サーバーレスアーキテクチャで何が変わるのか

ソリューションカンパニー
ソリューションビジネス第二部
鹿谷　崇志

はじめに

これまでもコンピュータを取り巻く環境、特にアプリケーション実行環境は大きく変化してきた。データ処理形態は、バッチ処理からオンラインリアルタイム処理へ、また、アプリ実行形態は、メインフレームを中心としたセンター集中処理モデルから、分散処理、3層アーキテクチャへと変化してきた。エンタープライズアプリケーションにおいても、HTML5やJQueryなどでAJAXを用いた3層アーキテクチャ（UI（データチェック、画面生成）、アプリケーション（ビジネスロジック）、データベース（RDB、NoSQL））が主流となりつつある（Webシステムの開発基盤であるAZAREA-Geneもこのモデルを採用）。

一方、アプリケーション実行インフラは、メインフレーム、オフコン、IA（インテルアーキテクチャ）サーバー、HW仮想化、パブリッククラウド（IaaS）と変化してきた。前半はアプリケーション実行形態の変化に応じたものであるが、後半はインフラ管理コスト削減を目的とした変化ととらえることができる。さらに最近ではインフラストラクチャのほとんどの部分にマネージドサービスを利用したサーバーレスアーキテクチャという形態も登場している。先進的な企業ではすでに試験的に採用を始めている。本稿ではサーバーレスアーキテクチャとは何か、どんな効果が期待できるのかを実際にAWSのLambdaを用いたPoC検証の結果を元に紹介していく。

1.　サーバーレスアーキテクチャの定義

おそらく残念ながら（執筆時点では）サーバーレスアーキテクチャの明確な定義は存在しない。クラウドといえばまず頭に浮かぶのがSalesforce等のSaaSである。SaaSでは、アプリケーションを含むインフラ管理のほとんどが不要である。当然アプリケーションの初期設定は必要となるが、利用者は業務に集中することができる。一方、企業向けアプリケーションの実行環境として広く使われているのが仮想HW環境をサービス提供するIaaSである。IaaSでは、ファシリティ、ネットワーク、ハードウェア、ハイパーバイザはクラウド事業者側で管理されているが、仮想HW上のOS、ミドルウェア、データ、アプリケーションの管理責任は利用者側にある。データベースサービス、メッセージングサービスなどのPaaSでは、OS、ミドルウェアまではクラウド事業者で管理されているが、処理量に応じたサーバー数の増減、メモリー設定のチューニングなどインフラ運用がなくなるわけではない。

では、サーバーレスアーキテクチャとはどのようなものであるか。サーバーレスとはいってもサーバーが不要なわけではなく、サーバーの管理が不要（クラウド提供者によって管理されている）ということである。利用者はアプリケーションやデータを配置するだけでよく、サーバーを意識する必要はない。課金の仕組みもサーバー台数や稼働時間によるものではなく、実際に処理に要した時間やトランザクション数、データ容量で課金される。一概には言えないが、散発的なトランザクション処理においてはIaaSのサーバー利用料金よりかなり低くなることが多い。AWSにおけるサーバーレスアーキテクチャとは、このようなサーバーレスなサービス群を組み合わせてアプリケーションを実行する環境を指す。本稿では、サーバーインフラを全く意識（管理・運用）する必要のないフルマネージドサービスを組み合わせたシステムを、サーバーレスアーキテクチャと定義することとした。

2.　AWSのクラウドサービス

本題に入る前にAWSの代表的なサービスをインフラ管理の面から分類してみた。

（ア）IaaSに代表される仮想HW提供サービス

① EC2（Elastic Computing Cloud）

仮想HWを提供するAWSの代表的なサービス。

インスタンス（CPU、メモリー）、EBS（ストレージ）、セキュリティグループ（ファイアウォール）、EIP（公開IP）等で構成さ

れ、Intelアーキテクチャのもと、幅広いOSに対応している。

利用者は、各構成要素について設計、設定する必要がある。

また、OSやミドルウェアの運用は利用者で行う必要があり、セキュリティパッチやアップデートなど通常のサーバーOSと同様なインフラ運用が必要となる。

（イ） PaaSに分類されるもの

マネージドサービスに分類されるが、CPU性能、メモリーサイズ、DISK容量などインフラ環境の管理・運用が必要となる。

① RDS（Relational Data Base Service）

Oracle、MySQL、MSSQLServerなどのデータベースを提供するサービス。

利用者は、サーバーのタイプ（メモリー、CPU数）、DISK容量、DISKのタイプ（SSD、Magnetic）などのインフラ設定と、データベースシステムの設定パラメータ（バッファサイズ、文字コードなど）を設定、管理する必要がある。ログのアーカイブ、定期バックアップ、リカバリ、障害時のフェールオーバなどの管理機能は提供される。

② ECS（EC2 Container Service）

Dockerコンテナを実行するための実行環境及びクラスタの管理サービス。

利用者は、CPUユニット数、メモリー制限などを設定する必要がある。コンテナのデプロイや、クラスタの維持などの管理機能は提供される。

（ウ） サーバーレスに分類されるもの

一般的にはアプリケーション実行プラットフォームを指すことが多いが、フルマネージドなデータストアサービスもここに分類した。

① Lambda

AWSのサーバーレスアーキテクチャの中心的な存在。イベントドリブンなアプリケーション実行基盤。実行言語はJava、C#、JavaScript（Node.js）、Pythonをサポートしている。利用者は、処理に必要なメモリーサイズを指定する。CPU処理能力はメモリーサイズに比例する。簡易なバージョン管理機能を備えているので、ステージング環境も容易に利用できる。

② API Gateway

LambdaやAWSのサービスAPIをREST（Web Service）で公開するためのゲートウェイサービス。

利用者は、簡単なインタフェースマッピング（REST JSONフォーマットとバックエンドI/Fのマッピング）と、認証ポリシーを指定するだけで、APIをインターネットへ公開することができる。リソース制御機能で、トランザクションレートやクオータを制限できる。

APIキャッシュ機能で、同一リクエストのバックエンドへの負荷を軽減する機能もある。

③ DynamoDB

フルマネージド型のNo-SQLデータベース。

利用者は、テーブルの作成の読み書きのスループットを指定するだけで、自動的にプロビジョニングされる。

指定したスループットに応じ課金される。

④ Kinesis

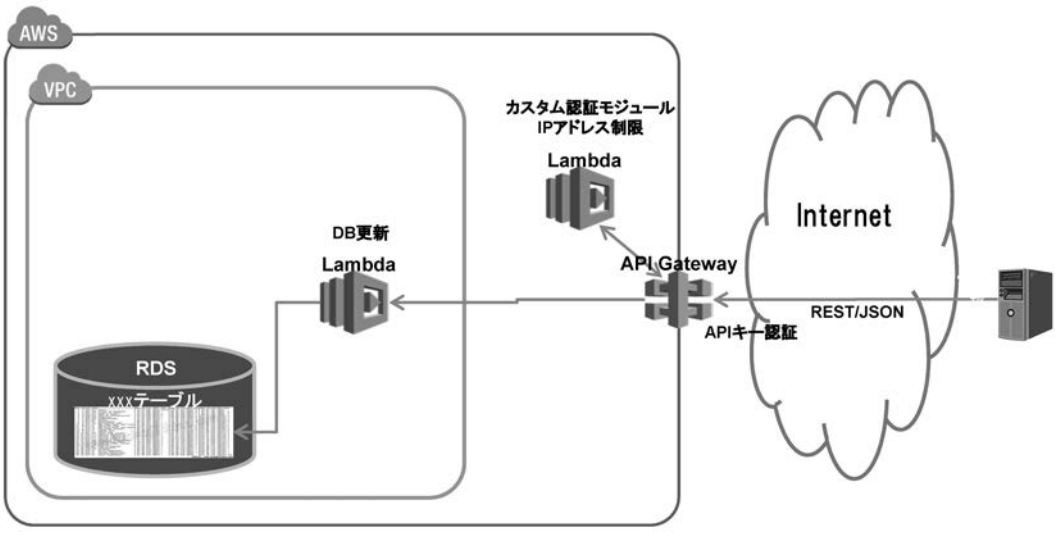
非常に多数の同時接続（センサーデバイスなど）からのデータをシャードというストリーミングキューへ収集し、連続的に処理を行うためのデータストア基盤。

実際のデータ処理にはLambdaなどが用いられる。

3. API GatewayとLambdaの検証

今回のPoCでは、外部サービスからREST APIを用いてデータをデータベースへ書き込むという処理を、API GatewayとLambdaでRDS（MySQL）へ書き込むというシナリオで検証した。API Gatewayは、インターネットへ公開されたAPIなので、利用サーバーを限定しセキュリティを保つため

図1 PoC検証イメージ



に、IPアドレスで利用制限するためのカスタム認証モジュールを作成し検証をした（図1）。

（ア） Lambdaの構築と運用

構築手順（概要）

① IAM Roleの作成

Lambda Functionの実行に必要なAWSサービスへのアクセス権限の設定を行う。

② Lambda Functionの作成

Function名を指定し、以下の設定パラメータを記述する。

設定パラメータは、事前に作成したIAM Roleと、メモリーサイズ、最大実行時間、実行環境などを指定する。当PoCではRDS（リレーショナルデータベースサービス）を用いるためVPC（仮想的なプライベートネットワーク）上での実行環境を選択した。

③ アプリケーションの作成とデプロイ

当PoCでは、RDBへのアクセスがあるため、O/Rマッパの豊富なJava言語で実装することにした。開発環境は、EclipseにAWSから提供されているLambda SDKを導入することで、通常のJavaアプリケーション開発と同等な生産性を得ることができた。プログラムのバージョン管理機能があらかじめ備わっているので、本番実行バージョンをそのままで、プログラム保守、テストが可能となっている。1つ難点としては単体テストが容易に行えないということである。Eclipse上でLambda SDKによりコードチェックなどは可能であるが、実行エンジンが提供されているわけではないので、実行テストを行うためには都度Lambdaへデプロイし実環境でテストを行う必要がある。クライアント側でのデバック用Lambdaエンジンの提供など今後の改善に期待したい。

運用・監視

監視機能としてはCloudWatchのパフォーマンス情報が提供されており、呼び出し回数、レスポンス、エラー回数などが監視可能だ。また、アプリケーションが書き出すログは、CloudWatch Logsで管理される。

インフラ監視が必要なものは唯一スロットル回数だけである。Lambdaの同時実行数はデフォルトで100となっている。同時実行数が100を超えるとスロットルが発生し、呼び出し元プログラムにはエラーが応答される（非同期呼び出しの場合は、AWS側で再処理が行われる）。呼び出し側アプリケーションは時間をおいて再度実行を要求しなければならない。頻繁にスロットルが発生する場合は、サポートに申請することで同時実行数の上限を増加することが可能だ。

アプリケーション監視は、Cloudwatchのエラー回数を監視し、エラー発生時はログを分析し原因究明を行うことができる。問題分析を容易に行うためには、アプリケーションはエラー発生時に問題分析に必要な内容を出力する必要がある。

パフォーマンスに関しての設定はメモリーサイズだけである。メモリーサイズを大きくするとCPU性能も比例するようになっており、目標のレスポンスタイムに応じメモリーサイズを増減させるだけである。

（イ） API Gatewayの構築と運用

設計と構築

① 外部設計（I/F設計）

API Gatewayは、Web API（REST）とバックエンド（Lambda Function）のインタフェース変換が主な機能である。

外部アプリケーション（クライアント）とAPI Gateway間のI/F（REST）設計と、Lambda Functionを呼び出すときのI/F（JSON）の設計を行う。API（アプリケーション）ごとに設計してもよいが、ある程度APIの目的別に外部I/Fは標準化しておいたほうが良いだろう。当PoCでは、APIの処理目的別に標準的なI/Fを設計することとした。考慮すべき点は、将来の拡張性を意識して設計することであろう。

② API Gatewayの設計

API Gatewayは、1つのAPIが1つの独立したURLを持つ。Lambda Functionとのマッピングは、1API:1Lambda Functionでもよいし、1API:複数Lambda Functionでもよい。この場合URLの下のPathごとにLambda Functionがマッピングされる。前者は、1対1の関係であるため、API Gateway設定変更時の影響範囲が限定されるという利点があるが、API数が非常に多数となってしまう。拡張可能ではあるが、アカウント当たりのAPI数のデフォルトの上限は60である。後者は、API Gateway変更時の影響範囲が配下のLambda Functionに及ぶ。利点としては、API数を削減できることと、同一URL下であることでCookieを共有可能ということである。当PoCでは、サブシステム（アプリケーション）単位でAPIを定義し、機能別（データ種別ごと）にLambda Functionを作成しPathにマッピングすることとした。

③ API Gatewayの構築

API Gatewayの構築は、Web API（REST）とLambda Functionのインタフェースのマッピングを定義するのが主な設定項目である。外部設計で作成したI/Fを基にリクエストとレスポンスのマッピングを定義する。

リクエスト（Web API→Lambda）のマッピングは、主にPOSTのBody部（JSON形式）とQueryストリング、Cookieなどのインタフェースを Lambdaへ受け渡すオブジェクト（JSON形式）に変換するためのマッピングを定義する。レスポンス（Lambda→Web API）のマッピングは、Lambdaからのレスポンスオブジェクト（JSON形式）に応じ、HTTPレスポンスコードへのマッピングと、HTTP Content Body（JSON形式）へのマッピングを定義する。そのほかに、認証方式の選択、スロットル制限（秒あたりのリクエスト数）の設定、APIキャッシュの設定、カスタム認証モジュールの設定などがある。

当PoCでは、クライアント側のIPアドレスによる利用制限が要件であったため、カスタム認証モジュールを作成し、IPアド

レスによる認証を検証した。

④ API（URL）とLambda Functionの関連付け

API Gatewayには、複数ステージで実行を行う機能が備わっており、URLの一部にステージ識別IDを含めることで、本番、ステージング、テストなどのステージング環境が簡単に得られる。Lambdaとの連携も容易で、ステージごとにLambda Functionのどのバージョンを実行するかを定義できるようになっている。

運用監視

監視機能としてCloudWatchのパフォーマンス情報と、API GatewayのエラーメッセージがCloudWatchログへ出力される。

インフラ面で監視が必要なものは、HTTPエラーコードが主なものとなる。5XX、4XXエラーの発生時には、ログを分析し、アプリケーションのエラーの場合は、Lambda Functionのログから原因究明を行うことができる。

スロットル制限でエラーとなっている場合は、制限値の増加を検討するだけである。従来のWebサーバーシステムのように、サーバーの台数、CPU、メモリー、DISK容量などの監視やチューニング、高可用性のための冗長化の検討などは不要だ。

4. サーバーレスで何が変わるのか

（ア）インフラ管理面で見たサーバーレスアーキテクチャ

インフラ管理面から見たサーバーレスアーキテクチャは、究極の運用レスいうことができる。これまでも、クラウド化（IaaSやPaaS）によってインフラ運用業務がなくなると言われていたが、実際のところは、OS以上の管理、運用は利用者側の責任範囲であり、DISKのバックアップやリカバリ、サーバーの稼働監視と障害発生時の対応、セキュリティパッチの適用、定期的なWindows Update、SPの適用など、サーバー単体で見たらそれほどインフラ運用作業が削減されることはなかった。

サーバーレス（LambdaとAPI Gateway）のケースでは、場合によっては構築フェーズから開発側で実施することが可能となっている。高度なインフラ知識や高可用性構成の設計などは不要であり、必要な性能目標と、アプリケーションインタフェースの設定さえ行えば、アプリケーションが実装できてしまうのである。

強いてインフラ担当の役割をあげるのであれば、同時実行プロセス数、スロットル制限などの制限値に達していないかの監視と制限値の増加と、AWS側のサービス障害時のサポート問い合わせなどが残された役割となる。

（イ）アプリケーション実行環境としてのサーバーレス

もともとアプリケーション開発ではインフラ環境を意識することは少なかった。

サーバーレスなアプリケーション実行環境であるLambdaは、java、C#、JavaScript（Node.js）、Pythonに対応している。API Gatewayが、HTTPの状態管理、認証機能の提供、リクエスト時のJSONからオブジェクトへの変換、レスポンス時のオブジェクトからJSONへの変換を行うため、Lambdaアプリケーションは、API Gatewayから受け渡されたオブジェクトを処理し、オブジェクトとして結果を応答するだけの簡単なプログラムで、REST APIを実装することができる。API GatewayとLambdaを組み合わせることでRESTfulなAPIを容易に作成することができるのである。

ただし、Apatch/Tomcat環境などのように細かなパラメータの設定等はない代わりに、いくつかの制約事項を守る必要がある。アプリケーション要件によってはその制約を回避するための機能を実装する必要がある場合がある（表1）。

表1　主要な制限事項

	リソースまたはオペレーション	デフォルトの制限	拡張可否
Storage Gateway	アカウントあたりのスロットル制限	1 秒あたり 1000 回のリクエスト (rps) で、バースト制限は 2000 rps	はい
	アカウントあたりの API 数	60	はい
	アカウントあたりの使用プラン	300	はい
	API ごとのカスタム認証	10	はい
	API あたりのステージ数	10	はい
	統合のタイムアウト	Lambda および HTTP 統合の両方について 30 秒	いいえ
	ペイロードサイズ	10 MB	いいえ
Lambda	リクエストあたりの最大実行時間	300 秒	いいえ
	「Invoke」リクエスト本文のペイロードサイズ(RequestResponse)	6 MB	いいえ
	「Invoke」リクエスト本文のペイロードサイズ(Event)	128 K	いいえ
	「Invoke」レスポンス本文のペイロードサイズ(RequestResponse)	6 MB	いいえ
	同時実行数	100	はい
	Lambda 関数デプロイパッケージのサイズ (.zip/.jar ファイル)	50 MB	いいえ
	リージョンあたりの、アップロードできるすべてのデプロイパッケージの合計サイズ	75 GB	いいえ
	デプロイパッケージ (非圧縮 zip/jar サイズ) に圧縮できるコード/依存関係のサイズ	250 MB	いいえ

今回の検証では、一度のリクエストで多数（Max5000件）のデータ更新を想定していたため、制約事項の1つであるGateway統合のタイムアウト30秒が課題となった。データ更新中に30秒に達してしまった場合、処理が中断され再処理を余儀なくされてしまうが、全件を再実行しても再度30秒を超過してしまうことが予想された。この問題を回避するために、一度のリクエストで受け取ったデータの部分再送を可能とするように再送機能を実装する必要があった。処理時間が30秒に達する前にそれまでの処理をコミットし、データ行ごとの処理ステータスをクライアント側に返すことで部分再送を実現した。

サーバーレスアーキテクチャでは、IaaSでの環境構築とは異なり、フレームワークの制約に合わせた外部設計（特にI/F設計）が重要となる。RESTの電文形式はもとより、HTTPレスポンスに対するクライアント側に求める動作も定義しておく必要がある（表2）。

（ウ）サーバーレスアーキテクチャで変わること

検証の結果判明したことは、インフラ運用すべき項目がほとんどないということである。オンプレミスのシステムの場合、性能やセキュリティを維持するための監視、運用や、H/W、S/Wの老朽化対応などのライフサイクルマネジメント、そしてHW障害の対応など、運用フェーズに入った後にも多くのインフラ運用作業が存在した。サーバーレスアーキテクチャではそのほとんどをマネージドサービスの組み合わせで実現するので、アプリケーションの保守、運用が必要なだけで、インフラ運用はほとんど存在しないことになる。

以下に、一般的なシステムとのインフラ運用項目の比較（表3）と、サーバーレス環境で管理可能な監視項目（表4）を記す。

表2　HTTPレスポンスとクライアント側動作

ステータス	概要	クライアント側の動作
200	正常（部分エラーを含む）	Resultコードを確認し部分エラーの場合は、エラーデータのみを再送する。
400	JSON形式エラー	プログラム・データを確認し再送
401	認証エラー	認証キーを確認し再送
403	APIキーエラー、無効なHTTPメソッド	プログラム・データを確認し再送
413	データサイズ超過	データを修正し再送
415	サポートされていないContext-Typeの指定	プログラム・データを確認し再送
429	リクエスト数が設定値を超えた場合	時間をおいて再送
500	サーバー側処理エラー（再送不可）	サーバー側の復旧を待って再送
503	サーバー側一時エラー（再送可）	時間をおいて再送
504	API Gateway タイムアウト(30秒)	時間をおいて再送

表3　運用項目比較

	管理対象・運用項目	オンプレミス	サーバーレス
ファシリティ	スペース・電源容量・空調能力	○	
	入退室管理、搬入搬出管理	○	
ネットワーク	回線帯域、品質	○	
	ルータ、ファイアウォール、LB、SWの余力、空きポート	○	
	ルータ、ファイアウォール、LBのパッチ管理	○	
ハードウェア	資産管理、余力、保守	○	
	障害管理、対応、冗長化	○	
ソフトウェア・フレームワーク	構成管理、導入、保守	○	△（パラメータ設定）
	パッチ適用、バージョンアップ	○	
アプリケーション	開発、実装	○	○
	アプリケーション保守	○	○
	障害対応	○	○

表4　監視項目一覧

	項目名	単位	説明
Lambda	Invocation	Count	Functionの呼び出し回数。成功および失敗した呼び出しが含まれる。
	Errors	Count	失敗した呼び出し回数。同時実行制限エラー、内部サービスエラーを除く回数。
	Throttles	Count	同時実行数制限により失敗した呼び出し回数。
	Duration	mSec	Function実行時間
	ログ	Text	アプリケーションが書きだしたログおよびLambdaのログ
API Gateway	Count	Count	APIメソッドの呼び出し回数
	IntegrationLatency	mSec	バックエンド処理時間
	Latency	mSec	APIの処理時間
	CacheHitCount	Count	APIキャッシュにヒットしたリクエストの数
	CacheMissCount	Count	APIキャッシュにヒットせず、バックエンドから提供されたリクエストの数
	4XXError	Count	クライアント側のエラー数
	5XXError	Count	サーバー側のエラー数
	ログ	Text	API Gatewayが出力するログ

日常の稼働監視では、性能（レスポンス、スロットル）とエラー（エラー回数、アプリケーションログ）の監視は最低限行う必要があるが、これらについてもAWSで用意されている監視機能（CloudWatch）を活用することで自動化が可能だ。

このように、サーバーレスアーキテクチャを利用することで、アプリケーションのみを意識すればよい理想的なコンピューティング環境を手に入れることができるのである。

おわりに

コンピュータシステムの黎明期であった1961年頃にすでにユーティリティコンピューティングという概念があった。電力、水道、電話と同じようにコンピューティングが公共設備となるであろうという考えである。それから50年余りたった現在、サーバーレスアーキテクチャを以てようやくその域へ到達したように思える。長らくコンピュータシステムは、ファシリティ（データセンター）、ネットワーク、ハードウェアなどのIT基盤と、そのうえで動くミドルウェア（データベース、アプリケーション処理基盤）、アプリケーションで構成されており、これらの構築、管理、運用がSIerの主な生業であった。近い将来、IT基盤、ミドルウェアは、クラウドサービスベンダーからサービス提供されるものとなり、狭義のインフラ運用技術者の仕事はなくなってしまうことが考えられる。このような変化の中を生き抜いていくためには、よりお客様視点（ビジネス、業務）に立って、どのようなIT技術（クラウドサービス）の組み合わせがお客様にとって最適なシステムアーキテクチャであるかを提案できる人材となっていく必要がある。クラウドサービスは、常に新しい概念、新しいサービスが生まれてくる。IT技術者はいつまでも勉強し続ける必要がありそうだ。

参考文献

「Serverless Architectures」、Martin Fowler、2016年8月4日
<http : //martinfowler.com/articles/serverless.html>
「ユーティリティ・コンピューティング　そのコンセプトの出自と背景」、P42、Open Enterprise Magazine Dec 2003
<http : //www.itmedia.co.jp/enterprise/0405/16/0517_feature.pdf>
「Amazon API Gatewayとは」、Amazon Web Services（AWS）ドキュメント
<http : //docs.aws.amazon.com/ja_jp/apigateway/latest/developerguide/welcome.html>
「AWS Lambdaとは」、Amazon Web Services（AWS）ドキュメント
<https : //docs.aws.amazon.com/ja_jp/lambda/latest/dg/welcome.html>