

# AWSによるクラウド入門

Tomoyuki Mano

Version 1.0, May 2020

# Table of Contents

1.はじめに .....	1
1.1. 本講義の目的・内容 .....	1
1.2. 本講義のフィロソフィー .....	1
1.3. 必要な計算機環境 .....	2
1.4. 前提知識 .....	2
1.5. AWSアカウント .....	3
1.6. 講義に関連する資料 .....	3
1.7. 本書で使用するノーテーションなど .....	3
2.クラウド概論 .....	4
2.1. クラウドとは? .....	4
2.2. なぜクラウドを使うのか? .....	5
3.AWS入門 .....	7
3.1. AWSとは? .....	7
3.2. AWSの機能・サービス .....	7
3.3. AWSでクラウドを作るときの基本となる部品 .....	8
3.4. Region と Availability Zone .....	8
3.5. AWSでのクラウドの開発 .....	10
3.6. CloudFormation と AWS CDK .....	13
4.Hands-on #1: 初めてのEC2インスタンスを起動する .....	17
4.1. 準備 .....	17
4.2. アプリケーションの説明 .....	18
4.3. プログラムを実行する .....	21
4.4. 講義第一回目のまとめ .....	26
5.クラウドで行う科学計算・機械学習 .....	28
5.1. なぜ機械学習をクラウドで行うのか? .....	28
5.2. GPU による機械学習の高速化 .....	29
6.Hands-on #2: AWSでディープラーニングの計算を走らせる .....	31
7.課題 .....	32
8.Appendix .....	33
8.1. AWS Educate のアカウント作成 .....	33
8.2. Python, node.js のインストール .....	34
8.3. AWS CLI のインストール .....	34
8.4. AWS CDK のインストール .....	35
8.5. Python <code>venv</code> クイックガイド .....	36
9.著者紹介 .....	38
10.ライセンス .....	39

# Chapter 1. はじめに

## 1.1. 本講義の目的・内容

本講義は、東京大学計数工学科で2020年度S1/S2タームに開講されている"システム情報学特論"の一部として行われるものである。

本講義(計3回)の目的は、クラウドの初心者を対象とし、クラウドの基礎的な知識・概念を解説する。また、Amazon Web Service (AWS)の提供するクラウド環境を実例として、具体的なクラウドの利用方法をハンズオンを通して学ぶ。

特に、科学・エンジニアリングの学生を対象として、研究などの目的でクラウドを利用するための実践的な手順を紹介する。知識・理論の説明は最小限に留め、実践を行う中で必要な概念の解説を行う予定である。受講生が研究などでクラウドを利用する際の、足がかりとなることができればこの講義の目的は十分達成されたことになる。

講義スケジュールは以下の通りである。

Table 1. 講義スケジュール

講義	ハンズオン
第一回 (6/24)	AWSに自分のサーバーを立ち上げてみる
第二回 (7/1)	AWSでディープラーニングをやってみる
第三回 (7/8)	AWSでデータベースを作ってみる

講義初回は、クラウドの基礎となる概念・知識を解説する。セキュリティやネットワークなど、クラウドを利用する上で最低限おさえなければいけないポイントを紹介する。ハンズオンでは、自分だけのマイ・サーバーをAWSに立ち上げる方法を紹介する。

第二回では、クラウド上で科学計算(特に、機械学習)を走らせるための手法を紹介する。併せて、[Docker](#)とよばれる仮想計算環境に自分のプログラムをパッケージングする手順を説明する。ハンズオンでは、AWSのクラウドでJupyter notebookを使って簡単な機械学習の計算を走らせる課題を実践する。

第三回では、Serverless architectureと呼ばれる最新のクラウドのアーキテクチャを紹介する。これは、サーバーの処理能力を負荷に応じてより柔軟に拡大・縮小するための概念であり、それ以前(Serverfullとしばしば呼ばれる)と質的に異なる設計思想をクラウドに導入するものである。この実践として、ハンズオンでは簡単なデータベースをクラウド上に作成する。

## 1.2. 本講義のフィロソフィー

本講義のフィロソフィーを一言で表すなら、"ロケットで宇宙まで飛んでいって一度地球を眺めてみよう!" である。どういうことか?

ここでいう"地球"とは、クラウドコンピューティングの全体像のことである。言うまでもなく、クラウドという技術は大変奥が深く、幾多の情報技術・ハードウェア・アルゴリズムが精緻に組み合わさってできた総体である。クラウドを理解するということは、それら一つ一つの概念の深い理解が求められる。

そして、ここでいう"ロケット"とはこの講義のことである。この講義では、ロケットに乗って宇宙まで飛び立ち、地球(クラウド)の全体を自身の目で眺めてもらう。その際、ロケットの成り立ちや仕組み(クラウドを支える要素技術)は深くは問わない。将来、自分が研究などの目的でクラウドを利用することになった時に、改めて学んでもらえれば良い。本講義の目的はむしろ、クラウドの最先端に実際に触れ、そこからどんな景色が見えるか(どんな応用が可能か)を実感してもらうことである。

そのような理由で、本講義はとてもとても盛りだくさんなものになっている。第一回はクラウドの基礎から始め、二回目では一気にレベルアップし機械学習(ディープラーニング)をクラウドで実行する手法を解説する。さらに第三回

目では,サーバーレス・アーキテクチャというここ数年のうちに確立した全く新しいクラウドの設計について解説し,それを用いて簡単なデータベースを作成する.正直に言って,一学期まるまるを費やして学んでもよいくらいの内容である.

が,この口ケットにしがみついてきてもらえば,とてもエキサイティングな景色が見られることを約束したい.



Figure 1. 宇宙からみた地球 (Image from NASA <https://www.nasa.gov/image-feature/planet-of-clouds>)

### 1.3. 必要な計算機環境

講義では,AWS上にクラウドを展開するハンズオンを実施する.そこで紹介するプログラムを実行するため,以下の計算機環境が必要である.

- **UNIX系コンソール:** ハンズオンで紹介するコマンドを実行したり,SSHでサーバーにアクセスするため,UNIX系のコンソール環境が必要である.MacまたはLinuxのユーザーは,OSに標準搭載のコンソールを使用すればよい.Windowsのユーザーは,[Windows Subsystem for Linux \(WSL\)](#)を使ってUbuntuの仮想環境をインストールすることを推奨する.WSLのインストールについては,[公式ドキュメンテーションを参照](#)のこと.
- **Dockerのインストール:** 講義ではDockerの使い方を解説する.予め自身の計算機にDockerのインストールをしておくこと.Linux/Mac/Windowsのインストール法については[公式ドキュメンテーションを参照](#).執筆時点において,[Windows 10 Home](#)へのインストールには注意が必要である.詳細は[こちらのドキュメンテーションを参照](#)のこと.
- **Python** (Version 3.6以上推奨)
- **node.js** (version 10.0以上推奨)

### 1.4. 前提知識

本講義を行うにあたり,前提知識は特に仮定しない.が,以下の前提知識があるとよりスムーズに理解をができるだろう.

- **Pythonの基本的な理解:** 本講義ではPythonを使ってプログラムの作成を行う.使用するライブラリは十分抽象化されており,関数の名前を見ただけで意味が明瞭なものがほとんどであるので,Pythonに詳しくなくても心配はない.
- **Linuxコマンドラインの基礎的な理解:** クラウドを利用する際,クラウド上に立ち上がるサーバーは基本的にLinuxである(Windowsを選択することもできなくはないが一般的ではない).Linuxのコマンドラインについて知識があると,トラブルシュートなどが容易になる.

## 1.5. AWSアカウント

本講義のハンズオンを実行するにあたり,個人のAWSアカウントの取得が必要である.

AWSには無料利用枠(Free tier)というものがあり,少量の計算であれば無料で行える(詳細).講義のハンズオンではなるべく無料枠内で完結できるよう努めるが,場合によっては多少のコストが発生する可能性がある点はご承知いただきたい.

また, [AWS Educate](#) を介してアカウントを作成すれば,\$30分の利用クーポンが手に入るので(2020/05時点),ぜひ利用していただきたい.



AWS Educate の登録手順についてはAppendixに記載してある.

## 1.6. 講義に関連する資料

ハンズオンで使うプログラム,および教科書とそのソースコードは以下のウェブページで公開している.

<https://gitlab.com/tomomano/intro-aws>

## 1.7. 本書で使用するノーテーションなど

- ・ プログラムのコードやシェルのコマンドは `monospace letter` で記述する.
- ・ シェルに入力するコマンドは,それがシェルコマンドであると明示する目的で,先頭に \$ がつけてある. \$ はコマンドをコピー&ペーストするときは除かなければならない. 逆に,コマンドの出力には \$ はついていない点に留意する.

また,以下のような形式で注意やチップスを提供する.



追加のコメントなどを記す.



発展的な議論やアイディアなどを紹介する.



陥りやすいミスなどの注意事項を述べる.



絶対に犯してはならないミスを指摘する.

### もっと勉強したい人へ

クラウドの実践的な教材としては,以下のリソースが役に立つだろう.

- ・ [AWS Educate](#) (AWSが公式で提供している学生向けのクラウド学習教材.オンラインの教科書・動画・コーディングなどで学べる.学生ならば受講は無料.)
- ・ [AWS公式ドキュメント](#) (クラウド技術は日進月歩であり,新しい機能やサービスが毎年のように更新され,昔の情報はどんどん古くなっていく.公式ドキュメンテーションで常に最新の情報を参照すべし.)

また,クラウドというのは様々な情報技術の総体である.クラウドのインフラを支える理論について興味がある場合は,ネットワーク・OS・データベースなど,個別のテーマに絞った本を参照することをおすすめする.

## Chapter 2. クラウド概論

## 2.1. クラウドとは？



クラウドとはなにか?クラウドという言葉は、それ自身がとても広い意味を持つので、厳密な定義付けを行うことは難しい。

学術的な意味でのクラウドの定義づけをするとしたら、NIST(米国・国立標準技術研究所)による [The NIST Definition of Cloud Computing](#) が引用されることが多い。

これによると、クラウドとは以下の要件が満たされたハードウェア/ソフトウェアの総体のことをいう。

- **On-demand self-service** - 利用者のリクエストに応じて計算資源が自動的に割り当てられる。
  - **Broad network access** - 利用者はネットワークを通じてクラウドにアクセスできる。
  - **Resource pooling** - クラウドプロバイダーは、所有する計算資源を分割することで複数の利用者に計算資源を割り当てる。
  - **Rapid elasticity** - 利用者のリクエストに応じて、迅速に計算資源の拡大あるいは縮小を行うことができる。
  - **Measured service** - 計算資源の利用量を計測・監視することができる。

…と、いわれても抽象的でよくわからないかもしれない。もう少し具体的な話をする。

個人が所有する計算機で、CPUの数を増やそうと思ったら、物理的に筐体を開け、CPUソケットを露出させ、新しいCPUに交換する必要があるだろう。あるいは、ストレージがいっぱいになってしまったなら、古いディスクを抜き取り、新しいディスクを挿入する必要がある。新しい計算機を買ったときには、LANケーブルを差し込まないとネットワークには接続できない。

クラウドでは、これらの操作がプログラマティックに(コマンドを通じて)実行できる。CPUが1000個欲しいと思ったらならば、そのようにクラウドプロバイダーにリクエストを送れば良い。すると、数分もしないうちにそのような計算資源が割り当てられる。ストレージ・ネットワークなどについても同様のことが行える。使い終わったら、そのことをプロバイダーに伝えれば、割り当て分はすぐさま削除される。クラウドプロバイダーは、使った計算資源の量を正確にモニタリングしており、その量をもとに利用料金の計算が行われる。

このように、クラウドの本質は物理的なハードウェアの仮想化・抽象化であり、利用者はコマンドを通じて、まるでソフトウェアの一部かのように、物理的なハードウェアの管理・運用を行うことができる。もちろん、背後では、データセンターに置かれた膨大な数の計算機が稼働しており、クラウドプロバイダーはそれらを上手にやりくりし、ソフトウェアとしてのインターフェースを提供することで、このような仮想化・抽象化を達成しているわけである。クラウドプロバイダーの視点からすると、大勢のユーザーに計算機を貸し出し、データセンターの稼働率を常時100%に近づけることで、利益率の最大化を図っている。が、利用者としてはそういった背後のあれこれを気にする必要は(基本的に)ない。

著者の言葉で、クラウドの重要な特性を定義するならば、以下のようなになる。

クラウドとは計算機ハードウェアの抽象化である。つまり、物理的なハードウェアをソフトウェアの一部かのように自在に操作・拡大・接続することを可能にする技術である。

## 2.2. なぜクラウドを使うのか？

上述のように、クラウドとは自由に計算資源をプログラマティックに操作することができる計算環境である。ここでは、自前の計算環境と比べて、なぜクラウドを使うと良いことがあるのかについて述べたい。

### 1. 自由にサーバーのサイズをスケールできる

なにか新しいプロジェクトを始めるとき、あらかじめ必要なサーバーのスペックを知るのは難しい。いきなり大きなサーバーを買うのはリスクが高い。一方で、小さすぎるサーバーでは、後のアップグレードが面倒である。クラウドを利用すれば、プロジェクトを進めながら、必要な分だけの計算資源を確保することができる。

### 2. 自分でサーバーをメンテナンスする必要がない

悲しいことに、コンピュータとは古くなるものである。最近の技術の進歩の速度からすると、5年も経てば、もはや当時の最新コンピューターも化石と同じである。5年ごとにサーバーを入れ替えるのは相当な手間である。またサーバーの停電や故障など不意の障害への対応も必要である。クラウドでは、そのようなインフラ整備はプロバイダーが自動でやってくれるので、心配する必要がない。

### 3. 初期コスト0

自前の計算環境とクラウドの、経済的なコストのイメージを示したのが下図である。クラウドを利用する場合の初期コストは基本的に0である。その後、使った利用量に応じてコストが増大していく。一方、自前の計算環境では、大きな初期コストが生じる。その分、初期投資後のコストの増加は、電気料金やサーバー維持費などに留まるため、クラウドを利用した場合よりも傾きは小さくなる。クラウドのコストのカーブが、自前計算環境のコストのカーブの下にある範囲においては、クラウドを使うことは経済的なコスト削減につながる。

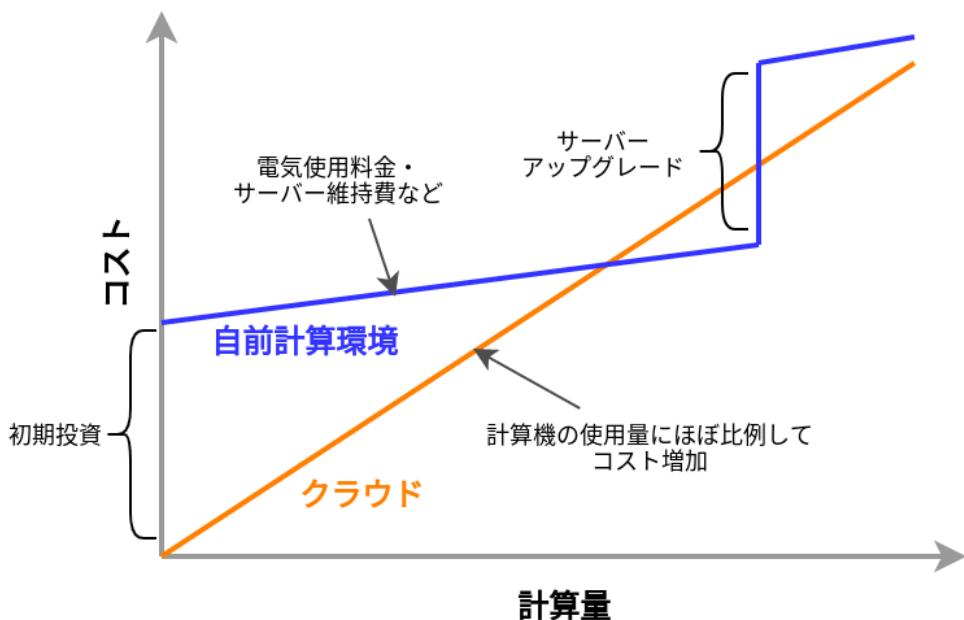


Figure 2. クラウドと自前計算機環境の経済的コストの比較

特に、1.の点は研究の場面では重要であると筆者は感じる。研究をやっていて、四六時中計算を走らせ続けるという場合は少ない。むしろ、新しいアルゴリズムが完成したとき・新しいデータが届いたとき、集中的・突発的に計算タスクが増大することが多いだろう。そういうときに、フレキシブルに計算力を増強させることができるのは、クラウドを使う大きなメリットである。

ここまでクラウドを使うメリットを述べたが、逆に、デメリットというのも当然存在する。

## 1. クラウドは賢く使わないといけない

上のコストのカーブにあるとおり、使い方によっては自前の計算環境のほうがコスト的に有利な場面は存在しうる。クラウドを利用する際は、使い終わった計算資源はすぐに削除するなど、利用者が賢く管理を行う必要があり、これを怠ると思もしない額の請求が届く可能性がある。

## 2. セキュリティ

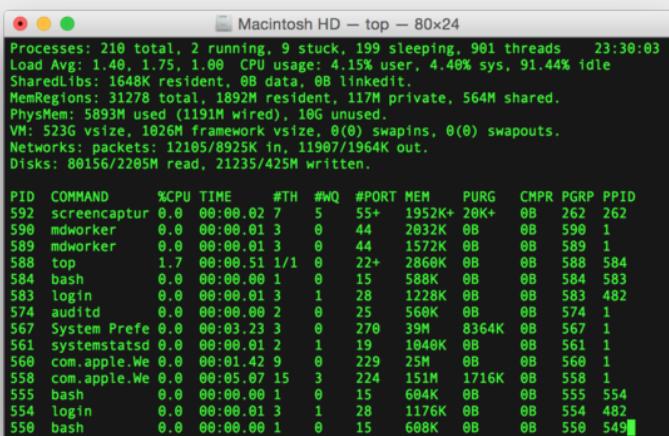
クラウドは、インターネットを通じて、世界のどこからでもアクセスできる状態にあり、セキュリティ管理を怠ると簡単にハッキングの対象となりうる。ハッキングを受けると、情報流出だけでなく、経済的な損失を被る可能性がある。また、個人の医療データなど、法律で管理のされ方が規定されているデータに関しては、クラウドを利用することはそもそもできない場合がある。

## 3. ラーニングカーブ

上記のように、コスト・セキュリティなど、クラウドを利用する際に留意しなければならない点は多い。賢くクラウドを使うには、十分なクラウドの理解が必要であり、そのラーニングカーブを乗り越える必要がある。

### 小嘶: Terminal の語源

Mac/Linuxなどでコマンドを入力するときに使用する、あの黒い画面のことを Terminal と呼んだりする。この言葉の語源をご存知だろうか？



Macintosh HD — top — 80x24

```
Processes: 210 total, 2 running, 9 stuck, 199 sleeping, 901 threads 23:30:03
Load Avg: 1.40, 1.75, 1.88 CPU usage: 4.15% user, 4.40% sys, 91.44% idle
SharedLibs: 1648K resident, 0B data, 0B linkedit
MemRegions: 31278 total, 1892M resident, 117M private, 564M shared.
PhysMem: 5893M used (1191M Wired), 18G unused.
VM: 523G vsize, 1826M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 12105/8925K in, 11907/1964K out.
Disks: 80156/2205M read, 21235/425M written.
```

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PURG	CMPR	PGRP	PPID
592	screenCaptur	0.0	00:00:02	7	5	55+	1952K+	28K+	0B	262	262
596	mdworker	0.0	00:00:01	3	0	44	2032K	0B	0B	590	1
589	mdworker	0.0	00:00:01	3	0	44	1572K	0B	0B	589	1
588	top	1.7	00:00:51	1/1	0	22+	2868K	0B	0B	588	584
584	bash	0.0	00:00:00	1	0	15	588K	0B	0B	584	583
583	login	0.0	00:00:01	3	1	28	1228K	0B	0B	583	482
574	audited	0.0	00:00:00	2	0	25	560K	0B	0B	574	1
567	System Prefe	0.0	00:03:23	3	0	270	39M	8364K	0B	567	1
561	systemstatsd	0.0	00:00:01	2	1	19	1048K	0B	0B	561	1
560	com.apple.We	0.0	00:01:42	9	0	229	25M	0B	0B	560	1
558	com.apple.We	0.0	00:05:07	15	3	224	151M	1716K	0B	558	1
555	bash	0.0	00:00:00	1	0	15	604K	0B	0B	555	554
554	login	0.0	00:00:01	3	1	28	1176K	0B	0B	554	482
550	bash	0.0	00:00:00	1	0	15	608K	0B	0B	550	549

この言葉の語源は、コンピュータが誕生して間もない頃の時代に遡る。その頃のコンピュータというと、何千何万のという大量の真空管が接続された、会議室一個分くらいのサイズのマシンであった。そのような高価でメンテが大変な機材であるから、当然みんなでシェアして使うことが前提となる。ユーザーがコンピュータにアクセスするため、マシンからは何本かのケーブルが伸び、それぞれにキーボードとスクリーンが接続されていた…これを Terminal と呼んでいたのである。人々は、代わる代わるTerminalの前に座って、計算機との対話を行っていた。

時代は流れ、WindowsやMacなどのいわゆるパーソナルコンピュータの出現により、コンピュータはみんなで共有するものではなく、個人が所有するものになった。

最近のクラウドの台頭は、みんなで大きなコンピュータをシェアするという、最初のコンピュータの使われ方に原点回帰していると捉えることもできる。一方で、スマートフォンやウェアラブルなどのエッジデバイスの普及も盛んであり、個人が複数の小さなコンピュータを所有する、という流れも同時に進行しているのである。

# Chapter 3. AWS入門

## 3.1. AWSとは？

本講義では、クラウドの実践を行うプラットフォームとして、AWSを用いる。

AWS (Amazon Web Service) はAmazon社が提供する総合的なクラウドプラットフォームである。

AWSはAmazonが持つ膨大な計算リソースを貸し出すクラウドサービスとして、2006年に誕生した。現在では、クラウドプロバイダーとして最大のマーケットシェア(約33%)を保持している(ソース: [Market report by canaly](#)s). Netflix, Slackをはじめとした多くのウェブ関連のサービスで、一部または全てのサーバーリソースがAWSから提供されているとのことである。よって、知らないうちにAWSの恩恵にあずかっている人も少なくないはずだ。

最大のシェアをもつだけに、とても幅広い機能・サービスが提供されており、科学・エンジニアリングの研究用途としても頻繁に用いられるようになってきている。

## 3.2. AWSの機能・サービス

Figure 3は、執筆時点においてAWSで提供されている主要な機能・サービスの一覧である。

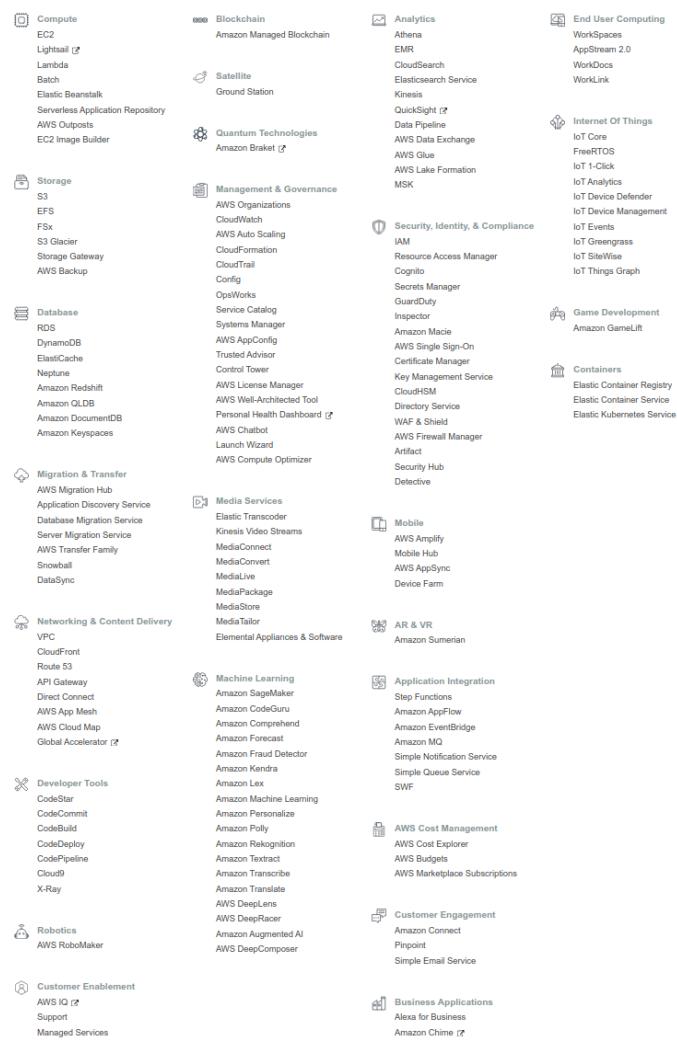


Figure 3. AWSで提供されている主要なサービス一覧

計算、ストレージ、データベース、ネットワーク、セキュリティなど、クラウドの構築に必要な様々な要素が独立したコンポーネントとして提供されている。基本的に、これらを組み合わせることでひとつのクラウドシステムができる。

る。

また、機械学習・音声認識・AR/VRなど、特定のアプリケーションにパッケージ済みのサービスも提供されている。これらを合計すると全部で176個のサービスが提供されているとのことである([出典](#))。

AWSの初心者は、この大量のサービスの数に圧倒され、どこから手をつけたらよいのかわからなくなる、という状況に陥りがちである。だが実のところ、基本的な構成要素はそのうちの数個のみに限られる。他の機能の多くは、基本の要素を組み合わせ、特定のアプリケーションとしてAWSがパッケージとして用意したものである。なので、基本要素となる機能の使い方を知れば、AWSのおおよそのリソースを使いこなすことが可能になる。

### 3.3. AWSでクラウドを作るときの基本となる部品

#### 3.3.1. 計算



**EC2 (Elastic Compute Cloud)** 様々なスペックの仮想マシンを作成し、計算を実行することができる。科学計算では、おそらくこれをメインで使うことになる。



**Lambda Function as a Service (FaaS)**と呼ばれる、小さな計算をサーバーなしで実行するためのサービス。Serverless architecture の章で詳しく解説する。

#### 3.3.2. ストレージ



**EBS (Elastic Block Store)** EC2にアタッチすることのできる仮想データドライブ。いわゆる"普通の"(一般的なOSで使われている)ファイルシステムを思い浮かべてくれたらよい。



**S3 (Simple Storage Service)** Object Storage と呼ばれる、APIをつかってデータの読み書きを行う、いうなれば"クラウド・ネイティブ"なデータの格納システムである。通常のファイルシステムの機能に加えてバージョン管理などの機能が付与されている。Serverless architecture の章で詳しく解説する。

#### 3.3.3. データベース



**DynamoDB** NoSQL型のデータベースサービス(知っている人は [MongoDB](#)などを思い浮かべたらよい)。Serverless architecture の章で詳しく解説する。

#### 3.3.4. ネットワーク



**VPC(Virtual Private Cloud)** AWS上に仮想ネットワーク環境を作成し、仮想サーバー間の接続を定義したり、外部からのアクセスなどを管理する。EC2はVPCの内部に配置されなければならない。

### 3.4. Region と Availability Zone

AWSを使用する際の重要な概念として、[Region](#) と [Availability Zone \(AZ\)](#) がある([Figure 4](#))。

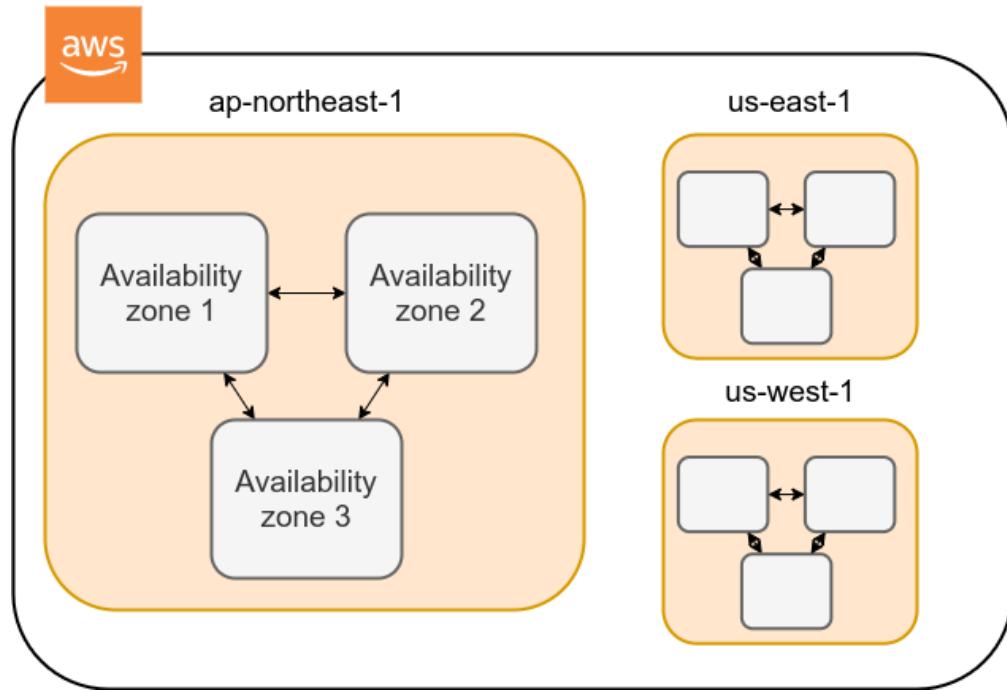


Figure 4. AWSにおけるRegionとAvailability Zones

**Region**とは、データセンターの所在地のことである。執筆時点において、AWSは世界の24の国と地域でデータセンターを所有している。[Figure 5](#)は2020/05時点で利用できるRegionの世界地図を示している。インターネットの接続などの観点から、地理的に一番近いRegionを使用するのが一般的によい。日本では東京にデータセンターがある。また大阪リージョンも2021年に提供開始予定とのことである。各Regionには固有のIDがついており、例えば東京は **ap-northeast-1**、米国・オハイオ州は **us-west-2**、などと定義されている。



Figure 5. Regions in AWS(出典: <https://aws.amazon.com/about-aws/global-infrastructure/>)

AWSコンソールにログインすると、画面右上でリージョンを選択することができる([Figure 6](#))。EC2, S3などのAWSのリソースは、リージョンごとに完全に独立である。したがって、リソースを新たにデプロイする時、あるいはデプロイ済みのリソースを閲覧するときは、コンソールのリージョンが正しく設定されているか、確認する必要がある。

ネットのウェビジネスを展開する場合などは、世界の各地にクラウドを展開する必要があるが、個人的な研究用途として用いる場合は、最寄りのリージョン(i.e. 東京)を使えば基本的に問題ない。

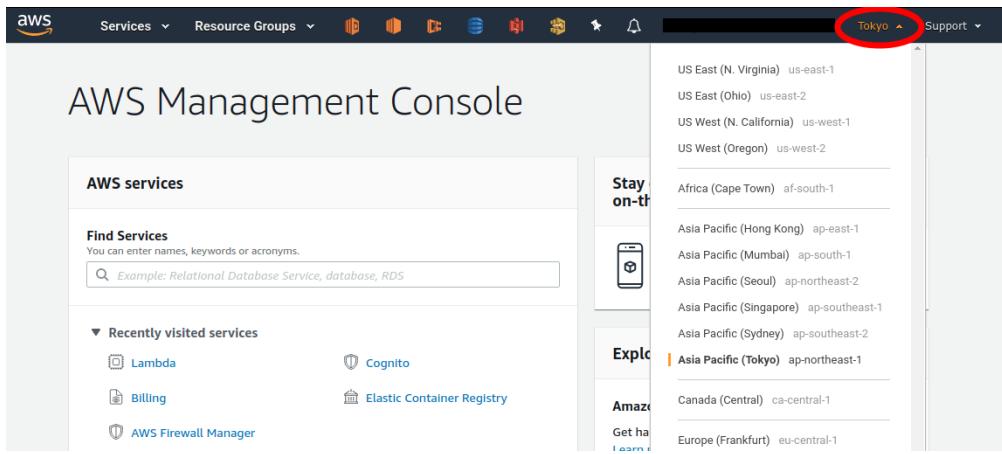


Figure 6. AWSコンソールでリージョンを選択

**Avaialibility Zone (AZ)** とは、Region 内で地理的に隔離されたデータセンターのことである。リージョンは2個以上のAZを有しており、もしひとつAZで火災や停電などが起きた場合でも、他のAZがその障害をカバーすることができる。また、AZ間は高速なAWS専用ネットワーク回線で結ばれているため、AZ間のデータ転送は極めて早い。

AZは、ネットのビジネスなどでサーバーダウンが許容されない場合などに注意すべき概念であり、個人的な用途で使う限りにおいてはあまり深く考慮する必要はない。言葉の意味だけ知っておけば十分である。

### Further reading

- AWS documentation "Regions, Availability Zones, and Local Zones"

## 3.5. AWSでのクラウドの開発

AWSのクラウドの全体像がわかつたところで、次のトピックとして、どのようにしてAWS上にクラウドの開発を行い、展開していくかについての概略を解説をしよう。

AWSのリソースを追加・編集・削除などの操作を実行するには、**コンソールを用いる方法**と、**APIを用いる方法**の、二つの経路がある。

### 3.5.1. コンソール画面からリソースを操作する

AWSのアカウントにログインすると、まず最初に表示されるのが**AWSコンソール**である。

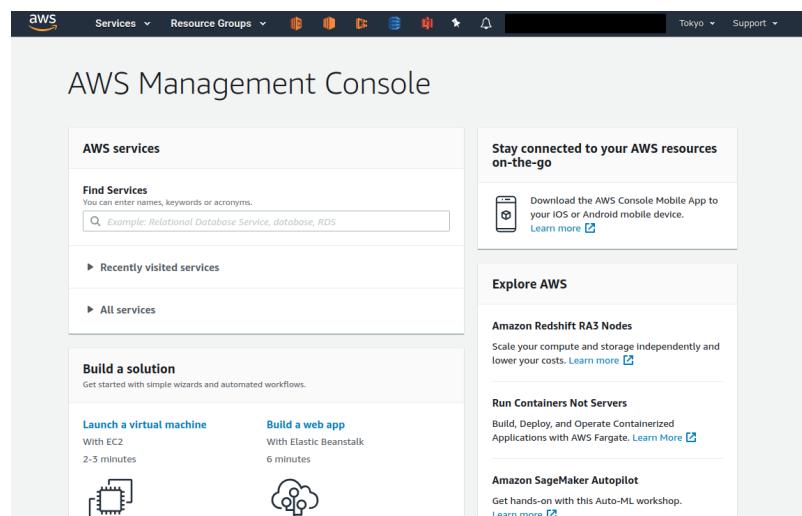


Figure 7. AWSマネージメントコンソール画面

コンソールを使うことで,EC2のインスタンスを立ち上げたり,S3のデータを追加・削除したり,ログを閲覧したりなど,あらゆるAWS上のあらゆるリソースの操作をGUI (Graphical User Interface) を使って実行することができる.初めて触る機能をポチポチと試したり,デバッグを行うときなどにとても便利である.

コンソールはさらっと機能を試すくらいの使用には便利なのであるが,実際にクラウドの開発をする場面でこれを直接いじることはあまりない.むしろ,次に紹介するAPIを使用して,プログラムとしてクラウドのリソースを記述することで開発を行うのが一般的である.

そのような理由で,本講義ではAWSコンソールを使ったAWSの使い方はあまり触れない.AWSのドキュメンテーションには,たくさんの[チュートリアル](#)が用意されており,コンソール画面から様々な操作を行う方法が記述されているので,興味がある読者はそちらを参照されたい.

### 3.5.2. APIからリソースを操作する

API(Application Programming Interface) を使うことで,コマンドをAWSに送信し,クラウドのリソースの操作をすることができます.

APIとは,簡単に言えばAWSが公開しているコマンドの一覧であり,**GET, POST, DELETE**などの**REST API**から構成されている.が,直接REST APIを入力するのは面倒であるので,その手間を解消するための様々なツールが提供されている.

[AWS CLI](#)は,UNIXのコンソールからAWS APIを送信するためのCLI (Command Line Interface) である.

CLIに加えて,いろいろなプログラミング言語でのSDK(Software Development Kit)が提供されている.

- Python ⇒ [boto3](#)
- Ruby ⇒ [AWS SDK for Ruby](#)
- node.js ⇒ [AWS SDK for Node.js](#)

具体的なAPIの使用例をあげよう.

S3に新しい保存領域(バケットと呼ばれる)を追加したいとしよう.AWS CLIを使った場合は,以下のコマンドを打てばよい.

```
$ aws s3 mb s3://my-bucket
```

上記のコマンドは,[my-bucket](#)という名前のバケットを,[ap-northeast-1](#)のregionに作成する.なお,上記のコマンドを実行する前提として,認証鍵を用いたAWSへのログインは済んでいるものとする(ハンズオンにて詳しく解説).

Pythonから上記と同じ操作を実行するには, [boto3](#) ライブラリを使って,以下のスクリプトを実行する.

```
import boto3

s3_client = boto3.client("s3", region_name="ap-northeast-1")
s3_client.create_bucket(Bucket="my-bucket")
```

もう一つ例をあげよう.

新しいEC2のインスタンス(インスタンスとは,起動状態にある仮想サーバーの意味である)を起動するには,以下のコマンドを打てば良い.

```
$ aws ec2 run-instances --image-id ami-xxxxxxxx --count 1 --instance-type t2.micro --key-name MyKeyPair
--security-group-ids sg-903004f8 --subnet-id subnet-6e7f829e
```

上記のコマンドにより, `t2.micro` というタイプ(1CPU, 1.0GB RAM)のインスタンスが起動する. ここでは他のパラメータの詳細の説明は省略する(ハンズオンで詳しく解説).

Pythonから上記と同じ操作を実行するには, 以下のようなスクリプトを使う.

```
import boto3

ec2_client = boto3.client("ec2")
ec2_client.run_instances(
    ImageId="ami-xxxxxxxxx",
    MinCount=1,
    MaxCount=1,
    KeyName="MyKeyPair",
    InstanceType="t2.micro",
    SecurityGroupIds=["sg-903004f8"],
    SubnetId="subnet-6e7f829e",
)
```

以上の具体例を通じて, APIによるクラウドのリソースの操作のイメージがつかめてきただろうか? コマンド一つで, 新しい仮想サーバーを起動したり, データの保存領域を追加したり, 任意の操作を実行することができるわけである. 基本的に, このようなコマンドを複数組み合わせていくことで, 自分の望むCPU・RAM・ネットワーク・ストレージが備わった計算環境を構築することができる.もちろん, 逆の操作(リソースの削除)もAPIを使って実行できる.

### 3.5.3. ミニ・ハンズオン: AWS CLI を使ってみよう

ここでは, ミニ・ハンズオンとして, AWS CLI を実際に使ってみる.



AWS CLI のインストールについては, [Section 8.3](#) を参照.



以下のコマンドを実行する前に, AWSの認証情報が正しく設定されていることを確認する. これには `~/.aws/credentials` のファイルに設定が書き込まれているか, 環境変数 (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_DEFAULT_REGION`) が定義されている必要がある.

ここでは一番シンプルな, S3を使ったファイルの読み書きを実践する (EC2の操作は少し複雑なので, 第一回ハンズオンで行う). `aws s3` コマンドの詳しい使い方は [公式ドキュメンテーション](#) を参照.

まず, 最初にS3にデータの格納領域 (`Bucket` と呼ばれる)を作成する.

```
$ bucketName="mybucket-$(openssl rand -hex 12)"
$ echo $bucketName
$ aws s3 mb "s3://${bucketName}"
```

S3のバケットの名前は, グローバルにユニークでなければならないことから, 上ではランダムな文字列を含んだバケットの名前を生成し, `$bucketName` という変数に格納している.

次に, バケットの一覧を取得してみよう.

```
$ aws s3 ls
2020-06-07 23:45:44 mybucket-c6f93855550a72b5b66f5efe
```

先ほど作成したバケットがリストにあることを確認できる.



本書のノーテーションとして、シェル(ターミナル)に入力するコマンドは、それがシェルであると明示する目的で先頭に \$ がつけてある。\$ はコマンドをコピー&ペーストするときは除かなければならない。逆に、コマンドの出力は \$ なしで表示されている。

次に、バケットにファイルをアップロードする。

```
$ echo "Hello world!" > hello_world.txt  
$ aws s3 cp hello_world.txt "s3://${bucketName}/hello_world.txt"
```

上では `hello_world.txt` というダミーのファイルを作成して、それをアップロードした。

それでは、バケットの中にあるファイルの一覧を取得してみる。

```
$ aws s3 ls "s3://${bucketName}" --human-readable  
2020-06-07 23:54:19    13 Bytes hello_world.txt
```

先ほどアップロードしたファイルがたしかに存在することがわかる。

次に

最後に、使い終わったバケットを削除する。

```
aws s3 rb "s3://${bucketName}" --force
```

デフォルトでは、バケットは空でないと削除できない。空でないバケットを強制的に削除するには `--force` のオプションを付ける。

以上のように、AWS CLI を使って、S3のバケットの操作を実行することができた。EC2やLambda、DynamoDBなどについても同様に AWS CLI を使ってあらゆる操作を実行することができる。

## 3.6. CloudFormation と AWS CDK

### 3.6.1. CloudFormation による Infrastructure as Code (IaC)

前節で述べたように、AWS API を使うことでクラウドのあらゆるリソースの作成・管理が可能である。よって、原理上は、APIのコマンドを組み合わせていくことで、自分の作りたいクラウドを設計することができる。

しかし、ここで実用上考慮しなければならない点がひとつある。AWS API には大きく分けて、**インフラを操作するコマンド**と、**タスクを実行するコマンド**があることである (Figure 8)。具体例をあげて説明しよう。

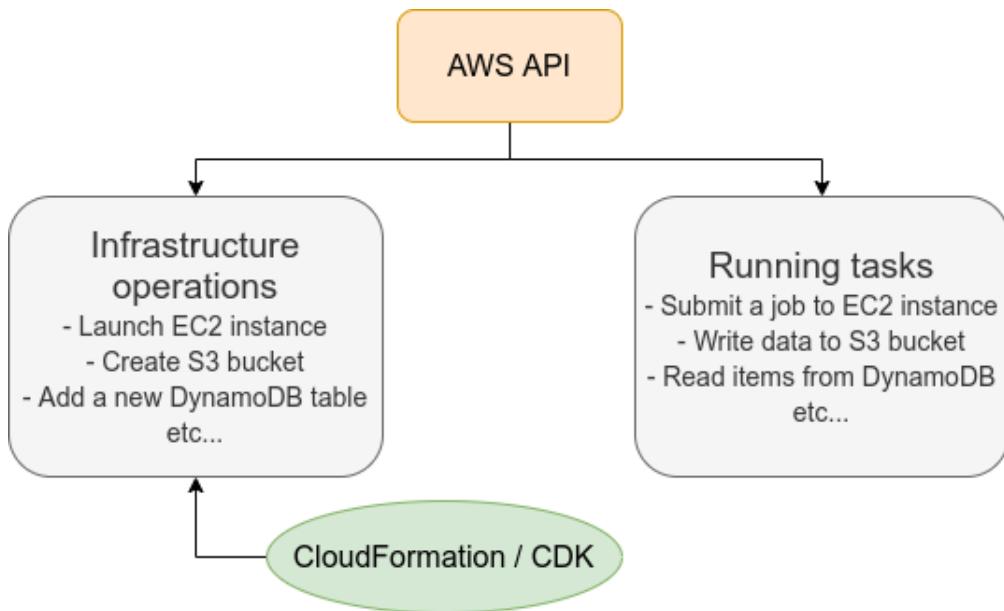


Figure 8. AWS APIはインフラを操作するコマンドとタスクを実行するコマンドに大きく分けられる

**インフラを操作するとは**, EC2のインスタンスを起動したり,S3の保存領域(バケット)をしたり, データベースに新たなテーブルを追加する,などの静的なリソースを準備する操作を指す. あるいは, 既にあるリソースを削除するなどの操作も考えられる. このようなコマンドは, クラウドのデプロイ時にのみ, 一度だけ実行されればよい.

**タスクを実行するコマンド**とは, EC2のインスタンスにジョブを投入したり, S3にデータを書き込んだりするなどの操作を指す. これは, 前のインフラを操作するコマンドで作られたインフラを前提として, その内部で実行されるべき計算を記述するものである. 前者に比べてこちらは動的な操作を担当する, と捉えることもできる.

そのような観点から, **インフラを記述するプログラム**と**タスクを実行するプログラム**はある程度分けて管理されるべきである. クラウドの開発は, クラウドの(静的な)インフラを記述するプログラムを作成するステップと, インフラ上で動く動的な操作を行うプログラムを作成するステップの, 二段階に分けて考えることができる.

AWSでのインフラを管理するための仕組みが, [CloudFormation](#) である. CloudFormation とは, CloudFormationのシンタックスに従ったテキストにより, AWSのインフラを記述するものである. CloudFormation を使って, 例えば, EC2のインスタンスをどれくらいのスペックで, 何個起動するか, インスタンス間はどのようなネットワークで結び, どのようなアクセス権限を付与するか, などのリソースの定義を逐次的に記述することができる. 一度CloudFormation ファイルが出来上がれば, それにしたがったクラウド・インフラをコマンド一つでAWS上に展開することができる. また, CloudFormation ファイルを交換することで, 全く同一のクラウド環境を他者が簡単に再現することが可能になる. このように, 本来は物理的な実体のあるハードウェアを, プログラムによって記述し, 管理するという考え方を, **Infrastructure as Code (IaC)**と呼ぶ.

CloudFormation を記述するには, **JSON** (JavaScript Object Notation) や **YAML** (YAML Ain't Markup Language) などのフォーマットを選択することができる. 以下は, JSONで記述された CloudFormation ファイルの一例(抜粋)である.

```

"Resources" : {
    ...
    "WebServer": {
        "Type" : "AWS::EC2::Instance",
        "Properties": {
            "ImageId" : { "Fn::FindInMap" : [ "AWSRegionArch2AMI", { "Ref" : "AWS::Region" },
                { "Fn::FindInMap" : [ "AWSInstanceType2Arch", { "Ref" : "InstanceType" }, "Arch" ]
            ] },
            "InstanceType" : { "Ref" : "InstanceType" },
            "SecurityGroups" : [ {"Ref" : "WebServerSecurityGroup"} ],
            "KeyName" : { "Ref" : "KeyName" },
            "UserData" : { "Fn::Base64" : { "Fn::Join" : [ "", [
                "#!/bin/bash -xe\n",
                "yum update -y aws-cfn-bootstrap\n",

                "/opt/aws/bin/cfn-init -v ",
                "    --stack ", { "Ref" : "AWS::StackName" },
                "    --resource WebServer ",
                "    --configsets wordpress_install ",
                "    --region ", { "Ref" : "AWS::Region" }, "\n",
                "/opt/aws/bin/cfn-signal -e $? ",
                "    --stack ", { "Ref" : "AWS::StackName" },
                "    --resource WebServer ",
                "    --region ", { "Ref" : "AWS::Region" }, "\n"
            ]]} }
        },
        ...
    },
    ...
    "WebServerSecurityGroup" : {
        "Type" : "AWS::EC2::SecurityGroup",
        "Properties" : {
            "GroupDescription" : "Enable HTTP access via port 80 locked down to the load balancer + SSH access",
            "SecurityGroupIngress" : [
                { "IpProtocol" : "tcp", "FromPort" : "80", "ToPort" : "80", "CidrIp" : "0.0.0.0/0" },
                { "IpProtocol" : "tcp", "FromPort" : "22", "ToPort" : "22", "CidrIp" : { "Ref" : "SSHLocation" } }
            ]
        }
    },
    ...
}

```

ここでは、"WebServer" という名前のつけられた EC2 インスタンスを定義している。かなり長大で複雑な記述であるが、これによって所望のスペック・OSをもつEC2インスタンスを自動的に生成することが可能になる。

### 3.6.2. AWS CDK

前節で紹介した CloudFormation は、見てわかるとおり大変記述が複雑であり、またそれのどれか一つにでも誤りがあつてはいけない。また、基本的に"テキスト"を書いていくことになるので、プログラミング言語で使うような便利な変数やクラスといった概念が使えない(厳密には、変数に相当するような機能は存在するのだが)。そのようなわけで、実際にCloudFormation 職人と呼ばれる専門のプロが存在するくらいである。一方、記述の多くの部分は繰り返しが多く、自動化できる部分も多い。

そのような悩みを解決してくれるのが、[AWS Cloud Development Kit \(CDK\)](#) である。CDKは Python などのプログラミング言語を使って CloudFormation を自動的に生成してくれるツールである。CDKは2019年にリリースされたばかりの比較的新しいツールで、日々改良が進められている([GitHub](#) のソースのリリースを見ればその開発のスピードの速さがわかるだろう)。CDKは TypeScript (JavaScript), Python, Java など複数の言語でサポートされている。

CDKを使うことで,CloudFormationに相当するリソースの記述を,より親しみのあるプログラミング言語を使って行うことができる.かつ,典型的なリソース操作に関してはパラメータの多くの部分を自動で決定してくれるので,記述しなければならない量もかなり削減される.

以下に Python を使った CDK のコードの一例(抜粋)を示す.

```
from aws_cdk import (
    core,
    aws_ec2 as ec2,
)

class MyFirstEc2(core.Stack):

    def __init__(self, scope, name, **kwargs):
        super().__init__(scope, name, **kwargs)

        vpc = ec2.Vpc(
            ...
            # some parameters
        )

        sg = ec2.SecurityGroup(
            ...
            # some parameters
        )

        host = ec2.Instance(
            self, "MyGreatEc2",
            instance_type=ec2.InstanceType("t2.micro"),
            machine_image=ec2.MachineImage.latest_amazon_linux(),
            vpc=vpc,
            vpc_subnets=ec2.SubnetSelection(subnet_type=ec2.SubnetType.PUBLIC),
            security_group=sg,
            ...
        )
```

上記のようなコードから,CloudFormationファイルを自動生成することができる.とても煩雑だったCloudFormationファイルに比べて,Python を使うことで格段に短く,わかりやすく記述できるのがわかるだろう.

本講義では,ハンズオンでCDKを使ってクラウド開発の体験をしてもらう.

### Further reading

- [AWS CDK Examples](#): CDKのexample project が多数紹介されている.ここにある例をテンプレートに自分の開発を進める良い.

# Chapter 4. Hands-on #1: 初めてのEC2インスタンスを起動する

ハンズオンの第一回では、CDKを使ってEC2のインスタンス(仮想サーバー)を起動し、SSHでサーバーにログインする、という演習を行う。このハンズオンを終えれば、あなたは自分だけのサーバーをAWS上に立ち上げ、自由に計算を走らせることができるようになるのである！

ハンズオンのソースコードはこちらのリンクに置いてある ⇒ <https://gitlab.com/tomomano/intro-aws/handson/01-ec2>

## 4.1. 準備

まずは、ハンズオンを実行するための環境を整える。これらの環境整備は、後のハンズオンでも前提となるものなので確実にミスなく行っていただきたい。

### 4.1.1. AWS Account

ハンズオンを実行するには個人のAWSアカウントが必要である。AWSアカウントの取得については [Section 1.5](#) および [Section 8.1](#) 参照。

### 4.1.2. Python と node.js

本ハンズオンを実行するには

- Python (3.7 以上)
- node.js (10.3.0 以上)

がインストールされていなければならない。インストールの方法については [Section 8.2](#) に簡単なガイドを記してある。

### 4.1.3. AWS CLI

AWS CLI のインストールについては、[Section 8.3](#) を参照。

### 4.1.4. AWS CDK

AWS CDK のインストールについては、[Section 8.4](#) を参照。

### 4.1.5. SSH

[SSH \(secure shell\)](#) は Unix 系のリモートサーバーに安全にアクセスするためのツールである。SSHによる通信はすべて暗号化されているので、機密情報をインターネット越しに安全に送受信することができる。本ハンズオンで、リモートのサーバーにアクセスするために SSH クライアントがインストールされている必要がある。SSH クライアントは Linux/Mac のシェルには標準搭載されている。Windows の場合は WSL をインストールすることを推奨する。詳しくは [Section 1.3](#) を参照。

SSH コマンドの基本的な使い方は

```
$ ssh <user name>@<host name>
```

である。[`<host name>`](#) はアクセスする先のサーバーの IP アドレスや DNS によるホストネーム (e.g. google.com) などが入る。[`<user name>`](#) は接続する先のユーザー名である。

SSH は平文のパスワードによる認証を行うこともできるが、より強固なセキュリティを施すため、**公開鍵暗号方式(Public Key Cryptography)**による認証を行うことが強く推奨されており、EC2はこの方法でしかアクセスを許していない。公開鍵暗号方式の仕組みについては各自勉強してほしい。本ハンズオンにおいて大事なことは、**EC2インスタンスが公開鍵(Public key)を保持し、クライアントとなるコンピューター(あなたの自身のコンピュータ)が秘密鍵(Private key)を保持する**、という点である。EC2のインスタンスには秘密鍵を持ったコンピュータのみがアクセスすることができる。逆に言うと、秘密鍵が漏洩すると第三者もサーバーにアクセスできることになるので、**秘密鍵は絶対に漏洩することのないよう注意して管理する**。

SSH コマンドでは、ログインのために使用する秘密鍵ファイルを `-i` もしくは `--identity_file` のオプションで指定することができる。例えば

```
$ ssh -i Ec2SecretKey.pem <user name>@<host name>
```

のように使う。



SSH コマンドはデフォルトの動作として、`~/.ssh/` のディレクトリにあるファイルをスキヤンし、`id_rsa` や `id_ed25519` など、適合するファイル名が存在するとその秘密鍵を使ってログインを試みる。

#### 4.1.6. ソースコードのダウンロード

以下のコマンドで GitLab からソースコードをダウンロードする。

```
$ git clone git@gitlab.com:tomomano/intro-aws.git
```

あるいは、<https://gitlab.com/tomomano/intro-aws> のページに行って、右上のダウンロードボタンから .zip アーカイブをダウンロードすることもできる。

## 4.2. アプリケーションの説明

このハンズオンで作成するアプリケーションの概要を [Figure 9](#) に示す。

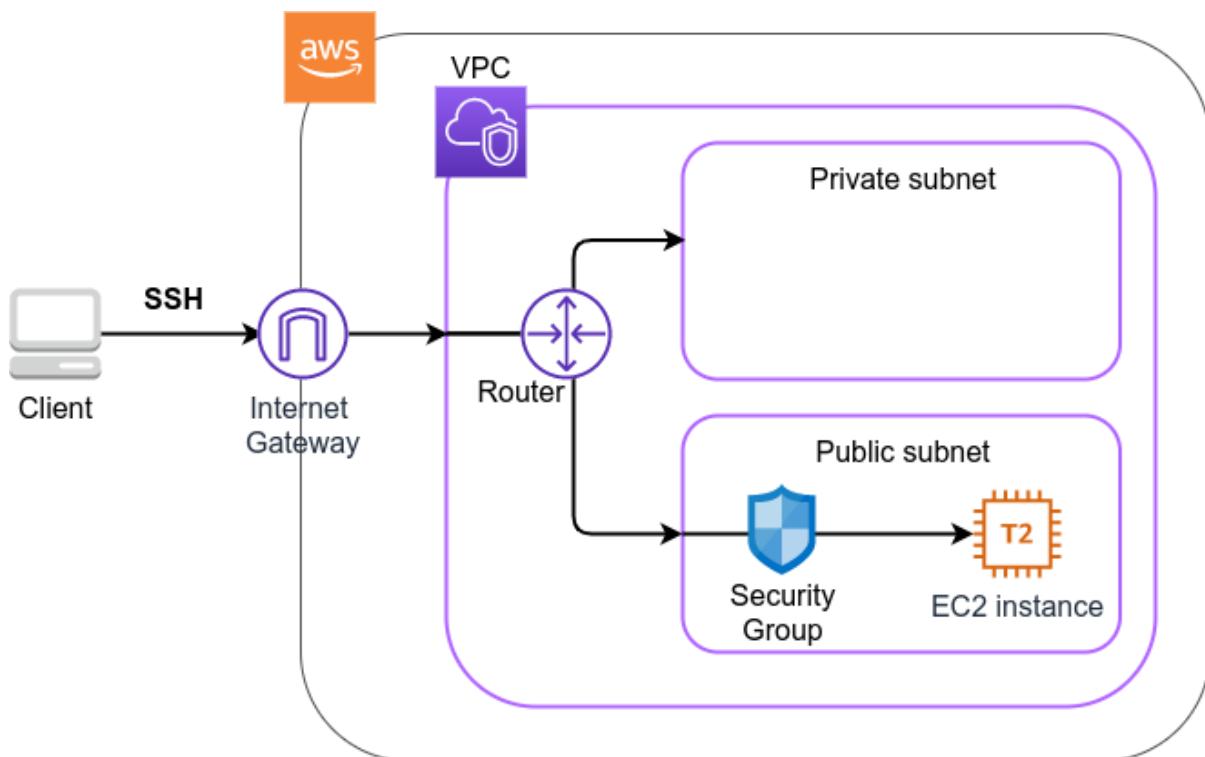


Figure 9. ハンズオン#1で作製するアプリケーションのアーキテクチャ

このアプリケーションは、ではまず、**VPC (Virtual Private Cloud)** を使ってプライベートな仮想ネットワーク環境を立ち上げている。そのVPCの public subnet の内側に、**EC2 (Elastic Compute Cloud)** の仮想サーバーを配置する。さらに、セキュリティのため、**Security Group** によるEC2インスタンスへのアクセス制限を設定している。

上記のようなアプリケーションを、CDKを使って構築する。

早速ではあるが、今回のハンズオンで使用するプログラムを見てみよう ([handson/01-ec2/app.py](#))。

```
class MyFirstEc2(core.Stack):

    def __init__(self, scope: core.App, name: str, key_name: str, **kwargs) -> None:
        super().__init__(scope, name, **kwargs)

    ①
    vpc = ec2.Vpc(
        self, "MyFirstEc2-Vpc",
        max_azs=1,
        cidr="10.10.0.0/23",
        subnet_configuration=[
            ec2.SubnetConfiguration(
                name="public",
                subnet_type=ec2.SubnetType.PUBLIC,
            )
        ],
        nat_gateways=0,
    )

    ②
    sg = ec2.SecurityGroup(
        self, "MyFirstEc2Vpc-Sg",
        vpc=vpc,
        allow_all_outbound=True,
    )
    sg.add_ingress_rule(
        peer=ec2.Peer.any_ipv4(),
        connection=ec2.Port.tcp(22),
    )

    ③
    host = ec2.Instance(
        self, "MyFirstEc2Instance",
        instance_type=ec2.InstanceType("t2.micro"),
        machine_image=ec2.MachineImage.latest_amazon_linux(),
        vpc=vpc,
        vpc_subnets=ec2.SubnetSelection(subnet_type=ec2.SubnetType.PUBLIC),
        security_group=sg,
        key_name=key_name
    )
```

① まず最初に、VPCを定義する。

② 次に、SGを定義している。ここでは、任意のIPv4のアドレスからの、ポート22 (SSHの接続に使用される)への接続を許容している。それ以外の接続は拒絶される。

③ 最後に、上記で作ったVPCとSGが付与されたEC2 のインスタンスを作成している。インスタンスタイプは **t2.micro** を選択し、**Amazon Linux** をOSとして設定している。

それぞれについて、もう少し詳しく説明しよう。

#### 4.2.1. VPC (Virtual Private Cloud)



VPCはAWS上にプライベートな仮想ネットワーク環境を構築するツールである。高度な計算システムを構築するには、複数のサーバーを連動させて計算を行う必要があるが、そのような場合に互いのアドレスなどを管理する必要があり、そのような場合にVPCは有用である。

本ハンズオンでは、サーバーは一つしか起動しないので、VPCの恩恵はよく分からないかもしれない。しかし、EC2インスタンスは必ずVPCの中に配置されなければならない、という制約があるので、このハンズオンでもミニマルなVPCを構成している。

##### Advanced tips

興味のある読者のために、VPCのコードについてもう少し詳しく説明しよう。



- `max_azs=1` : このパラメータは、前章で説明した availability zone を設定している。このハンズオンでは、特にデータセンターの障害などを気にする必要はないので1にしている。
- `cidr="10.10.0.0/23"` : このパラメーターは、VPC内のIPv4のレンジを指定している。CIDR記法については、[Wikipedia](#)などを参照。`10.10.0.0/23` は `10.10.0.0` から `10.10.1.255` までの512個の連続したアドレス範囲を指している。つまり、このVPCでは最大で512個のユニークなIPv4アドレスが使えることになる。今回はサーバーは一つなので512個は明らかに多すぎるが、VPCはアドレスの数はどれだけ作成しても無料なので、多めに作成した。
- `subnet_configuration=…` : このパラメータは、VPCにどのようなサブネットを作るか、を決めている。サブネットの種類には **private subnet** と **public subnet** の二種類がある。**private subnet** は基本的にインターネットとは遮断されたサブネット環境である。インターネットと繋がっていないので、セキュリティは極めて高く、VPC内のサーバーとのみ通信を行えばよいEC2インスタンスは、ここに配置する。**Public subnet** とはインターネットに繋がったサブネットである。本ハンズオンで作成するサーバーは、外からSSHでログインを行いたいので、**Public subnet** 内に配置する。
- `natgateways=0` : これは少し高度な内容なので省略する（興味のある読者は [公式ドキュメンテーション](#) を参照）。が、これを0にしておかないと、NAT Gateway の利用料金が発生してしまうので、注意！

#### 4.2.2. Security Group

Security group (SG) は、EC2インスタンスに付与することのできる仮想ファイアウォールである。例えば、特定のIPアドレスから来た接続を許したり（インバウンド・トラフィック）、逆に特定のIPアドレスへのアクセスを禁止したり（アウトバウンド・トラフィック）することができる。

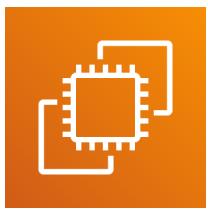
本ハンズオンでは、SSHによる外部からの接続を許容するため、`sg.add_ingress_rule(peer=ec2.Peer.any_ipv4(), connection=ec2.Port.tcp(22))` により、すべてのIPv4アドレスからのポート22番へのアクセスを許容している。

また、SSHでEC2インスタンスにログインしたのち、インターネットからプログラムなどをダウンロードできるよう、`allow_all_outbound=True` のパラメータを設定している。



セキュリティ上の観点からは、SSHの接続は自宅や大学などの特定の地点からの接続のみを許す方が望ましい。

### 4.2.3. EC2 (Elastic Compute Cloud)



EC2はAWS上に仮想サーバーを立ち上げるサービスである。個々の起動状態にある仮想サーバーのことをインスタンス (instance) と呼ぶ。

EC2では用途に応じて様々なインスタンスタイプが提供されている。以下に、代表的なインスタンスタイプの例を挙げる(2020/06時点での情報)。EC2のインスタンスタイプのすべてのリストは [公式ドキュメンテーション](#)で見ることができる。

Table 2. EC2 instance types

Instance	vCPU	Memory (GiB)	Network bandwidth (Gbps)	Price per hour (\$)
t2.micro	1	1	-	0.0116
t2.small	1	2	-	0.023
t2.medium	2	4	-	0.0464
c5.24xlarge	96	192	25	4.08
c5n.18xlarge	72	192	100	3.888
x1e.16xlarge	64	1952	10	13.344

このようにCPUは1コアから96コアまで、メモリーは1GBから3000GB以上まで、ネットワークは最大で100Gbpsまで、幅広く選択することができる。また、時間あたりの料金は、CPU・メモリーの占有数にほぼ比例する形で増加する。EC2はサーバーの起動時間を秒単位で記録しており、**利用料金は使用時間に比例する形で決定される**。例えば、**t2.micro** のインスタンスを10時間起動した場合、 $0.0116 \times 10 = 0.16$  ドルの料金が発生する。



上記で t2.micro の \$0.0116 / hour という金額は、on-demandインスタンスというタイプを選択した場合の価格である。EC2 では他に、Spot instance と呼ばれるインスタンスも存在する。Spot instance は、AWSのデータセンターの負荷が増えた場合、AWSの判断により強制シャットダウンされる可能性がある、という不便さを抱えているのだが、その分大幅に安い料金設定になっている。いうなれば、"すき間の空きCPUで計算を行う"といった格好である。科学計算で、コストを削減する目的で、このSpot Instanceを使う事例も報告されている ([Wu+, 2019](#))。

## 4.3. プログラムを実行する

さて、ハンズオンのコードの理解ができたところで、プログラムを実際に実行してみよう。繰り返しになるが、[Section 4.1](#) での準備ができていることが前提である。

### 4.3.1. Python の依存ライブラリのインストール

まずは、Python の依存ライブラリをインストールする。以下では、Python のライブラリを管理するツールとして、[venv](#) を使用する。

まずは、[handson/01-ec2](#) のディレクトリに移動しよう。

```
$ cd intro-aws/handson/01-ec2
```

ディレクトリを移動したら, `venv` で新しい仮想環境を作成し, インストールを実行する.

```
$ python3 -m venv .env  
$ source .env/bin/activate  
$ pip install -r requirements.txt
```

これで Python の環境構築は完了だ.



`venv` の簡単な説明は [Section 8.5](#) に記述してある.

#### 4.3.2. AWS の認証情報をセットする

AWS CLI および AWS CDK を使うには, AWSの認証鍵が設定されている必要がある. 以下のようにして環境変数を設定する.

```
export AWS_ACCESS_KEY_ID=XXXXXX  
export AWS_SECRET_ACCESS_KEY=YYYYYY  
export AWS_DEFAULT_REGION=ap-northeast-1
```

上の `XXXXXX`, `YYYYYY` としたところは自分の鍵に置き換えることを忘れずに.

AWS の認証鍵の取得については [Section 8.1](#) を参照. コマンドラインでの AWS の認証の設定の仕方は [Section 8.3](#) を参照.

#### 4.3.3. SSH鍵を生成

EC2 インスタンスには SSH を使ってログインする. EC2インスタンスを起動するのにさきがけて, SSHの公開鍵・秘密鍵のペアを準備する必要がある.

以下の aws-cli コマンドにより, `HirakeGoma` という名前のついた鍵を生成する.

```
$ export KEY_NAME="HirakeGoma"  
$ aws ec2 create-key-pair --key-name ${KEY_NAME} --query 'KeyMaterial' --output text > ${KEY_NAME}.pem
```

上のコマンドを実行すると, 現在のディレクトリに `HirakeGoma.pem` というファイルが作成される. これが, サーバーにアクセスするための秘密鍵である. SSH でこの鍵を使うため, `~/.ssh/` のディレクトリに鍵を移動する. さらに, 秘密鍵が書き換えられたり第三者に閲覧されないよう, ファイルのアクセス権限を `400` に設定する.

```
$ mv HirakeGoma.pem ~/.ssh/  
$ chmod 400 ~/.ssh/HirakeGoma.pem
```

#### 4.3.4. デプロイを実行

これまでのステップで準備は整った!

早速, アプリケーションをAWSにデプロイしてみよう.

```
$ cdk deploy -c key_name="HirakeGoma"
```

`-c key_name="HirakeGoma"` というオプションで, 先程生成した `HirakeGoma` という名前の鍵を使うよう指定している.

上記のコマンドを実行すると、VPC、EC2 などが実際に展開される。また、コマンドの出力の最後に Figure 10 のような出力が得られるはずである。

✓ MyFirstEc2

Outputs:

```
MyFirstEc2.InstancePublicIp = 54.238.112.5
MyFirstEc2.InstancePublicDnsName = ec2-54-238-112-5.ap-northeast-1.compute.amazonaws.com
```

Stack ARN:

```
arn:aws:cloudformation:ap-northeast-1:606887060834:stack/MyFirstEc2/46ed0490-aa2d-
```

Figure 10. CDKデプロイ実行後の出力

ここでの `InstancePublicIp` として書かれているのが、起動したインスタンスのパブリックIPアドレスである。アドレスはランダムに割り当てられるので、上の画像のアドレスとは異なっているはずである。

早速、SSHで接続してみよう。

```
$ ssh -i ~/.ssh/HirakeGoma.pem ec2-user@<IP address>
```

**-i** オプションで、先程生成した秘密鍵を指定している。EC2 インスタンスにはデフォルトで `ec2-user` という名前のユーザーが作られている。最後に、`<IP address>` の部分は自分の値で置き換える (`54.238.112.5` など)。

ログインに成功すると、以下のような画面が表示される。リモートのサーバーにログインしているので、プロンプトが`[ec2-user@ip-10-10-1-217 ~]\$`となっている。

```
(.env) tomoyuki@eiffel:01-ec2$ ssh -i ~/.ssh/HirakeGoma.pem ec2-user@54.238.112.5
Last login: Tue Jun  9 09:18:09 2020 from 157.82.122.171
```

\_ | ( \_ \_ / ) Amazon Linux AMI  
  | \ | |

```
https://aws.amazon.com/amazon-linux-ami/2018.03-release-notes/
5 package(s) needed for security, out of 7 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-10-10-1-217 ~]$
```

Figure 11. SSH で EC2 インスタンスにログイン

おめでとう! これで、めでたくAWS上にEC2仮想サーバーを起動し、リモートからアクセスすることができるようになった!

せっかくサーバーを起動したので、少し遊んでみよう。

次のコマンドで、CPUの情報を取得することができる。

```
$ cat /proc/cpuinfo
```

```
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model    : 63
model name: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
stepping   : 2
microcode  : 0x43
cpu MHz    : 2400.096
cache size : 30720 KB
```

次に, 実行中のプロセスやメモリの消費を見てみよう.

```
$ top -n 1

top - 09:29:19 up 43 min,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 76 total,   1 running,  51 sleeping,   0 stopped,   0 zombie
Cpu(s): 0.3%us, 0.3%sy, 0.1%ni, 98.9%id, 0.2%wa, 0.0%hi, 0.0%si, 0.2%st
Mem: 1009140k total, 270760k used, 738380k free, 14340k buffers
Swap:      0k total,      0k used,      0k free, 185856k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1 root      20   0 19696 2596 2268 S  0.0  0.3  0:01.21  init
 2 root      20   0     0     0 S  0.0  0.0  0:00.00 kthreadd
 3 root      20   0     0     0 I  0.0  0.0  0:00.00 kworker/0:0
```

t2.micro インスタンスなので, 1009140k = 1GB のメモリーがあることがわかる.

今回起動したインスタンスには Python2 はインストール済みだが, Python 3 は入っていない. 最後の課題として, Python 3.6 のインストールを行ってみよう. インストールは簡単である.

```
$ sudo yum update -y
$ sudo yum install -y python36 python36-pip
```

インストールしたPythonを起動してみよう.

```
$ python3
Python 3.6.10 (default, Feb 10 2020, 19:55:14)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python のインターフェリタが起動した!

さて, サーバーでのお遊びはこんなところにしておこう(興味があれば各自いろいろと試してみると良い). 次のコマンドでログアウトする.

```
$ exit
```

#### 4.3.5. AWS コンソールから確認

これまで, すべてコマンドラインからEC2に関連する諸々の操作を行ってきた. EC2インスタンスの状態を確認したり, サーバーをシャットダウンなどの操作は, AWS コンソールから実行することもできる. 軽くこれを紹介しよう.

まず, AWS コンソールにログインする.

ログインしたら, Services から EC2 を検索(選択)する. 次に, 左のサイドバーの Instances とページを辿る. すると, Figure 12 のような画面が得られるはずである. この画面で, 自分のアカウントの管理下にあるインスタンスを確認することができる.

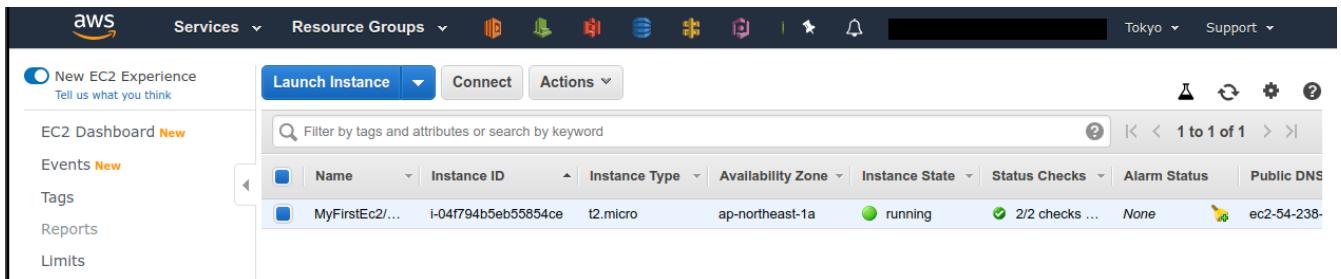


Figure 12. EC2 コンソール画面



コンソール右上で、正しいリージョン（今回の場合は ap-northeast-1, Tokyo）が選択されているか、注意する！

同様に、VPC・SGについてもコンソールから確認することができる。

前章で CloudFormation について触れたが、今回デプロイしたアプリケーションも、CloudFormation の "スタック" として管理されている。スタック (stack) とは、AWSリソースの集合のことを指す。今回の場合は、VPC/EC2/SGなどがスタックの中に含まれている。

コンソールで CloudFormation のページに行ってみよう (Figure 13)。

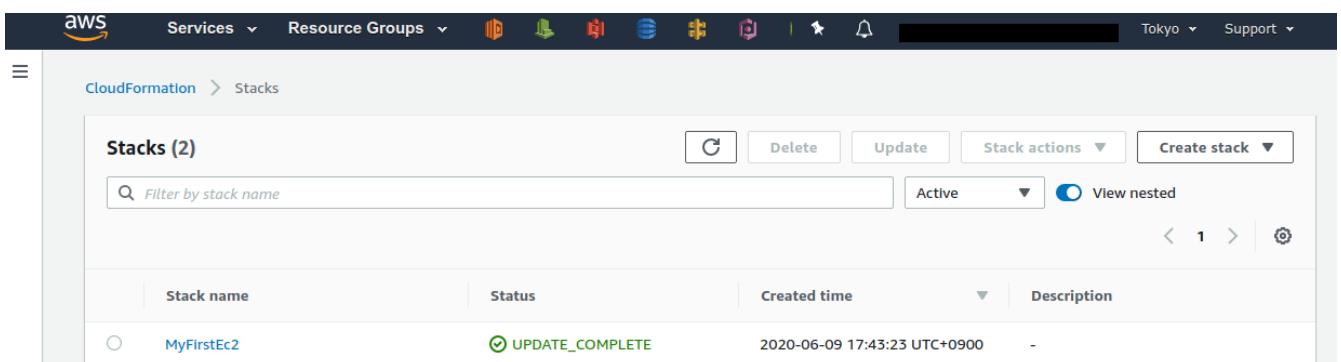


Figure 13. CloudFormation コンソール画面

"MyFirstEc2" という名前のスタックがあることが確認できる。クリックをして中身を見てみると、EC2、VPNなどのリソースがこのスタックに紐付いていることがわかる。

#### 4.3.6. スタックを削除

これにて、第一回のハンズオンで説明すべき事柄はすべて完了した。最後に、使わなくなったスタックを削除しよう。

スタックの削除には、2つの方法がある。

1つめの方法は、前節の Cloudformation のコンソール画面で、"Delete" ボタンを押すことである (Figure 14)。

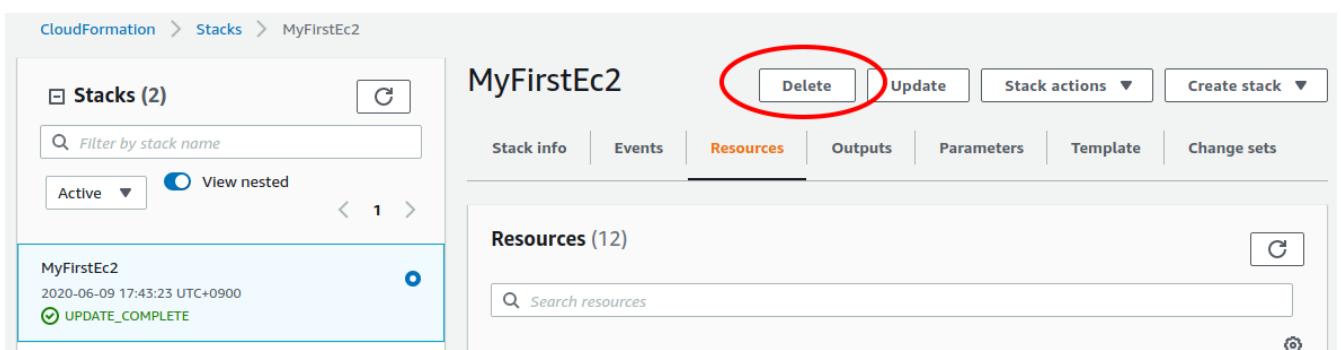


Figure 14. CloudFormationコンソール画面から、スタックを削除

2つめの方法は、コマンドラインから行う方法である。

先ほど、デプロイを行ったコマンドラインに戻ろう。そうしたら、

```
$ cdk destroy
```

と実行する。すると、スタックの削除が始まる。

削除した後は、VPC、EC2など、すべて跡形もなく消え去っている。

このように、自分の使いたいときにだけ、サーバーを立ち上げ、使い終わったら直ちに削除する、というのが現代のクラウドの正しい使い方である。

**！** **スタックの削除は各自で必ず行うこと!** 行わなかった場合、EC2インスタンスの料金が発生し続けることになる! t2.micro は \$0.0116 / hour の料金設定なので、一ヶ月起動しつづけると約\$8の請求が発生することになる!

また、本ハンズオンのために作成したSSH鍵ペアも不要なので、削除しておく。

まず、EC2側に登録してある公開鍵を削除する。これも、コンソールおよびコマンドラインの2つの方法で実行できる。

コンソールから実行するには、EC2の画面に行き、左のサイドバーの **Key Pairs** を選択。鍵の一覧が表示されるので、**HirakeGoma** とある鍵にチェックを入れ、画面右上の **Actions** から、**Delete** を実行 (Figure 15)。

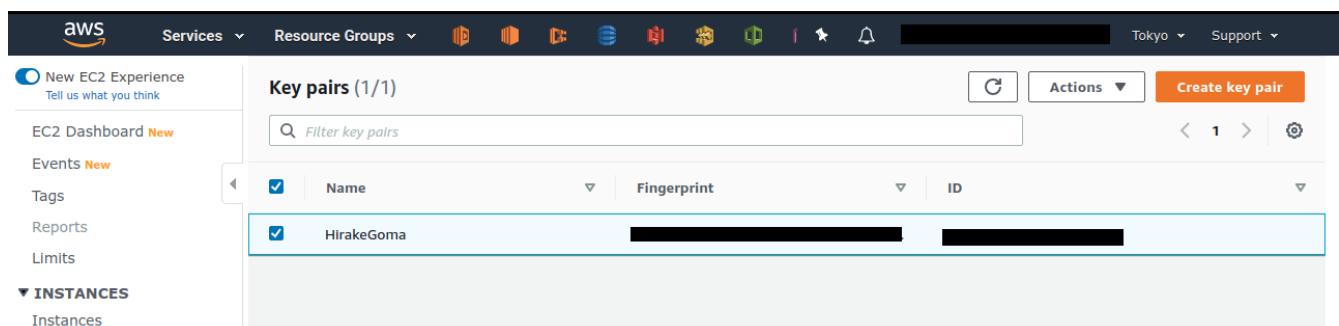


Figure 15. EC2でSSH鍵ペアを削除

コマンドラインから実行するには、以下のコマンドを使う。

```
$ aws ec2 delete-key-pair --key-name "HirakeGoma"
```

最後に、手元のコンピュータから鍵を削除する。

```
$ rm -f ~/.ssh/HirakeGoma.pem
```

これで、クラウドの片付けもすべて終了だ。



なお、頻繁にEC2インスタンスを起動したりする場合は、いちいちSSH鍵を削除する必要はない。

## 4.4. 講義第一回目のまとめ

ここまでが、第一回目の講義の内容である。盛りだくさんの内容であったが、ついてこれたであろうか？

第一回では、クラウドの概要と、なぜクラウドを使うのか、という点を議論した。また、クラウドを学ぶ具体的な題材としてAWSを取り上げ、AWSの概要説明を行った。さらに、ハンズオンではAWS CLI/CDKを使って、自分のマイ・サーバーをAWS上に立ち上げる演習を行った。

ハンズオンなどを通じて、いかに簡単に(たった数行のコマンドで!)仮想サーバーを立ち上げたり、削除したりすることができるか、体験することができます。このように、**ダイナミックに計算リソースを拡大・縮小をできることが、クラウドの最も本質的な側面であると、筆者は考えている。**

次回以降の講義では、今回学んだクラウドの技術を基に、より現実的な問題を解くことを体験してもらう。お楽しみに！

# Chapter 5. クラウドで行う科学計算・機械学習

ここからが第二回目の講義の内容になる。

第二回目は、前回学んだクラウドの知識・技術を使って、現実的な問題を解くことを考える。

計算機が発達した現代では、計算機によるシミュレーションやビッグデータの解析は、科学・エンジニアリングの研究の主要な柱である。これらの大規模な計算を実行するには、クラウドは最適である。本講義では、どのようにしてクラウド上で科学計算を実行するのかを、ハンズオンとともに体験してもらう。科学計算の具体的な題材として、今回は機械学習(ディープラーニング)を取り上げる。

なお、本講義では [PyTorch](#) ライブラリを使ってディープラーニングのアルゴリズムを走らせるが、ディープラーニングおよび PyTorch の知識は不要である。講義ではなぜ・どうやってディープラーニングをクラウドで実行するか、に主眼を置いてるので、実行するプログラムの詳細には立ち入らない。将来自分でディープラーニングを使う機会が来たときに、詳しく学んでもらいたい。

## 5.1. なぜ機械学習をクラウドで行うのか？

2010年代後半に始まったAIブームのおかげで、研究だけでなく社会・ビジネスの文脈でも機械学習に高い関心が寄せられている。特に、ディープラーニングと呼ばれる多層のレイヤーからなるニューラルネットワークを用いたアルゴリズムは、画像認識や自然言語処理などの分野で圧倒的に高い性能を実現し、革命をもたらしている。

ディープラーニングの特徴は、なんといってもそのパラメータの多さである。層が深くなるほど、層間のニューロンを結ぶ“重み”パラメータの数が増大していく。例えば、最新の言語モデルである [GPT-3](#) には **1750億個** のパラメータが含まれている！このような膨大なパラメータを有することで、ディープラーニングは高い表現力と汎化性能を実現しているのである。

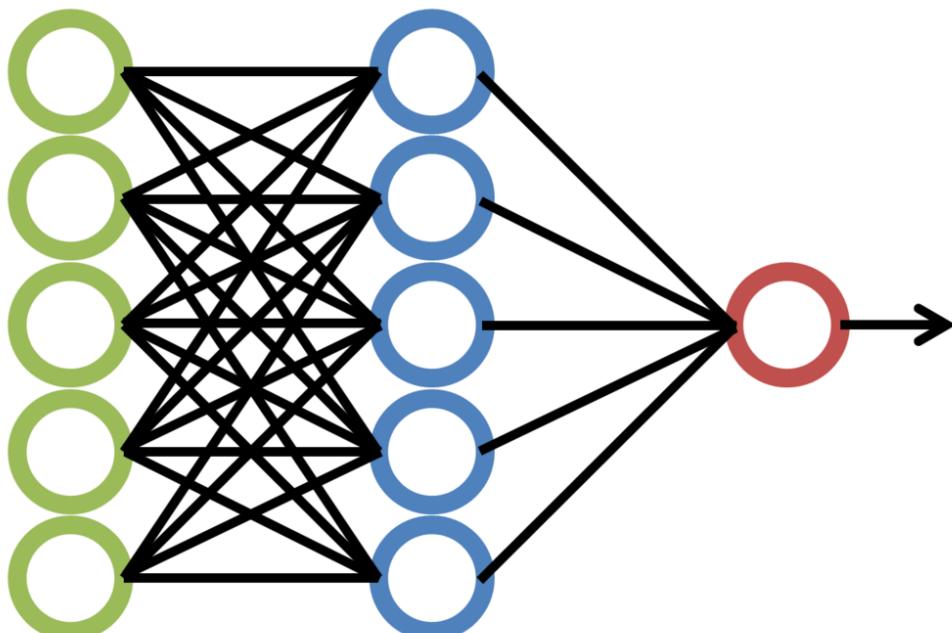


Figure 16. ニューラルネットワークにおける畳み込み演算。

GPT-3に限らず、最近の SOTA (State-of-the-art) の性能を達成するニューラルネットでは、百万から億のオーダーのパラメータを有することは頻繁になってきている。そのような巨大なニューラルネットを学習(トレイン)させるのは、当然のことながら巨大な計算コストがかかる。そんな巨大な計算に最適なのが、クラウドである！事実、GPT-3の学習も、詳細は明かされていないが、Microsoft社のクラウドを使って行われたと報告されている。



GPT-3 が発表された時、そのモデルがもつ表現能力には多くの人が驚嘆させられた。OpenAI のブログに、モデルが出力した翻訳や文章要約のタスクの結果が紹介されている。

## 5.2. GPUによる機械学習の高速化

ディープラーニングの計算で欠かすことのできない技術として, **GPU (Graphics Processing Unit)**について少し説明する。

GPUは,その名のとおり,元々はコンピュータグラフィックスを出力するための専用計算チップである.CPU (Central Processing Unit)に対し,グラフィックスの演算に特化した設計がなされている.身近なところでXBoxやPS4などのゲーム機などに搭載されているし,ハイスペックなノートパソコンやデスクトップコンピュータにも搭載されていることがある.コンピュータグラフィックスでは,スクリーンにアレイ状に並んだ数百万個の画素をリアルタイムで処理する必要がある.そのため,GPUは,コアあたりの演算能力は比較的弱いかわりに,チップあたり数百から数千のコアを搭載しており(Figure 17),スクリーンの画素を並列的に処理することで,リアルタイムでの描画を実現している.



Figure 17. GPUのアーキテクチャ.GPUには数百から数千の独立した計算コアが搭載されている。(画像出典:  
<https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/>)

このように,コンピュータグラフィックスの目的で生まれたGPUだが,2010年前後から,その高い並列計算能力をグラフィックス以外の計算(科学計算など)に用いるという流れ(**General-purpose computing on GPU; GPGPU**)が生まれた.GPUのコアは,その設計から,行列の四則演算など,単純かつ規則的な演算が得意であり,そのような演算に対しては数個程度のコアしか持たないCPUに比べて圧倒的に高い計算速度を実現することができる.現在ではGPGPUは分子動力学や気象シミュレーション,そして機械学習など多くの分野で使われている.

ディープラーニングで最も頻繁に起こる演算が,ニューロンの出力を次の層のニューロンに伝える畳み込み(Convolution)演算である.畳み込み演算は,まさにGPUが得意とする演算であり,CPUではなくGPUを用いることで学習を飛躍的に(数百倍程度!)加速させることができる.

このようにGPUは機械学習の計算で欠かせないものであるが,なかなか高価である.例えば,科学計算・機械学習に専用設計されたNVIDIA社のTesla V100というチップは,一台で約百万円の価格が設定されている.機械学習を始めるのに,いきなり百万円の投資はなかなか大きい.だが,クラウドを使えば,そのような初期コスト0でGPUを使用することができるのである!

 機械学習を行うのに、V100が必ずしも必要というわけではない。むしろ、研究者などではしばしば行われるのは、コンピュータゲームに使われるグラフィックス用のGPUを買ってきて(NVIDIA GeForceシリーズなど)、それを機械学習に用いる、というアプローチである。グラフィックス用のいわゆる"コンシューマGPU"は、市場の需要が大きいおかげで、10万円前後の価格で購入することができる。V100と比べると、コンシューマGPUはコアの数が少なかつたり、メモリーが小さかっただり、倍数精度の計算が遅かったりなどで劣る点があるが、ディープラーニングの計算は特に問題なく実行することができる。

ローカル環境でディープラーニングの開発を行うには、コンシューマGPUで十分であると筆者は考える。プログラムができあがって、ビッグデータの解析や、モデルをさらに大きくしたいときなどに、クラウドは有効だろう。

クラウドでGPUを使うには、GPUが搭載された特別なインスタンスタイプ ([P3](#), [P2](#), [G3](#), [G4](#) など) を選択しなければならない。[Table 3](#) に、代表的なGPU搭載のインスタンスタイプを挙げる(執筆時点(2020/06)での情報)。

Table 3. GPUを搭載したEC2インスタンスタイプ

Instance	GPUs	GPU model	GPU Mem (GiB)	vCPU	Mem (GiB)	Price per hour (\$)
p3.2xlarge	1	NVIDIA V100	16	8	61	3.06
p3n.16xlarge	8	NVIDIA V100	128	64	488	24.48
p2.xlarge	1	NVIDIA K80	12	4	61	0.9
g4dn.xlarge	1	NVIDIA T4	16	4	16	0.526

[Table 3](#) からわかるとおり、CPUのみのインスタンスと比べると少し高い価格設定になっている。また、古い世代のGPU (V100に対してのK80) はより安価な価格で提供されている。GPUの搭載数は1台から最大で8台まで選択することが可能である。

GPUを搭載した一番安いインスタンスタイプは、[g2dn.xlarge](#) であり、これには廉価かつ省エネルギー設計のNVIDIA T4 が搭載されている。今回のハンズオンでは、このインスタンスを使用して、ディープラーニングの計算を行ってみる。

 V100を一台搭載した [p3.2xlarge](#) の利用料金は一時間あたり \$3.06 である。V100が約百万円で売られていることを考えると、約3000時間 (= 124日間)、通算で計算を行った場合に、クラウドを使うよりもV100を自分で買ったほうがお得になる、という計算になる。

(実際には、自前でV100を用意する場合は、V100だけでなく、CPUやネットワーク機器、電気使用料もかかるので、百万円よりもさらにコストがかかる。)

GPT-3で使われた計算リソースの詳細は論文でも明かされていないのだが、[Lambda社のブログ](#)で興味深い考察が行われている (Lambda社は機械学習に特化したクラウドサービスを提供している)。

 記事によると、1750億のパラメータを学習するには、一台のGPU (NVIDIA V100)を用いた場合、342年の月日と460万ドルのクラウド利用料が必要となる、とのことである。GPT-3のチームは、複数のGPUに処理を分散することで現実的な時間のうちに学習を完了させたのであるが、このレベルのモデルになってくるとクラウド技術の限界を攻めないと達成できることは確かである。

# Chapter 6. Hands-on #2: AWSでディープラーニングの計算を走らせる

ハンズオン第二回では、前章で学んだ

ハンズオンのソースコードはこちらのリンクに置いてある ⇒ <https://gitlab.com/tomomano/intro-aws/handson/01-ec2>

ログイン

```
$ ssh -i ~/.ssh/HirakeGoma.pem -L localhost:8888:localhost:8888 ubuntu@<IP address>
```

Jupyter notebook の設定

```
$ jupyter notebook password  
Enter password:  
Verify password:  
[NotebookPasswordApp] Wrote hashed password to /home/ubuntu/.jupyter/jupyter_notebook_config.json
```

```
$ cd ~  
$ mkdir ssl  
$ cd ssl  
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout mykey.key -out mycert.pem  
Generating a RSA private key  
.....+++++  
.....+++++  
writing new private key to 'mykey.key'  
-----  
You are about to be asked to enter information that will be incorporated  
into your certificate request.  
What you are about to enter is what is called a Distinguished Name or a DN.  
There are quite a few fields but you can leave some blank  
For some fields there will be a default value,  
If you enter '.', the field will be left blank.  
-----  
Country Name (2 letter code) [AU]:JP  
State or Province Name (full name) [Some-State]:Tokyo  
Locality Name (eg, city) []:  
Organization Name (eg, company) [Internet Widgits Pty Ltd]::  
Organizational Unit Name (eg, section) []:  
Common Name (e.g. server FQDN or YOUR name) []:  
Email Address []:
```

サーバー起動

```
$ jupyter notebook --certfile=~/ssl/mycert.pem --keyfile ~/ssl/mykey.key
```

# Chapter 7. 課題

以下の課題を提出.

# Chapter 8. Appendix

## 8.1. AWS Educate のアカウント作成

執筆時点において、AWS Educateに参加すると、\$30分のAWS利用クーポンが手に入る。また、AWS Educateの提供する各種オンライン教材にアクセスすることができる。以下に登録の手順を示す。



ここに示すのは執筆時点(2020/05)での情報である。将来的に手順が変更される可能性があることに注意。

### AWS Educate アカウントの作成

1. AWS Educate のページへ行く。<https://aws.amazon.com/education/awseducate/>
2. "Join AWS Educate" > "Student" を選択。
3. アカウント情報を入力。東大ECCSのメールアドレス([g.ecc.u-tokyo.ac.jp](mailto:g.ecc.u-tokyo.ac.jp))を使用する。
4. メールアドレスの確認がメールで届くので、リンクに従って認証する。
5. アカウントが認可されるまで少し時間がかかるので、待つ。
6. アカウントが認可されると、AWS Educate にログインできるようになる。

### AWS Educate から AWS アカウントを使用する

1. AWS Educate にログインしたら、トップバーの"AWS Account"をクリック。そこから"AWS Educate Starter Account"をクリック(下図参照)。
2. 遷移した先のページ(vocareum)にある "AWS Console" をクリックすると、AWSのコンソール画面へ遷移する(下図参照)。
  - このコンソール画面から、クーポンの利用分だけ任意の機能を使うことができる。
3. 遷移した先のページ(vocareum)にある "Account Details" をクリックすると、CLIからAWSにアクセスするためのアクセスキーなどが表示される(下図参照)。
  - `aws_access_key_id`, `aws_secret_access_key` の値を確認。
  - これらの値を `~/.aws/credentials` などのファイルに保存する(参照)。

Figure 18. AWS Educate スクリーンショット1

The screenshot shows the 'Your AWS Account Status' section of the AWS Educate dashboard. It displays three metrics: 'Active' (full access), '\$30' remaining credits (estimated), and '2:55' session time. Below these metrics are two buttons: 'Account Details' and 'AWS Console'. The 'AWS Console' button is circled in red. A note below the metrics advises users to use the account responsibly, mentioning instance shutdown and logout.

Figure 19. AWS Educate スクリーンショット2

## 8.2. Python, node.js のインストール

## 8.3. AWS CLI のインストール

公式のドキュメンテーションに従い, インストールを行う.

Linuxマシンならば, 以下のコマンドを実行すれば良い.

```
$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
$ unzip awscliv2.zip
$ sudo ./aws/install
```

インストールできたか確認するため, 以下のコマンドを打って正しくバージョンが表示されることを確認する.

```
$ aws --version
```

インストールができたら, 以下のコマンドにより初期設定を行う ([参照](#)).

```
$ aws configure
```

コマンドを実行すると, **AWS Access Key ID**, **AWS Secret Access Key** を入力するよう指示される. これらの認証情報の取得については [Section 8.1](#) を参照. コマンドは加えて, **Default region name** を訊いてくる. ここには **ap-northeast-1** (東京リージョン)を指定するのがよい. 最後の **Default output format** は **JSON** としておくとよい.

上記のコマンドを完了すると, **~/.aws/credentials** と **~/.aws/config** という名前のファイルに設定が保存されているはずである. 念の為, 中身をしてみるとよい.

```
$ cat ~/.aws/credentials
[default]
aws_access_key_id = XXXXXXXXXXXXXXXXXXXX
aws_secret_access_key = YYYYYYYYYYYYYYYYYYYYY

$ cat ~/.aws/config
[profile default]
region = ap-northeast-1
output = json
```

上記のような出力が得られるはずである。

`~/.aws/credentials` には認証鍵の情報が、`~/.aws/config` には各設定が記録されている。

デフォルトでは、`[default]` という名前でプロファイルが保存される。いくつかのプロファイルを使い分けたければ、`default` の例に従って、例えば `[myprofile]` という名前でプロファイルを追加すればよい。

AWS CLI でコマンドを打つときに、プロファイルを使い分けるには、

```
$ aws s3 ls --profile myprofile
```

のように、`--profile` というオプションをつけてコマンドを実行する。

いちいち `--profile` オプションをつけるのが面倒だと感じる場合は、以下のように環境変数を設定することもできる。

```
export AWS_ACCESS_KEY_ID=XXXXXX
export AWS_SECRET_ACCESS_KEY=YYYYYY
export AWS_DEFAULT_REGION=ap-northeast-1
```

上の環境変数は、`~/.aws/credentials` よりも高い優先度を持つので、環境変数が設定されていればそちらの情報が使用される（参照）。

## 8.4. AWS CDK のインストール

公式ドキュメントに従いインストールを行う。

node.js がインストールされてれば、基本的に以下のコマンドを実行すれば良い。

```
$ npm install -g aws-cdk
```

本書のハンズオンはAWS CDK version 1.30.0 で開発した。CDK は開発途上のライブラリなので、将来的にAPIが変更される可能性がある。APIの変更によりエラーが生じた場合は、version 1.30.0 を使用することを推奨する。

```
$ npm install -g aws-cdk@1.30
```

インストールできたか確認するため、以下のコマンドを打って正しくバージョンが表示されることを確認する。

```
$ cdk --version
```

インストールができたら、以下のコマンドによりAWS側の初期設定を行う。これは一度実行すればOK。

```
$ cdk bootstrap
```



`cdk bootstrap` を実行するときは、AWSの認証情報とリージョンが正しく設定されていることを確認する。デフォルトでは `~/.aws/config` にあるデフォルトのプロファイルが使用される。デフォルト以外のプロファイルを用いるときは `AWS_ACCESS_KEY_ID` などの環境変数を設定する（[参照](#)）。



AWS CDK の認証情報の設定は AWS CLI と基本的に同じである。詳しくは [Section 8.3](#) を参照。

## 8.5. Python `venv` クイックガイド

他人からもらったプログラムで、`numpy` や `scipy` のバージョンが違う！などの理由で、プログラムが動かない、という経験をしたことがある人は多いのではないだろうか。もし、自分のPCの中に一つしかPython環境がないとすると、プロジェクトを切り替えるごとに正しいバージョンをインストールし直さなければならず、これは大変な手間である。

コードのシェアをよりスムーズにするためには、ライブラリのバージョンはプロジェクトごとに管理されるべきである。それを可能にするのが Python 仮想環境と呼ばれるツールであり、`venv`, `pyenv`, `conda` などがよく使われる。

そのなかでも、`venv` は Python に標準搭載されているので、とても便利である。`pyenv` や `conda` は、別途インストールの必要があるが、それぞれの長所もある。

```
$ python -m venv .env
```

というコマンドを実行することで、`venv` モジュールにより `.env/` というディレクトリが作られる。

この仮想環境を起動するには

```
$ source .env/bin/activate
```

と実行する。

シェルのプロンプトに `(.env)` という文字が追加されていることを確認しよう。これが、"いまあなたは `venv` の中にいますよ" というしるしになる。

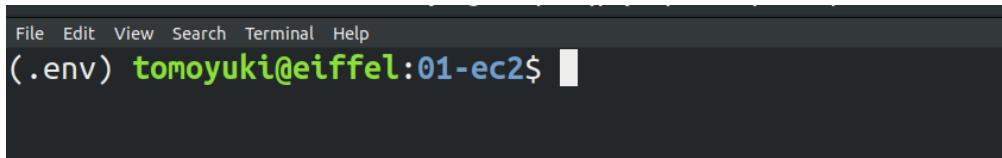


Figure 20. `venv` を起動したときのプロンプト

仮想環境を起動すると、それ以降実行する `pip` コマンドは、`.env/` 以下にインストールされる。このようにして、プロジェクトごとに使うライブラリのバージョンを切り分けることができる。

Python では `requirements.txt` というファイルに依存ライブラリを記述するのが一般的な慣例である。他人からもらったプログラムに、`requirements.txt` が定義されていれば、

```
$ pip install -r requirements.txt
```

と実行することで、必要なライブラリをインストールし、瞬時にPython環境を再現することができる。

# Chapter 9. 著者紹介

真野 智之 (Tomoyuki Mano) 東京大学大学院情報理工学系研究科博士課程

研究分野 神経科学・神経情報学

現在の研究テーマ クラウドを用いた脳画像解析・データベース構築

趣味 料理・ランニング・鉄道・アニメ・村上春樹

連絡先 [tomoyukimano@gmail.com](mailto:tomoyukimano@gmail.com)

GitLab <https://gitlab.com/tomomano>

# Chapter 10. ライセンス