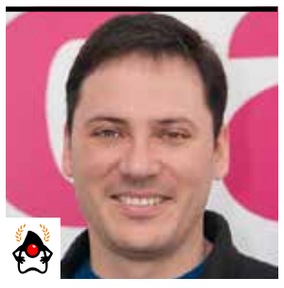

ANDRÉS ALMIRAY

JDeferred：PromiseとFutureの簡単な処理

非同期操作の悩みを解消

開発者は、連続的に発生するイベントは難なく処理できますが、並列イベントや遅延イベントの処理には難儀します。とはいえ、後から取得される結果の処理を容易にする手法は幸いにもちゃんと用意されています。そうした手法の主たるものがpromiseとfutureです。本記事では、この2つの手法と、これらの手法の使用を大幅に簡素化するライブラリ、JDeferredについて説明します。

Wikipediaによると、最初の時点ではわかっていない結果の代わりとして機能するオブジェクト、というのがこの背景にある主要な概念だと定義されています。futureは変数の読取り専用プレースホルダ・ビューであり、値を保持する以外の役割はありません。promiseは、futureの値を設定する、書込み可能な単一代入コンテナです。promiseにはAPIを定義でき、解決中の値、（予測済みまたは不測の）エラーが原因で拒否されようとしている値、計算タスクの中止といったfutureの状態変化にこのAPIで対応できます。次に、これについて詳しく見ていきます。

Javaでのpromise

標準Javaライブラリには、java.util.concurrent.Future<V>に基づくfutureの概念のさまざまな実装が含まれています。最近Java 8で追加されたものの1つにCompletableFutureがあります。このクラスを使用してできることは次のとおりです。

- 非同期方式で計算される可能性がある値の取得
- 計算結果に働きかけるミューテータ関数の登録（完了時）
- 結果を受け入れ、他の結果と結合することもある一連の機能の確立
- 予期した結果を計算するバックグラウンド・タスクの初期化

この型に含まれるファクトリ・メソッドのペアを使用すれば、すぐにCompletableFuture（以降はこのタイプをpromiseと呼びます）の使用を開始できます。次の文を実行すると、値を返さないpromiseを作成できます。

```
CompletableFuture.runAsync(new Runnable() { ... });
```

このバージョンでは、何らかの計算を実行するタスクを定義できますが、重要なのは結果ではありません。重要なのは、タスクが問題なく完了したかどうかです。次のような対応処理を付加できます。

```
CompletableFuture<Void> promise =
  CompletableFuture.runAsync(() -> ...);
promise.thenApply(result -> {
  System.out.println("Task is finished!");
});
```

計算結果を知りたい場合は、次のような、Supplierを引数として取る別のファクトリ・メソッドを起動する必要があります。

```
CompletableFuture<String> promise =
  CompletableFuture.runAsync(() -> "hello");
promise.thenApply(result -> {
  System.out.println("Task result was " + result);
});
```

一度promiseを参照したら、これにさらに操作を付加し、計算中の結果、計算中にスローされる例外、返された値への追加の変換に対応できます。

ここで、GitHub上である組織の名前を使用しているリポジトリのリストを表示する必要があるとします。そのためには、REST APIをコールし、結果を処理して表示する必要があります。そこでさらに、コードをJavaFXアプリケーションとして作成する必要があるとします。この最後の要件があると、promiseの概念の使用を検討せざるを得なくなります。というのも、リポジトリ・リストの計算はバックグラウンド・スレッドで実行する必要がありますが、結果はUIスレッド内で公開する必要があるからです。これは、対話型のJavaFXアプリケーションを作成する場合の原則です。言い換えると、UIに無関係な操作（この例の場合はネットワーク・コールなど）はUIスレッド以外のスレッドで行う必要があり、逆にUIに関係のある操作（ウィジェットのプロパティの更新など）はUIスレッド内で行う必要があります。実際のネットワーク・コールのやり方については詳しく説明しませんが、動作する完全なコードがGitHubにあります。次のスニペットは、promiseを使用してバックグラウンドで計算を実行する方法を示しています。このプロジェクトでは、関連リソースの一部を注入していることがわかります。

```
public class GithubImpl implements Github {
 @Inject private GithubAPI api;
 @Inject private ExecutorService executorService;
 @Override
 public
 CompletableFuture<List<Repository>> repositories(
    final String organization) {
  Supplier<List<Repository>> supplier = () -> {
   Response<List<Repository>> r = null;
   try {
    r = api.repositories(organization).execute();
   } catch (IOException e) {
    throw new WebApplicationException(e);
   }

   if (r.isSuccessful()) {
    return r.body();
   }
   throw new IllegalStateException(r.message());
  };

  return CompletableFuture.supplyAsync(supplier,
        executorService);
 }
}
```

このコードでは、execute()を使用してネットワーク・コールを発行します。通信上の問題や解析エラーが発生した場合は、IOExceptionがスローされます。コールが成功した場合は解析済みのボディが返され、成功しなかった場合はIllegalStateExceptionがスローされます。最後に、ターゲットとなるExecutorを指定してpromiseを作成します。前述のスニペットを見てお気付きかと思いますが、executorを明示的には定義しませんでした。というのも、Executorを定義しない場合は共通プールForkJoinが使用されるためです。

次は、promiseに返された結果を使用します。ここでは、別のコンポーネント（たとえば、コントローラ）があるとします。先ほど定義したばかりのサービスをこのコンポーネントで起動して、結果のリストを移入します。このコンポーネントには、サービスの起動時に例外が発生した場合にエラーを表示する役割もあります。

```
public class AppController {
 @Inject private AppModel
model;
 @Inject private Github github;
 @Inject private Applica-
tionEventBus eventBus;

 public void loadRepositories() {
  model.setState(RUNNING);
```

promiseによって供給される結果を処理するときのステップの順序には十分に注意してください。異なる順序でステップを記述した場合、動作が変わり、おそらく想定外の処理が実行されます。

```
    github.repositories(model.
getOrganization())
      .thenAccept(model.
getRepositories()::addAll)
      .exceptionally(t -> {
        eventBus.publishAsync(new
ThrowableEvent(t));
        return null;
      })
      .thenAccept(result -> model.
setState(READY));
  }
}
```

JDeferred では、担当する処理別にコールバックをグループ化できるため、CompletableFuture のところで説明した順序の問題がなくなります。

最後の行がどのように実行されるのかはこの後すぐに説明します。それでは、前述のコード・スニペットを1行ずつ見ていきます。まず、コントローラが状態を設定します。この状態は、計算が終了するまで他の操作ができないようにするためにUIで使用するものです。次に、サービスを起動し、promise（前述のスニペットに記述されているrepositories）を取得します。promiseを使用することで、結果を処理するといったその後の操作をコントローラに設定できます。この例の場合は、リポジトリのリストをモデルに追加する操作が該当します。多くの場合、UIでの表示にはこのモデルが使用されることになります。続いて、サービスの実行中に発生する可能性がある例外を、exceptionally()内のラムダ式を使用して処理します。最後に、処理が成功したか失敗したかに関係なく、thenAccept()内のラムダ式を使用して状態をもう一度設定します。

注意点：JDeferredへの期待も込めて

promiseによって供給される結果を処理するときのステップの順序には十分に注意してください。異なる順序でステップを記述した場合、動作が変わり、おそらく想定外の処理が実行されます。各ステップにSUCCESS、FAILURE、ALWAYSというラベルを付けます。現在の実行順序は、SUCCESS、FAILURE、ALWAYSです。

順序を変えた場合、異なる結果が生じます。

- ALWAYS、SUCCESS、FAILUREの順序ではコンパイルすら通りません。なぜなら、ラムダ式から値が返されなかった場合、ALWAYSは値が返される代わりとして結果の型をVoidに変更するからです。
- SUCCESS、ALWAYS、FAILUREの順に実行した場合、エラーが発生したときはUIが無効のままになります。これは、UIが待機しているモデルの状態が永遠に更新されないためです。
- FAILURE、SUCCESS、ALWAYSの順の場合も、エラーが発生したときはUIが無効のままになります。これも、状態が更新されないためです。

したがって、このタイプのpromiseに付加する操作の順序には十分すぎるほど注意する必要があります。

CompletableFutureには、futureでもありpromiseでもあるという潜在的な問題があります。promiseの場合はノンブロッキングの非同期方式で振る舞いますが、futureの場合はブロッキング方式のメソッドget()を持っています。つまり、ノンブロッキングでの使用例をいつでもブロッキングでの使用例に転換でき、それが不注意によるものであっても同様ということです。というのも、こうしたメソッドを提供する型（Optionalなど）に対してget()を呼び出すことがあまりにも一般的だからです。

すると、このような疑問を持つ方がいるかもしれません。「それが何だというのですか。getメソッドを呼び出さない限り、何も問題はないはずでしょう」と。しかし、このタイプのpromiseがfutureであることを踏まえると、futureを処理できる別のAPIがはるか下流でgetメソッドを呼び出さないという保証はないのです。このpromiseがそもそもfutureでなかったのであれば、ずっとよかったでしょう。次は、「CompletableFutureをpromise専用のAPIでラッピングしたらどうだろうか」という疑問が湧くかもしれません。その方法でうまくいくと思いますが、既製のpromiseライブラリに切り替えるという方法もあります。つまり、JDeferredに切り替えるのです。

JDeferredの概要

JDeferredはpromiseの概念を実現するライブラリです。着想の元にあるのはjQueryとAndroid Deferred Objectで、JDK1.6以降に対応するように設計されています。JDeferredのAPIは非常に単純ですが、その単純さに惑わされてはいけません。このAPIを使用すれば、安定性があり正しく動

作する、可読性の高いコードを作成できます。JDeferredを使用して前述の例をもう一度検討してみます。詳しく調査したい方は、GitHubからコードの完全版をダウンロードしてください。

次のMavenエントリを使用して、JDeferredをプロジェクトに追加できます。

```
<dependency>
  <groupId>org.jdeferred</groupId>
  <artifactId>jdeferred-core</artifactId>
  <version>1.2.5</version>
</dependency>
```

Gradleがお好みであれば次のエントリを使用します。

```
compile 'org.jdeferred:jdeferred-core:1.2.5'
```

JDeferredが提供する基本型はorg.jdeferred.Promiseであり、操作またはコールバックの登録に使用できます。promiseでは、完了時に値を返す、エラーが発生した場合にObject（Throwableだけでなく任意のObject）をスローする、計算途中の結果を返す、という処理を実行できます。最後の2つのオプションは、CompletableFutureでは実行できません。JDeferredでは、担当する処理別にコールバックをグループ化できるため、先ほどCompletableFutureの部分で説明した、順序の問題がなくなります。promiseは通常、DeferredManagerと呼ばれる別のコンポーネントによって作成されます。タスク作成メカニズムとpromise自体は別々の概念であるため、ライブラリではこのようにしてこの2つを分離しています。前述したGitHubのサービスをJDeferredで実装するとどのようになるか見てみます。

```
public class GithubImpl implements Github {
 @Inject private GithubAPI api;
 @Inject private DeferredManager deferredManager;

 @Override
 public Promise<List<Repository>, Throwable, Void>
    repositories(final String organization) {
   return deferredManager.when(() -> {
    Response<List<Repository>> r =
     api.repositories(organization).execute();
    if (r.isSuccessful())
     { return r.body(); }
    throw new IllegalStateException(r.message());
   });
 }
}
```

上記のコードは、機能的には先ほど詳しく見たコードと同等ですが、格段にすっきりしています。この方法で実行したタスクには、DeferredManagerによってエラーが自動的に処理されるというメリットがあります。通信エラーや解析エラーをこれまでのように明示的に処理する必要がないのはそのためです。そうしたエラーが発生すると、promiseの状態がfailedに設定され、失敗のコールバックでエラーを受信するようにエラーが記録されます。

この例では中間結果が生成されません。Promiseの3つ目の引数がVoidに設定されているのはそのためです。

promiseを使用するためには、次のようにします。

```
public class AppController {
 @Inject private AppModel model;
 @Inject private Github github;
 @Inject private ApplicationEventBus eventBus;

 public void loadRepositories() {
  model.setState(RUNNING);
  github.repositories(model.getOrganization())
   .done(model.getRepositories()::addAll)
   .fail(t ->
    eventBus.publishAsync(new ThrowableEvent(t)))
   .always((state, resolved, rejected) ->
    model.setState(READY));
 }
}
```

コントローラが実行する機能は先ほどと同じですが、コードははるかにすっきりしています。SUCCESS、FAILURE、ALWAYSの各コールバックは、ケースごとに必要と判断した任意の順序で定義できます。最後になりますが、結果が渡されるまでブロッキング方式でpromiseを強制的に待機させる方法はありません。このAPIではできないようになっているのです。

JDeferredで実装するAPIは、よりシンプルでありながら同じ機能を持ち、欠点がありません。

必要であれば、DeferredObjectを使用し、より手動の度合いが高い実装に切り替えてpromiseを生成することもできます。この型を使用すると、計算された値または拒否された値の設定や中間結果の公開を必要に応じて可能になります。SwingWorker APIの使用経験がある方は、この動作がどのような役割を果たすかご存知でしょう。主な相違点は、DeferredObjectから送信される通知がバックグラウンド・スレッドで送信されるのに対し、SwingWorkerからの通知はUIスレッド内で送信されるところです。次に、DeferredObjectを使用してpromiseに返された結果を手動で設定する方法と失敗を手動でトリガーする方法の例を示します。

```
public class GithubImpl implements Github {
 @Inject private GithubAPI api;
 @Inject private ExecutorService executorService;

 @Override
 public Promise<List<Repository>, Throwable, Void>
 repositories(final String organization) {
  Deferred<List<Repository>, Throwable, Void> d =
   new DeferredObject<>();
  executorService.submit(() -> {
   Response<List<Repository>> r = null;
   try {
    r = api.repositories(organization).execute();
   } catch (IOException e) {
    d.reject(e);
    return;
   }
   if (r.isSuccessful()) { d.resolve(r.body()); }
   d.reject(new IllegalStateException(r.message()));
  });
  return d.promise();
 }
}
```

今回は、Executorまたは同様の手段を使用して、通信エラーや解析エラーを処理するほか、バックグラウンド・タスクを明示的にスケジュールする必要があります。好きなときにpromiseの解決または拒否が可能であるため、DeferredObjectのこの用法はテストを記述するときに重宝します。次のテスト・ケースは、JDeferred、Mockito、および依存性注入を併用してまさにそうした使用例（テストの記述）を実装する方法を示しています。

```
@RunWith(JukitoRunner.class)
public class AppControllerTest {
 @Inject private AppController controller;
 @Inject private AppModel model;

 @Test
 public void happyPath(Github github) {
  // 前提：
  Collection<Repository> rs =
    TestHelper.createSampleRepositories();
  Promise<List<Repository>, Throwable, Void> p =
    new DeferredObject<List<Repository>,
      Throwable, Void>().resolve(rs);
  when(github.repositories("foo")).thenReturn(p);

  // 次の場合：
  model.setOrganization("foo");
  controller.loadRepositories();
```

```
    // 次に処理を実行:
    assertThat(model.getRepositories(), hasSize(3));
    assertThat(model.getRepositories(), equalTo(rs));
    verify(github, only()).repositories("foo");
  }
}
```

上記のコードを見ると、DeferredObjectを使用して、Githubクラスのモック・インスタンスとともに予測結果を設定する方法がわかります。このテストでは、すべてが予測どおりに動作するハッピー・パスをチェックします。代わりにrejected()を起動して、失敗するパスを設定すれば、予測した例外の発生をチェックすることもできます。

まとめ

promiseを使用すると、遅延（非同期）方式で計算結果を処理できます。Java8にはCompletableFutureという名前の型が用意されており、この型をpromiseとして使用できます。これにより、結果の処理や、結果を別の値に変換すること、結果を他の結果と結合することのほか、エラーが発生した場合の例外事例の処理が可能です。

ただし、そのようなpromiseに付加する操作の順序には注意を払う必要があります。また、そのようなpromiseは、get()メソッドを起動するだけでいつでもブロックできます。JDeferredで実装するAPIは、よりシンプルでありながら同じ機能を持ち、欠点がありません。また、バックグラウンドでの計算中に任意のタイミングで中間結果を公開できます。この2つ目の動作の例は、GitHubで入手できるコードで確認できます。</article>

Andrés Almiray：JavaおよびGroovyの開発者で、Java Champion。17年を超えるソフトウェアの設計および開発の経験があり、Javaの初期の頃から、Webアプリケーションやデスクトップ・アプリケーションの開発に関わっている。熱心なオープンソースの信奉者で、Groovy、JMatter、Asciidoctorなどの有名プロジェクトに参加。Griffonフレームワークの創設メンバーであり、現在のプロジェクト・リードである。JSR 377の仕様リードも務めている。

learn more

- JDeferredライブラリ
- future、promise、JDeferredのチュートリアル
- Java 8 CompletableFuture（Javadoc）
- Java 8 CompletableFutureのチュートリアル