Java 開発 2.0: Amazon SQS によるクラウド・ベースのメッセージング

Amazon のメッセージ・キューイング・システムを利用して、使った分だけ料金を支払う

Andrew Glover
Author and developer
Beacon50

2011年 3月 22日

Amazon SQS (Simple Queue Service) はメッセージ指向ミドルウェア (MOM: Message-Oriented Middleware) から必要な概念を採り入れていますが、ある特定の実装言語またはフレームワークしか使用できないわけではありません。メッセージ・キューイング・システムをインストールして保守する負担を軽減するとともに、AWS の従量課金によるスケーラビリティーを活用するために Amazon SQS を利用する方法を学びましょう。

このシリーズの他の記事を見る

この連載について

Java™ 技術が初めて登場してから現在に至るまでに、Java 開発の様相は劇的に変化しました。成熟したオープンソースのフレームワーク、そしてサービスとして提供される信頼性の高いデプロイメント・インフラストラクチャーを利用できる (借りられる) おかげで、今では Java アプリケーションを短時間かつ低コストでアセンブルし、テスト、実行、保守することが可能になっています。この連載では Andrew Glover が、この新たな Java 開発パラダイムを可能にする多種多様な技術とツールを詳しく探ります。

メッセージ・キューは金融システムや、医療、旅行業界をはじめ、さまざまなソフトウェア・アーキテクチャーおよびドメインで一般的に使用されています。けれども、メッセージ・キューを使用するには、キューイング・システムをインストールして保守する必要があります。これは、分散システムの主要なメッセージング・パラダイムとなっているメッセージ指向ミドルウェア (MOM) では特に言えることです。今月の記事では、このような作業負荷が高いメッセージング手法に代わるクラウド・ベースの手法を紹介します。それは、Amazon の SQS (Queue Service) です。

Web アプリケーションを Google App Engine や Amazon Elastic Beanstalk でホストするほうが理にかなっている場合は多々あります (「参考文献」を参照)。それと同じように、クラウド・メッ

セージング・システムを利用することに価値がある場合も珍しくありません。いずれの場合にしても、基礎となるインフラストラクチャーのインストールと保守に時間を費やす代わりに、アプリケーションの作成により多くの時間をかけられるようになります。

この記事では、メッセージ・キューイング・システムをインストールして保守する負担を Amazon SQS が引き受ける仕組みを説明します。また、SQS メッセージ・キューを作成して、そのキューにメッセージを入れ、そこから取得する方法を実際に演習する機会も用意しています。そして最後に、先月の記事で Amazon Elastic Beanstalk について説明するために使用したモバイル Web アプリケーション、Magnus にメッセージング機能を追加する場合の例を紹介します。

「もしもし、どちら様ですか?」、「こちらは MOM です」

メッセージ指向ミドルウェア (Message-Oriented Middleware、通称 MOM) は、メッセージ・キューを介して互いに通信する、疎結合されたシステムを表す用語です。疎結合システムでは、コンポーネント同士が (例えばコンパイル時の依存関係などによって) 密接に結合されるのではなく、ネットワーク全体に分散されます。このようにコンポーネントを分散し、コンポーネント間の通信をメッセージ・キューで仲介することによって、メッセージング・システムのスケーリングが可能になります。

従来のメッセージ指向システムでは、アーキテクトが、どのコンポーネントとどのコンポーネントが通信するのかを決めるのが通常です。すべての通信はメッセージ・パッシングによって行われる一方、メッセージそのものは多くの場合、特定のプラットフォームに依存しない汎用フォーマットとなります。メッセージは単純なストリングである場合もあれば、XML または JSON を使用してコード化された文書の場合もあります。

MOM アーキテクチャーはそれぞれのコンポーネントを独立させて、プラットフォームに依存しないコンポーネント間の通信を可能にすることから、個々のユニットにはさまざまな種類を混在させることができます。つまり、分散アーキテクチャーのコンポーネントは、Java 言語や C#、Ruby などのさまざまな言語で作成できるということです。また、これらのコンポーネントは、UNIX® および Windows® といったように、異なるプラットフォームに配することも可能です。さらに MOM は、システムの統合を容易にします。MOM はミドルウェアとして、レガシー・システムでも、それよりも新しいシステムでも同様に接続できるからです。これは、コンポーネント間の API はメッセージでしかなく、しかもそのメッセージは XML 文書からシリアライズされたオブジェクト、さらに単純な `String` に至るまで、あらゆるものにできるためです。

GAE はみんなの MOM です！

MOM システムでのメッセージ・キューは Web のパイプ役を果たし、システム・コンポーネント間でメッセージが自由にやりとりされるように各種のコンポーネントを接続します。結局のところ、GAE はメッセージ指向ミドルウェア・システムの好例と言えます。

優れた MOM の例に洩れず、Google App Engine はメッセージ・キューを使用してシステム・プロセスを切り離します。具体的には、GAE キューによって、長期間実行されるプロセスを Web リクエストからアンロードすることが可能になるということです。GAE を使用して、サーブレットや JSP を指す URL をメッセージ・キューに入れると、あとは GAE サービスがこれらのメッセージ・キューを選び出して処理します。サーブレットについては、Web アプリケーションの主要な論理

シーケンスに関連して非同期で呼び出されることになります (GAE についての詳細は、「参考文献」を参照してください)。

GAE は、主要なプロセスの実行期間を管理するために、長期間実行されるプロセスをキューに入れますが、これは GAE だけが行っていることではありません。この MOM 的な機能は、例えば Heroku などの PaaS 実装にも提供されています。けれども Amazon SQS の場合には、プラットフォームに関わらず、あらゆる Web アプリケーションで簡単にこのキューイングを行うことができます。

Amazon SQS の概要

Amazon SQS には、JMS でメッセージ・キューを使用したことがあればお馴染みの豊富な機能が用意されています。

Amazon SQS は JMS ではありません

Java プラットフォームでのメッセージ・キューは、JMS 仕様によって裏付けられるように、何も新しいものではありません。JMS は 10 年以上も前に登場した仕様で、RabbitMQ および Apache の ActiveMQ、さらには IBM の Websphere® MQ まで、見事な実装が揃っています。けれども、Amazon SQS API は JMS インターフェースを 1 つも実装しません。実のところ、この API はほぼ間違いなく JMS よりも単純で、遥かに簡単に使いこなせます。

Amazon SQS には以下の特徴があります。

- 1 つのキューに対して複数のプロセスが読み取り/書き込みを行うことができます。また、処理が行われている間、メッセージはロックされるため、複数のプロセスが 1 つのキューから読み取りを行うとしても、メッセージを処理するのは 1 つのリーダーだけに限られます。
- 同時アクセスに際して極めて高い可用性を実現するために、Amazon の大規模な冗長アーキテクチャーを利用しています。さらに、(少なくとも 1 つの) メッセージの配信を保証します
- 料金は、使用した分に対してしか発生しません。Amazon SQS の場合、メッセージの単価は $0.000001 です。AWS では現在、1 ヶ月あたり最初の 100,000 件のメッセージが無料となる無料利用枠を設けています。ただし、ギガバイト単位で請求される帯域幅使用料があることに注意してください。これは、すべての AWS 製品に共通しています。

SQS を導入する方法は、AWS のあらゆるサービスと同じく簡単です。AWS アカウントをまだ持っていない場合には、まずアカウントを作成してください。次に、Amazon SQS を有効に設定します。そして最後に、AWS SDK for Java を使用して、クラウド・ベースのメッセージをパブリッシュして読み込みます (この後、メッセージを実際に作成する方法について詳しく説明します)。

SQS メッセージの作成方法

Amazon Simple Queue Service という名前にふさわしく、キューに対する読み取り/書き込みの背後にあるロジックは、単純そのものです。まずは、有効なアクセス・キーと秘密鍵を使って AWS との接続を確立してください (リスト 1 を参照)。

リスト 1. AWS との接続の確立

```
AmazonSQS sqs = new AmazonSQSClient(new BasicAWSCredentials(AWS_KEY, AWS_SECRET));
```

次に必要となるのは、キューです。AWS API で `createQueue` を呼び出すと (リスト 2 を参照)、常に新しいキューが作成されるわけではありません。キューがすでに存在する場合は、そのハンドルが返されます。SQS でのキューは単なる URL にすぎないため、キュー・ハンドルも同じく URL です。ただし、AWS SDK API では、`Queue` という URL は `String` 型であり、Java の `URL` 型ではないことに注意してください。

リスト 2. Queue のハンドルの取得

```
String url = sqs.createQueue(new CreateQueueRequest("a_queue")).getQueueUrl();
```

キューを取得すると、そのキューへのメッセージの書き込みが可能になります。SQS のメッセージ・フォーマットは、メッセージが `String` であるという点で SimpleDB のフォーマット (「参考文献」を参照) と同様です。ただし、`String` は簡単に構造化することができます。そのため、メッセージのフォーマットを有効な JSON または XML にすれば、簡単に構文解析することができます。

リスト 3. SQS を介したメッセージの送信

```
sqs.sendMessage(new SendMessageRequest(url, "It's a wonderful life!"));
```

SQS はあくまでも単純です

Amazon SQS は何よりもまず、単純であることを忘れないでください。これは、いつもは慣れているようなオマケがないことを意味します。例えば、SQS ではキューに新しいメッセージが追加されても積極的な通知を行いません。そのため、SQS キューのリーダーはキューを定期的にポーリングして、新しいメッセージの有無を確認する必要があります。ポーリング自体は難しい操作ではありませんが、アプリケーションのオーバーヘッドになることは確かです。場合によっては、このようなオーバーヘッドは許容されない可能性もあります。この問題は、Amazon SNS (Simple Notification Service) が解決しますが、それについてはこの記事では触れません。

メッセージ長には制限があります。デフォルトでは、メッセージは 8KB を超えてはなりません。これより長いメッセージを使用する必要がある場合は、いつでもメッセージを分割して、個々の部分をシーケンス ID を使って識別するという方法を使えます。こうすれば、受信側でメッセージを再び組み立てることができます。

メッセージを SQS キューに入れるためのコードは、以上のたった 3 行だけです。

AWS SDK について

SimpleDB について私が紹介した記事 (「参考文献」を参照) を読んでいれば尚更のこと、AWS SDK にはお馴染みのパターンがあることにお気付きかもしれません。AWS 内のあらゆるものは Web サービスであるため、すべての通信は HTTP 上で行われます。そのため、AWS SDK の API は `Request` 型のオブジェクトを使用して論理リクエストを模倣します。例えば、`SendMessageRequest` や `CreateQueueRequest` などのオブジェクトです。いずれの場合も、オブジェクトの名前がそのオブジェクトの意図を説明します。

他に注目する点は、SQS に置かれるメッセージは永続的であることです。メッセージは、明示的に削除されるまで存在し続けます (削除されなければ、メッセージは最終的に消滅します。自動期限切れのデフォルト値は 4 日間です)。Amazon SQS では、読み取りのためにメッセージを取得す

る際に、単純なロック・ストラテジーを適用します。したがって、読み取りイベントの場合、そのメッセージには一定期間、他の同時読み取りプロセスからアクセスすることができません。この期間は、メッセージの可視性タイムアウトと呼ばれます。可視性タイムアウトの値は、デフォルトで 30 秒に設定されますが、適当と思われる任意の期間に変更することができます。

Amazon のインフラストラクチャー内に存在するメッセージの永続性は、安心の種です。SimpleDB や、さらには S3 のように、AWS の世界にあるコンポーネントには相当な冗長性があります。そのため、1 つ、または複数のリーダー・プロセスがメッセージの処理中に突然終了したとしても、そのメッセージがまだ有効な可能性は十分にあります。その上、AWS ネットワーク内の何らかのアセットがサービスを提供しなくなったとしても、ミッション・クリティカルなメッセージが失われることはありません。メッセージは他のマシンにいくらでも存在しているはずです。最後に、他のあらゆる AWS 製品の例に洩れず、メッセージ・インフラストラクチャーの物理ロケーションは、米国やヨーロッパなど、地域別に設定することができます。

SQS メッセージを読み取る方法

SQS メッセージは、3 行のコードで SQS キューに書き込むことができます。メッセージを読み取るためのコードは、それよりもわずか数行多いだけです。実のところ、AWS との接続および同じ SQS キューのハンドルが必要な場合、最初の 2 行はまったく変わりません。Amazon SQS では、コールバック機能を提供していません。また、メッセージ到着を積極的に通知することもないため、SQS キューを定期的にポーリングして、キューに何か配信するものがないかどうかを確かめる必要があります。SQS キューを読み取る場合に必要な追加の数行は、そのためのコードです。

ポーリング・ストラテジーの実装には、少し注意しなければならない点があります。それは、メッセージを処理する前に、実際に有効なメッセージを受け取ったことを確認する必要があることです。これを怠ると、あの非情な `NullPointerException` を目にすることになります。

例えば、AWS との接続が有効になっていて、メッセージが含まれるキューのハンドルを取得済みである場合、リスト 4 に記載するコードによってメッセージを取得することができます。

リスト 4. SQS を介したメッセージの受信

```
while (true) {
 List<Message> msgs = sqs.receiveMessage(
    new ReceiveMessageRequest(url).withMaxNumberOfMessages(1)).getMessages();

 if (msgs.size() > 0) {
  Message message = msgs.get(0);
  System.out.println("The message is " + message.getBody());
  sqs.deleteMessage(new DeleteMessageRequest(url, message.getReceiptHandle()));
 } else {
   System.out.println("nothing found, trying again in 30 seconds");
   Thread.sleep(3000);
 }
}
```

リスト 4 の `sqs` への参照は、リスト 1 に示されている `AmazonSQS` 型です。このオブジェクトが提供する `receiveMessage` メソッドは、`ReceiveMessageRequest` を受け入れます。`ReceiveMessageRequest` は、キューに含まれるメッセージを規定の数だけリクエストするように構成することができます。上記の例では、一度に 1 つのメッセージだけを取得するように構

成しましたが、いくつのメッセージをリクエストするかに関わらず、`receiveMessage` メソッドは `Message` 型からなる `List` を返します。

ポリシー・ストラテジーを実装する

前述のとおり、SQS の読み取りはポーリング方式で行われます。その上、`receiveMessage` メソッドは非ブロック方式です。したがって、対応する `List` (`msgs`) に実際に何かが含まれていることを確認する必要があります。キューから何も取得できなかった場合、`ReceiveMessageRequest` での `getMessages` 呼び出しが返すのは、`null` ではなく、空の `List` です。

有効なメッセージを取得したとすれば、`getBody` 呼び出しによってメッセージのペイロードまたは本体を入手することができます。有効なメッセージのハンドルを取得すると、SQS がそのメッセージをロックすることに注意してください。デフォルトでは、メッセージで何らかの操作を行うために 30 秒の猶予が与えられます。メッセージを永久に処理しないことに決めた場合は、メッセージを削除しなければなりません。上記で `DeleteMessageRequest` を引数に取る `deleteMessage` を呼び出しているのは、そのためです。

`Message` インスタンスはその受信ハンドル (例えば `id` など) によって区別されます。ハンドルはメッセージには直接関係しません。それよりも、読み取り中のイベントに関係します。複数回読み取られたメッセージ (例えば、メッセージが削除されなかった場合や、読み取りプロセスが失敗した場合) は、複数の、ただし異なる受信ハンドルを持つことになります。そのため、メッセージを削除する場合には、`getReceiptHandle` 呼び出しによってメッセージの受信ハンドルを渡さなければなりせん。

キューを絶えずチェックしてメッセージの有無を確かめる代わりに、sleep 関数を指定して、メッセージを 1 つも取得しなかった場合には 30 秒間待機するようにします。もちろん場合によっては、スリープ状態にするのが賢明ではないことや、休止期間が長期化する結果になることもあります。

上記の数行のコードには、Amazon SQS の大部分が網羅されています。AWS SDK には他にも多数の関数と機能がありますが、SQS キューのメッセージの読み取り/書き込みを行うには、このコードで十分事が足ります。

今度は、このコードを実際に使用した場合の例を見てみましょう。

Magnus と Amazon SQS との出会い

先月の記事では、Magnus という単純なモバイル Web アプリケーションを作成して、Amazon Elastic Beanstalk が持つ機能のいくつかをデモンストレーションしました (「参考文献」を参照)。Magnus には、アカウント所有者のモバイル機器から受信した位置情報を保管する巧妙な機能があります。位置情報は、多くの人々が提供したい、または使用したいと思う情報の一例にすぎません。

誰かの居場所に関する情報を取得するというだけでも役に立ちますが、人々が実際に見たいのは図 (図と角の丸いピカピカのボタン) で表現された情報です。受け渡しするデータが大量にあるとすると、処理の観点から、図の作成および分析には相当なコストがかかることが予想されます (Hadoop のことを考えている方はいませんか？)。これに対処する 1 つの方法は、実証済みの抽

出、変換、ロードの手法、つまり ETL (Extract, Transform, and Load) 手法です。ETL はかなり広義の用語で、これにはさまざまな内容が含まれます (この ETL に関連して人々はキャリアを積み、企業はビジネスを構築しています)。しかし、ここで言う ETL は、MongoDB データを分析し、そのデータに基づいて新しい文書を作成することを意味するだけです。

Amazon SQS での ETL

データの分析となると、データを分析する目的、そして分析によって提供できる答えには、数え切れないほどの可能性があります。Magnus Web アプリケーションはこの可能性のうち、ごくわずかな部分を行い、地理座標、時刻、ユーザー・アカウントに関するデータを抽出して表示します。厳密には、Magnus が対象とするデータは、位置の緯度と経度、ユーザー・アカウント ID、タイムスタンプ、そしてこれらのデータの間の関係のみです。

Magnus では、このデータを図の形式にして、(例えば、任意の時点におけるアカウント所有者の位置をマーカーで示す地図を使用して) 地理的地域を基準にユーザー・アカウントを表示することも、アカウント所有者/ユーザーが特定の地域をどのように移動したかを (別の地図で) 示すことも考えられます。この種の情報を提供するには、オフラインで行われる ETL 式のプロセスが必要です。このデータをリアルタイムで (生成されると同時に) 提供するとしたら、処理の点であまりにもコストがかかり過ぎます。したがって、これらの分析は、「ほぼ」リアルタイムの分析であると考えてください。

Magnus で Amazon SQS を使用するためには、事前のセットアップが必要です。第一に、AWS のクレデンシャルを取得する手段がなければなりません。私のお気に入りは Play (「参考文献」を参照) なので、これをアプリケーション開発フレームワークとして使用することにします。クレデンシャルを取得するには、Play の `application.conf` ファイルを使用することができます。このファイルは、自動的に読み込まれるプロパティー・ファイルです。

リスト 5. Play の application.conf への AWS 構成データの追加

```
#AWS configuration
aws_access_key_id=1S.........MR2
aws_secret_access_key=S3.........ZM
```

プロパティーが定義された後は、Play の `Play` オブジェクトを呼び出すことによって簡単にプロパティーを取得することができます (リスト 6 を参照)。

リスト 6. Play での AWS 情報の取得

```
public class Application extends Controller {

 private static final String AWS_KEY =
    Play.configuration.get("aws_access_key_id").toString();
 private static final String AWS_SECRET =
    Play.configuration.get("aws_secret_access_key").toString();

//....
}
```

基本部分が定義されたところで、本来の仕事に取り掛かれます。リスト 7 に記載するコードは、先月の Amazon Elastic Beanstalk の紹介で使用したスニペットと似ていますが、今回は `saveLocation` を、単純な JSON 文書を「`locations_queue`」という名前のキューに置くためのコードで更新しています。この JSON は基本的に、`{"id":"4d6baeb52a54f1000001"}` のようになりま

す。保存された位置の ID が渡されるのは、メッセージの受信側が検索して分析できるようにするためです。

リスト 7. メッセージを SQS に格納するための saveLocation メソッド

```
public static void saveLocation(String id, JsonObject body) throws Exception {
 String eventname = body.getAsJsonPrimitive("name").getAsString();
 double latitude = body.getAsJsonPrimitive("latitude").getAsDouble();
 double longitude = body.getAsJsonPrimitive("longitude").getAsDouble();
 String when = body.getAsJsonPrimitive("timestamp").getAsString();

 SimpleDateFormat formatter =
   new SimpleDateFormat("dd-MM-yyyy HH:mm");
 Date dt = formatter.parse(when);

 ObjectId oid = new Location(id, dt, latitude, longitude).save();

 AmazonSQS sqs = new AmazonSQSClient(new BasicAWSCredentials(AWS_KEY, AWS_SECRET));

 Map mp = new HashMap<String, String>();
 mp.put("id", oid.toString());

 String url = sqs.createQueue(new CreateQueueRequest("locations_queue")).getQueueUrl();
 sqs.sendMessage(new SendMessageRequest(url, new Gson().toJson(mp)));

 renderJSON(getSuccessMessage());
}
```

Ruby とのデート？

メッセージが SQS キューに入れられたので、今度はキューからメッセージを取り出して、何らかの処理を行う必要があります。MOM の利点の 1 つに、異種混合のアーキテクチャーを使用できることがあったことを覚えているでしょうか。このことから、SQS のリーダー側は、Java 以外の言語で作成することも、さらには別のプラットフォームで実行することもできます。

基本的に、この分析処理に使用する手段は私の一存で決められます。そこで、流行の先端を行く開発者たちからの信頼を勝ち得るために、Ruby で処理することにします。

リスト 8 では、`right_aws` という Ruby gem の助けを借りて、SQS を処理しています。いろいろな意味で、gem は jar ファイルと見なすことができます。`right_aws` ライブラリーは Amazon の SDK for Java とよく似ていますが、right_aws ライブラリーのほうが簡潔で、使い方も遥かに簡単です。

リスト 8. Ruby での SQS を対象とした接続およびキューの作成

```
require "right_aws"
#...
sqs  = RightAws::SqsGen2.new(aws_access_key_id, aws_secret_access_key)
queue = sqs.queue('locations_queue')
```

ご覧のとおり、リスト 8 の該当する 2 行のコードが AWS との接続を確立し、「`locations_queue`」という名前のキューのハンドルを取得します。

次に、ポーリング・メカニズムを導入します (リスト 9 を参照)。`@queue` への参照は、リスト 8 と同じ `queue` 変数ですが、この場合にはクラスの一部として定義されています。そのため、リスト 9 では Ruby の `@` 構文を使用してインスタンス変数を直接参照しています。

リスト 9. SQS からのメッセージの処理

```
def process_messages()
  while true
    msg = @queue.pop
    if !msg.nil?
      handle_message(msg) # impl of which does neat stuff
   msg.delete
    else
      sleep 10
    end
  end
end
```

メッセージを `handle_message` メソッドに渡した後は、メッセージを削除することができます。1 つもメッセージが見つからない場合、メイン・スレッドが 10 秒間スリープ状態になります。`! msg.nil?` の行は、Java コードでの `msg != null` に相当します。ただし、Ruby では `null` でさえも 1 つのオブジェクトです。オブジェクトに (`nil?` メソッド呼び出しによって) `nil` 型であるかどうかを問い合わせると、ブール値が返されます。

まとめ

AWS は Web サービス・オファリングです。したがって、多種多様なプラットフォーム・ライブラリーがアクセスして利用します。Magnus には、この AWS によってもたらされる柔軟性が明らかに示されています。Magnus では Java コードを使用してメッセージを SQS キューに入れる一方、ちょっとした Ruby プログラムによって、メッセージを SQS から取り出すことができました。キューを使用するアーキテクチャーの長所の 1 つは、暗黙のうちにコンポーネントが疎結合になることです。

Web アプリケーションを GAE や Amazon Elastic Beanstalk でホストする価値があるのと同じく、クラウド・メッセージング・システムを利用することにも価値があります。Amazon の SQS を利用すれば、キューイング・システムをインストールして保守するという負担を取り除いてくれるため、あとはキューを作成し、メッセージをそこに入れ、取得するだけで済みます。あとのことは、Amazon に任せておけばよいのです。

著者について

Andrew Glover

Andrew Glover は、ビヘイビア駆動開発、継続的インテグレーション、アジャイル・ソフトウェア開発に情熱を持つ開発者であるとともに、著者、講演者、起業家でもあります。また、easyb BDD (Behavior-Driven Development) フレームワークの創始者、そして『継続的インテグレーション入門 開発プロセスを自動化する47の作法』、『Groovy in Action』、『Java Testing Patterns』の 3 冊の本の共著者でもあります。詳細は彼のブログにアクセスするか、Twitter で彼をフォローしてください。