

ADAM BIEN

Adam Bien：Java EE 6、Java EE 7、EJB 3.X、JAX-RS、JPA 2.X の各 JSR の Expert Group メンバー。JDK 1.0 から Java テクノロジーに関わり、各サーブレットや EJB 1.0 に携わった後、現在は Java SE および Java EE のアーキテクトおよび開発者。『Real World Java EE Patterns—Rethinking Best Practices』および『Real World Java EE Night Hacks』の著者。

写真：THOMAS EINBERGER/GETTY IMAGES

Java EE 7 と Java SE 8 による並行プログラミング

Java SE 8 で Java EE 7 アプリケーションを動作させ、双方の長所を活用する

Java EE 7 と Java SE 8 では、並行プログラミングや非同期プログラミングに関して新たなサポート形式が導入されました。メソッド・ハンドル、関数型インタフェース、ラムダ式などの Java SE 8 言語の新機能は、メソッドの非同期実行を大幅に効率化します。新しい API（特に Stream API）や CompletableFuture に加え、並列配列やストリームといった java.util パッケージのさまざまな拡張によって、Java 開発者がコードをパラレル化する新しい方法が導入されています。

同時に、Java EE 7 API でも、JAX-RS 2.0（JSR 339、Java API for RESTful Web Services）での非同期 HTTP 処理の直接サポートや、WebSocket のサポートが行われました。さらに、Concurrency Utilities for Java EE（JSR 236）によってまったく新しい API も導入されています。

たとえ別々に考えても、Java EE 7 と Java SE 8 はいずれも新しい API、機能、パラダイムを並行プログラミングにもたらしています。本記事では、Java EE 7 アプリケーションを Java SE 8 で動かすという「パーフェクト・ストーム」について紹介します。

パーフェクト・ストーム

たとえ別々に考えても、**Java EE 7 と Java SE 8 はいずれも新しい API、機能、パラダイム**を並行プログラミングにもたらしています。本記事では、Java EE 7 アプリケーションを Java SE 8 で動かすという「パーフェクト・ストーム」について紹介します。

はじまりは Java EE 6

Java EE 6 では、EJB 3.1 に @Asynchronous アノテーションが導入されました。javax.ejb.Asynchronous アノテーションが指定されたメソッドは、別のスレッドとトランザクションで非同期的に実行されます。**リスト 1** に、同期メソッドを非同期的に実行する例を示します。

通常、どのような EJB Bean でも汎用の java.util.Executor になることができます。たとえば、**リスト 2** は EJB ベースの Executor の実装です。

汎用 Executor は java.lang.Runnable 実装を受け取り、実装依存の方法で実行します。**リスト 3** では、アプリケーションのコードを Runnable でラップしています。

リスト 3 の Executor は EJB Bean で実装されているため、トランザクションでの実行が保証されます。したがって、Runnable#run メソッドは新しいトランザクションで実行されます。

ラムダ式を使用すると、大変効率的に匿名 Runnable インスタンスを作成できます。

```
Runnable run = () -> { //… };
```

実際、save メソッドを**リスト 4** のように 1 行に収めることができます。なお、この Runnable の実装は間に合わせのものです。

対応するシグネチャを持つメソッドであれば、任意のメソッドを Runnable 実装の代わりに使用できます。Runnable インタフェースはシングル抽象メソッド（SAM）インタフェース、つまり 1 つの抽象メソッドのみを持つインタフェースです。SAM は、インスタンスや静的メソッド・ハンドルで直接実装できます。

同様に、インジェクションされた POJO（Plain Old Java Object）のメソッドも、Runnable として Executor に渡すことができます（**リスト 5**

参照）。

Java EE 7 によるコード削減

EJB **ベースの** Executor のカスタム実装には、大きな欠点があります。並列リクエスト数の制限や、基盤となるスレッド・プールの設定変更を行う標準的な方法がないことです。JSR 236（Concurrency Utilities for Java EE）に対応した Java EE 7 には、Java SE の java.util.concurrent.ExecutorService のマネージド実装が付属しています。ExecutorService は Executor インタフェースを拡張しており、その結果、カスタム EJB 実装は過去のものとなりました。

リスト 6 の ManagedExecutorService が、ExecutorService のマネージド実装です。これはアプリケーション・サーバーのマネージド・リソースであり、@Resource アノテーションによってインジェクションされます。

リスト 7 は、POJO の内部で String を加工する例で、保存の前に Normalizer#normalize メソッドで入力を処理しています。

Java SE 8 では、java.util.function パッケージに便利な SAM が数多く組み込まれています。**リスト 8** に示すように、Consumer<String> インタフェースは Storage#store メソッドの抽象化に使用でき、Function<String,String> は Normalizer#normalize メソッドのプレースホルダとして使用できます。

コードの削減

Java SE 8 の力を少し借りると、複数のコンポーネントを連結して、コードを 1 行に減らすことができます。同時に、Java EE 7 が実用的な機能を提供しています。

Java SE 8 でさらにコードを削減

Java SE 8 には、協調動作をサポートする魅力的なユーティリティとして java.util.concurrent.CompletableFuture クラスが付属しています。CompletableFuture は Supplier インスタンスを Function や Consumer に接続し、柔軟な実行パイプラインを実現します。**リスト 8** の例は、CompletableFuture によって大幅に簡略化できます。**リスト 9** では、Storage と Normalizer を CompletableFuture で接続しています。

Function インタフェースと Consumer インタフェースへの割り当ては、単なるデモの目的で行われています。なお、**リスト 9** のコードは、アプリケーション・サーバーのマネージド・スレッド・プールでは実行できません。このコードを実行できるのは、「共有プール」と呼ばれる ForkJoinPool のインスタンス ForkJoinPool.commonPool() です。

名前が「Async」で終わるすべてのメソッドには、2 つ目のパラメータとして Executor インスタンスを渡せるようになっています。ManagedExecutorService は Executor であり、マネージド・アプリケーション・サーバーのスレッドでパイプラインを実行するために使用できます。**リスト 10** では、Executor として ManagedExecutorService を渡し、SAM としてメソッド・ハンドルを使用しています。

Java SE 8 の力を少し借りると、複数のコンポーネントを連結して、コードを 1 行に減らすことができます。それと同時に、Java EE 7 が、ManagedExecutorService の管理、監視、設定などの実用的な機能を提供しています。

発生する例外

リスト 10 のパイプラインで発生した例外はすべて、通知されることなく飲み込まれてしまいます。この「便りがないのは悪い知らせ」的な振る舞いは、出発点としてはよくても、大半のユースケースでは許容できないでしょう。外側の StorageService#save メソッドは、パイプラインがタスクを処理する前に終了する可能性があります。CompletableFuture は、ブロッキング式の例外処理と、非同期式の例外処理の両方を提供しています。Future#get メソッドは呼出し元をブロックし、戻り値を返すか、または ExecutionException を返して発生したエラー原因を知らせます。**リスト 11** は、例外のブロッキングの例です。

リスト 11 の get() メソッドによって例外は飲み込まれなくなりますが、呼び出すメソッドは CompletableFuture がすべてのタス

リスト1 リスト2 ／ リスト3 ／ リスト4 ／ リスト5 ／ リスト6

```
@Stateless
public class StorageService {
   @Asynchronous
   public void save(String content) {
      //heavy lifting
      System.out.println("Storing " + content);
   }
}
```

すべてのリストのテキストをダウンロード

クを終えるまでブロックされてしまいます。**リスト 12** は、非同期式の例外処理の例です。例外発生時に非同期で呼び出される exceptionally() メソッドには、パラメータとして Function<Throwable,[RETURN_VALUE]> を渡します。

exceptionally メソッドは、RETURN_VALUE の型が様々であり、任意の段階で適用できます。たとえば、supplyAsync や thenApplyAsync の直後では、2 つ目の総称型パラメータは String となります。このメソッドに渡すのは SAM であり、メソッド・ハンドルによって簡単に実装できます。エラー処理コードを専用のハンドラに移すことによって、さらにコードを効率化できます。**リスト 13** では、専用ハンドラで例外処理を行っています。

例外は非同期的に発生するため、StorageService クラス内で意味のある処理を行うことはできません。そのため、例外処理は完全に専用ハンドラにまとめることができます。save メソッドの実装は、処理を呼び出すだけで結果は考慮しない形になっています。エラー処理戦略は、メッセージ駆動型 Bean（MDB）や Java Message Service（JMS）と同様です。

非同期コードのエラーを呼出し元に伝えることはできません。そのため、エラーはデッド・レター・キューとも呼ばれる専用の JMS キューに送られます。CompletableFuture にも、同じメカニズムを JMS を使用せずに適用できます。非同期の例外処理には、単純な Contexts and Dependency Injection（CDI）イベントがもっとも適しています。**リスト 14** では、例外をイベントとして送信しています。

補足すると、javax.enterprise.event.Event#fire メソッドは、総称型の Consumer<Object> です。このメソッドを使用すると、Storage への直接参照を、インジェクションされた別の Event インスタンスで完全に置き換えることができます。**リスト 15** では、CDI イベントを Consumer<String> として使用しています。

Storage クラスは、ほとんどそのまま使用できます。save メソッドの String パラメータに @Observes アノテーションを追加する必要があるだけです。**リスト 16** では、Storage クラスを CDI リスナーにしています。

2 つのパワー

Java SE 8 **がアプリケーション・アーキテクチャに与えるインパクトと比べれば、**Java EE 7 **によるインパクトはわずかなものです。本当に面白くなるのは、この 2 つを組み合わせたときです。**

Java SE 8 のパワーが Java EE 7 の生産性と融合します。

非同期 JAX-RS 2.0

現在のところ、JAX-RS リソースでは大したことはしていません。**リスト 17** の @POST メソッドと @GET メソッドは、いずれも同期的に実行されます。

JAX-RS 2.0 では、非同期リクエスト処理のために AsyncResponse パラメータが導入されました。AsyncResponse に @Suspended アノテーションを追加すると、リソースのメソッドはリクエストが完了する前に終了できるようになります。このアプローチによって、実際のビジネス・ロジックの処理から接続処理スレッドを分離できるようになります。**リスト 18** は、そのような非同期リクエスト処理の例です。

待機中のリクエストは、AsyncResponse#resume メソッドの呼出しによって再開できます。通信プロトコルはビジネス・ロジックの実装から分離した方が良いため、ビジネス・ロジックに AsyncResponse を公開するべきではありません。そのため、ビジネス・ロジックはこの外側で実装します。ビジネス・ロジックに AsyncResponse を送信する際は、その存在を隠ぺいしたまま送信する必要があります。

Java SE 8 より前では、AsyncResponse をカスタム・ラッパー内でカプセル化することが一般的でした。しかし、Java SE 8 では、AsyncResponse#resume メソッドを Consumer<String> として送信するだけで大丈夫です。AsyncResponse をビジネス・ロジックと密接に統合する際に、抽象化やインタフェースの追加を行う必要ありません。

また、@POST メソッドを手直しして、StorageService への参照を完全に取り除くこともできます。StorageResource#store メソッ

リスト7 リスト8 ／ リスト9 ／ リスト10 ／ リスト11 ／ リスト12

```
public class Normalizer {
   public String normalize(String input) {
      return "# " + input;
   }
}
//…
  @Inject
   Normalizer normalizer;

   public void save(String input) {
      String content = normalizer.normalize(input);
      executor.execute(() -> storage.store(content));
   }
```

すべてのリストのテキストをダウンロード

ドは入力パラメータをバックエンドに送信します。バックエンドは、Supplier<String> によって簡単に抽象化できます。パラメータはラムダ式を使用して即座に Supplier に変換され、通常の CDI イベントとして送信されます。すでに JAX-RS リソースに存在している Java SE 8「ネイティブ」のインタフェースを使用すると、StorageService のコードをさらに簡略化できます。**リスト 19** では、非同期に String を配信しています。

リスト 19 の getContent メソッドは、Storage#retrieve メソッドを Supplier<String> として使用し、その出力を Consumer <Object> の入力として接続します。また、この例では、CompletableFuture とインジェクションされた ManagedExecutorService をパイプラインとして使用しています。

モニタリング
非同期アプリケーションでは、同期アプリケーションよりもモニタリングやデバッグが難しくなります。インターセプタは、個々のタスクや境界のモニタリングに非常に適しています。

Java EE 7 によるモニタリング

非同期アプリケーションでは、同期アプリケーションよりもモニタリングやデバッグが難しくなります。Java EE 5 で導入されたインターセプタは、個々のタスク（制御）や境界のモニタリングに非常に適しています。**リスト 20** は単純なパフォーマンス測定インターセプタです。

アノテーションは 1 つだけ（@Interceptors(Monitor.class)）でよく、アノテーションが付加されたクラスのすべてのパブリック・メソッドがインターセプトされてモニタリングされます。**リスト 20** の Monitor インターセプタは、ロギングに System.out.println を使用しています。実環境では、代わりに CDI イベントを使用してモニタリング・データをセンターに送信し、今後の分析に役立てることができます。

Java SE 8 による JPA コードの削除

Java Persistence API（JPA）を使用すると、Store クラスを拡張して簡単に永続化機能を持たせることができます。**リスト 21** では、簡単な JPA 2.1 コードによって永続化を行っています。各ユーザーに対して、String の content がそれぞれ格納されています。

ここで統計をとってみます。例として、ユーザー 1 人当たりの平均 String サイズを計算します。このような計算は、CPU に負荷がかかる可能性があるため、並列処理を行います。JPA 問合せには、ストリーム可能な Java SE 8 コレクションが返されます。Java SE 8 コレクションによって、インメモリの処理や分析を Java SE 8 でシンプルに行う道が開かれます。**リスト 22** では、単純な Store の統計を作成しています。

リスト13 リスト14 ／ リスト15 ／ リスト16 ／ リスト17 ／ リスト18

```
public void save(String input) {
    CompletableFuture.supplyAsync(() -> input, executor).
            thenApplyAsync(normalizer::normalize, executor).
            thenAcceptAsync(storage::store, executor).
            exceptionally(this::handle);
}

public Void handle(Throwable error) {
    System.out.println("error = " + error);
    return null;
}
```

すべてのリストのテキストをダウンロード

結果リスト全体が RAM（日々安くなっています）に収まるなら、**リスト 22** のように、JPA エンティティ上で Java SE 8 のラムダ式を使用し、簡単に実装やテスト、問合せの実行を行うことができます。Java SE 8 には、さまざまな Collector 統計ユーティリティが導入されており、コードを追加することなく必要な計算を行うことができます。IntSummaryStatistics ユーティリティ・クラスは、JsonObject として簡単に公開できます。たとえば、**リスト 23** では、IntSummaryStatistics を JsonObject に変換しています。

並列ストリームにはスレッド管理のオーバーヘッドがかかるため、単純すぎるユースケースでは十分なパフォーマンスは得られません。また、多くのユースケースでは、**リスト 22** は単一スレッドの stream を使用する方が parallelStream を使用するよりパフォーマンスがおそらく良いでしょう。さらに、非同期 parallelStream は、アプリケーション・サーバーのマネージド・スレッドではなく、ForkJoinPool が提供す

るスレッドで実行されます。使用可能なフォーク / ジョイン・スレッドの数は使用可能な CPU の数によって異なりますが、システム・プロパティ java.util.concurrent.ForkJoinPool.common.parallelism で設定できます。アプリケーション・サーバーは、このような「勝手に」作られたフォーク / ジョイン・スレッドは認識しないため、大量のスレッドによってパフォーマンスのボトルネックや堅牢性の問題が発生する可能性があります。

大変な注目
Java EE 7 は多くの注目を集めています。「Java EE」という名前がついているだけで、会議室は一杯になり、ブログ記事は人気になります。

まとめ

Java EE 7 は多くの注目を集めています。「Java EE」という名前がついているだけで、会議室は一杯になり、ブログ記事は人気になります。しかし、Java SE 8 がアプリケーション・アーキテクチャに与えるインパクトと比べれば、Java EE 7 によるインパクトはわずかなものです。本当に面白くなるのは、この 2 つを組み合わせたときです。Java SE 8 のパワーが Java EE 7 の生産性と融合します。

本記事の目的は、無駄なプログラミングをなくし、きれいなコードを書くためのアイデアを提供することです。また、本記事に掲載したサンプル・コードは大幅に簡略化しています。String の永続化も、Java SE 8 と Java EE 7 の API を最大限に使用して行いました。しかし実環境では、実際のユーザーには意味のないクールなテクノロジーを追い求めるよりも、中核となるビジネス・ロジックに集中するべきでしょう。

</article>

LEARN MORE

- この記事で使用されたサンプル・プロジェクト
- JSR 346：Contexts and Dependency Injection for Java EE 1.1
- 「Going Asynchronous With CompletableFuture and Java EE 7」スクリーンキャスト

リスト19 リスト20 ／ リスト21 ／ リスト22 ／ リスト23

```
@Stateless
public class StorageService {
   @Resource
   ManagedExecutorService executor;
   @Inject
   Normalizer normalizer;
   @Inject
   Event<Throwable> exceptionHandler;
   @Inject
   Event<String> sink;
   @Inject
   Storage storage;

   public void save(@Observes Supplier<String> supplier) {
      CompletableFuture.supplyAsync(supplier, executor).
            thenApplyAsync(normalizer::normalize, executor).
            thenAcceptAsync(this.sink::fire, executor).
            exceptionally(this::handle);
   }

   public Void handle(Throwable error) {
      exceptionHandler.fire(error);
      return null;
   }

   public void getContent(
@Observes Consumer<Object> listener) {
      CompletableFuture.supplyAsync(
            thenAcceptAsync(listener, executor).
            exceptionally(this::handle);
   }
}
```

すべてのリストのテキストをダウンロード