)
                } catch (e: IOException) {
                    handleException(e)
                }
            }
    }
}

private suspend fun postToot(
    queue: PostTootQueue,
    postTootQueueRepository: PostTootQueueRepository,
    userCredentialRepository: UserCredentialRepository
) {
    val userCredential = userCredentialRepository.find(
        queue.instanceUrl,
        queue.username
    ) ?: return

    val tootRepository = TootRepository(userCredential)

    val medias = postTootQueueRepository
        .findMediasByTootId(queue.id)

    val mediaIds = uploadMedias(
        medias,
        tootRepository,
        postTootQueueRepository
    )

    tootRepository.postToot(
        queue.status,
        mediaIds
    )
    postTootQueueRepository.delete(queue)
}

private suspend fun uploadMedias(
    medias: List<PostMediaQueue>,
    tootRepository: TootRepository,
```

② ③ ④ ⑤ ⑥ ⑦

```
↘
        postTootQueueRepository: PostTootQueueRepository
    ): List<String> {
        return medias.mapNotNull { mediaQueue ->
            if (mediaQueue.mediaId != null) {
                return@mapNotNull mediaQueue.mediaId
            }                                                    ⑧

            val file = File(mediaQueue.file)
            val media = tootRepository.postMedia(
                file,                                            ⑨
                mediaQueue.mediaType
            )
            mediaQueue.mediaId = media.id                        ⑩
            postTootQueueRepository.upsert(mediaQueue)
            file.deleteOnExit()     ⑪

            return@mapNotNull mediaQueue.mediaId
        }
    }

    private fun handleException(e: HttpException) {
        val message = when (e.code()) {
            HttpURLConnection.HTTP_FORBIDDEN -> "権限がありません: ${e.message}"
            else -> "不明なエラーです ${e.message}"
        }
        Log.e(TAG, message)
    }

    private fun handleException(e: IOException) {
        Log.e(TAG, "IOException", e)
    }
}
```

① IntentServiceクラスを継承する
② すべての投稿キューについて1つずつ投稿処理を実行。中断関数を実行するためrunBlocking内に記述
③ 投稿キューに指定されたインスタンス・ユーザー名の認可情報を取得
④ 投稿キューに添付されたメディアキューのリストを取得
⑤ すべての添付メディアをアップロードして、メディアIDのリストとして取得
⑥ 投稿を実行
⑦ 投稿キューを削除（投稿キューに関連した添付メディアのキューも削除される）
⑧ レコードにメディアIDが保存されている（アップロード済みの）添付メディアはアップロードしない
⑨ メディアのアップロード処理
⑩ アップロードが完了して得られるメディアIDをメディアキューに記録（重複アップロードの防止）
⑪ アップロードが完了したメディアのファイルを削除

▶ AndroidManifest.xmlへの登録

Serviceは、Androidのシステムコンポーネントの1つです。使用するにはAndroid Manifest.xmlに登録している必要があります。

AndroidManifest.xmlを**リスト48.12**のように変更します。

○リスト48.12：app/src/main/AndroidManifest.xml

```
        <activity
            android:name=".ui.login.LoginActivity"
            android:launchMode="singleTask"
            android:windowSoftInputMode="adjustPan">

            <!-- 省略 -->

        </activity>

+       <service
+           android:name=".service.PostTootService"
+           android:exported="false" />
    </application>
```

投稿キューの追加

バックグラウンドで投稿処理をするサービスを追加したので、TootEditViewModelの処理を調整します。調整後のTootEditViewModelでは投稿とメディアのキューをデータベースに追加して、投稿処理は実行しません。

.ui.toot_edit.TootEditViewModelを**リスト48.13**のように変更します。

○リスト48.13：.ui.toot_edit.TootEditViewModel

```
    import io.keiji.sample.mastodonclient.repository.MediaFileRepository
+   import io.keiji.sample.mastodonclient.repository.PostTootQueueRepository
    import io.keiji.sample.mastodonclient.repository.TootRepository
    // 省略

    class TootEditViewModel(
            private val instanceUrl: String,
            private val username: String,
            private val coroutineScope: CoroutineScope,
            application: Application
    ) : AndroidViewModel(application) {
        private val userCredentialRepository = UserCredentialRepository(
            application
        )
+       private val postTootQueueRepository = PostTootQueueRepository(application)
        private val mediaFileRepository = MediaFileRepository(application)
                                                                          ↗
```

```
        val loginRequired = MutableLiveData<Boolean>()

        val postComplete = MutableLiveData<Boolean>()
        val errorMessage = MutableLiveData<String>()

+       fun addTootQueue() {
+           val statusSnapshot = status.value ?: return
+           if (statusSnapshot.isBlank()) {
+               errorMessage.postValue("投稿内容がありません")
+               return
+           }
+
+           coroutineScope.launch {
+               postTootQueueRepository.addQueue(                    ①
+                   instanceUrl,
+                   username,
+                   statusSnapshot,
+                   mediaAttachments.value
+               )
+
+               postComplete.postValue(true)
+           }
+       }

        fun postToot(status: Editable?) {
            // 省略
        }
```

① 投稿キューに追加

サービスの起動

投稿処理のサービスを起動します。.ui.toot_edit.TootEditFragmentを**リスト48.14**のように変更します。

○リスト48.14：.ui.toot_edit.TootEditFragment

```
    import io.keiji.sample.mastodonclient.databinding.FragmentTootEditBinding
+   import io.keiji.sample.mastodonclient.service.PostTootService
    import io.keiji.sample.mastodonclient.ui.login.LoginActivity

    class TootEditFragment : Fragment(R.layout.fragment_toot_edit) {
        override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

            // 省略

            viewModel.mediaAttachments.observe(viewLifecycleOwner, Observer {
                adapter.mediaAttachments = it
            })
            viewModel.postComplete.observe(viewLifecycleOwner, Observer {
```

```
-                   Toast.makeText(requireContext(), "投稿完了しました", Toast.LENGTH_LONG).
                    show()
+                   Toast.makeText(requireContext(), "投稿キューに追加しました",
                    Toast.LENGTH_LONG).show()
+                   launchPostService()      ①

                        callback?.onPostComplete()
                    })
                    viewModel.errorMessage.observe(viewLifecycleOwner, Observer {
                        Snackbar.make(view, it, Snackbar.LENGTH_LONG).show()
                    })
                }

+               private fun launchPostService() {
+                   val intent = Intent(requireContext(), PostTootService::class.java)
+                   requireContext().startService(intent)                                  ②
+               }

                private fun openMediaChooser() {
                    // 省略
                }

                // 省略

                override fun onOptionsItemSelected(item: MenuItem): Boolean {
                    return when (item.itemId) {
                        R.id.menu_post -> {
-                           viewModel.postToot()
+                           viewModel.addTootQueue()                                       ③
                            true
                        }
                        else -> super.onOptionsItemSelected(item)
                    }
                }
```

① 投稿サービスを起動

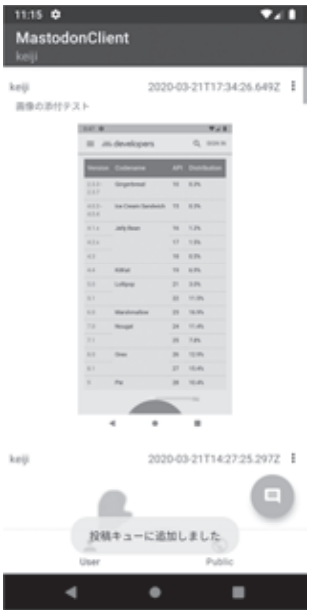
② 明示的 Intent で起動するサービスを指定

③ 投稿操作で実行するメソッドの切り替え

実行

ここまでの変更を終えてアプリを実行します。投稿画面を開き、投稿内容を入力して投稿ボタンを押すと画面に「投稿キューに追加しました」と表示され、タイムラインに戻って操作が可能になります（図48.2）。

○図48.2：



Step 49 Timberでログ出力をする

前のステップでApplicationクラスのonCreate実行を確認するためLogcatにログを出力しました。

ログ出力は便利な反面、課題もあります。Logcatを眺めてみるとアプリに関係のないログが大量に流れているのに気づくでしょう。1つのアプリが大量のログを出力すると、他のアプリのログが追いづらくなります。また、プロセスや文字列でフィルターすることはできるものの、ログ出力そのものをアプリの外から停止することはできません。

たとえば、本書でもアクセストークンの取得する際、デバッグ目的でアクセストークンをログに出力しています（**リスト49.1**）。

○リスト49.1：.ui.login.LoginViewModel

```
        coroutineScope.launch {
            val responseToken = authRepository.token(
                    clientId,
                    clientSecret,
                    redirectUri,
                    scopes,
                    code
            )

            Log.d(TAG, responseToken.accessToken)
```

公開（リリース）され、ユーザーの手元に届く前にログ出力を削除できていれば問題はありませんが、万が一、削除を忘れてリリースされれば、アクセストークンが漏洩する危険があります。

他にも、エラーをそのままログ出力をすると通信先のURLや、アプリの構造など、本来は不要な情報を周囲に知らせることになります（筆者は、エラーが発生したときに対象となるJSONデータを全部ログに出力しているアプリを見たことがあります）。

このようなトラブルを避けるためにも、不要なログ出力は避け、リリース時には十分なチェックが必要です。しかしログ出力は、デバッグの都合上、まったく使わないでアプリ開発をすることも現実的ではありません。

そこで本書ではログ出力に「Timber」を使います。Timberは、Jake Wharton氏がオープンソースで開発しているログ出力ライブラリで、シンプルながらも、ログの切り替えが柔軟にできる仕組みを備えています。

Timberによるログ出力は、次の手順で行います。

・ライブラリの追加
・Timberの初期化

ライブラリの追加（timber）

build.gradleを開いて、dependenciesにTimberを追加します（**リスト49.2**）。

○リスト49.2：app/build.gradle

```
    dependencies {
        // 省略
        kapt "androidx.room:room-compiler:2.2.4"
+       implementation 'com.jakewharton.timber:timber:4.7.1'    ①
    }
```

① timberを追加。バージョンは本書執筆時点で最新の4.7.1を指定する。

Timberの初期化

.MyApplicationを**リスト49.3**のように変更します。

○リスト49.3：.MyApplication

```
    import io.keiji.sample.mastodonclient.db.AppDatabase
+   import timber.log.Timber
+   import timber.log.Timber.DebugTree
                                                              ↗
```

```
    class MyApplication : Application() {

        // 省略

        override fun onCreate() {
            super.onCreate()
+           if (BuildConfig.DEBUG) {
+               Timber.plant(DebugTree())
+           }

            Log.d(TAG, "Application is created.")
+           Timber.d("Application is created.")

            initDatabase(this)
        }
```

① デバッグビルド（BuildConfig.DEBUG が true のとき）だけログ出力を有効

ここまでのアプリを実行すると、アプリ起動時に Logcat に「Application is created.」が2回表示されます。

次に、Timber.plant の行を消してからもう一度起動すると、「Application is created.」は1回しか出力されません。Timber.plant が実行されない状態では Timber はログを出力しないことがわかります。

リスト 49.3 で Timber.plant が実行されるのは BuildConfig.DEBUG が true のときだけです。したがってリリース時（BuildConfig.DEBUG が false）には、Timber の出力は停止します。

▶アクセストークンのログ出力を削除

.ui.login.LoginViewModel を**リスト 49.4** のように変更します。

○リスト 49.4：.ui.login.LoginViewModel

```
        coroutineScope.launch {
            val responseToken = authRepository.token(
                    clientId,
                    clientSecret,
                    redirectUri,
                    scopes,
                    code
            )

-           Log.d(TAG, responseToken.accessToken)
```

Timber を利用するしないにかかわらず、アクセストークンをログに出力すべきではありません。

Step 50　WorkManagerにバックグラウンド処理を依頼する

これまではServiceで実行していたバックグラウンド処理を、「WorkManager」に切り替えます。

WorkManagerは、直ちに実行する必要がないタスクや、時間を指定したり定期的に実行するタスクの「スケジューリング実行」を目的として開発されています。

WorkManagerはServiceの代替ではないですが、バックグラウンド処理を実行する上で便利な機能となっています。たとえば、Serviceの場合、投稿処理でエラーが発生するなどしてしまえばServiceは終了します。タスクが完了するまでにデバイスの電源を切った場合もServiceは終了して、再び電源を入れてもServiceは自動的に復帰しません。一度Serviceが終了すると、投稿キューの内容が次に投稿されるのは、今の設計では新しい投稿を実行してServiceを再び起動したタイミングとなります。

WorkManagerは、タスク（ワーカーと呼ばれる）を実行中、何らかのエラーで完了しなくても自動的に再実行を指示することができます。また、デバイスを再起動してもタスクは消えずに継続します。さらに、ネットワーク接続があるときや、充電（給電）中に限定してタスクを実行するといった条件を設定することもできます。ただし、10分を超えるタスクはシステムによって停止されるなどの制約[注1)]もあります。

WorkManagerの導入は、次の手順で行います。

・ライブラリの追加
・Workerの作成
・WorkManagerにWorkerを登録

ライブラリの追加（work-runtime-ktx）

build.gradleを開いて、dependenciesにWorkManagerを追加します（**リスト50.1**）。

○リスト50.1：app/build.gradle

```
    dependencies {
        // 省略
        implementation 'com.jakewharton.timber:timber:4.7.1'
+       implementation "androidx.work:work-runtime-ktx:2.4.0"     ①
    }
```

① work-runtime-ktxを追加。バージョンは本書執筆時点で最新の2.4.0を指定する。

注1）URL https://developer.android.com/topic/libraries/architecture/workmanager/how-to/managing-work?hl=en#cancelling

Workerの作成

パッケージ.workerにクラスPostTootWorkerを作成して、リスト50.2のようにします。

○リスト50.2：.worker.PostTootWorker

```
package io.keiji.sample.mastodonclient.worker

import android.app.Application
import android.content.Context
import androidx.work.CoroutineWorker
import androidx.work.WorkerParameters
import io.keiji.sample.mastodonclient.db.entity.PostMediaQueue
import io.keiji.sample.mastodonclient.repository.PostTootQueueRepository
import io.keiji.sample.mastodonclient.repository.TootRepository
import io.keiji.sample.mastodonclient.repository.UserCredentialRepository
import retrofit2.HttpException
import timber.log.Timber
import java.io.File
import java.io.IOException
import java.net.HttpURLConnection

class PostTootWorker(
    context: Context,
    workerParams: WorkerParameters
) : CoroutineWorker(context, workerParams) {     ①

    private val postTootQueueRepository = PostTootQueueRepository(
        applicationContext as Application
    )
    private val userCredentialRepository = UserCredentialRepository(
        applicationContext as Application
    )

    override suspend fun doWork(): Result {
                                                  ②
        return postQueuedToots()
    }

    private suspend fun postQueuedToots(): Result {
        return try {
            postTootQueueRepository
                .allTootQueues()
                .forEach { queue ->
                    val userCredential = userCredentialRepository
                        .find(queue.instanceUrl, queue.username)
                    userCredential ?: return@forEach

                    val tootRepository = TootRepository(userCredential)

                    val medias = postTootQueueRepository
```

```
                        .findMediasByTootId(queue.id)

                    val mediaIds = uploadMedias(
                        medias,
                        tootRepository
                    )

                    tootRepository.postToot(
                        queue.status,
                        mediaIds
                    )

                    postTootQueueRepository.delete(queue)
                }
            Result.success()     ③
        } catch (e: HttpException) {
            handleException(e)
        } catch (e: IOException) {
            handleException(e)
        }
    }

    private suspend fun uploadMedias(
        medias: List<PostMediaQueue>,
        tootRepository: TootRepository
    ): List<String> {
        return medias.mapNotNull { mediaQueue ->
            if (mediaQueue.mediaId != null) {
                return@mapNotNull mediaQueue.mediaId
            }

            val file = File(mediaQueue.file)
            val media = tootRepository.postMedia(
                file,
                mediaQueue.mediaType
            )
            mediaQueue.mediaId = media.id
            postTootQueueRepository.upsert(mediaQueue)

            file.deleteOnExit()

            return@mapNotNull mediaQueue.mediaId
        }
    }

    private fun handleException(e: HttpException): Result {
        val (message, result) = when (e.code()) {
            HttpURLConnection.HTTP_FORBIDDEN -> {
                Pair("権限がありません: ${e.message}", Result.failure())     ③
            }
            else -> {
                Pair("不明なエラーです ${e.message}", Result.retry())     ④
            }
        }
```

```
↘
        Timber.e(message)
        return result
    }

    private fun handleException(e: IOException): Result {
        Timber.e(e)
        return Result.retry()     ⑤
    }
}
```

① WorkManagerのWorkerは、Workerクラスを継承する（CoroutineWorkerはコルーチンを使う場合の親クラス）
② Workerの起動で実行されるエントリーポイント
③ ステータスコードがForbiddenであれば失敗する
④ ステータスコードがForbidden以外であれば再試行する
⑤ ネットワーク接続に関するエラーは再試行する

WorkManagerにWorkerを登録

WorkManagerにWorkerを登録します。.ui.toot_edit.TootEditViewModelを**リスト50.3**のように変更します。

○リスト50.3：.ui.toot_edit.TootEditViewModel

```
    import androidx.lifecycle.MutableLiveData
+   import androidx.work.Constraints
+   import androidx.work.NetworkType
+   import androidx.work.OneTimeWorkRequestBuilder
+   import androidx.work.WorkManager
    import io.keiji.sample.mastodonclient.entity.LocalMedia
    // 省略
    import io.keiji.sample.mastodonclient.repository.UserCredentialRepository
+   import io.keiji.sample.mastodonclient.worker.PostTootWorker
    import kotlinx.coroutines.CoroutineScope
    // 省略

    class TootEditViewModel(
            private val instanceUrl: String,
            private val username: String,
            private val coroutineScope: CoroutineScope,
            application: Application
    ) : AndroidViewModel(application) {

        // 省略

        private val mediaFileRepository = MediaFileRepository(application)

+       private val workManager = WorkManager.getInstance(application)    ①

                                                                          ↗
```

```
        val status = MutableLiveData<String>()

        // 省略

        fun addTootQueue() {
            val statusSnapshot = status.value ?: return
            if (statusSnapshot.isBlank()) {
                errorMessage.postValue("投稿内容がありません")
                return
            }

            coroutineScope.launch {
                postTootQueueRepository.addQueue(
                    instanceUrl,
                    username,
                    statusSnapshot,
                    mediaAttachments.value
                )

+               enqueueWorker()     ②

                postComplete.postValue(true)
            }
        }

+       private fun enqueueWorker(): OneTimeWorkRequest {
+           val constraints = Constraints.Builder()
+               .setRequiredNetworkType(NetworkType.CONNECTED)   ③
+               .build()
+           val postWorkRequest = OneTimeWorkRequestBuilder<PostTootWorker>()
+               .setConstraints(constraints)     ⑤                          ④
+               .build()
+           workManager.enqueue(postWorkRequest)     ⑥
+       }
```

① WorkManagerのインスタンスを取得

② Workerの登録を実行

③ Workerを実行時の制約（Constraint）を作成

④ WorkRequestの作成。OneTimeWorkRequestBuilderは一度だけ実行するWorkRequestを扱う

⑤ WorkRequestに制約を設定

⑥ WorkManagerにWorkerを登録

WorkRequestには、さまざまな制約を設定できます（**表50.1**）。

○表50.1：WorkRequestに設定できる主な制約

メソッド	引数	解説
setRequiredNetworkType	NetworkType	CONNECTED ：ネットワークに接続されている状態で実行 METERED ：従量課金のネットワークに接続されている状態で実行 UNMETERED ：課金が発生しないネットワークに接続されている状態で実行 NOT_ROAMING：ローミングでないネットワークに接続されている状態で実行
setRequiresBatteryNotLow	Boolean	バッテリ残量が低い状態では実行しない
setRequiresDeviceIdle	Boolean	デバイスが待機状態のときに実行する
setRequiresCharging	Boolean	デバイスが充電中であれば実行する

Service起動の停止

WorkManagerの導入に伴い、Serviceの起動を停止します。.ui.toot_edit.TootEditFragmentをリスト50.4のように変更します。

○リスト50.4：.ui.toot_edit.TootEditFragment

```
        viewModel.postComplete.observe(viewLifecycleOner, Observer {
            Toast.makeText(requireContext(), "投稿完了しました", Toast.LENGTH_LONG).show()

-           launchPostService()

            callback?.onPostComplete()
        })
```

実行

ここまでの変更を終えてアプリを実行します。Tootを投稿するときに、Serviceを使っていたときはネットワーク接続がない状態で投稿をすると、その後ネットワーク接続を復帰しても再投稿されませんでした。
WorkManagerへの切り替え後は、ネットワーク接続を復帰すると自動的に再投稿が実行されます。

Step 51 投稿完了時に通知を表示する

現状でユーザーは、投稿が完了したのか。エラーが発生したのか、知る方法がありません。投稿の状態を通知で示すことで投稿の完了とエラーの発生をユーザーにわかりやすく伝えます。

画面の上端に小さなアイコンが表示される領域（ステータスバー）があります。ここはアプリから、処理の状況や結果などを通知する目的で使われます。

○図51.1：

通知の表示は、次の手順で行います。

・通知チャンネルの設定
・通知の表示

通知チャンネルの作成

通知を表示する前に、通知チャンネル（Notification Channel）を作成する必要があります。Notification Channelは、Android 8.0（API Level 26）から導入された仕組みで、通知の種類や重要度に応じた出し分けをするために使います。

パッケージのトップ（io.keiji.sample.mastodonclient）にファイル`NotificationUtil.kt`を作成して、**リスト51.1**のようにします。

○リスト51.1：.NotificationUtil.kt

```
package io.keiji.sample.mastodonclient

import android.app.NotificationChannel
import android.content.Context
import androidx.core.app.NotificationManagerCompat

private const val CHANNEL_ID_GENERAL = "general"     ①
private const val CHANNEL_ID_ERROR = "error"     ②

fun registerNotificationChannelGeneral(
    applicationContext: Context
) {
    registerNotificationChannel(
        CHANNEL_ID_GENERAL,
        "一般",
        NotificationManagerCompat.IMPORTANCE_DEFAULT,
        applicationContext
    )
}

fun registerNotificationChannelError(
    applicationContext: Context
) {
    registerNotificationChannel(
        CHANNEL_ID_ERROR,
        "投稿エラー",
        NotificationManagerCompat.IMPORTANCE_HIGH,
        applicationContext
    )
}

private fun registerNotificationChannel(
    channelId: String,
    name: String,
    importance: Int,
    context: Context
) {
    val channel = NotificationChannel(
        channelId,
        name,                                    ③
        importance
    )
    val nm = NotificationManagerCompat.from(context)
    nm.createNotificationChannel(channel)     ④
}
```

① 一般の通知に使うチャンネル

② エラーを通知するときのチャンネル

③ 通知チャンネルのインスタンスを生成

④ 通知チャンネルを作成

通知チャンネルの作成は、MyApplication内で行います[注2)]。.MyApplicationを**リスト51.2**のように変更します。

○リスト51.2：.MyApplication

```
    override fun onCreate() {
        super.onCreate()
        if (BuildConfig.DEBUG) {
            Timber.plant(DebugTree())
        }

        Log.d(TAG, "Application is created.")
        Timber.d("Application is created.")

+       registerNotificationChannelGeneral(this)
+       registerNotificationChannelError(this)

        initDatabase(this)
    }
```

実行 ここまでの変更を終えて、一度アプリを起動して終了します。ホームの「アプリ情報」から「通知」を選択すると、アプリが利用する（登録した）通知チャンネルが確認できます（図51.2）。

○図51.2：

注2）本書ではMyApplicationで通知チャンネルの作成を行っていますが、通知の直前にチャンネルを作成することもできます。

＜ユーザーによる通知の無効化＞

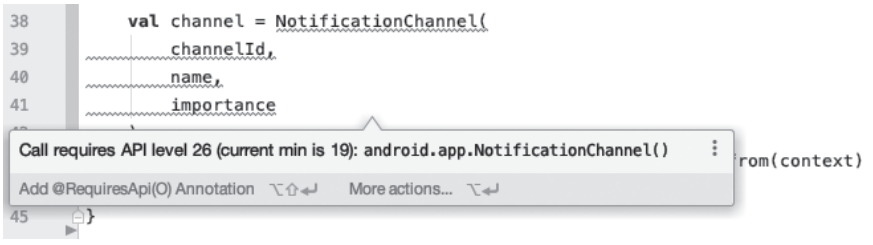
ユーザーは、通知チャンネル毎に通知の有効・無効を設定できます。ユーザーが無効に設定した通知チャンネルについては以後、通知が表示されなくなります。また、ユーザーは個別の通知チャンネルだけでなく、アプリ自体の通知を無効にすることもできます。

あまりに頻繁に通知を表示したり、通知を使って広告を表示したり、ユーザーの望まない動作をするとユーザーが通知を無効にして、本来必要な情報が届けられなくなる可能性があるので、十分な注意が必要です。

▶ API Levelに応じたNotificationChannelの取り扱い

NotificationChannelのコードに赤い下線が表示されています（**図51.3**）。これはアプリをインストール可能なAndroidのバージョン（API Level）に合致しないAPIが使われていることが原因です。

○図51.3：



プロジェクトを作成したときにminSdkVersion（Minimum SDK Version）を設定しました。minSdkVersionを19に設定したアプリは、API Levelが19以上のAndroidにインストールできます。今回の場合、API Level 26以上でしか利用できないAPIを使っているため、API Level 19 ～ 25のAndroidにアプリをインストールした場合、前述の理由でエラーが発生します。

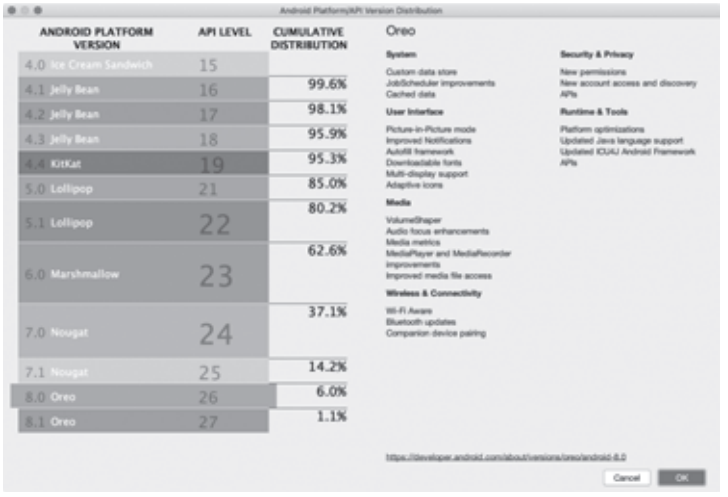
エラーを回避する方法は2つあります。

1つは、プロジェクトを作成したときに設定した「アプリがどのAPI Levelで動作するかの最低値（minSdkVersion）」を26以上に設定します。しかしこの方法だとアプリが動作するデバイスが限定されてしまいます。

本書執筆時点でAPI Levelが26（Android 8.0 開発コード「Oreo」）以上のシェアは、全Androidデバイスの6%しかありません。一方、API Level 19以上は95.3%となっています[注3)]。この圧倒的とも言える差を許容できるかどうかです（**図51.4**）。

注3） Web公開用に手直ししている現在時点では、API Level 19以上の割合は「60.8%」と躍進しています。

○図51.4：



※API Level別のシェア

もう一つのエラー回避方法は、プログラム側でAPI Levelに応じて処理を分けることです。具体的に言えば、アプリが動作しているAndroidのAPI Levelが26未満であれば、NotificationChannelを使わないようにします。

.NotificationUtil.ktを**リスト51.3**のように変更します。

○リスト51.3：.NotificationUtil.kt

```
    import android.app.NotificationChannel
    import android.content.Context
+   import android.os.Build
    import androidx.core.app.NotificationManagerCompat

    // 省略

    private fun registerNotificationChannel(
        channelId: String,
        name: String,
        importance: Int,
        context: Context
    ) {
+       if (Build.VERSION.SDK_INT < Build.VERSION_CODES.O) {
+           return
+       }

        val channel = NotificationChannel(
            channelId,
            name,
            importance
```

①

```
        )
        val nm = NotificationManagerCompat.from(context)
        nm.createNotificationChannel(channel)
    }
```

① アプリが動作しているAndroidのAPI Levelが、「Android O（API Level 26）」未満であれば処理を抜ける（returnする）

通知の表示

通知チャンネルの準備ができたら通知の表示を行います。

まずはじめに、Material Designから通知に使うアイコン素材を取得します（**図51.5**、**図51.6**）。それぞれダウンロードしてres/drawable-xxxhdpiディレクトリに配置します。

○図51.5：

※report_done（black, 18dp）。投稿完了。

○図51.6：

※report_problem（black, 18dp）。エラー発生。

アイコンを準備したら、次に.NotificationUtil.ktを**リスト51.4**のように変更します。

○リスト51.4：.NotificationUtil.kt

```
    import android.os.Build
+   import androidx.core.app.NotificationCompat
    import androidx.core.app.NotificationManagerCompat

    // 省略

    private const val CHANNEL_ID_GENERAL = "general"
    private const val CHANNEL_ID_ERROR = "error"

+   private const val NOTIFY_ID_GENERAL = 0
+   private const val NOTIFY_ID_ERROR = -1

+   fun showNotification(context: Context, message: String) {
+       val notification = NotificationCompat.Builder(
+               context,
+               CHANNEL_ID_GENERAL     ①
+           )
+           .setContentTitle("投稿を完了しました")
+           .setContentText(message)
+           .setSmallIcon(R.drawable.baseline_done_black_18)
+           .build()
```

```
+        val nm = NotificationManagerCompat.from(context)
+        nm.notify(NOTIFY_ID_GENERAL, notification)
+    }

+    fun showErrorNotification(context: Context, message: String) {
+        val notification = NotificationCompat.Builder(
+                context,
+                CHANNEL_ID_ERROR     ②
+            )
+            .setContentTitle("投稿でエラーが発生しました")
+            .setContentText(message)
+            .setSmallIcon(R.drawable.baseline_report_problem_black_18)
+            .build()
+        val nm = NotificationManagerCompat.from(context)
+        nm.notify(NOTIFY_ID_ERROR, notification)
    }
```

① 一般的な通知を「general チャンネル」で表示
② エラーの通知を「error チェンネル」で表示

最後にWorkerで通知を表示します。.worker.PostTootWorkerを**リスト51.5**のように変更します。

○リスト51.5：.worker.PostTootWorker

```
    import io.keiji.sample.mastodonclient.repository.UserCredentialRepository
+   import io.keiji.sample.mastodonclient.showErrorNotification
+   import io.keiji.sample.mastodonclient.showNotification
    import retrofit2.HttpException

    class PostTootWorker(
        context: Context,
        workerParams: WorkerParameters
    ) : CoroutineWorker(context, workerParams) {

        // 省略

        private suspend fun postQueuedToots(): Result {
            return try {
                postTootQueueRepository
                    .allTootQueues()
                    .forEach { queue ->
                        // 省略
                    }

+               showNotification(
+                   applicationContext,
+                   "Tootの投稿を完了しました"
+               )
```

```
                Result.success()
            } catch (e: HttpException) {
                handleException(e)
            } catch (e: IOException) {
                handleException(e)
            }
        }

        private fun handleException(e: HttpException) {
            val (message, result) = when (e.code()) {
                HttpURLConnection.HTTP_FORBIDDEN -> {
                    Pair("権限がありません: ${e.message}", Result.failure())
                }
                else -> {
                    Pair("不明なエラーです ${e.message}", Result.retry())
                }
            }

+           showErrorNotification(
+               applicationContext,
+               message
+           )

            Timber.e(message)
            return result
        }

        private fun handleException(e: IOException) {
+           showErrorNotification(
+               applicationContext,
+               "サーバーに接続できませんでした"
+           )

            Timber.e(e)
            return Result.retry()
        }
    }
```

実行 ここまでの変更を終えてアプリを起動します。投稿が正常に完了すれば、通知領域に投稿完了のアイコンが表示されます（図51.7）。投稿でエラーが発生すれば、通知領域にエラーのアイコンが表示されます（図51.8）。

○図51.7：

※投稿完了の通知

○図51.8：

※エラーの通知

Step 52　アプリからのIntentを受け取る

他のアプリからIntentを使ってMastodonClientを起動できることはアクセストークンを取得したときに説明しました。ここではもう少し一般的なIntent連携、たとえばテキストや画像を共有して投稿画面を開く機能を追加します。

テキスト共有からの投稿

テキストの共有は、たとえばブラウザであるページを開いて［共有］を選択したとき、あるいはテキストの一部を選択して長押しして、表示されたメニューから［共有］を選択する操作のときです（**図52.1**）。

○図52.1：

テキストの共有に対応するにはAndroidManifest.xmlにIntent-Filterを設定します。AndroidManifest.xmlを**リスト52.1**のように変更します。

○リスト52.1：app/src/main/AndroidManifest.xml

```
        <activity android:name=".ui.toot_edit.TootEditActivity"
-               android:windowSoftInputMode="adjustResize" />
+               android:windowSoftInputMode="adjustResize">
+           <intent-filter>
+               <action android:name="android.intent.action.SEND" />
+
+               <category android:name="android.intent.category.DEFAULT" />
+
+               <data android:mimeType="text/plain" />
+
+           </intent-filter>
        </activity>
```

① テキスト共有を受け取るIntent-Filter

次に、TootEditActivityを起動したときに共有されたデータがあれば取得して表示します（**リスト52.2**）。

○リスト 52.2：.ui.toot_edit.TootEditFragment

```
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

        // 省略

        bindingData.lifecycleOwner = viewLifecycleOwner
        bindingData.viewModel = viewModel

+       handleIntentExtras(requireActivity().intent)

        adapter = MediaPreviewAdapter(layoutInflater)

        // 省略
    }

+   private fun handleIntentExtras(intent: Intent) {
+       val extras = intent.extras ?: return ①
+       val mimeType = intent.type
+       when {
+           mimeType == null -> {
+               // nothing to do
+           }                                    ②
+           mimeType == "text/plain" -> {
+               val text = extras.getString(Intent.EXTRA_TEXT) ③
+               viewModel.status.postValue(text) ④
+           }
+       }
+   }

    private fun launchPostService() {
        val intent = Intent(requireContext(), PostTootService::class.java)
        requireActivity().startService(intent)
    }
```

① 共有されたデータ（intent.extras）がなければ処理を抜ける（return する）

② SmartCast 用のコード。取得した mimeType は Nullable だが、ここで null を受け取ることで以降 NonNull として扱われる

④ 共有テキストを取得

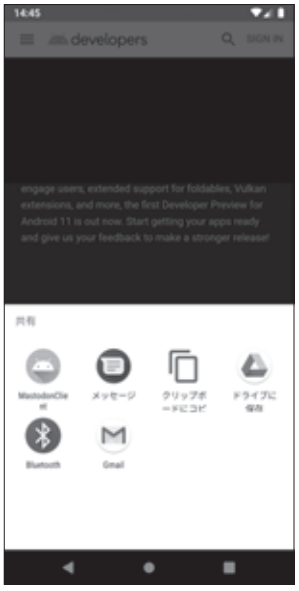
⑤ テキスト入力領域を更新。双方向 DataBinding により、LiveData を更新すれば画面に反映される

実行

ここまでの変更を終えてアプリを起動して、一度終了します。ブラウザーなどの他のアプリを起動して表示されているテキストを選択して［共有］すると、共有候補に MastodonClient が表示されます（図 52.2）。

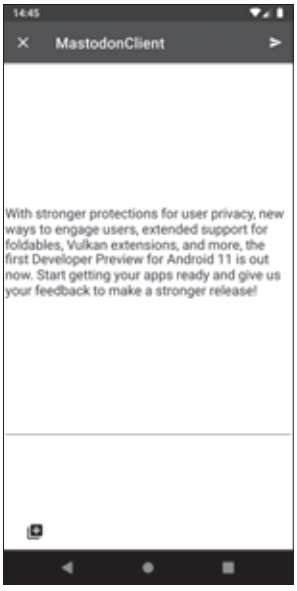
MastodonClient を選択すると、選択していたテキストが入力済みの状態で投稿編集画面が起動します（図 52.3）。

○図52.2：



※テキストの共有候補にMastdonClientが表示されている

○図52.3：



※投稿編集画面に共有したテキストが表示されている

画像共有からの投稿

画像の共有は、カメラアプリで撮影した画像や、ギャラリーなどで開いた画像を「共有」する操作です。画像の共有に対応するにはAndroidManifest.xmlにIntent-Filterを設定します。

AndroidManifest.xmlを、**リスト52.3**のように変更します。

○リスト52.3：app/src/main/AndroidManifest.xml

```
        <activity android:name=".ui.toot_edit.TootEditActivity"
                android:windowSoftInputMode="adjustResize">
            <intent-filter>
                <!-- 省略 -->

            </intent-filter>
+           <intent-filter>
+               <action android:name="android.intent.action.SEND" />
+
+               <category android:name="android.intent.category.DEFAULT" />
+
+               <data android:mimeType="image/*" />
+
+           </intent-filter>
        </activity>
```

① 画像の共有を受け取るIntent-Filter

次に、TootEditActivityを起動したときに共有されたデータがあれば取得して表示します（**リスト52.4**）。

.ui.toot_edit.TootEditFragmentを**リスト52.4**のように変更します。

○リスト52.4：.ui.toot_edit.TootEditFragment

```
    import android.content.Intent
+   import android.net.Uri
    import android.os.Bundle

    class TootEditFragment : Fragment(R.layout.fragment_toot_edit) {

        // 省略

        private fun handleIntentExtras(intent: Intent) {
            val extras = intent.extras ?: return
            val mimeType = intent.type
            when {
                mimeType == null -> {
                    // nothing to do
                }
                mimeType == "text/plain" -> {
                    val text = extras.getString(Intent.EXTRA_TEXT)
                    viewModel.status.postValue(text)
                }
+               mimeType.startsWith("image/") -> {
+                   extras.getParcelable<Uri>(Intent.EXTRA_STREAM)?.also {
+                       viewModel.addMedia(it)      ③
+                   }                                                    } ②
+               }                                                        } ①
            }
        }
```

① mimeTypeがimage/から始まる画像を処理

② 共有画像を取得

③ 添付画像に追加

実行

ここまでの変更を終えてアプリを起動して、一度終了します。ギャラリーなどの他のアプリを起動して画像を［共有］すると、共有候補にMastodonClientが表示されます（図52.4）。

MastodonClientを選択すると、画像が添付された状態で投稿編集画面が起動します（図52.5）。

○図52.4：

※画像の共有候補にMastdonClientが表示されている

○図52.5：

※投稿編集画面に共有した画像が表示されている

Step 53 Activity/Fragmentの再生成に対応する

Androidのシステムは、メモリが足りない場合などに非表示になったActivityを終了（廃棄）することがあることはすでに説明しました。

Activityを廃棄する決定はAndroidのシステムが行い、アプリ側から逃れることはできません。開発者ができることは、Activityがいつ廃棄されてもいいように準備しておくことしかありません。

開発者向けオプション（**図53.1**）で「アクティビティを保持しない」を有効にすることで、Activityが廃棄されたときの動作をテストできます。

「アクティビティを保持しない」を有効にすると、非表示になったActivityは必ず廃棄されるようになります。たとえば、「アクティビティを保持しない」を有効にしてから、投稿画面を開いて画像を添付してテキストを入力します（**図53.2**）。

その後、ホームボタンを押したり、タスクを切り替えたりしてアプリをバックグランドに遷移します（**図53.3**）。

○図53.2：

○図53.3：

再びMastodonClientアプリにタスクを切り替えると、入力したはずのテキストと、添付した画像が消えてしまいます（**図53.4**）。

○図53.4：

Activityが廃棄された状態でユーザーがアプリに戻ると、Androidのシステムは、アプリが表示していたActivityとFragmentを再生成します。しかし、再生成されたActivityとFragmentからは廃棄以前の以前の状態が失われています。

まず、状態の保存と復元をするために.ui.toot_edit.TootEditFragmentを、**リスト53.1**のように変更します。

○リスト53.1：.ui.toot_edit.TootEditFragment

```
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

        // 省略

        handleIntentExtras(requireActivity().intent)
+       viewModel.onRestoreInstanceState(savedInstanceState)    ②

        // 省略

    }                                                                    ①

+   override fun onSaveInstanceState(outState: Bundle) {
+       super.onSaveInstanceState(outState)
+                                                                        ③
+       viewModel.onSaveInstanceState(outState)    ④
+   }
```

① 保存された状態がBundleオブジェクトで渡される。保存された状態がなければ（再生成でなければ）savedInstanceStateはnullになる

② BundleオブジェクトをviewModelのonRestoreInstanceStateに渡して復元する

③ Androidのシステムが実行するFragmentのライフサイクル。ここで値を保存したBundleオブジェクトが再生成時に渡される

④ ViewModelのonSaveInstanceStateにBundleオブジェクトを渡す

次に、ViewModel側でも状態の保存と復元に対応します。.ui.toot_edit.TootEditViewModelを、**リスト53.2**のように変更します。

○リスト53.2：.ui.toot_edit.TootEditViewModel

```
    import android.net.Uri
+   import android.os.Bundle
    import androidx.lifecycle.AndroidViewModel
    // 省略

    class TootEditViewModel(
            private val instanceUrl: String,
            private val username: String,
                                                                    ↗
```

```
            private val coroutineScope: CoroutineScope,
            application: Application
    ) : AndroidViewModel(application) {

+       companion object {
+           private const val KEY_STATUS = "status"     ①
+           private const val KEY_MEDIA_ATTACHMENTS = "media_attachments"     ②
+       }

        // 省略

        private fun handleMediaException(mediaUri: Uri, e: IOException) {
            errorMessage.postValue("メディアを読み込めません ${e.message} ${mediaUri}")
        }

+       fun onSaveInstanceState(outState: Bundle) {
+           outState.putString(KEY_STATUS, status.value)
+           outState.putParcelableArrayList(
+               KEY_MEDIA_ATTACHMENTS,                                    ③
+               mediaAttachments.value
+           )
+       }

+       fun onRestoreInstanceState(savedInstanceState: Bundle?) {
+           savedInstanceState ?: return
+
+           status.postValue(savedInstanceState.getString(KEY_STATUS))
+           val mediaList: ArrayList<LocalMedia> = savedInstanceState
+               .getParcelableArrayList(KEY_MEDIA_ATTACHMENTS) ?: ArrayList()   ④
+           mediaAttachments.postValue(mediaList)
+       }
    }
```

① 入力テキストをBundleオブジェクトに保存するときのキー
② 添付メディアリストをBundleオブジェクト保存するときのキー
③ 状態をBundleオブジェクトに保存する処理
④ 状態をBundleオブジェクトから復元する処理

ViewModelにおけるonSaveInstanceStateとonRestoreInstanceStateの名前は、ActivityやFragmentにある同名のメソッドを踏襲しています。ViewModelのメソッドをオーバーライドしているわけではないので注意してください。

なお、onSaveInstanceStateに対してActivityではonRestoreInstanceStateというメソッドが用意されていますが、Fragmentにはありません。onCreateやonViewCreatedなど、いくつかのメソッドの引数にBundleが渡されるので、それを使って状態を復元します。

同様に、TootListFragmentとTootListViewModelについても状態の保存と復元を追加します。.ui.toot_list.TootListFragmentと.ui.toot_list.TootListViewModelを、それぞれ**リスト53.3**、**リスト53.4**のように変更します。

○リスト 53.3：.ui.toot_list.TootListFragment

```
    class TootListFragment : Fragment(R.layout.fragment_toot_list),
        TootListAdapter.Callback {

        // 省略

        override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

            // 省略

            val bindingData: FragmentTootListBinding? = DataBindingUtil.bind(view)
            binding = bindingData ?: return

+           viewModel.onRestoreInstanceState(savedInstanceState)

            bindingData.recyclerView.also {
                it.layoutManager = layoutManager
                it.adapter = adapter
                it.addOnScrollListener(loadNextScrollListener)
            }

            // 省略

        }

+           override fun onSaveInstanceState(outState: Bundle) {
+               super.onSaveInstanceState(outState)
+
+               viewModel.onSaveInstanceState(outState)
+           }

            private fun launchLoginActivity() {
                // 省略
            }
```

○リスト 53.4：.ui.toot_list.TootListViewModel

```
    import android.app.Application
+   import android.os.Bundle
    import androidx.lifecycle.*

    class TootListViewModel(
        private val instanceUrl: String,
        private val username: String,
        private val timelineType: TimelineType,
        private val coroutineScope: CoroutineScope,
        application: Application
    ) : AndroidViewModel(application), LifecycleObserver {
        // 省略

        fun reloadUserCredential() {
            // 省略
```

```
↘
        }

+       fun onSaveInstanceState(outState: Bundle) {
+           // Do nothing                                    } ①
+       }

+       fun onRestoreInstanceState(savedInstanceState: Bundle?) {
+           savedInstanceState ?: return
+
+           reloadUserCredential()     ②
+       }
    }
```

① メソッドとして用意はするが、現状は保存する値がないので何もしない

② 認可情報はBundleに保持せず再読み込みする

実行 ここまでの変更を終えてから、ふたたびActivityの再生成のテストをします。タスクの切り替えをしても入力したテキストと添付画像が消えず、正常に投稿できます。