ビッグデータ処理基盤における プログラミング言語に関する四方山話

2026/08/29

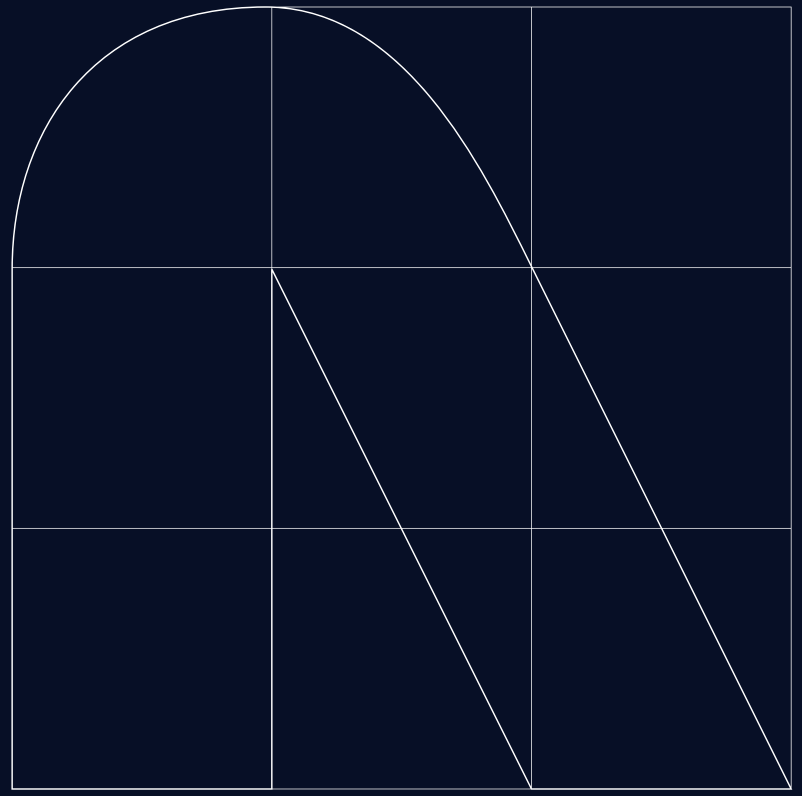
株式会社NTTデータ
岩崎 正剛

自己紹介

岩崎 正剛 / Masatake Iwasaki

- Open source software developer / data engineer @NTT DATA Japan
- Committer & PMC member of Apache Hadoop and Apache Bigtop
- ASF member

Hadoopエコシステムの紹介



Apache Hadoopとは何か?

- Hadoopは大規模データ処理のためのミドルウェア
- 汎用的なLinuxサーバ(～10000台)でクラスタを構成し、大量のデータを格納しつつ、並列分散処理することができる。
- 大きく分けて、以下の3種類の要素からなる。
  - 分散ファイルシステム(HDFS)
  - 計算リソース管理機構(YARN)
  - YARN上に実装された分散処理フレームワーク(MapReduce)

- Googleが自社の検索サービスに利用していた技術を、論文の形で紹介した内容を参考にして開発されたOSS
  - 論文の内容は、書籍『Googleを支える技術』などで紹介され、日本でも知られるところとなった
- 2006年に最初のバージョンが公開された
  - 初期の開発者はFacebook、Yahoo!、Twitterなどに所属
  - 汎用的なHWで安価に処理できるデータ規模が上がった
  - 「ビッグデータ」がbuzzwordだったのは2011～2014くらい?
  - その後、クラウド、コンテナ、機械学習、AI...とキーワードが推移する

Hadoopエコシステム

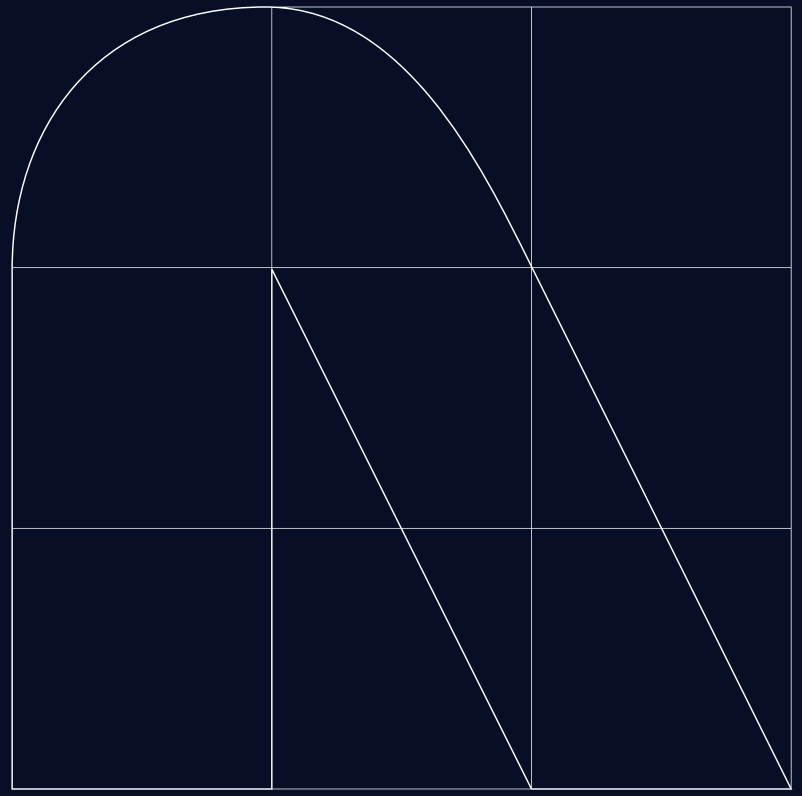
- Hadoopは実用上、多様な周辺ミドルウェアと組み合わせて利用されるようになった
- 周辺ミドルウェアの代表的な例を挙げると
  - Hive: SQL(ライクな)言語で並列分散データ処理を実行できる処理系
  - Spark: 汎用的なAPIで並列分散データ処理を実装できるフレームワーク
  - Flink: ストリームデータ処理が得意な並列分散処理フレームワーク
  - HBase: GoogleのBigtableを参考に実装された分散キーバリューストア
  - Ozone: オブジェクトストレージ
  - Zeppelin: ノートブック
  - Ranger: アクセス制御基盤
- Hadoopおよび周辺ミドルウェアの実装言語は、Java/JVM言語が主流。
- 製品間には依存関係があり、あるプロダクトに非互換な変更が入ると、それに依存するプロダクト側でも修正が必要。
- 各製品はそれぞれ独立の開発者グループによって維持され、製品間の互換性は時々壊れる。
- どのバージョンを組み合わせると機能するかは、各製品の開発者にとっても、自明ではない。

Hadoop as a services

- Hadoopが登場した200x年頃は、まだクラウド上で大規模データ処理を行う下地がなかった
  - 汎用IAサーバを並べて使うので、個人ユーザにはハードルが高い?

- いまではHadoopのPaaSが、各種主要クラウドサービスでも提供されている
  - Amazon EMR
  - Azure HDInsight
  - Google Cloud Dataproc
- 各サービスのデータストアのケイパビリティも上がり、Hadoopと組み合わせての利用もしやすい
  - Amazon S3
  - Azure Data Lake Storage
  - Google Cloud Storage

HadoopとJava



HadoopとJava

- Hadoopは、Javaで実装されたNutchというWebクローラから派生して生まれた
  - NutchはJavaで実装されたLuceneという全文検索ライブラリの関連プロジェクト
- Hadoopのサブプロジェクトとして(同じソースツリー内から)出発したHBaseやZooKeeperなども、結果としてJavaに
- Javaの恩恵
  - マルチプラットフォーム(write once, run anywhere)
    - ただしHadoop本体はネイティブコードも使う結果、サーバプラットフォームは事実上Linux前提(後述)
  - 自動メモリ管理
  - ライブラリの充実度
    - 標準ライブラリも充実していて高品質
    - モジュール管理の仕組み(Maven)で、サードパーティーの豊富なライブラリも利用しやすい
  - エンタープライズ用途で広く利用されてきた
- Javaの?制約
  - ヒープサイズの上限が、スケーラビリティの上限 (HDFSのファイルシステムメタデータサイズについて)
  - ミドルウェアおよびユーザアプリケーションとの依存関係の衝突(dependency hell)には苦労する

HadoopがサポートするJavaのバージョン

- 長いことJava 8から脱却できずにいた
- 2.xと3.xは、それなりの期間、並行して維持されていた
- コード修正をcherry-pickすることを考慮すると、新しくサポートされたJavaの新機能の利用を控えがち
  - 例) 2.x維持中はJava 8で導入されたlambdaを積極的に使わなかった

Version	Release date	Compile	Runtime
2.7.0	Jul 2015	Java 7	Java 7, Java 8
...			
2.10.0	Oct 2019	Java 7	Java 7, Java 8
3.0.0	Dec 2017	Java 8	Java 8
...			
3.2.0	Jan 2019	Java 8	Java 8
3.3.0	Jul 2020	Java 8	Java 8, Java 11
3.4.0	Mar 2024	Java 8, (Java 11)	Java 8, Java 11
3.5.0	Apr 2026	Java 17	Java 17

Javaのリリースサイクル

- (Java 9以降)半年ごとにfeature releaseを出す
- 約2年ごとにLTS(Long Term Support)版を出す

バージョン	GA
6	2006-12-12
7	2011-07-11
8 (LTS)	2014-03-18
9	2017-09-21
10	2018-03-20
11 (LTS)	2018-09-25
12	2019-03-19
13	2019-09-17
14	2020-03-17
15	2020-09-15
16	2021-03-16
17 (LTS)	2021-09-14
18	2022-03-22
19	2022-09-20
20	2023-03-21
21 (LTS)	2023-09-19
22	2024-03-19
23	2024-09-17
24	2025-03-18
25 (LTS)	2025-09-16
26	2026-03-17

OpenJDKのディストリビューション

- (基本的には)LTS版をベースに、OSディストリビュータなどがOpenJDKディストリビューションを提供

ディストリビューション名	提供主体	特徴
Red Hat build of OpenJDK	Red Hat	RHELにバンドルし、企業向けに提供。8/11/17/21/25 を長期メンテナンス対象とするライフサイクルを公開。
Eclipse Temurin	Eclipse Adoptium プロジェクト	旧 AdoptOpenJDK。コミュニティベースの無償ビルド。8/11/17/21 をLTSとして長期提供。
Amazon Corretto	AWS (Amazon Web Services)	AWS 環境での利用を前提とした案内が多く、8/11/17/21 をLTSとして長期サポート。
Microsoft Build of OpenJDK	Microsoft	Azureなど Microsoft 環境向けに提供。11/17/21 をLTSとして長期サポートすることを明示。短期版は長期サポート対象外。
Oracle JDK	Oracle	Oracleが提供する商用ディストリビューション。利用には有償サブスクリプションが必要。8/11/17/21/25 をLTSとして Premier / Extended Support を設定。
Azul Zulu	Azul	Azul Platform Core の一部として提供。8/11/17/21/25 などを中心に長期サポート(有償)。無償の Zulu Community 版はLTS対象外。

Red Hat build of OpenJDKのサポートライフサイクル

- OS(Red Hat Enterprise Linux)とは独立のサポートライフサイクルがある
- OpenJDK 8のEOLが11より長い
  - 何度か延長された結果
  - Java 8から11へのギャップが大きく、移行に苦労してるミドルウェア/ライブラリが多いことを反映?
- https://access.redhat.com/articles/1299013

OpenJDK on RHEL Support Table

	Supported RHEL Versions	End of Full Support	End of ELS-1
OpenJDK 6 (1.6)	RHEL 5.3+, 6.0+, 7.0+	December 31, 2016	N/A
OpenJDK 7 (1.7)	RHEL 5.9+, 6.3+, 7.0+	June 30, 2020	N/A
OpenJDK 8 (1.8)	RHEL 6.6*, 7.1+, 8.0+, 9.0+*	**November 30, 2026***	**December 31, 2030**
OpenJDK 11	RHEL 7.6+, 8.0+, 9.0+	**October 31, 2024**	**October 31, 2027**
OpenJDK 17	RHEL 8.4+, 9.0+	**December 31, 2027**	N/A
OpenJDK 21	RHEL 8.9+, 9.3+, 10.0+	**December 31, 2029**	N/A
OpenJDK 25	RHEL 9.7+, 10.1+	**December 31, 2030**	N/A

Javaのアップグレードに必要な作業

- コンパイルエラーになるHadoop自体のコードの修正
- エラーになるテストケースの確認と対応
- ビルドツールをJavaバージョン対応版にアップグレード
  - Mavenおよびそのプラグイン
- 依存ライブラリを新しいJavaバージョン対応版にアップグレード
  - JUnit、Mockit、Jersey
- 開発者/CI環境向けのDockerfileの更新
- ドキュメント中のJavaに関する記述の更新

- 必要な修正量が多いパターンの例
  - これまでwarningだったJavadocの書き方がerrorになる
  - エラー時の挙動の変化で、通らなくなるテストの修正
    - 例: テスト上期待されている例外のメッセージの文言が変わった
  - JUnitのアップグレード: JUnitのテストケースを4の流儀から5に書き換え
    - JUnit 5上で古いAPIを使う仕組み(junite-vintage-engine)があるので、書き換えを後回しにしてゆっくりやれる
  - Jerseyのアップグレード: REST APIを定義するコードの修正

HADOOP-15984: Update jersey from 1.19 to 2.x

- JAX-RS (REST APIのための標準/仕様)の実装であるJerseyのアップグレード
- 解決までに5年以上経過した。
- 難しい部分:
  - REST APIは各デーモンで実装されていて、影響範囲が多い。
  - バージョン差分で使い方が変わる部分があり、コード修正が避けられない。
  - Jersey 1と2を共存させて、サブモジュール単位で徐々に移行していく戦略がとれない。

Jerseyを利用してREST APIを実装するコードの例:

```
@GET
@Path("{" + UriFsPathParam.NAME + ":.*}")
@Produces({MediaType.APPLICATION_OCTET_STREAM + "; " + JettyUtils.UTF_8,
    MediaType.APPLICATION_JSON + "; " + JettyUtils.UTF_8})
public Response get(
    @Context final UserGroupInformation ugi,
    @Context final UriInfo uriInfo,
    @QueryParam(DelegationParam.NAME) @DefaultValue(DelegationParam.DEFAULT)
        final DelegationParam delegation,
    @QueryParam(UserParam.NAME) @DefaultValue(UserParam.DEFAULT)
        final UserParam username,
    @QueryParam(DoAsParam.NAME) @DefaultValue(DoAsParam.DEFAULT)
        final DoAsParam doAsUser,
    @QueryParam(GetOpParam.NAME) @DefaultValue(GetOpParam.DEFAULT)
        final GetOpParam op,
    ...
```

巨大なパッチ

- Hadoopは4つのサブプロジェクト(Common, HDFS, YARN, MapReduce)の集まり
  - 過去に分割するかどうかの議論はあったが、しないことになった
- Mavenのサブモジュール数としては100以上
- 各種デーモンはREST APIを持ってるので、多くのサブモジュールが影響を受ける
- 各モジュールで開発が並行しているため、巨大なパッチは、すぐにconflictしがち
- 関連するテストをすべて流しきるのに数時間以上かかる

Jersey 1.xと2.xを併用できない問題

- Javaでは、同一クラスローダ上に同じクラスの異なるバージョンは並存できない
  - Mavenにおいて、同一artifactの異なるバージョンを併用できない
- Jersey 1.x と 2.xはパッケージ名は変わっているので、Jerseyのコード自体は併用できるはず
  - com.sun.jersey.* -> org.glassfish.jersey.*
- Jersey 1.xはJAX-RS 1.1の実装なのに対して、Jersey 2.xはJAX-RS 2.0の実装
  - javax.ws.rs.* の異なるバージョンに依存する
    - (jsr311.jarとrs-api.jarが共存できない)
  - 関係するすべて(JAX-RSとJerseyとJackson?)をrelocation(後述)して共存させ、使い分けるのは困難

- Jersey 2.xをスキップしてJersey 3.x(JAX-RS 3.xの実装)に移行するのはどうか?
  - OracleからEclipse FoundationにJava EEが移管され、Jakarta EEになった結果、APIのパッケージ名が変わった
    - javax.ws.rs.* -> jakarta.ws.rs.*
  - Jersey 3.xにするためには、Jettyを11にアップグレードする必要があり、影響範囲が大きくなってしまう

解決へのステップ

- 部分的に実施可能な修正を先に行い、ある程度パッチのサイズを減らす
  - Jerseyの依存ライブラリのバージョン差分に起因する修正
  - テストコードから本質的には必要ないJersey依存部分を消す
- 実行環境 でJava 11や17対応するために、標準ライブラリから消えてしまった依存ライブラリ(JAXB)の代用品を足す
  - Hadoop 3.3はJava 11を、Hadoop 3.4はJava 17を、実行環境としてサポート

- 最終的には巨大なパッチをマージ
  - 最終的には: 392 files changed, 32897 insertions(+), 26930 deletions(-)
  - .diffのサイズとしては25kBくらい?
- Hadoop 3.5はビルドも実行環境もJava 17前提になった

- 時間経過により、開発コミュニティの活動量/コード修正ペースも落ち着き、大きなパッチがマージしやすくなった?
- AIの進化と普及により、いまなら力業が容易?
  - HADOOP-15984のコミット時点ではまだ使われていなかった(はず)

MavenのTransitive dependencies

- https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Transitive_Dependencies
- 依存ライブラリの依存ライブラリも依存ライブラリ
- 依存関係ツリー上に複数のバージョンがある場合、近いものが勝つ (dependency mediation)
- mediationの結果、問題なくビルドできて動くという保証はない
- Hadoopのクライアントを使うミドルウェアの依存関係ツリーはとても深い
- ミドルウェアとして、無駄に依存関係を増やさない/外に見せないべきという意識は、広がった気がする
  - 今日では、AIで車輪の再発明的なコードを作って抱えるのも容易?

hadoop-commonのcommon-loggingに関するdependency mediation:

```
$ mvn dependency:tree -Dmaven-dependency-plugin.version=2.10 -Dverbose
...
[INFO] org.apache.hadoop:hadoop-common:jar:3.4.0-SNAPSHOT
...
[INFO] +- org.apache.httpcomponents:httpclient:jar:4.5.13:compile
[INFO] |  +- org.apache.httpcomponents:httpcore:jar:4.4.13:compile
[INFO] |  +- (commons-logging:commons-logging:jar:1.1.3:compile - version managed from 1.2; omitted for duplicate)
...
[INFO] +- commons-logging:commons-logging:jar:1.1.3:compile
...
[INFO] +- commons-beanutils:commons-beanutils:jar:1.9.4:compile
[INFO] |  +- (commons-logging:commons-logging:jar:1.1.3:compile - version managed from 1.2; omitted for duplicate)
[INFO] |  ¥- (commons-collections:commons-collections:jar:3.2.2:compile - omitted for duplicate)
[INFO] +- org.apache.commons:commons-configuration2:jar:2.1.1:compile
[INFO] |  ¥- (commons-logging:commons-logging:jar:1.1.3:compile - version managed from 1.2; omitted for duplicate)
```

maven-shade-pluginを利用したパッケージ名の置換

- Javaの仕組み上、同一パッケージ名のライブラリの、異なるバージョンを併用することはできない。
- maven-shade-pluginの機能で、既存ライブラリのパッケージ名を置き換えたshadedライブラリを作ることができる。
- Hadoopでは、他のミドルウェアやユーザアプリケーションとの依存ライブラリのバージョン競合を防ぐ目的で利用されている。
- https://github.com/apache/hadoop-thirdparty

hadoop-thirdparty/hadoop-shaded-protobuf_3_25のpom.xml から抜粋(した内容を分かりやすさのために変数展開したもの):

```
<relocation>
  <pattern>com/google/protobuf</pattern>
  <shadedPattern>org.apache.hadoop.thirdparty.protobuf</shadedPattern>
</relocation>
```

同一ライブラリの新旧バージョンの併用

- shadedライブラリと元のライブラリは共存できるので、同じライブラリの新旧バージョンを併用できる。
- 併用状態にすることで、サブモジュール単位で、バージョン競合に関する問題をい、段階的に修正できる。
- (Jerseyの場合は、JAX-RSのバージョン競合に起因する問題で、この手法が簡単に適用できなかった)

新旧ライブラリを併用:

```
<dependencies>
   <dependency>
     <groupId>org.apache.hadoop.thirdparty</groupId>
     <artifactId>hadoop-shaded-protobuf_3_25</artifactId>
   </dependency>
   ...
   <dependency>
     <groupId>com.google.protobuf</groupId>
     <artifactId>protobuf-java</artifactId>
     <version>2.5.0</version>
     <scope>provided</scope>
   </dependency>
```

relocateされたライブラリを明示的に利用:

```
import org.apache.hadoop.thirdparty.protobuf.BlockingService;
```

hadoop-client-apiとhadoop-client-runtime (aka shaded client)

hadoop-clientのtransitive dependenciesを隠したもの
Hadoop 3.0.0で登場 (HADOOP-11804)
maven-shade-pluginのrelocation機能を利用

- hadoop-clientの依存ライブラリと、その呼び出し箇所のパッケージ名をバイトコード上書き換え
- 書き換え後の依存ライブラリをhadoop-client内部用に内包

hadoop-client-api: relocateされたorg.apache.hadoop.*が入ったfat jar
hadoop-client-runtime: relocateされた依存ライブラリが入ったfat jar

hadoop-client-apiとhadoop-client runtime:

```
$ ls -lh hadoop-client-api-3.4.0-SNAPSHOT.jar
-rw-rw-r--. 1 centos centos 19M Dec 11 11:05 hadoop-client-api-3.4.0-SNAPSHOT.jar

$ ls -lh hadoop-client-runtime-3.4.0-SNAPSHOT.jar
-rw-rw-r--. 1 centos centos 30M Dec 11 11:07 hadoop-client-runtime-3.4.0-SNAPSHOT.jar

$ jar tvf hadoop-client-runtime-3.4.0-SNAPSHOT.jar | grep '/shaded/' | awk '{print $8}'
...
org/apache/hadoop/shaded/com/google/common/annotations/VisibleForTesting.class
org/apache/hadoop/shaded/com/google/common/base/Absent.class
...
```

Hadoopのネイティブコード (C)

- libhadoop.so (Hadoop自身がJNIで呼び出すネイティブコード)
  - compression codec(LZ4, Snappy, zlib, Zstd)
  - 高速化(CRC32 by pipelined SSE42, user/group mapping, OpenSSL cipher)
  - HDFSの諸機能
    - Short Circuit Local Read: sendmsg/recvmsgでのオープン済ファイルディスクリプタ受け渡し
    - read ahead/drop behind: posix_fadviseによるキャッシュ戦略指定
    - cacheadmin: mmap/mlockによるデータブロックのメモリ保持
- libhdfs (HDFSのCクライアント)
- Linux container-executor(YARNのセキュリティ機能)

- (Hadoop以外の周辺ミドルウェアでは、ネイティブコードの利用箇所は少ない)

HadoopのネイティブコードとJavaの進化

- Javaのバージョンが新しければ、不要になる部分もある
  - 例えばCRC32の計算のパイプライン化は、Java 9以降の標準ライブラリでも行われている
    - AVX-512をサポートしているCPU上なら、それを使えるJava標準ライブラリの方が効率がよいはず
  - Javaアップグレードの恩恵を受けられるまでのタイムラグが(現状)大きいが…
- LZ4, Snappy, Zstdなどのコーデックは、サードパーティのJavaライブラリが利用できるようになって移行した
- Java 22で導入されたFFMも、JNIの置き換えとして期待される

bulk_crc32_x86.cから抜粋:

```
switch (num_blocks) {
   case 3:
     /* Do three blocks */
     while (likely(counter)) {
       __asm__ __volatile__(
       "crc32l (%7), %0;¥n¥t"
       "crc32l (%7,%6,1), %1;¥n¥t"
       "crc32l (%7,%6,2), %2;¥n¥t"
        : "=r"(c1), "=r"(c2), "=r"(c3)
        : "r"(c1), "r"(c2), "r"(c3), "r"(block_size), "r"(data)
       );
       data++;
       counter--;
     }...
```

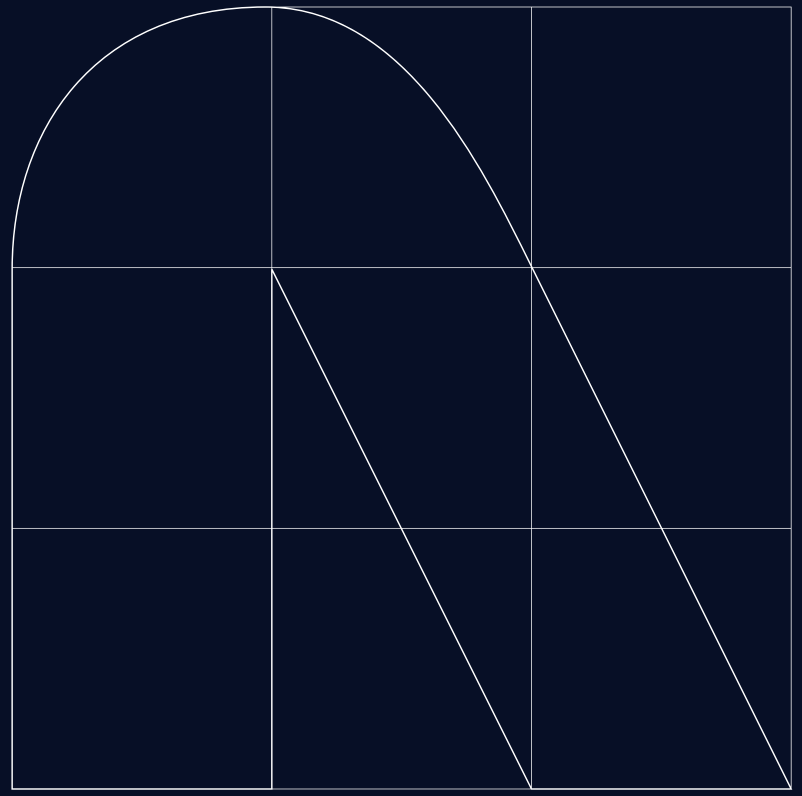
Hadoopのネイティブコード (C++)

- HadoopにおけるC++の利用場面は限定的
  - libhdfspp (HDFSのC++クライアント)
    - Apache ImpalaのようなC++で実装された周辺ミドルウェアが利用
  - libnativetask (MapReduceの内部処理を高速化するための仕組み(optional))

ネイティブライブラリのビルド

- MavenでnativeプロファイルをActivate化することでビルドできる
  - `mvn install -Pnative ...`
- これらをビルドするために、開発環境に必要な資材が重め。
  - GCC、CMake、Boost...

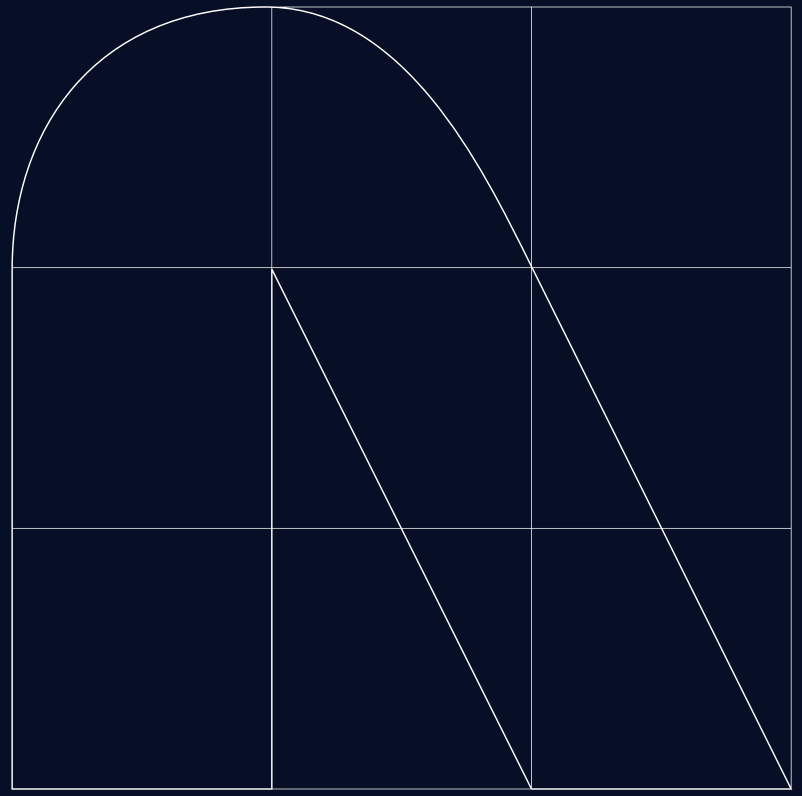
KuduとC++



Apache Kudu

- 分散カラムナデータストア
- HDFSおよびHBaseのコア開発者だったエンジニアが、その経験を踏まえて作った
- 構造化データを対象：データファイルのフォーマットはParquetに似ている
  - ユースケースの大半は、構造化データを扱うものだったので、そこに最適化
  - SQL処理系(ImpalaのようなMPP)から使われるのに適した構造
  - (HBaseは非構造化データ/キーバリューのバイト列を扱う分散KVSだった)
- (HDFSのような)外部データストアに非依存
  - 冗長化のためにRaftのロジックを、自前で実装
  - (HBaseは、HDFSとのバージョン整合性を取るのに苦労した)
- C++で実装
  - JavaヒープとGCに起因する制限に縛られないように自前でメモリ管理
  - Linuxの新しい機能をいち早く使えるように
    - JNIを使ったコードはメンテナンスしにくい
  - dependency hellの回避
    - B-treeのロジックも自前実装：インメモリ用と、オンディスク用の2種類ある

SparkとScala



Apache Spark

- Collection APIとほぼ同じインターフェースで、簡潔に記述したデータ操作が、並列分散処理されるフレームワーク
  - データ処理を多段MapReduceジョブに落とし込む場合の非効率を解消する
- Scala + Javaで実装されている
- Hadoopのライブラリを利用している
  - データの置き場所として、クラウドストレージなども利用できる
- サーバリソース管理の仕組みはいろいろ選べる
  - ビルトイン
  - Hadoop (YARN)
  - Kubernetes

SparkのQuick Startから抜粋:

```
scala> val textFile = spark.read.textFile("README.md")
scala> val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).count()
scala> wordCounts.collect()
```

Scala

- JVM言語
- オブジェクト指向言語かつ関数型言語
- 静的型付けかつ、case classと型推論により、定型的な記述が少なくて済む
- イミュータブルなコレクションと高階関数で、データ処理を宣言的に記述できる

Scalaのサンプルコード:

```
case class Sale(item: String, amount: Int)
val sales = List(Sale("A", 120), Sale("B", 80), Sale("A", 50))
val total = sales.groupBy(_.item).mapValues(_.map(_.amount).sum)
// totalの値はMap(A -> 170, B -> 80)で、型は Map[String, Int] と推論される
```

Scalaのcompatibility

- Scala 2では、major/minorバージョン間で互換性は担保されなかった
- 2.12 -> 2.13で大きく構造が変わったため、アップグレードが難しかった
- Spark 3.3以降の3.xは、Scala 2.12向けと、2.13向けに、それぞれビルドされたバイナリを提供
- Scala 3では、セマンティックバージョニングに従い、同一メジャーバージョン内での後方互換性を保つポリシーになった
- 追加された中間表現(TASTy)により、Scala 2.13.4からScala 3のライブラリを利用可能に
- SparkのScala 3対応はこれから (SPARK-54150)

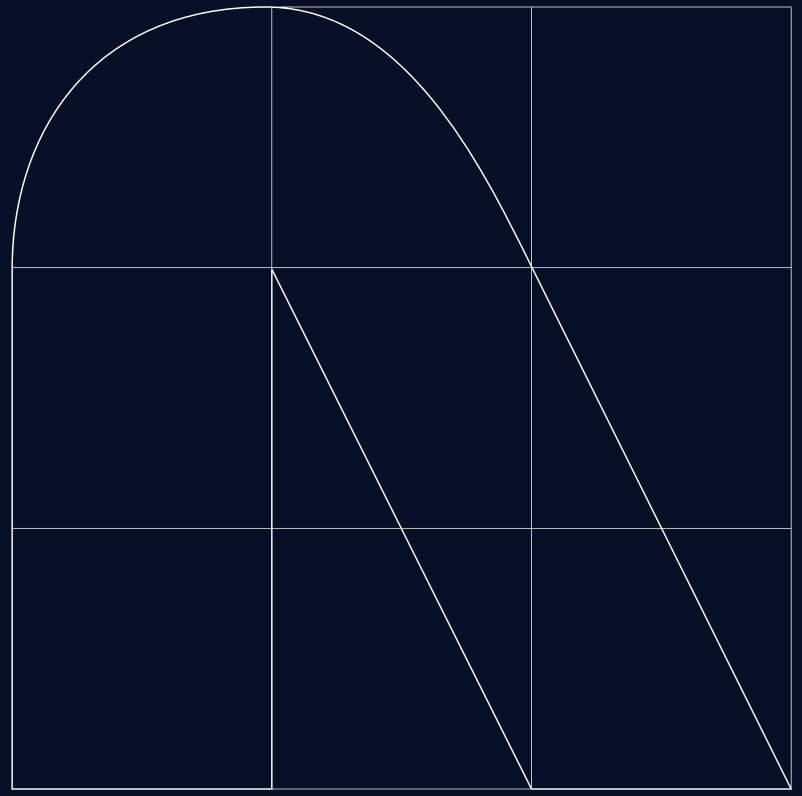
PySpark

- SparkをPythonから利用するためのAPI
- データサイエンス分野ではPythonがポピュラー
  - 数値計算や機械学習のライブラリが充実している
  - SparkのAPI(DataFrame)が影響を受けた部分もある
- 旧来のPySparkは、Py4Jで起動したJavaプロセスとやりとりして処理を行う仕組み
- Spark Connectは、置き換えとして利用できる、新しい仕組み
  - gRPCでSpark Connectサーバとやりとりする
- PySpark上で実装されたPandasもある。

Pandas on PySpark:

```
>>> import pyspark.pandas as ps
>>> psdf = ps.range(10)
>>> sdf = psdf.to_spark().filter("id > 5")
>>> sdf.show()
```

StormとClojure



Apache Storm

- ストリーミングデータ処理のためのミドルウェア
- データの到着しだい、細かい単位で、低レイテンシで処理するのが得意
- 当初はClojure(とJava)で実装されていた
- スピーディに実装するため(生みの親であるNathan Marz談)
- インターフェースはJavaなのでユーザアプリケーションはJavaでも実装できる
- Thriftを利用した他言語とも連携するための仕組みもあり

Clojure

- Lisp系のJVM言語
- 丸括弧(parenthesis)以外の括弧も、馴染みのあるconventionで使われていて、多少読みやすい
  - 各括弧(bracket)でvector: `[1 2 3]`
  - 波括弧(brace)でmap: `{:foo 1 :bar 2 :baz 3}`
- 並行プログラミング向けの道具立てを提供
- 破壊的な変更が言語処理系に入ったことがない

JavaとLispとClojureを比較

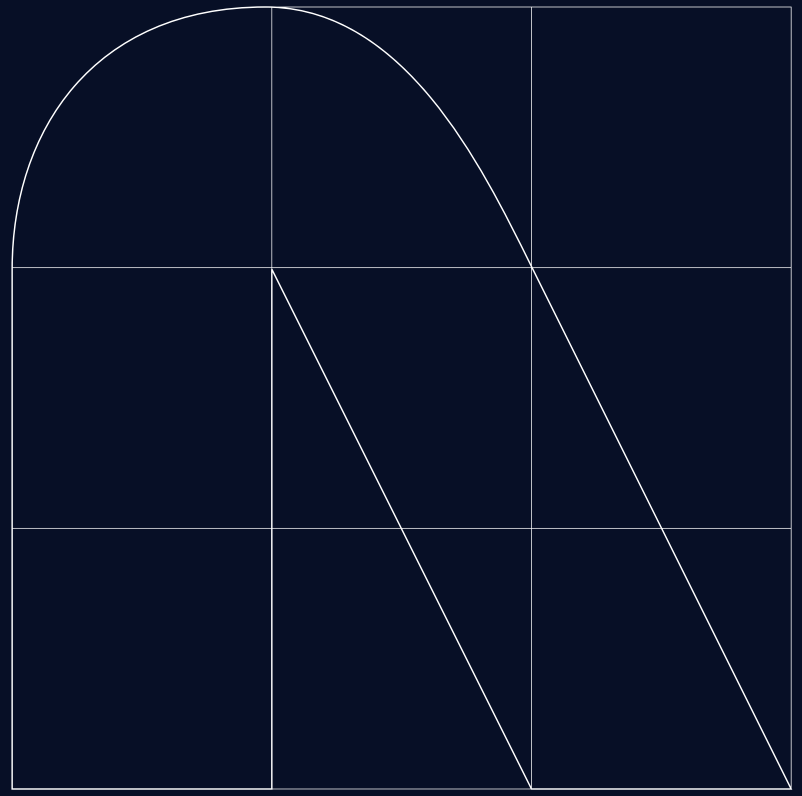
```
// Java
users.stream()
     .filter(u -> "active".equals(u.get("status")))
     .mapToDouble(u -> (Double) u.get("salary") * 1.10)
     .sum();

; Lisp
(reduce +
  (map (lambda (s) (* s 1.10))
    (map (lambda (u) (cdr (assoc 'salary u)))
      (filter (lambda (u) (string=? (cdr (assoc 'status u)) "active")) users))))

;; Clojure
(->> users
     (filter #(= (:status %) "active"))
     (map :salary)
     (map #(* % 1.10))
     (reduce +)))
```

Migration to Java

- Storm 2.0.0は、コードベースがJavaに移行した形でリリース(STORM-1225)
- Contributorの敷居を下げるため
- JStormのコードを取り込むため
  - AlibabaがStormをJavaでリライトし、改良を加えたもの

- Storm 3.0.0で、ユーザAPIのClojureサポートを削除 (#8537)
- メンテナンスを楽にするため
- 近年activeなcommitterもClojure得意なわけではないらしい

フロントエンドとスクリプト言語



CLI

- Java/JVM言語で実装された製品の場合、オプションや引数を整理して、Javaプロセスを起動するもの
- シェルスクリプトが主流
  - 可読性が低く、デバッグがしにくい
  - Hadoopのシェルスクリプトは、結構複雑
  - モジュール化やユニットテストの仕組み(Bats)はある程度活用されている

- Pythonもよく利用されている
  - シェルよりはコードが維持しやすい
  - ポピュラーなLinuxディストリビューションに標準でバンドルされている
  - 標準ライブラリが充実している
  - Python 2 -> 3への移行のような問題が発生することはある

対話環境(REPL)

- ScalaやClojureは対話環境をビルトインで提供していることも、アピール材料のひとつ
- SparkはScalaの対話環境をベースとしたspark-shellを提供
- Java標準の対話環境(REPL)としてjshellが追加されたのはJava 9
- spark-shellやhbase shellは、Hadoopの動作実験にも便利だった
  - Hadoop含む関連ライブラリにclasspathが通った状態で対話環境が起動してくるので

対話環境: HBaseとJRuby

- HBaseはCLI(hbase-shell)を、JRubyの対話環境(IRB)ベースで実装
- メソッドの引数を囲む括弧が省略可能なことでコマンド的な雰囲気
- 運用のためのちょっとした処理をJRubyスクリプトで記述できる

HBase shellでのコマンド実行例:

```
> create 'test', 'f'
> put "test", "r00001", "f:", 1
> (2..10).each { |i| put "test", "r%05d" % i, "f:", i}
> scan 'test', {LIMIT => 3}
ROW                   COLUMN+CELL
 r00001               column=f:, timestamp=2026-08-27T20:53:33.351, value=1
 r00002               column=f:, timestamp=2026-08-27T20:53:46.795, value=2
 r00003               column=f:, timestamp=2026-08-27T20:53:46.799, value=3
3 row(s)
```

Web UI

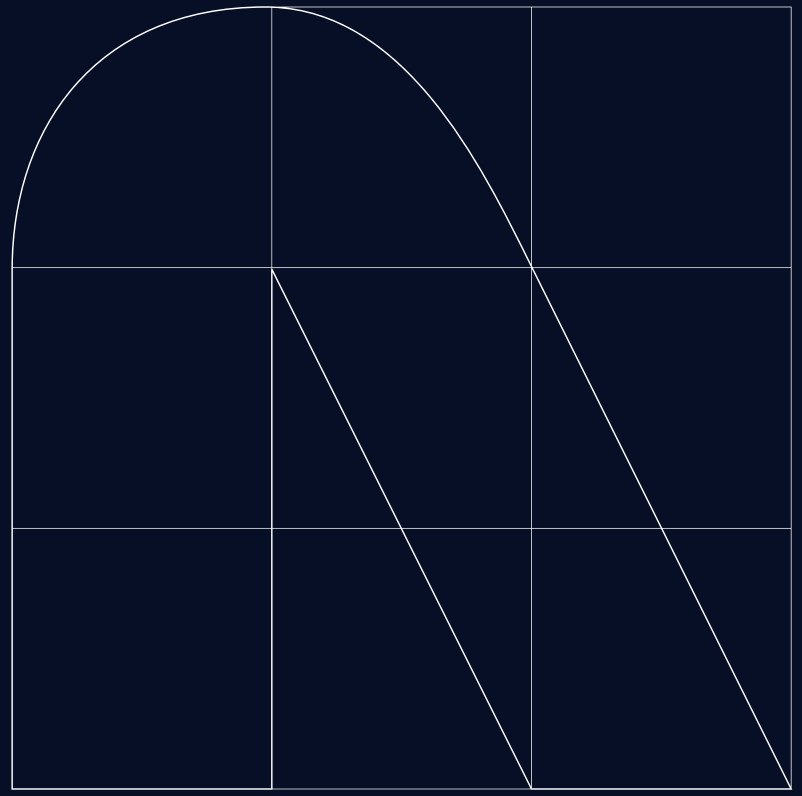
- 運用管理のためのWeb UIの多くはJavaScriptで実装されている
  - HadoopやSparkの基本的なWeb UIは凝ってない
    - JQuery、Bootstrap、DataTables、D3等をシンプルに使う
- 本格的なフレームーワーク(React/Vue/Angular)やパッケージマネージャ(Yarn)を使うと、維持に苦労する
  - Hadoop YARN-UI V2が該当
  - 脆弱性対応のためのバージョンの追従
  - ライブラリ間のバージョン整合性
  - frontend-maven-pluginを使うことでビルドはシームレスに実行できるが
  - 別ソースツリーで管理したほうが無難
- データ処理基盤のミドルウェア開発者の得意分野ではない?

ノートブック: Apache Zeppelin

コードがそのまま動くドキュメントをつくれるような環境
分析結果のデータを動的に、表や図として視覚化
データサイエンス分野ではJupyter Notebookがポピュラー

観点	Apache Zeppelin	Jupyter Notebook
実行基盤	JVM上で動作し、インタプリタを差し替えて利用	Pythonを中心としたカーネル方式
対応言語	1つのノート内でScala / SQL / Pythonなどを段落ごとに併用	カーネル単位で言語を選択(Python中心)
Spark連携	Sparkインタプリタを組み込み	PySpark + カーネルの組み合わせが一般的
可視化・共有	表やグラフの描画、ノート共有・スケジュール実行を内蔵	描画は各種ライブラリ、共有はnbconvertなど周辺ツール
エコシステム	Hadoopエコシステムの一員(Bigtopなどに同梱)	データ分析・機械学習分野のライブラリが非常に充実
主な使いどころ	JVMベースの分散処理基盤をベースに利用	Python資産を活かした分析ワークフロー

エディタ、IDE、ビルドツール



エディタとIDE

- Emacs + gtags(GNU GLOBAL)は複数の言語を横断的にカバーできて意外と便利
  - Pygments(シンタックスハイライトのためのライブラリ)と組み合わせると、多くの言語に対応できる
  - Lisp系のClojureでは、Emacsがデファクトの開発環境
- Javaに関しては、IntelliJ IDEAが長らく一番人気だった印象
- Visual Studio Code(VS Code)とLSPが普及してからは、こちらのユーザが多い

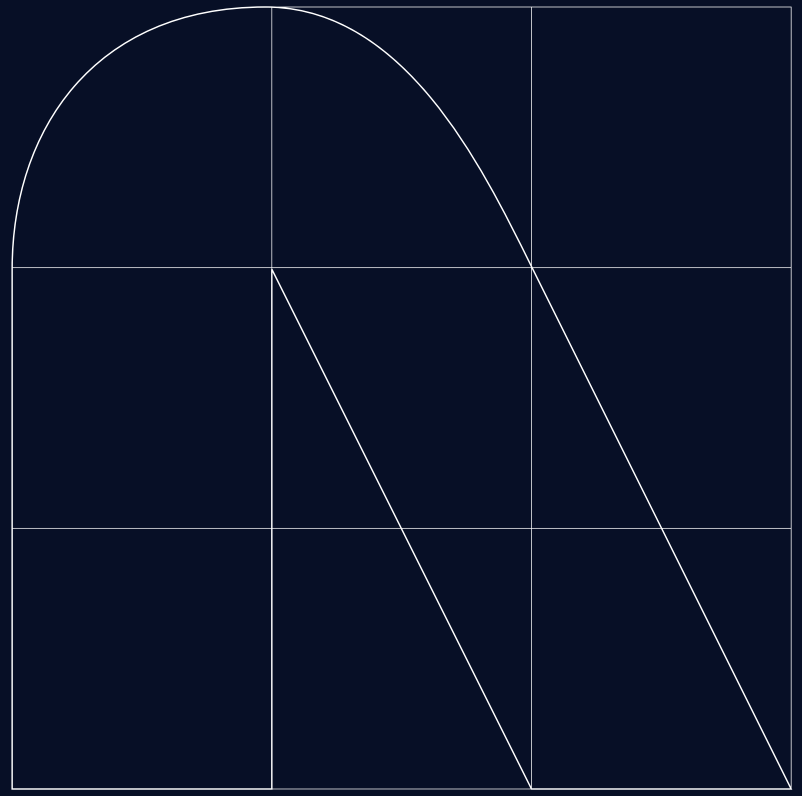
ビルドツールのバージョンすり合わせ

- JavaのビルドツールとしてMavenが利用される
- Mavenのバージョンと、使われているプラグイン(のバージョン)との兼ね合いなどでビルドが通らないこともある
  - 失敗の原因が分かりにくいので、Mavenの仕組みでビルド時にチェックすることもできる

maven-enforcer-pluginによるバージョン指定の例:

```
<plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-enforcer-plugin</artifactId>
          <version>3.3.0</version>
          <executions>
            <execution>
              <id>enforce-versions</id>
              <goals>
                <goal>enforce</goal>
              </goals>
              <configuration>
                <rules>
                  <requireMavenVersion>
                    <version>3.9.6</version>
                  </requireMavenVersion>
                  <requireJavaVersion>
                    <version>1.8</version>
                  </requireJavaVersion>
```

おわりに



まとめ

- Hadoopエコシステムでは、Javaを中心としつつ、様々なプログラミング言語が利用されている
  - 各製品の開発者のスキルや興味、技術的流行を反映した多様性
  - メンテナーはそれなりに幅広くキャッチアップする必要あり
- 成熟/普及したミドルウェアでは、コードの追加よりも、修正に労力が割かれる
  - 書きやすさよりも、読みやすさや、テスト/デバッグのしやすさが重要
- 互換性を維持する都合上、言語処理系のバージョンアップの恩恵をすぐ受けられない
  - バージョンアップ作業も苦労することが多い

AIについて

- AIのコード生成の普及が、プログラミング言語の選択に与える影響は?
  - 学習データが豊富という観点でポピュラーな言語が有利
    - 現状、Clojureでもアセンブラでも、難なくコードを生成/解説してくれるけど
    - 書く人による個性が出にくい方がよい? GoやPythonのような
  - コンパクトに書ける言語は、トークン効率がよいかもしれないが、そこまで重要ではない
  - 静的型付けがあった方が、コードの問題をコンパイル時に検出できてよい
  - 人間にとっての書きやすさ、書く速さの重要性が下がる
  - 生成されたコードを人間がレビューするための読みやすさの重要性は上がる
  - テストと修正のループを速くまわすために、ビルドの速さは重要

- 言語処理系のバージョンアップの追随が楽になるので、非互換な変更を入れるハードルが下がる?
  - ミドルウェアも、ユーザアプリケーションも、簡単に修正できてしまうなら
  - 検証/テストのコストは消えない

- AIとのやりとりに最適な言語は、自然言語でも、プログラミング言語でもない?


NTT DATA