ORACLE

次世代Java高速実行基盤 GraalVM

Jun Suzuki
Java Value Engineering
Java Global Business Unit
日本オラクル株式会社

Agenda

1. GraalVM Enterprise概要
2. JITコンパイラの最適化
3. Native Image
4. 多言語プログラミング対応
5. Demo

GraalVM Enterpriseとは

Oracle JDKをベースに開発した次世代JDKディストリビューション

高性能Just-in-Time(JIT)コンパイラ

- JVM内部アルゴリズムの最適化
- 既存Javaワークロードのパフォーマンス向上

Ahead-of-Time(AOT) コンパイラによる高速ネイティブ実行

- JVMに依存しないネイティブ実行ファイル（Native Image）を生成
- 高速起動と小さいフットプリントによるマイクロサービスへの親和性

多言語プログラミング対応

- Java, Scala, Kotlin, Gloovy、JavaScript, Python, Ruby, R, WebAssembly, C/C++
- 同一プログラム内、同一プロセス上の複数言語間の相互運用性

GraalVM Enterprise Editionの位置づけ

Oracle Java SE Universal Subscriptionの主要付加価値の一つ

- アプリケーションのモダナイゼーション
- クラウドネイティブ・アーキテクチャへのシフト

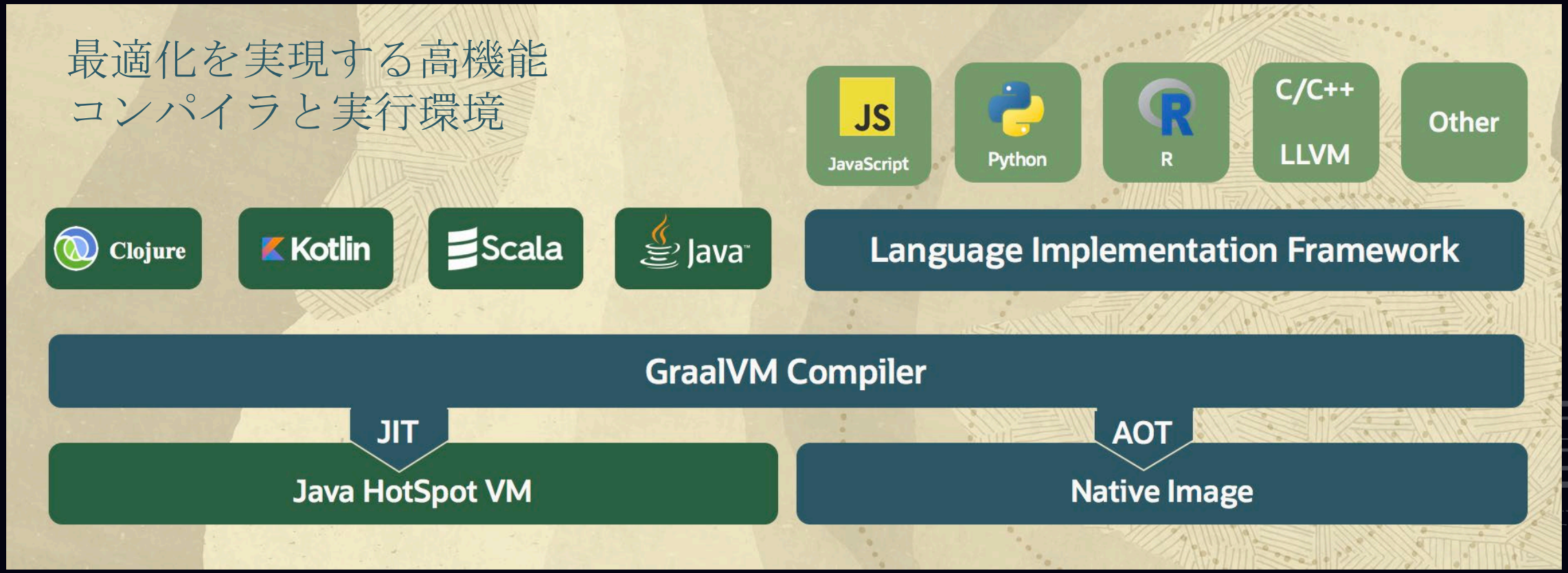
Oracle JDK	• Javaの主要主要コントリビュータ兼スチュワードによる信頼性と安全性を提供 • 長期間のサポートと、適時かつ予測可能なスケジュールのアップデートを保証
Global Oracle support	• 24 x 7 x 365 グローバルサポート • Javaエクスパートによるトリアージサポート
GraalVM Enterprise	• アプリケーション・パフォーマンス向上 • メモリーとCPUリソースの削減 • マイクロサービスとクラウド・ネイティブに最適
Enterprise Performance Pack	• JDK8のワークロードにJDK17のパフォーマンスを実現 • 既存JDK8ベースの限定的なインフラリソースで即効性のあるパフォーマンス向上を実現
Java Management Service	• 全Javaインスタンスの検出と使用状況のモニタリング • クラウドサービスによるJava運用保守作業の簡素化省力化 • 脆弱性リスクの識別とセキュリティー向上

Java SE Universal Subscriptionが提供するビジネスバリュー

- セキュリティーとコンプライアンス
- マネージメントとサポート
- モダナイゼーション
- パフォーマンスと効率性

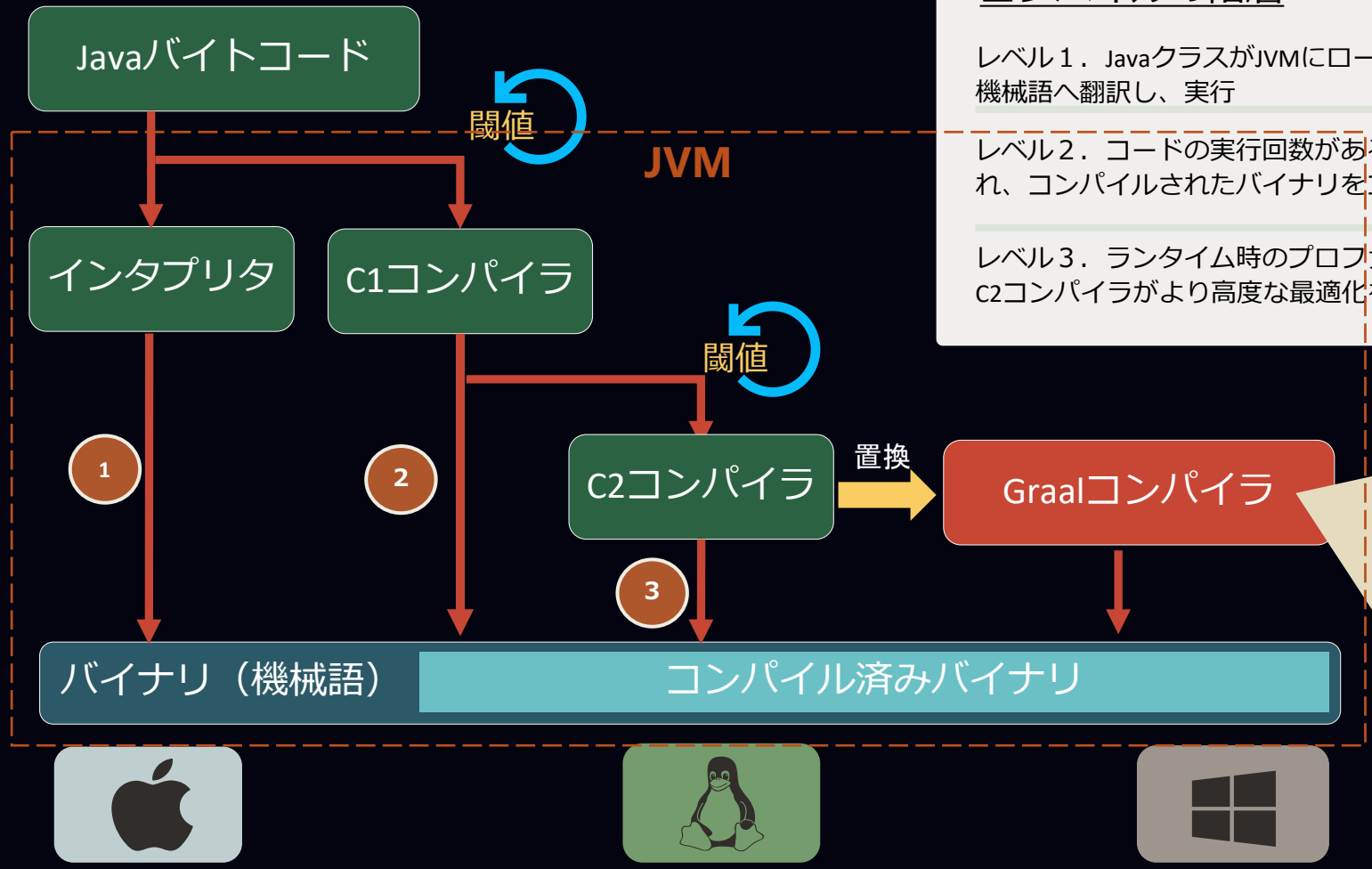
GraalVM Enterprise アーキテクチャ

100% Oracle JDK互換の次世代多言語プログラミング実行環境



JITコンパイラの最適化

高性能JITコンパイラ



コンパイルの階層

レベル１．JavaクラスがJVMにロードされた後、インタプリタがソース・コードを一行づつ機械語へ翻訳し、実行

レベル２．コードの実行回数がある閾値を超えた場合、C1コンパイラによる最適化が行われ、コンパイルされたバイナリをコードキャッシュに保存

レベル３．ランタイム時のプロファイリングにより呼び出し経路が解析、特定され、C2コンパイラがより高度な最適化を実施

◆**適切なインライン化**

getter/setterメソッドの代わりに、実際の変数を特定し代入することにより、メソッドコールの回数を減らし、オーバヘッドを軽減

◆**部分Escape解析**

変数の適用範囲を特定し、Synchronization作業を最低限に抑え、またローカル変数オブジェクトをJVMヒープからスタックに移動することで、JVMの使用メモリーを減らす

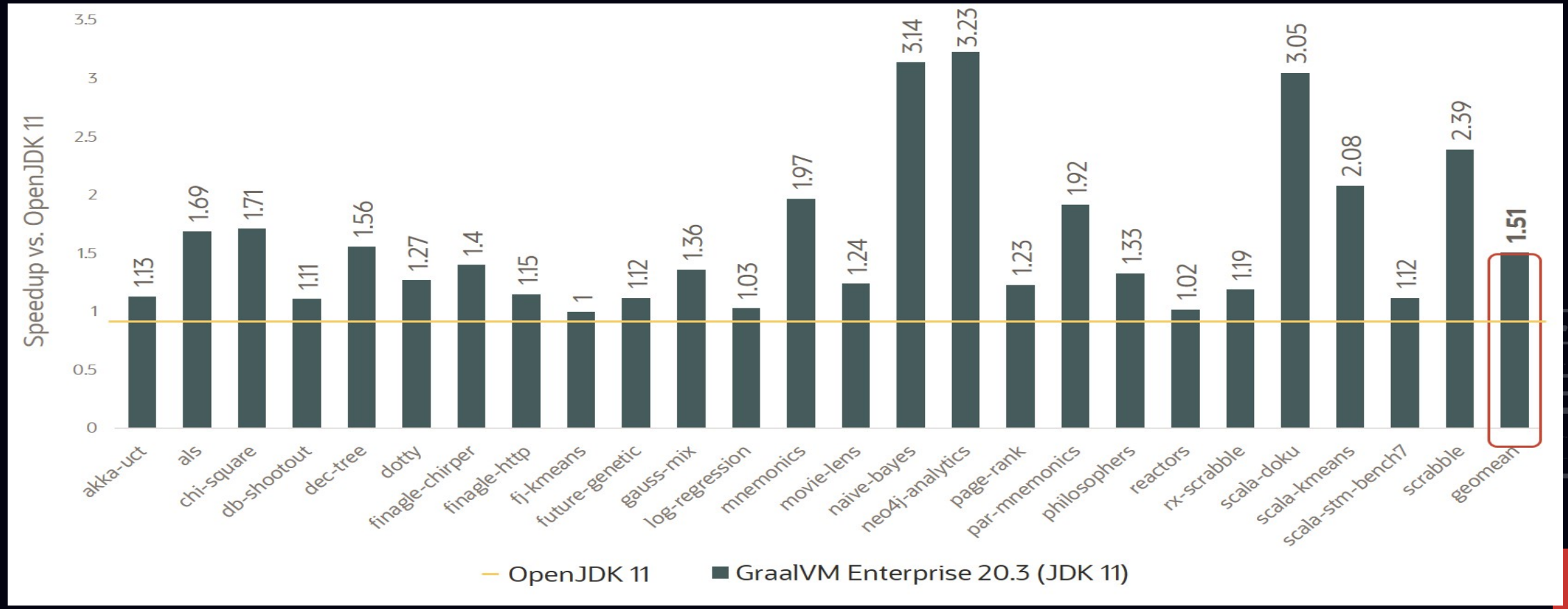
◆**DeOptimization**

最適化対象クラスを自動的に見直すことによりパフォーマンスを高め、また不要となるオブジェクトを特定してキャッシュから除去することによりメモリー負担を軽減

OpenJDKとのパフォーマンス比較

GraalVM EE 20.3のベンチマーク結果

- CPU、メモリー密集型のアプリケーションパターンに効果的
- より抽象化され、ストリームやラムダなどの最新のJava機能を使用するコードでは効果が顕著
- I/O、メモリー割当て、ガベージ・コレクションなどに収束するコードでは、改善の効果は小さい



<事例> Twitter

レスポンスの向上とレイテンシの低減

- 特徴と課題
  - トランザクション年間17%増加
  - 可用性の確保
  - インフラコストの削減
- ソリューション
  - GraalVM適用によるパフォーマンス向上
  - 全体アーキテクチャにマイクロサービスを取り入れ
- 結果
  - CPU使用率8～11%削減（$127 per CPU/year）
  - 物理サーバ5～12%削減
  - マルチ言語対応プラットフォームの実現

GraalVM採用のメリット

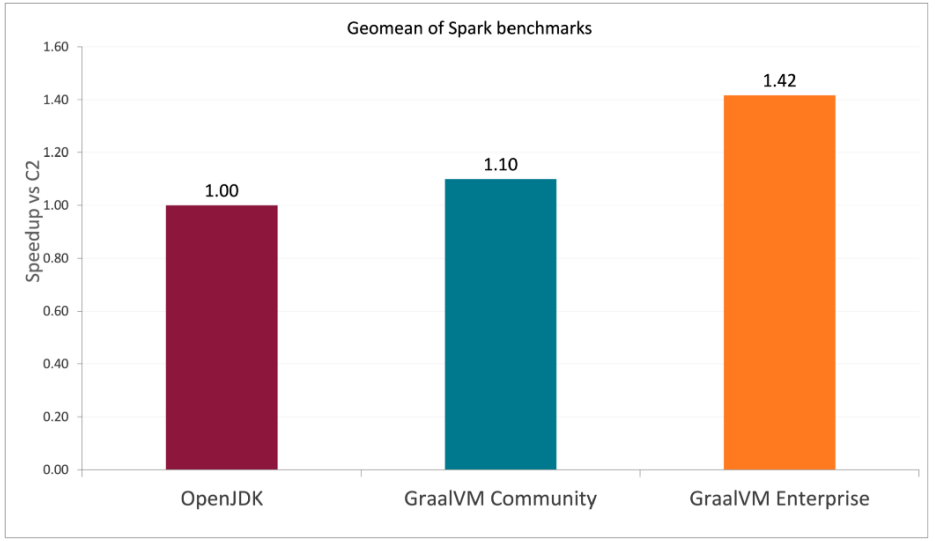
・インフラ・コスト削減（$127 per CPU/year)
・プラットフォームのFlexbility
・開発リソースの確保

https://www.youtube.com/watch?v=jLnedMcXYEs

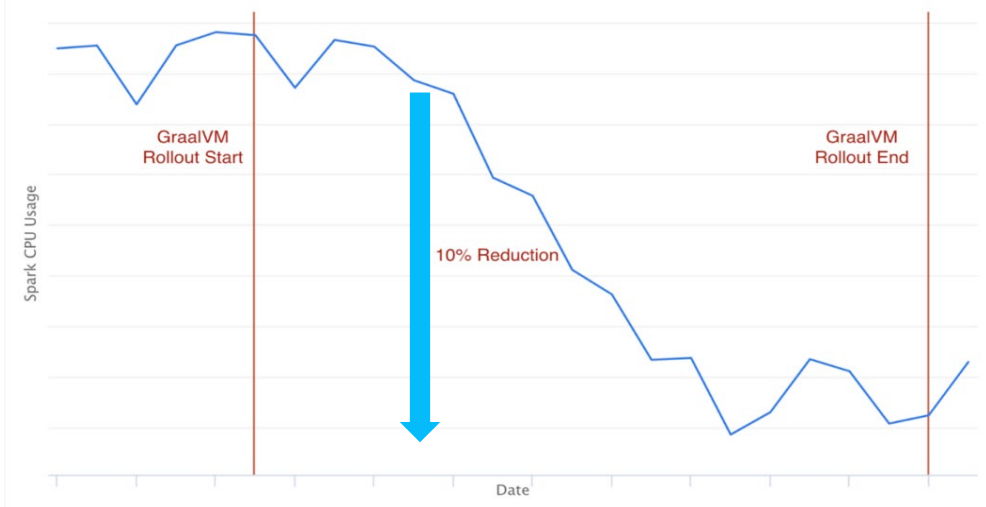
<事例> Facebook

大規模データ処理のパフォーマンス向上

- 特徴と課題
  - ユーザエクスペリエンスの改善
  - インフラコストの削減
  - Big Data処理基盤のパフォーマンス向上



- ソリューション
  - Big Data処理エンジンのJava環境をOpen JDKよりGraalVMに切り替え
    - Java環境変数の変更以外は、アプリの変更やチューニングは一切必要なし



- 効果
  - Apache SparkによるBig Data処理速度が**1.42倍**向上
  - CPU使用率が**10%**削減

https://blogs.oracle.com/graalvm/simplifying-the-cloud-native-journey-with-graalvm-and-helidon

<事例>Javaバッチ処理の高速化

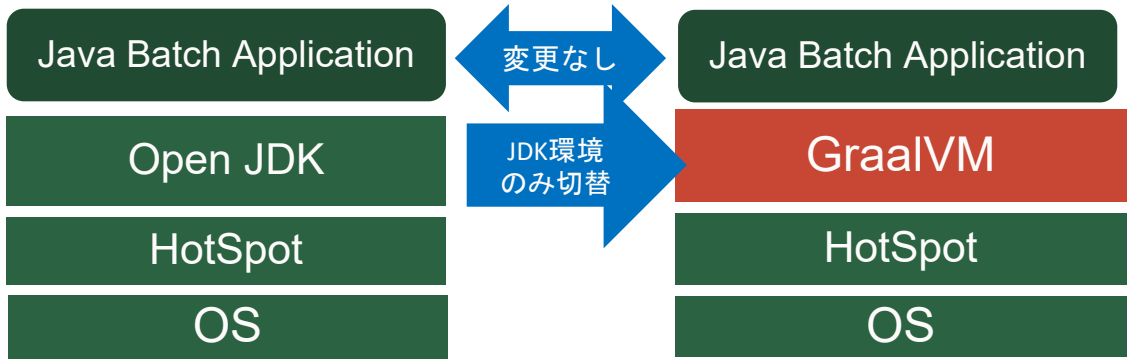
レガシーシステムのモダナイゼーション

背景と課題

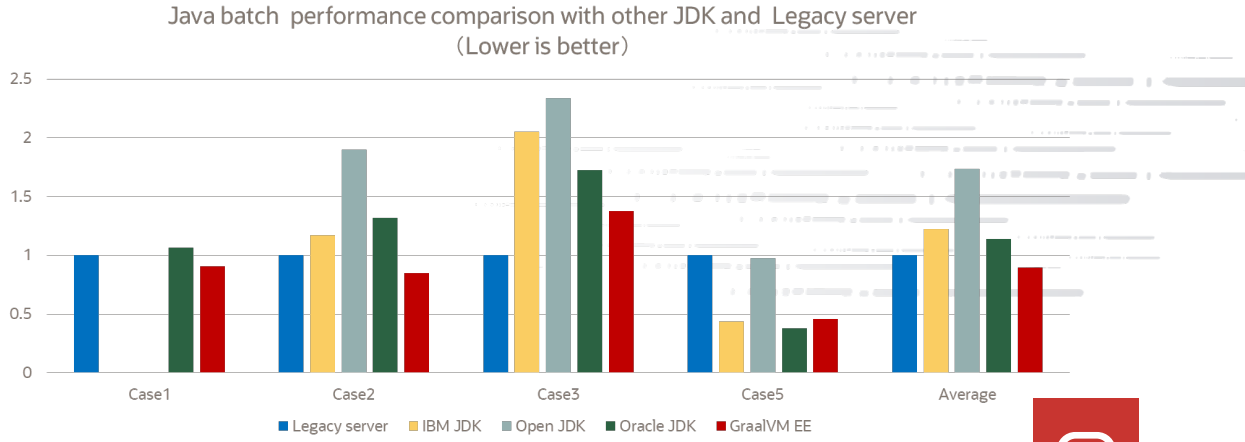
- 国内大手建築会社レガシー資産のインフラ保守維持コスト削減
- オフコン上数千本に及ぶバッチプログラムのJavaコンバージョンを実施
- オープン系サーバ上Javaバッチのパフォーマンス課題

ソリューション

- GraalVM EEを導入し、Open JDKや他社JDKとパフォーマンス比較を実施した上、最適なJavaランタイムを選択



オープン系サーバ上、オフコンと同等かそれ以上のパフォーマンス（10%～15%増）を実現



Native Image

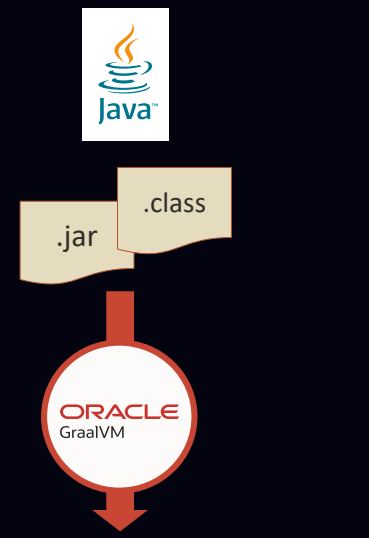
Native Imageの概要

- **Native Image**
  - Javaバイトコード（Javaクラス）をネイティブ実行ファイル(native image)に事前にコンパイルするテクノロジ
  - 負荷の重いコンパイル処理を実行時からビルド時に移動すること（AOT）により実行時の高速起動を実現
  - GraalVMのnative-imageツールより生成

```
$ native-image [options] HelloWorld
```

  - Maven, Gradleにnative image生成、テスト用のプラグインも提供（Native Build Tool）

```
$ mvn -Pnative package
```



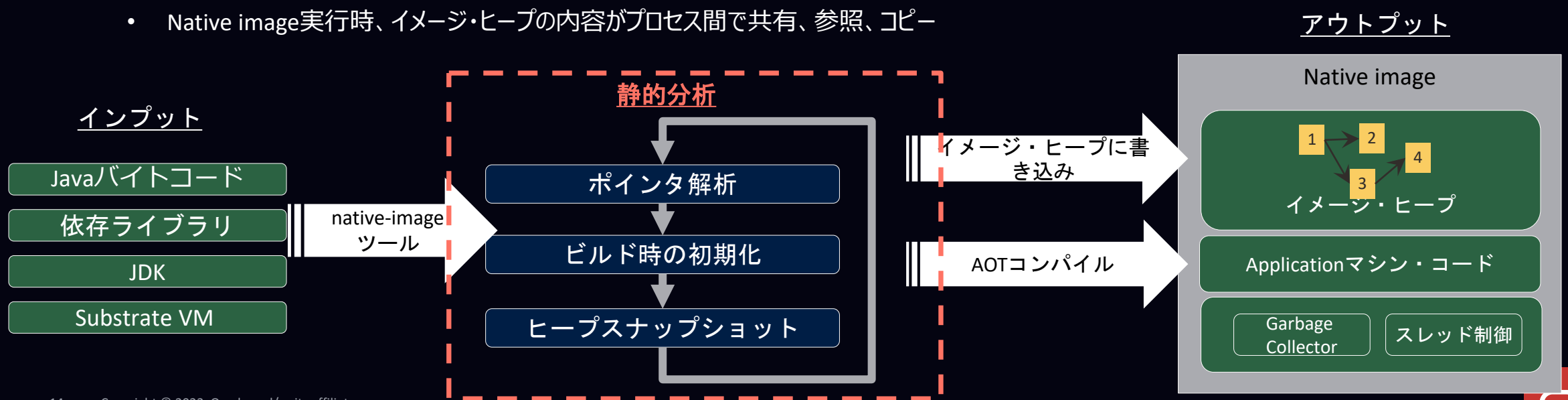
- **Native Imageの構成**
  - イメージ・ヒープ（静的分析による初期化済みのJavaオブジェクト）
  - アプリケーションのマシン・コード(実行に必要なJDK標準ライブラリもリンクされる)
  - ランタイム（Substrate VMと呼ばれ、Garbage Collector、スレッド制御機能を含む)

- **Native Imageの利点**
  - JVMに依存せず自己完結型実行ファイルのため、メモリー消費は低い
  - ミリ秒単位で起動
  - ウォームアップなしで即座にピーク・パフォーマンスを実現
  - 軽量のコンテナ・イメージにパッケージ化して、迅速かつ効率的にクラウドへデプロイ可能
  - 攻撃対象領域が縮小

Native Imageのビルド

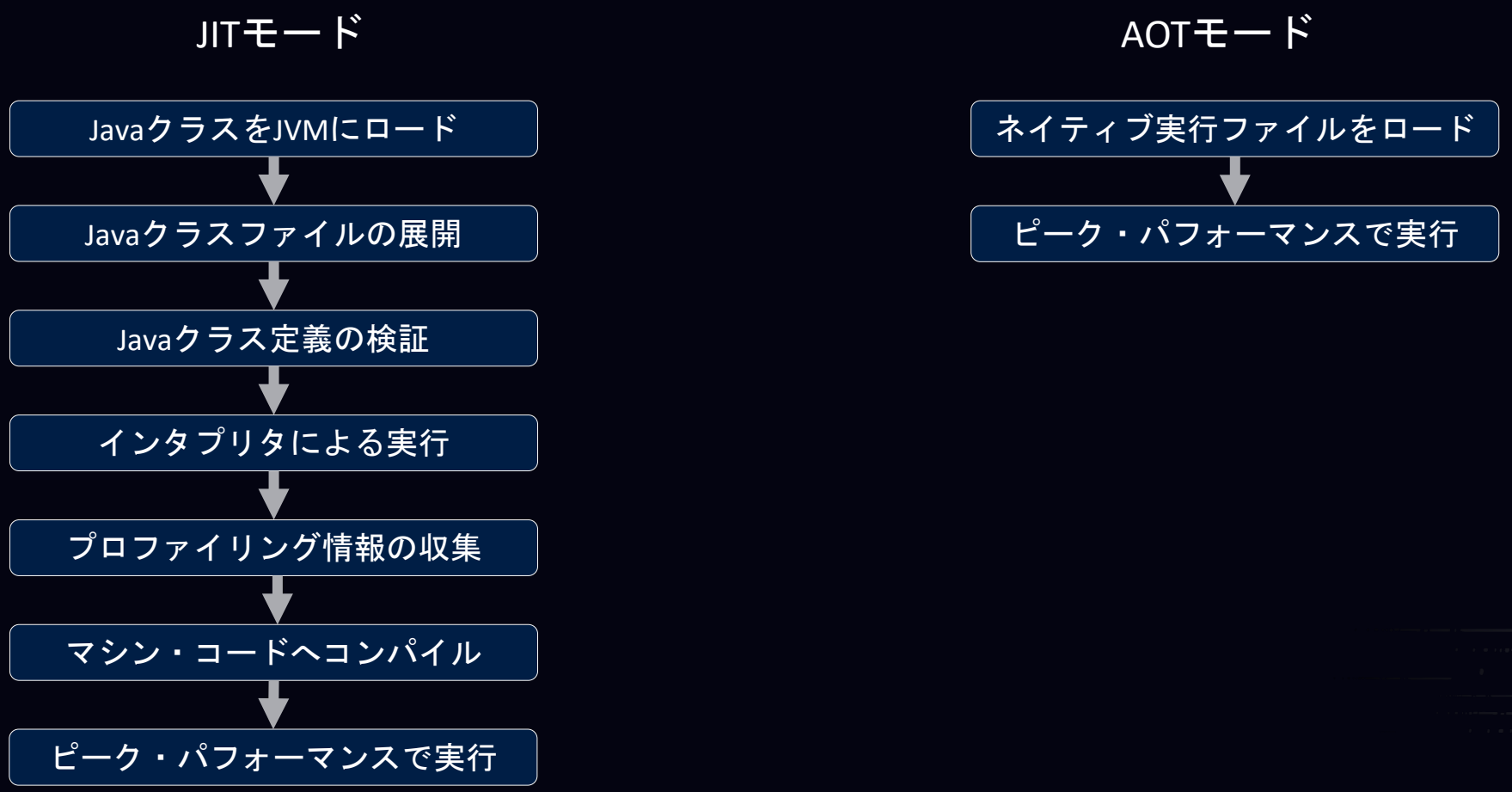
native-imageツールによる実行ファイル(native image)のビルド

- 静的分析
  - アプリケーションで使用されるプログラム要素(クラス、メソッド、およびフィールド)を決定するプロセス
    - ポインタ解析（Points-to Analysis）：実行時アクセス可能なクラス、メソッド、フィールドを判定
    - ビルド時の初期化(Initializations at build time)：ビルド時クラスと静的フィールドを初期化
    - ヒープスナップショット(Heap snapshotting)：実行時必要なオブジェクトを事前にヒープに割り当て（スナップショット）
- イメージ・ヒープ
  - ヒープスナップショットによって確定したオブジェクトがnative imageのデータ領域に書き込まれる
    - 初期化済みクラス、static フィールド、Enum定数、java.lang.Classオブジェクト
    - Native image実行時、イメージ・ヒープの内容がプロセス間で共有、参照、コピー



Native Imageの実行ステップ

JITモード vs AOTモード

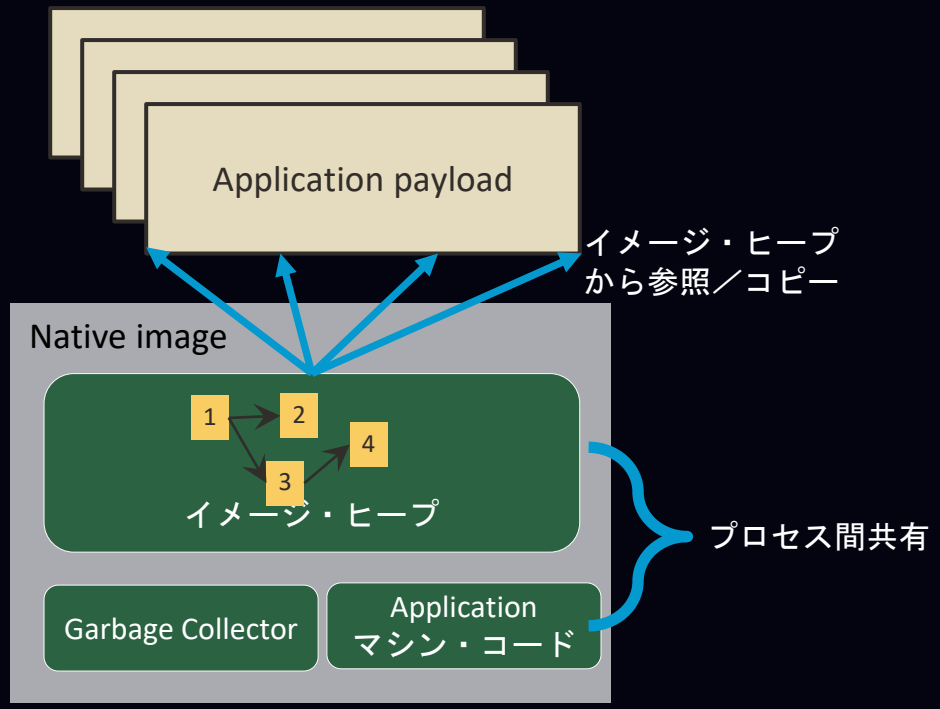


Native Imageの実行とメモリー消費

Native Imageの高速実行と低いフットプリント

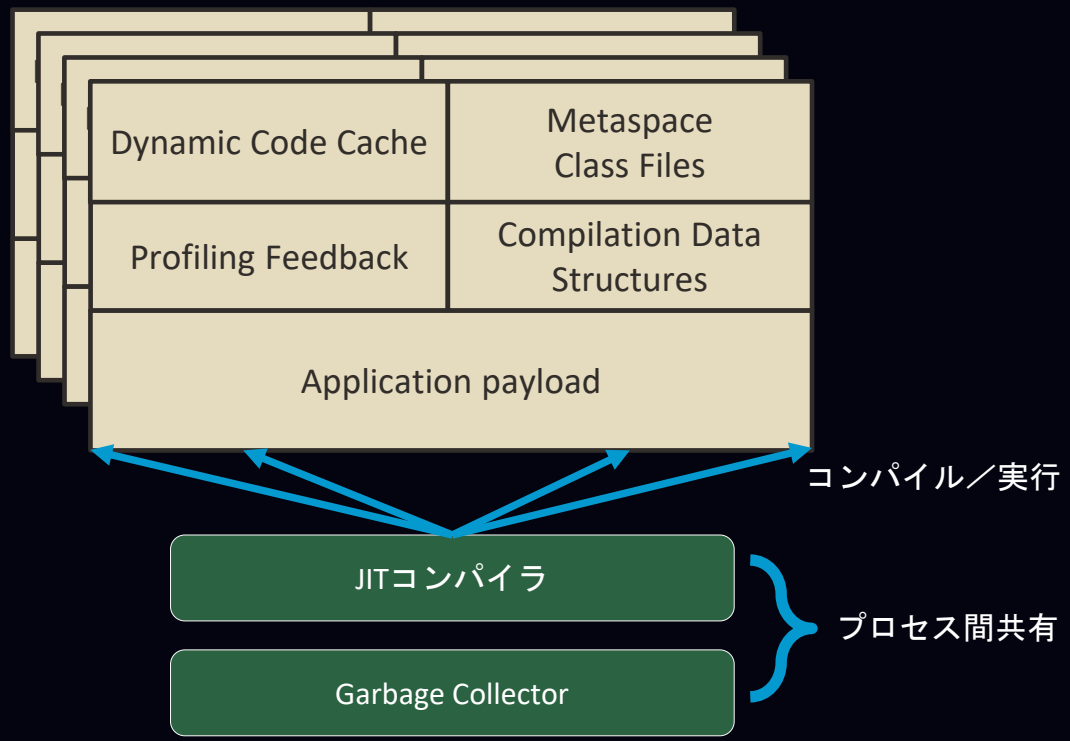
AOTモード

- 各プロセス（native imageの実行）から共通イメージ・ヒープから初期化済みのオブジェクトを参照／コピー



JITモード

- プロセス起動ごとにJITコンパイル実施に伴う多くのメモリーを必要とする



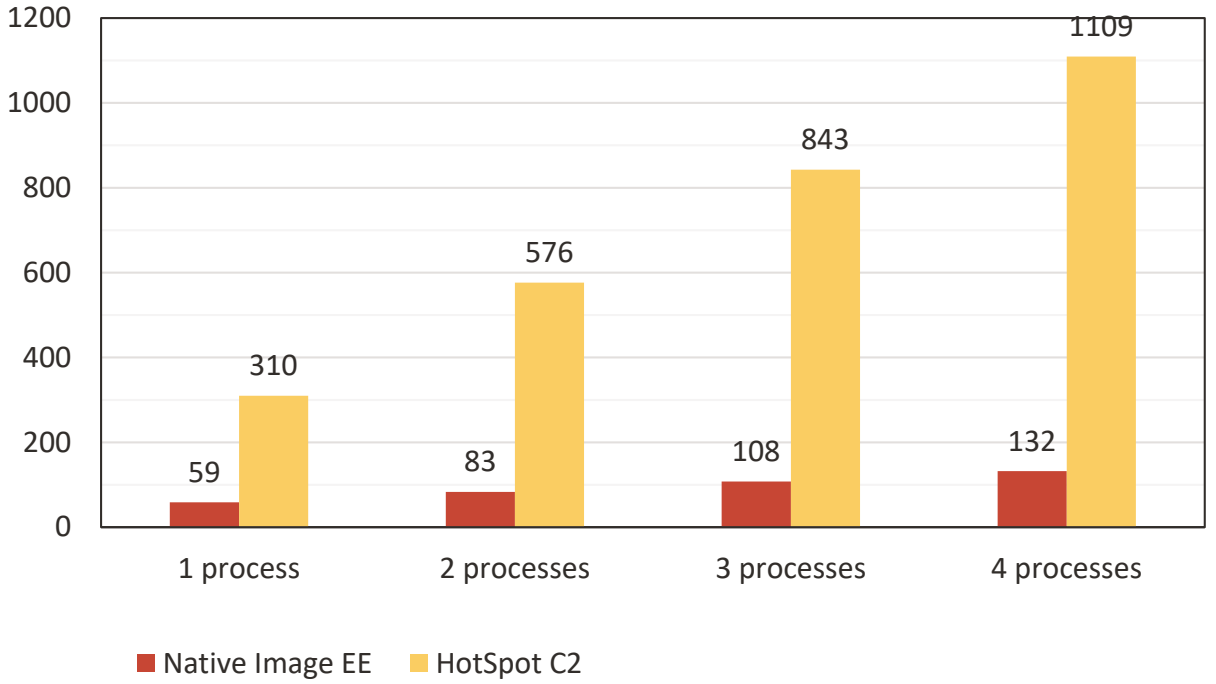
マイクロサービスの水平分散時のメモリー消費

AOT vs JIT

- **JVMをスケールアウトする際のメモリー消費パターン**
  - Quarkus Apache Tika ODT in a “tiny” configuration and with the serial GC
  - 1 CPU core per process, -Xms32m -Xmx128m)
  - JDK 11

- Java HotSpot VM
  - **4 VM instances = 4 times the memory**

- Native Image
  - **4 VM instances = 2 times the memory**
  - Image heap shared between processes
  - Machine code shared between processes

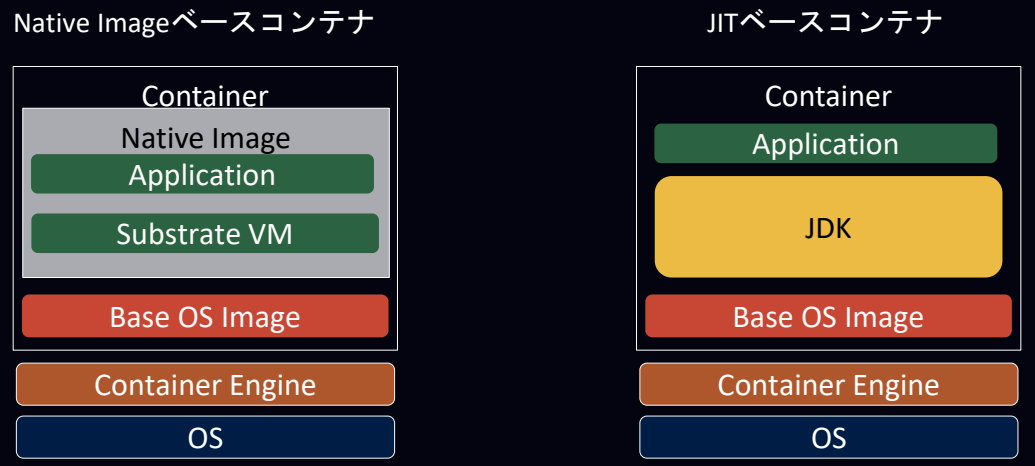


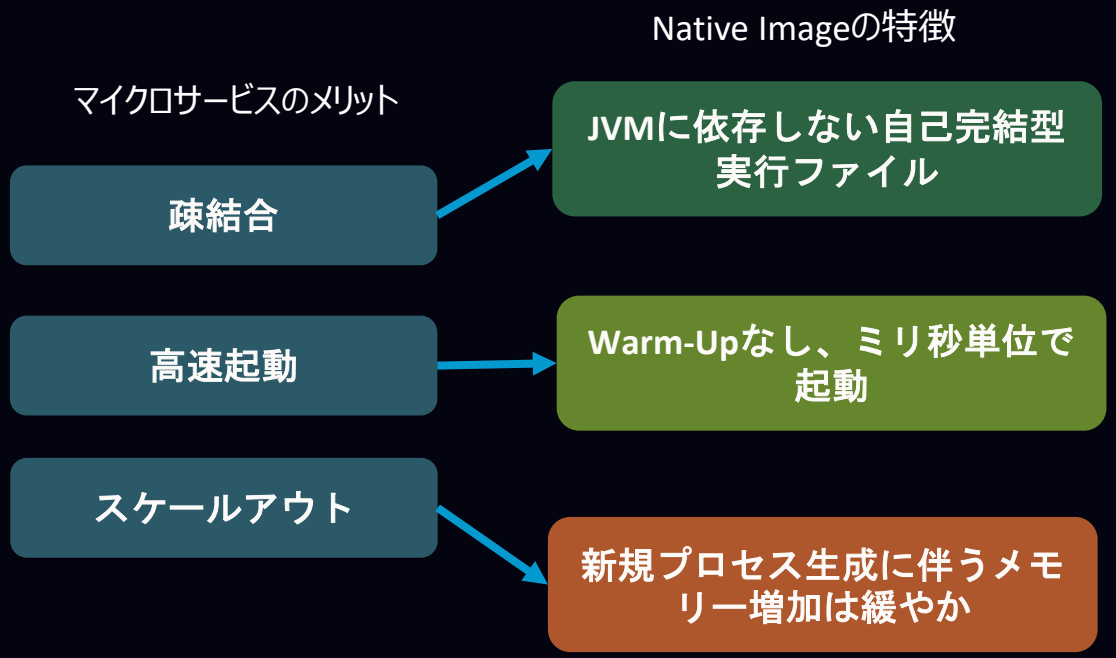
出典：https://www.youtube.com/watch?v=mhmqomex1zk

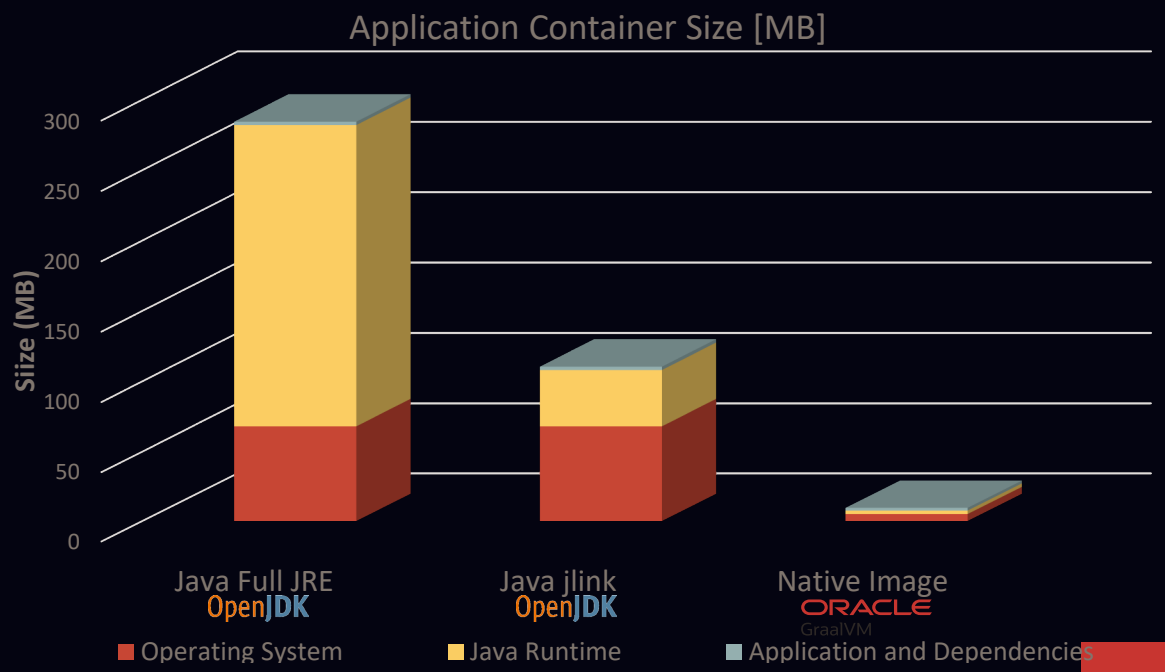
Native Imageとコンテナ

Native Imageとコンテナ化マイクロサービス

- Native Imageとコンテナ化マイクロサービスの相性が良い
  - コンテナサイズがよりコンパクト
  - 高速起動
  - スケールアウト



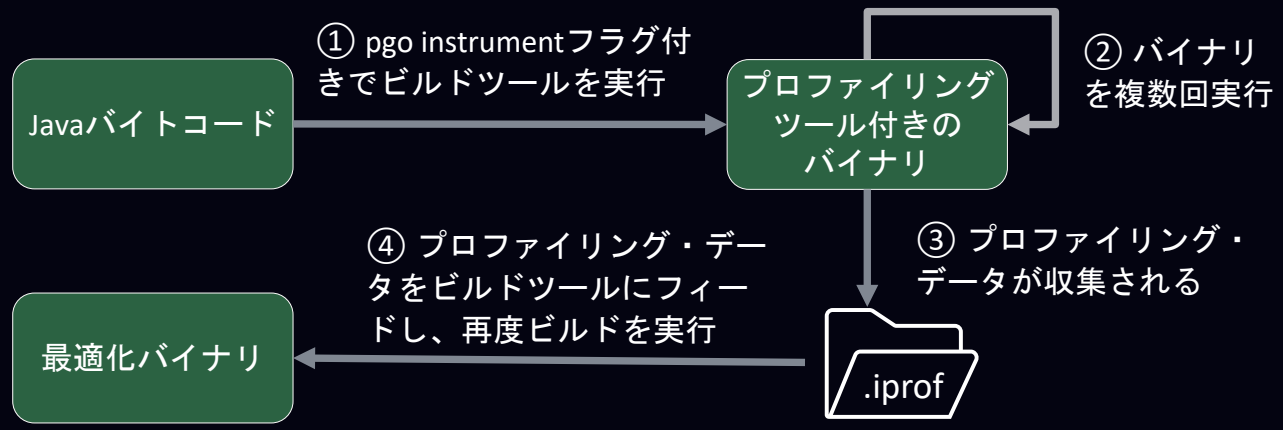




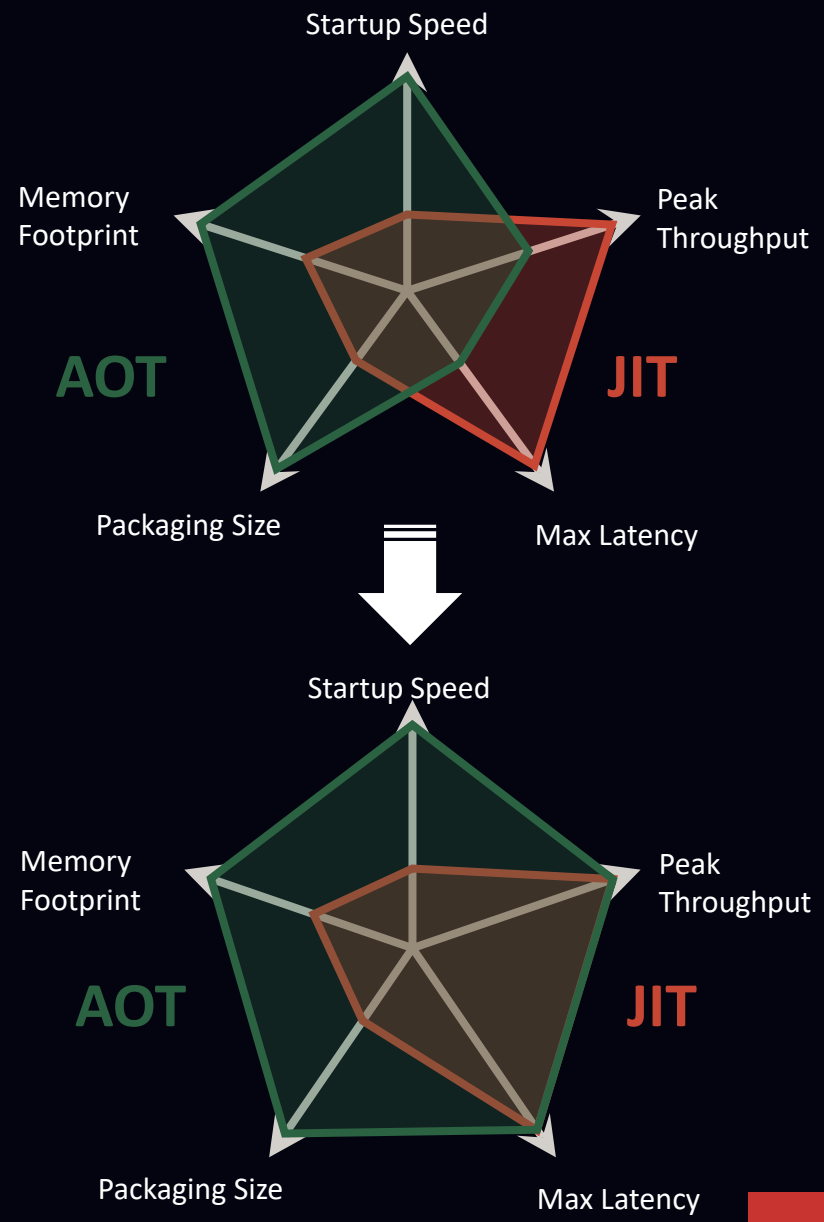
Native Imageの考慮点（１）

プロファイルに基づくNative Imageの最適化

- Native Imageのスループットを向上させる方法
  - プロファイルに基づく最適化ツールを利用してNative Imageのスループットを最大化
    - Native ImageはJITのように実行しながら最適化を行うことはできないため、ピーク時スループットはJITモード実行より低い



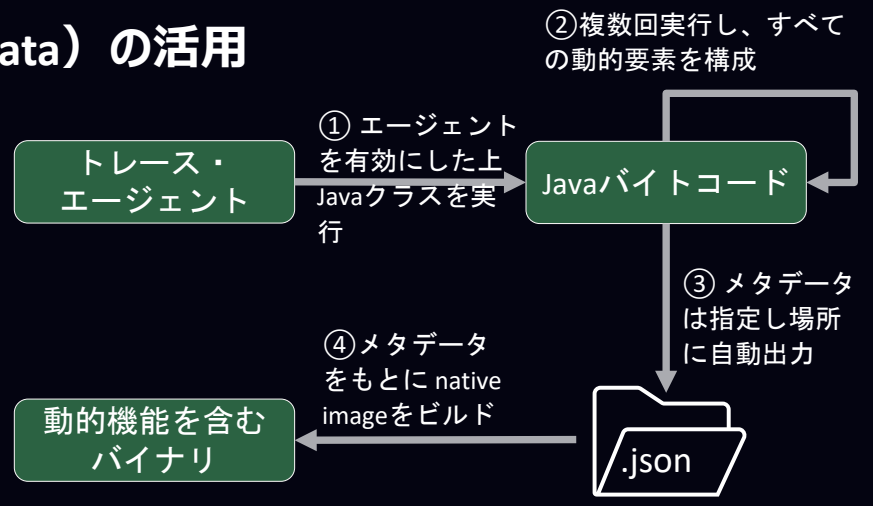
① $JAVA_HOME/bin/native-image --pgo-instrument myClass
② ./myNativeImage
③デフォルトではdefault.iprofが生成される
④$JAVA_HOME/bin/native-image --pgo=default.iprof myClass



Native Imageの考慮点（２）

Java動的機能を利用時の対応：到達可能性メタデータ（Reachability Metadata）の活用

- Native Imageの制約：Closed-world仮説
  - 通常の静的分析は、実行時必要なすべての要素が到達可能であることを前提
  - 最終的にビルドされたNative Imageは到達可能な要素のみ含む
  - Native Image実行時、クラスロードによる新しい要素を追加できない
    - Java Reflection、JNI(Java Native Interface)、リソースおよびリソース・バンドル、動的プロキシ、シリアライズ、事前定義済みクラス
    - 上記動的機能を利用したnative imageを実行する場合ランタイム・エラーが発生



トレース・エージェントツールによるメタデータの自動収集

- 対応方法：到達可能性メタデータを提供し、動的機能を静的分析に知らせる
  - トレース・エージェントツールを使用してメタデータを自動生成
    - エージェントはjava実行中に動的機能の使用をすべて追跡し、メタデータを自動生成

```
java -agentlib:native-image-agent=config-output-dir=META-INF/native-image
ReflectionExample StringReverser reverse "hello"
```

各Java動的機能に対応したメタデータJSONファイル

```
META-INF
└── native-image
    ├── jni-config.json
    ├── proxy-config.json
    ├── reflect-config.json
    └── resource-config.json
```

Reflect-config.jsonの例

- native-imageツールはMETA-INF/native-image配下のメタデータを認識

```
{
  "name":"StringReverser",
  "methods":[{"name":“reverse","parameterTypes":["java.lang.String"] }]
}
```

Native imageに含める要素を通知するエントリで構成される

Native Imageの考慮点（３）

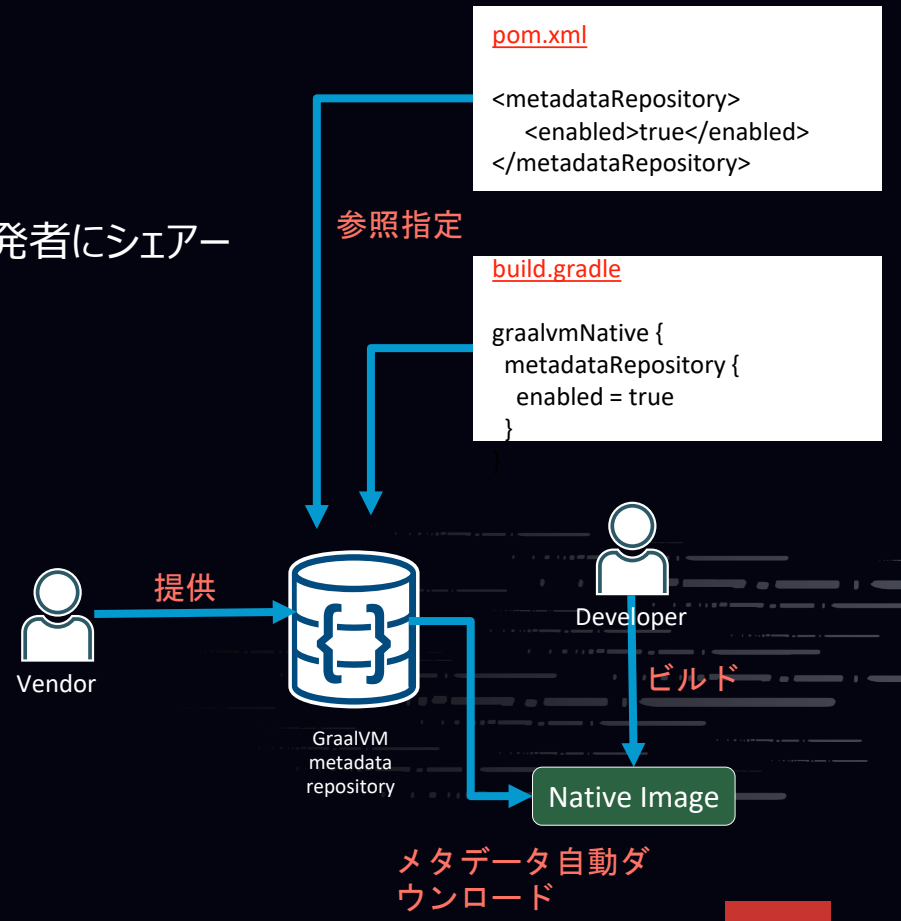
3rd パーティ製ライブラリやフレームワーク利用時の対応： GraalVM到達可能性メタデータ・リポジトリの活用

- 3rd パーティ製ライブラリを使用した場合のメタデータ問題
  - ブラックボックス化のためメータデータ情報の提供やメインテナンスは困難

- GraalVM到達可能性メタデータ・リポジトリ
  - GraalVM, Micronaut, Spring Boot, Quarkusチームの協業によって実現
  - 3rdパーティ製 ライブラリとフレームワークのNative Imageメタデータを蓄積し、全開発者にシェアー
  - Native Imageビルド時メタデータは自動ダウンロード

- リポジトリに対して、新規メタデータを追加可能
  - メタデータをリポジトリに簡単に提供するアプローチ／ツールが用意されている

Native imageから3rdパーティライブラリ、フレームワークの利用が飛躍的便利に！

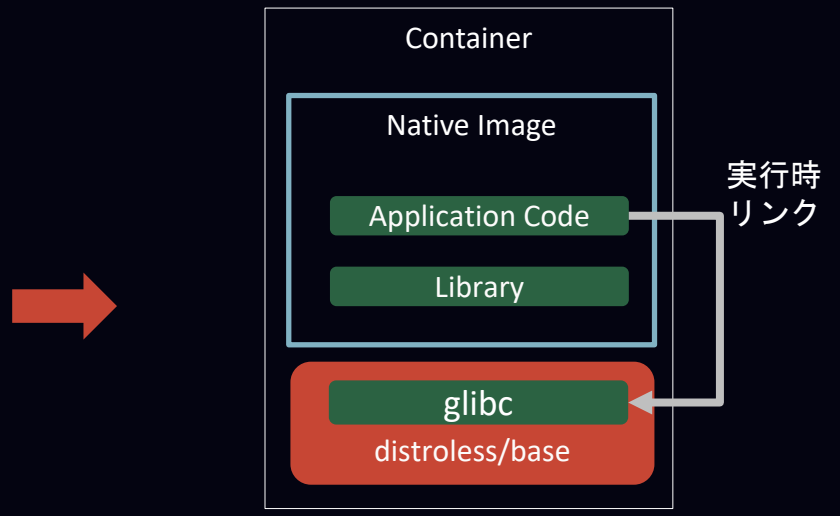


https://graalvm.github.io/native-build-tools/latest/maven-plugin.html#metadata-support
https://graalvm.github.io/native-build-tools/latest/gradle-plugin.html#metadata-support

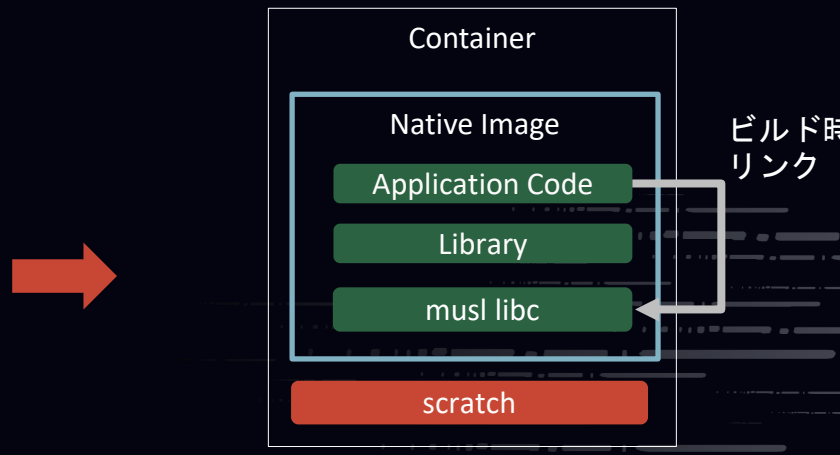
Native Imageの考慮点（4）

静的/ほぼ静的Native Imageを利用してよりコンパクトなコンテナを生成

- ほぼ静的Native Image
  - Native Imageビルドオプション（Mostly Static）
    - C標準ライブラリ(libc)以外のすべての依存ライブラリが静的にリンクされる
    - 実行時オーバーヘッドが小さい
    - native-image -H:+StaticExecutableWithDynamicLibC
  - コンテナベースOSイメージ：Distroless/base
    - 最小限のライブラリを含む軽量Baseイメージ(glibcを含む)

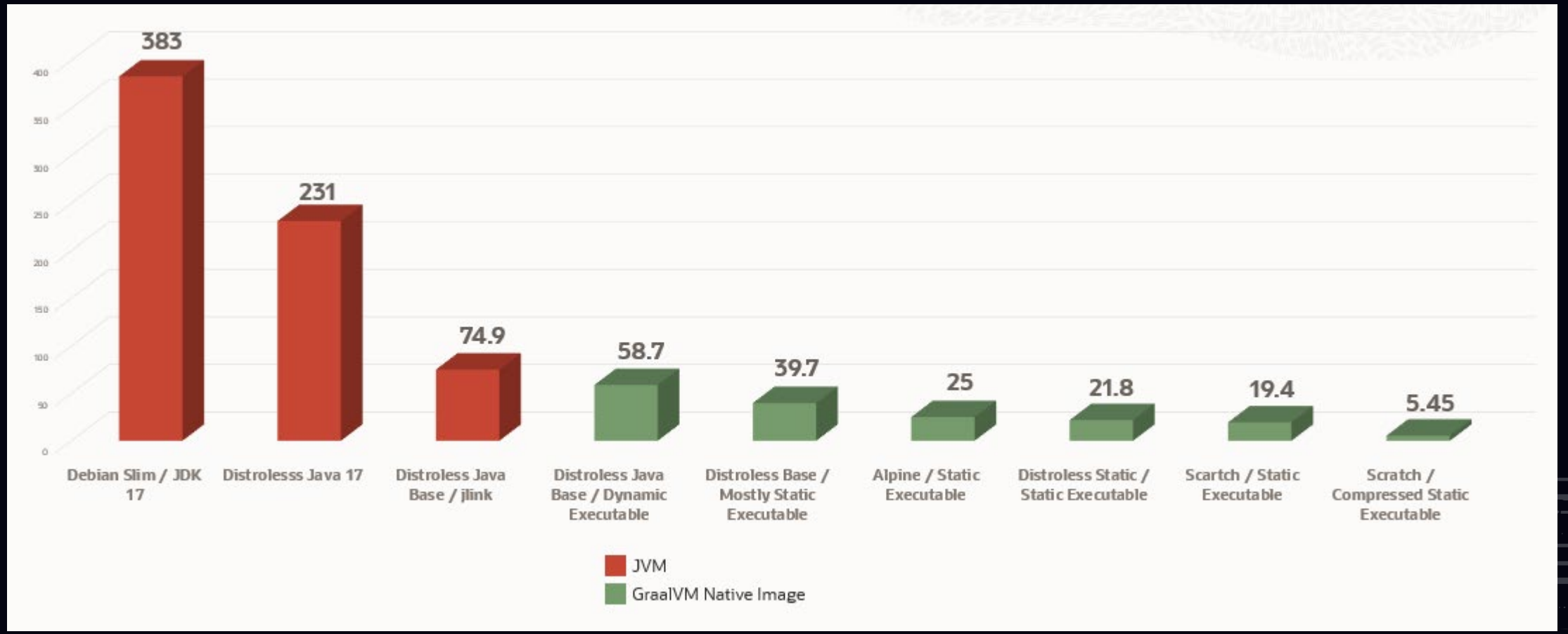


- 静的Native Image
  - Native Imageビルドオプション（Full Static）
    - すべての依存ライブラリが静的にリンクされる
    - 軽量、高速かつ単純なlibc実装であるmusl-libcを使用
    - 実行時オーバーヘッドがもっとも小さい
    - 現時点でJava11のみサポート
    - native-image --static –libc=musl
  - コンテナベースOSイメージ：Scratch
    - 最軽量Baseイメージ



コンテナイメージサイズ

異なるベースイメージでのコンテナサイズ比較



出典：https://www.youtube.com/watch?v=6wYrAtngIVo

Native Imageの考慮点（5）

Quick Buildモード（22.x～）

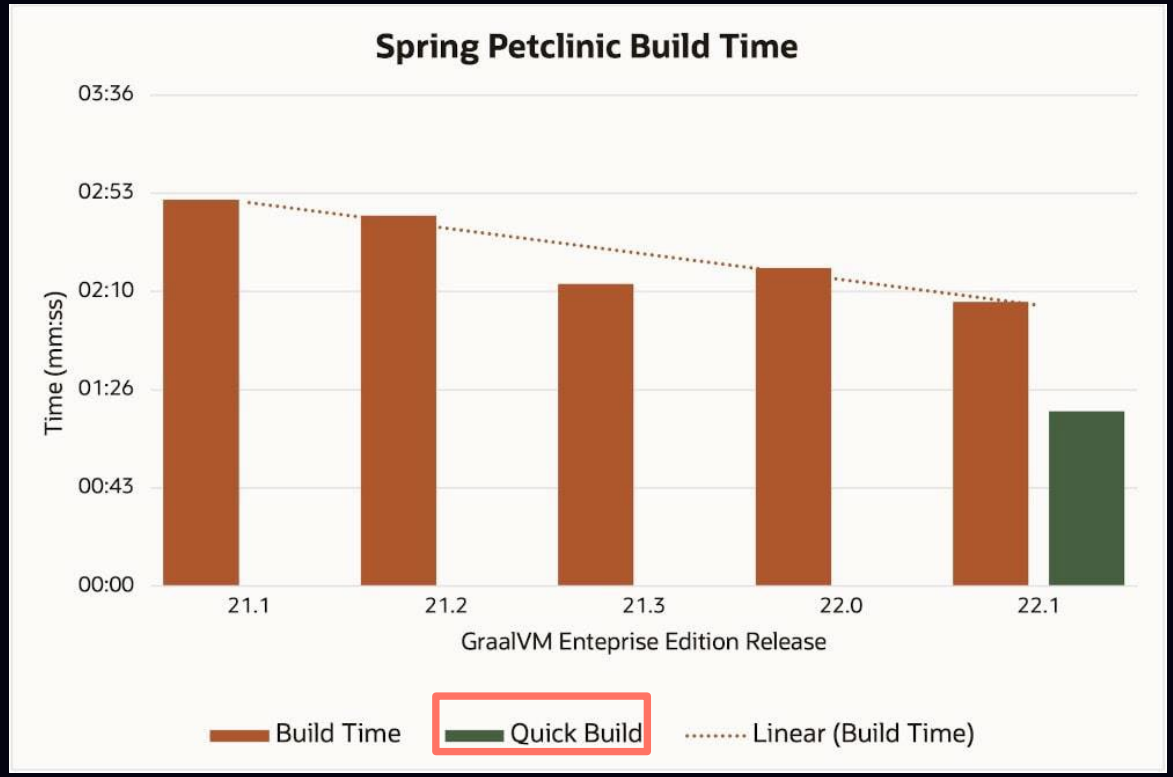
- 開発フェーズでQuick Buildモードを有効にしてNative Imageをビルド
  - Native Imageのビルド時間を短縮
  - 本番環境デプロイ時モードをオフに

- Native Imageビルドプラグイン設定
  - Mavenプロジェクト(pom.xml)

```
<buildArgs>
   <quickBuild>true</quickBuild>
</buildArgs>
```

  - Gradleプロジェクト(build.gradle)

```
graalvmNative {
  binaries {
    main {
      quickBuild = false
    }
  }
}
```

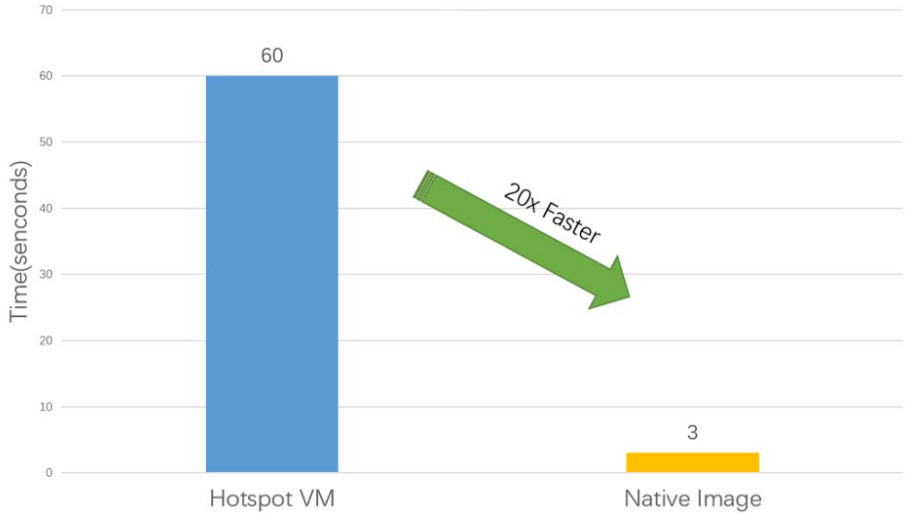


<事例>Alibaba

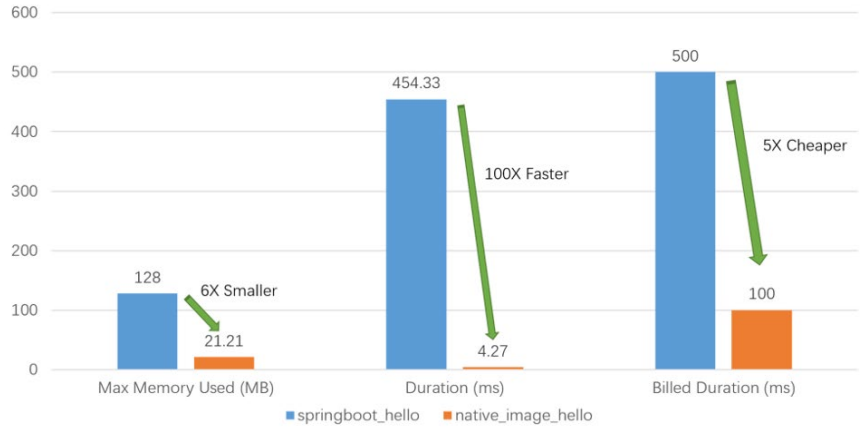
Java によるマイクロサービス構築

- 特徴と課題
  - 大規模オンライン・トランザクション
  - Spring Bootアプリの水平分散スケーラビリティ
  - 新規インスタンスのスタートアップ時間
- ソリューション
  - 既存アプリをNative Imageに変換
- 効果
  - 起動時間が 100 分の 1 に短縮
  - メモリ使用量の 83%削減
  - ガベージコレクション一時停止時間の 93%削減
  - 顧客コストの 80%削減

https://medium.com/graalvm/static-compilation-of-java-applications-at-alibaba-at-scale-2944163c92e



Application Startup Time Comparison

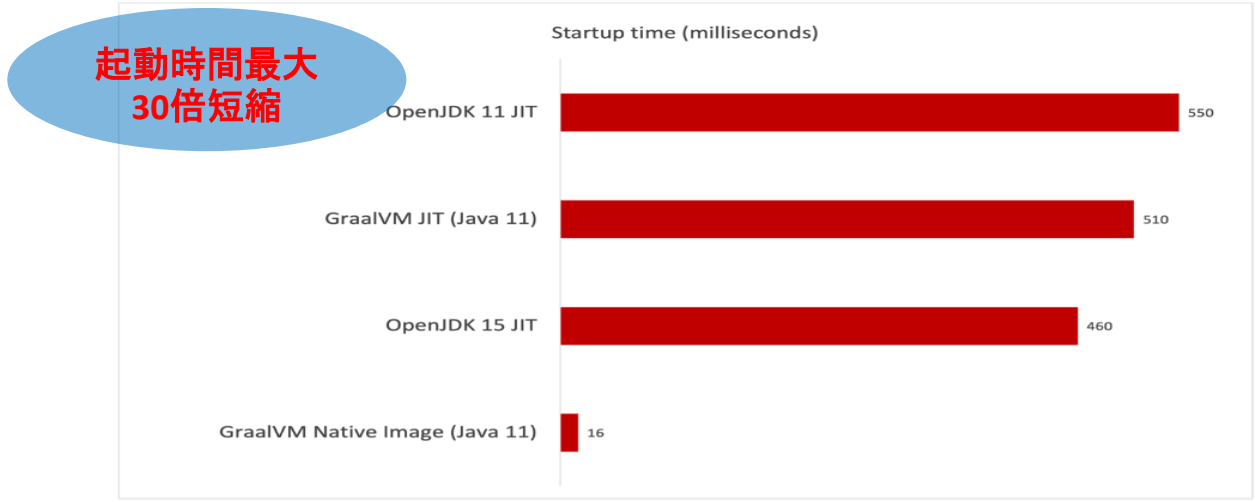


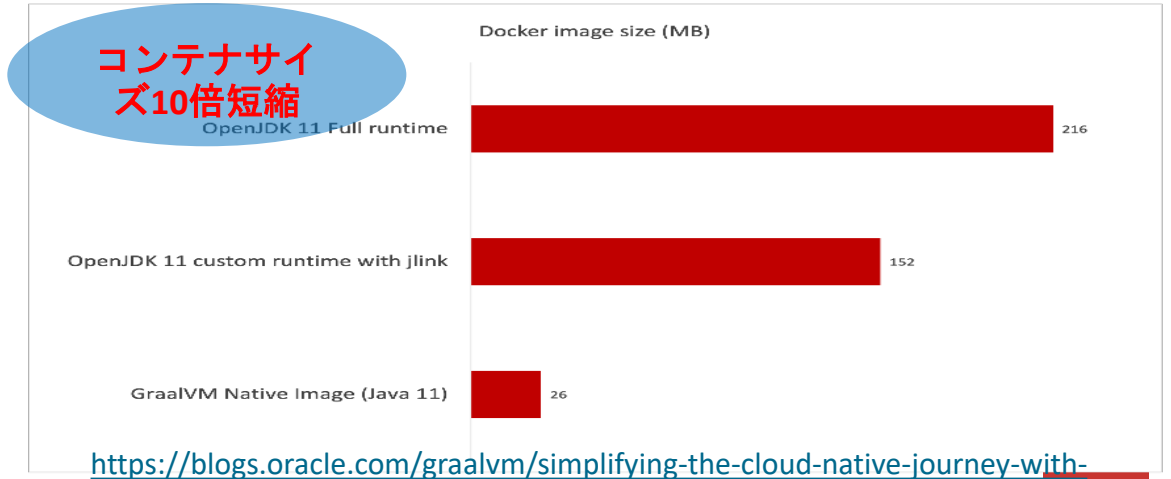
Traditional Java Function vs Statically Compiled Function

<事例>Standard Chartered Bank

マイクロサービスの高速起動と配布

- 特徴
  - 全世界1026の支店と86000の従業員を有する商業銀行
  - 既存JBossアプリからKubernetesへ移行
  - Java, Python, Spring Boot
- 課題
  - 既存Javaアプリスタート時間の短縮
  - クラウドへの迅速なデプロイ（小さいフットプリント）
  - 処理ピーク時自在なスケールアップ
- ソリューション
  - 高速ランタイムGraalVMと軽量フレームワークHelidonを導入
- 効果
  - Native Image化によるJavaコールドスタート遅延の解消
  - Native Imageをベースに生成するコンテナの軽量化を実現





https://blogs.oracle.com/graalvm/simplifying-the-cloud-native-journey-with-graalvm-and-helidon

<事例>Enterprise JavaのMicroservices化

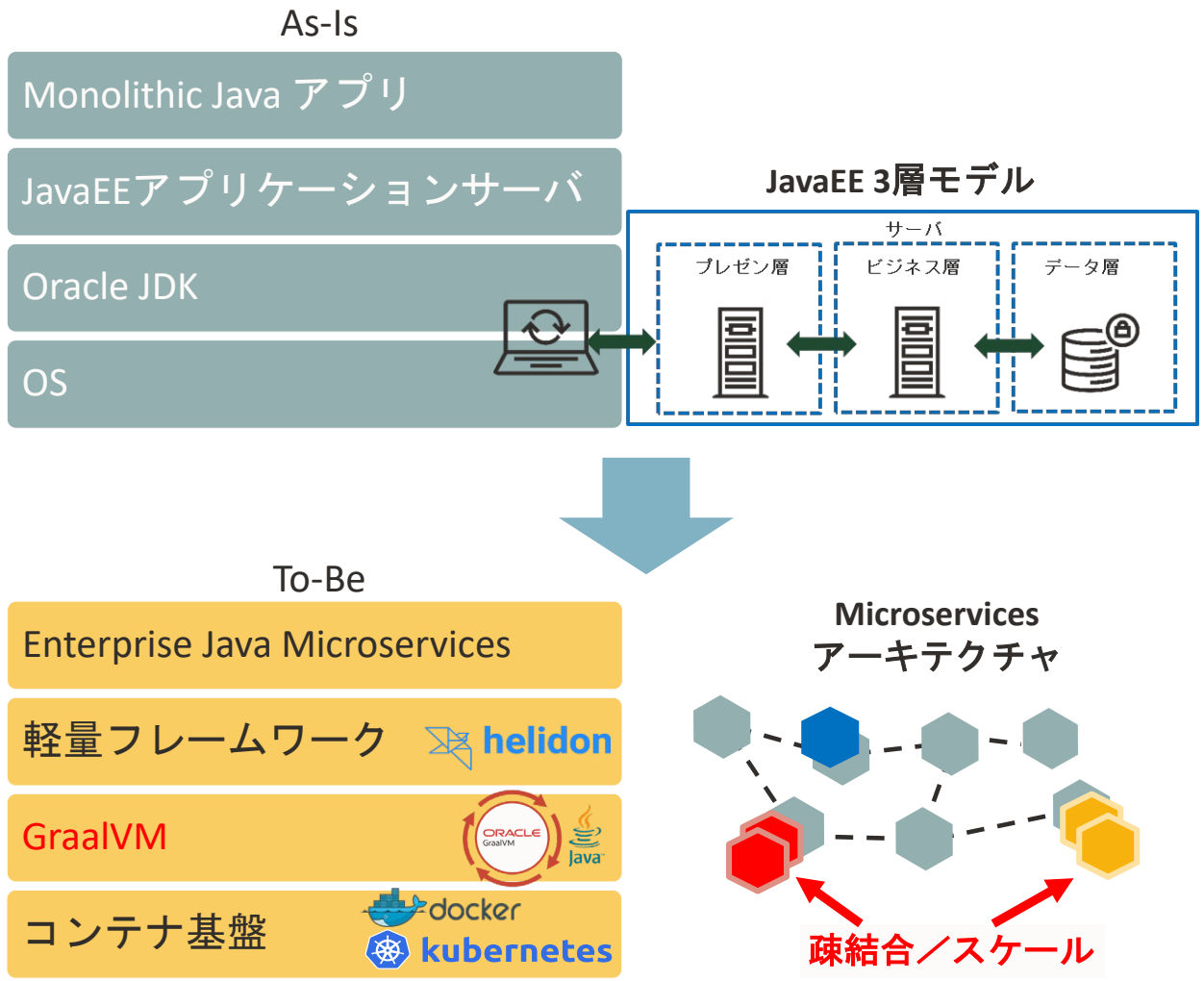
モノリシックからマイクロサービスへ転換

背景と課題

- 国内大手カード会社が既存モノリシックのアプリケーションからMicroservicesアーキテクチャへ転換
- Microprofile準拠の軽量フレームワークとGraalVMを組み合わせ
- Security, Agility重視

ソリューション

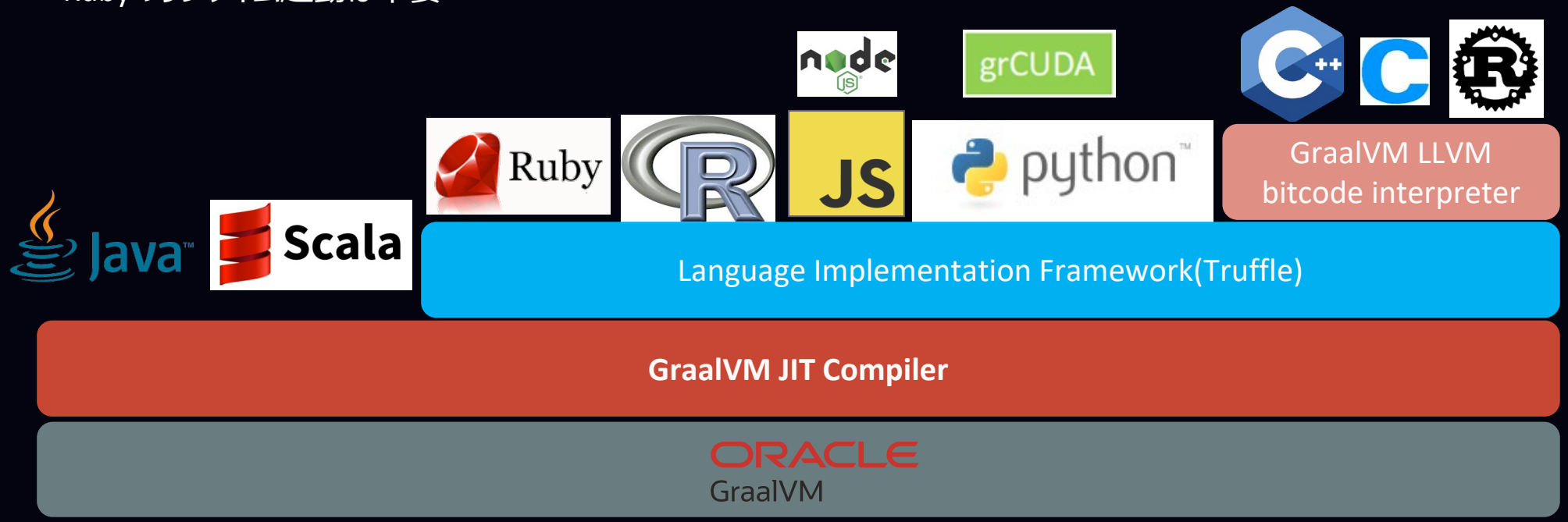
- Microprofile準拠軽量フレームワークHelidonとGraalVM EEを組み合わせ
- Kubernatesによるコンテナ管理



Polyglot

多言語プログラミング対応

- Java以外の言語のインタープリターを実装するためのフレームワークを提供
  - フレームワーク経由でGraalVMのJITやGCを利用可能
- JavaScript, R, Ruby, Python, LLVM, Web Assemblyなど各言語の実行環境を提供
  - １つのアプリケーションの中、複数言語の相互運用性を提供
  - 例：JavaScript のコードからRubyのメソッドを呼び出せる
    - Rubyのランタイム起動は不要



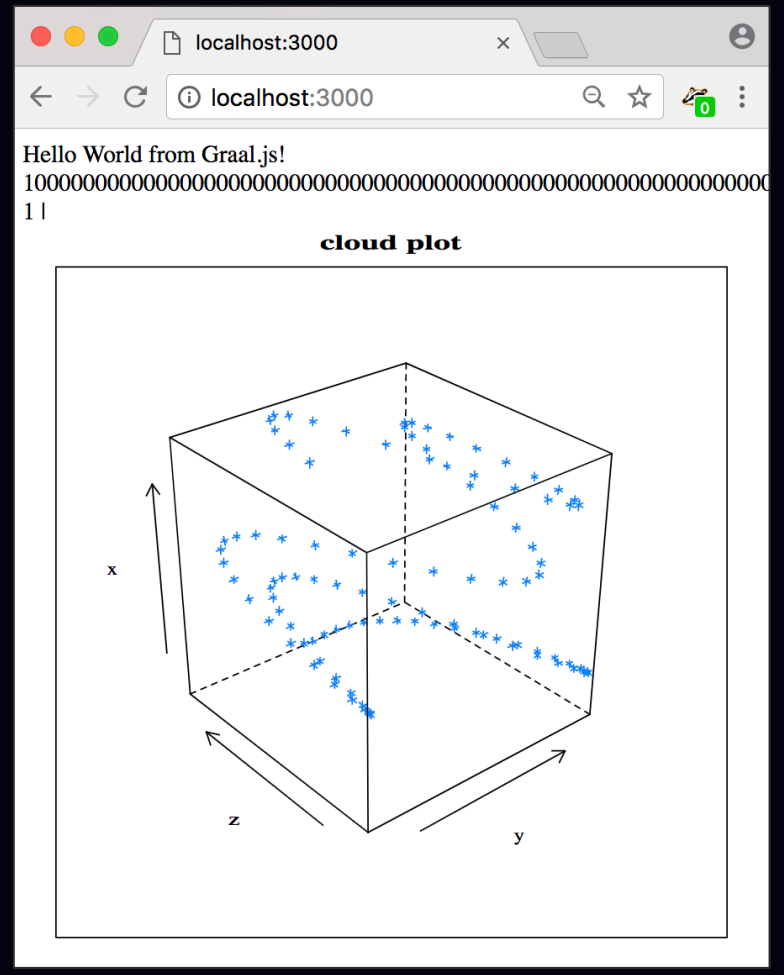
多言語プログラミング対応

- JavaScript/node.jsプログラムの中からJavaとR言語を呼ぶ例

```
const express = require('express')
const app = express()

const BigInteger = Java.type('java.math.BigInteger')

app.get('/', function (req, res) {
  var text = 'Hello World from Graal.js!<br> '
  // Using Java standard library classes
  text += BigInteger.valueOf(10).pow(100)
        .add(BigInteger.valueOf(1)).toString() + '<br>'
  // Using R interoperability to create graphs
  text += Polyglot.eval('R',
    `svg();
    require(lattice);
    x <- 1:100
    y <- sin(x/10)
    z <- cos(x^1.3/(runif(1)*5+10))
    print(cloud(x~y*z, main="cloud plot"))
    grDevices:::svg.off()
    `);
  res.send(text)
})
app.listen(3000, function () {
  console.log('Example app listening on port 3000!')
})
```



GraalVM Enterprise の適用分野とビジネスニーズのマッピング

適用分野	Javaシステム	クラウドシフト	DX推進	レガシーシステム	IoT
ビジネスニーズ	既存インフラへの投資・価値を最大化	運用コストの削減	モダンアーキテクチャの採用（マイクロサービス）	レガシー資産の再利用とモダナイゼーション	デバイスの高機能化とインターネットへの親和性
GraalVM付加価値	• 既存Java SE/Java EEアプリの高速化 • 既存インフラでより高いスループットを実現	• CPU使用率/メモリ使用量の削減 • 使用リソースの効率化 • スケールアウトの抑制	• マイクロサービスの高速起動により高スループット・低遅延 • 小フットプリント	• 多言語環境経由でレガシープログラムをWindows/Linuxで実行 • モダンな言語との連携による機能拡張	• 組み込みデバイスで動作可能な多言語実行環境 • ARMアーキテクチャのサポート
GraalVM機能	高性能JITコンパイラ	高性能JITコンパイラ Native Image	Native Image	多言語プログラミング対応	Native Image

GraalVM Enterprise Edition

エコシステム

参考情報

- GraalVM Enterprise ドキュメント
  - https://docs.oracle.com/cd/F44923_01/enterprise/22/index.html
  - https://docs.oracle.com/en/graalvm/enterprise/22/index.html

- GraalVMブログ
  - https://blogs.oracle.com/java/category/j-graalvm-technology
  - https://medium.com/graalvm

- 製品情報
  - https://www.oracle.com/java/graalvm/
  - https://www.oracle.com/jp/java/graalvm/
  - https://www.oracle.com/downloads/graalvm-downloads.html

Demo

デモ

- パフォーマンス比較
  - OpenJDK vs GraalVM(AOT)
    - フットプリント
    - アプリ起動時間
- サンプルアプリケーション
  - 素数を割り出すプログラム
  - Spring Boot
  - Jar形式 vs Native Image

Thank you