jzbrooks

Dispatchers.Main & Android

Coroutines provide a “lightweight thread” abstraction that—among other things—makes getting off of the main thread dead simple. How, though, do coroutines avoid slowing down the main thread if you never leave it?

Dispatchers.Main

kotlinx.coroutines provides implementations of a main dispatcher bound to the corresponding framework’s main thread. This is pretty important for Android, as many operations can only be done from the main thread, like manipulating the view hierarchy. To understand the Android implementation of `Dispatchers.Main`, we’ll first need to brush up on some framework-specific details.

Message Queues, Loopers, & Handlers

Message queues provide a mechanism to schedule work for the main thread. The scheduled work isn’t necessarily run right away. If the main thread is busy (it usually stays pretty busy in Android’s case), the tasks will accumulate until the main thread can pull another work item off of the queue.

```
while (process_is_alive) {
  val runnable = queue.remove()
  runnable.run()
}
```

> A gross over-simplification of Android’s message loop

On Android, the main looper manages this queue. A `Looper` associates a message queue with a thread. A `Looper` is often manipulated by way of a `Handler`. Computers are pretty fast, so the system generally runs through this work queue pretty quickly.

Dispatched on Main

The Android implementation of `Dispatchers.Main` makes use of these constructs to avoid blocking your main thread as much as possible. If you’re rusty on how coroutines work under the hood, check out this post.

When coroutines are dispatched on the main dispatcher, they’re posted to the message loop. When a coroutine is suspended, the code to resume the coroutine will again dispatch the continuation to the main thread, queuing its work at the end of the message queue.

```
class LibraryViewModel : ViewModel() {

  // properties omitted for brevity

  fun setup() {
    viewModelScope.launch { // coroutine dispatched on main loop
      _playlists.value = fetchPlaylists()
      _albums.value = fetchAlbums()
    }
  }

  private suspend fun fetchPlaylists(): List<Playlist> {
    return withContext(Dispatchers.IO) { // dispatched on IO thr
      // network request for playlists
    } // the result is dispatched back onto the caller's dispatc
  }

  private suspend fun fetchAlbums(): List<Album> {
    return withContext(Dispatchers.IO) { // dispatched on IO thr
      // network request for albums
    } // the result is dispatched back onto the caller's dispatc
  }
}
```

> `viewModelScope` dispatches to the main dispatcher by default.

Inside of a coroutine, suspending functions run sequentially, just like normal code. When each of the `withContext` calls resume, they resume on the caller's dispatcher by posting the continuation (in this case, just returning the result) on the main dispatcher.

Suspending on Main

So handlers are responsible for scheduling things on the main thread's message queue. What happens if we suspend on the main dispatcher without switching contexts then?

```
class LibraryViewModel : ViewModel() {

  // properties omitted for brevity

  fun setup() {
    viewModelScope.launch { // coroutine dispatched on main loop
      Log.d("About to warm up.")

      delay(10_000) // suspends on the main dispatcher!
                    // (but doesn't block the main thread)

      Log.d("I'm warmed up now.")
    }
  }
}
```

The coroutines starts and logs a message, then it suspends for 10 seconds. `delay` ultimately translates to a call to `Handler.postDelayed`. The continuation of that call is just the second log message in this case, so after the looper notices that the time has elapsed, the second log statement will print. The main thread is saved by using the message queue to prevent blocking for 10 seconds. The following code is functionally equivalent.

```
class LibraryViewModel : ViewModel() {

  // properties omitted for brevity

  fun setup() {
    viewModelScope.launch { // coroutine dispatched on main loop
      Log.d("About to warm up.")

      Handler(Looper.getMainLooper()).postDelayed(10_000) {
        Log.d("I'm warmed up now.")
      }

      Log.e("I'm warming up.")
    }
  }
}

// Resulting in:
// About to warm up.
// I'm warming up.
// I'm warmed up. (ten seconds later)
```

This way, *your work* is done after a ten second delay, but the system can keep chewing through other messages on the queue.

Replacing `delay` with `Thread.sleep`, on the other hand, would cause the entire thread of execution to be paused for ten seconds. Since the main thread is asleep, no messages will be handled until the thread resumes.

Gumming Up the Main Thread Anyway

`NetworkOnMainThreadException` exists to prevent scheduling a chunk of work that may take hundreds of milliseconds to complete. Often the framework and tools like Android Studio/Lint help you avoid doing too much on the main thread. It's still very possible to gum up the main thread, even with coroutines.

```
class FibonacciViewModel : ViewModel() {

  fun setup() {
    viewModelScope.launch { // coroutine dispatched on main loop
      fib(45)
    }
  }

  private fun fib(n: Int): Int {
    return if (n < 2) {
      n
    } else {
      fib(n - 1) + fib(n - 2)
    }
  }
}
```

The `fib` call that is scheduled would take several *seconds* to complete on most phones today. Since that chunk of work is dispatched on the main dispatcher, your main looper's message queue just queued a ticking timebomb. When the looper pulls that message off of the queue, the main thread will be busy computing fibonacci numbers for longer than you might like. This means view updates can't be drawn, `RenderThread` dispatches don't happen, and input events can't be recieved. Yikes.

Coroutines are helpful, but they don't give your phone new hardware or change the fundamental substrate of the Android Framework.

Justin Brooks
Email Me - Github - RSS Feed