Back-end 개발에 사용되는 프로그래밍 언어는 C++, C#, Java, Python, Go, Javascript 등 매우 다양합니다. 제가 담당하는 크래시리포트는 이 중에서 자바를 이용해서 백엔드 어플리케이션 개발을 했습니다.

자바는 오픈소스 소프트웨어 생태계가 매우 광범위하고 역사도 오래되었습니다. 이 이유 때문에 많은 개발자들의 선택을 받고 있습니다.

그 오픈소스 소프트웨어 중에 스프링부트 프레임워크를 활용하여 크래시리포트의 개별 서브시스템들이 개발되어 있습니다.

자바로 개발된 어플리케이션이 다른 언어 대비 장점만 있는 것은 아닙니다. 자바로 만들어졌기 때문에 감내해야 하는 단점을 갖고 있습니다.

본 글에서는 자바 어플리케이션의 단점 중 하나인 초기 시작 시간이 오래 걸리는 문제에 대해서 이야기 하고자 합니다.

자바를 이용하는 개발자 진영에서도 초기 시작 시간이 오래 걸리는 문제를 해소하기 위해서 여러 접근 및 해소 방법이 제안 되어 있습니다.

1) GraalVM Native Image

2) JVM Checkpoint Restore: Project CRaC

3) Project Leyden

위 3가지 방법의 특징은 간단히 정리해 보겠습니다.

첫번째인 GraalVM Native Image 는 미리 컴파일해서 바로 실행 가능한 상태로 만들어 주는 방법입니다. 이로 인해서 바이트 코드를 기계 언어로 바꾸는 작업이 없어져 초기 실행 시간이 단축되고 성능을 개선할 수 있습니다. 하지만, 네이티브이미지는 생각보다 제약사항이 많습니다. 컴파일 시간이 오래 걸리고 그 과정에서 오류도 발생합니다. 컴파일이 된다 하더라도 실행 시 오류가 많이 발생합니다.

특히 Mybatis 프레임워크를 사용한 어플리케이션의 경우, 네이티브이미지를 만들기 더더욱 어렵습니다. 기존 Mybatis 를 사용한 코드를 네이티브 이미지에서 실행 되도록 대대적인 수정이 이루어져야 합니다. 기존 서비스 중인 코드에 대해서 수정하면서까지 네이티브 이미지를 적용하는 것은 큰 부담이 됩니다. GraalVM Native Image 에 대한 좀더 자세한 내용은 지난번 블로그를 참고해 주세요.

두번째 방법인 Project CRaC 를 간단하게 설명하면 GraalVM 처럼 Azul Systems에서 확장 개발한 JDK 내 포함된 CRIU 프로그램을 사용하여 실행 중인 인스턴스의 snopshot 이미지를 생성하고 이를 새로운 어플리케이션이 시작할 때 활용하도록 하여 실행 시간와 초기 성능을 개선하는 것입니다. 현재는 리눅스 환경에서만 기능을 지원하며 스프링부트 3.2 에서 초기 지원이 시작되어 아직은 시작 단계라 트러블슈팅에 어려움이 있습니다. 저도 시도하다가 중단 멈춘 상태입니다.

세번째 방법인 Project Leyden 은 정적이미지를 활용하여 초기 시작 시간과 성능을 개선하겠다고 했습니다. 이 때 핫스팟 JVM(HotSpot JVM), C2 컴파일러(C2 compiler), 애플리케이션 클래스-데이터 공유(application class-data sharing), 제이링크 코드 도구(jlink code tool) JDK 의 기본 구성요소를 활용할 것이라고만 알려져 있습니다.

위와 같은 방법들이 지속적으로 제안되고 개발되는 것은 아마도 최근 서비스 환경이 마이크로서비스에 기반한 쿠버네티스 환경이 널리 퍼지면서 오토스케일링을 통한 효율적인 인입 트래픽 관리 그리고 효율적인 클라우드 자원관리에 대한 관심이 점점 늘어나고 있는 것에 대한 영향인지도 모르겠습니다. 실제로 Project CRaC 는 AWS Lambda and IBM OpenLiberty 의 지원을 받습니다.

제 첫번째 블로그도 예측하기 어려운 대량의 인입 트래픽을 대응하기 위한 접근 방법 중에 하나였었습니다. 지금도 잘 돌아가서 늘어난 트래픽을 잘 대응해주고 있습니다. 위에서 언급한 1,2 번 방법들은 효과는 강력하지만 그 과정이 복잡하고 제약 사항도 많습니다. 그래서 Project Leyden 이 시작된 것 같습니다. 좀더 범용적이고 단순하게 문제를 해결하고 싶었을 것 같습니다. VirtualThred 를 만든 Project Loom 처럼요. VirtualThread 에 대한 가능성과 제한요소가 아직은 많지만 지속적으로 생태계가 대응한다면 큰 변화를 줄 것이라고 생각합니다. 나중에 기회가 되면 VirtualThread를 실험하고 적용했던 과정과 결과를 공유해드리겠습니다.

저는 Project Leyden 에서 활용하는 애플리케이션 클래스-데이터 공유(application class-data sharing)을 활용하여 Mybatis 가 적용된 시스템의 초기 실행 시간 개선을 해보고자 합니다.

애플리케이션 클래스-데이터 공유(application class-data sharing) 기능은 OpenJDK 12부터 제공되었던 기능입니다. 최근 Spring Framework 6.1.3 에서 정식 지원하면서 좀더 쉽게 사용할 수 있게 되었습니다. 기존 스프링부트 어플리케이션에서 어떻게 적용하는지에 대해서 정리해보려 합니다.

저의 개발환경 JDK 21, Spring boot 3.2.2, Mybatis 3.0.3 그리고 기타 등등의 라이브러리로 되어 있습니다. 위에서 언급했던 것처럼 Mybatis 는 GraalVM 을 이용한 네이티브이미지로 만들기가

상당히 어렵습니다. 해내신 분들은 정말 고수이 십니다. 저는 어려운 길보다는 효과는 떨어지지만 안정적이고 간단한 방법으로 시도해 보겠습니다.

```
 java -XX:ArchiveClassesAtExit=application.jsa -jar -
Dspring.context.exit=onRefresh -Dserver.port=8081 demo.jar
```

위 명령어를 사용하면 프로그램이 초기 실행이 끝나면 바로 종료됩니다. 그리고 application.jsa 라는 파일이 생성됩니다. 이렇게 생성된 파일을 이용해서 다음 명령어를 실행합니다.

```
java -Xlog:cds:file=dynamic-cds.log -Xlog:class+load:file=cds.log -
XX:SharedArchiveFile=application.jsa -jar -Dserver.port=8081 demo.jar
```

위 명령어로 실행하면 실행된 결과가 두 종류의 로그 파일로 남습니다. 여기서 저희는 cds-log-parser 를 통해서 cds.log 파일을 분석해보겠습니다.

그 결과를 다음과 같습니다. 81.25% 가 캐시를 통해서 로드 되었네요. Top 10 package 에 org.springframework 가 보입니다.

```
---------------------------------------------------------------------------
Class Loading Report:
    18034 classes and JDK proxies loaded
    14653 (81.25%) from cache
     3381 (18.75%) from classpath

Categories:
   Lambdas  2064 (11.45%): 9.84% from cache
   Proxies   198 ( 1.10%): 51.52% from cache
   Classes 15774 (87.47%): 90.97% from cache

Top 10 locations from classpath:
      755 /home/appuser/unpack/BOOT-INF/lib/byte-buddy-1.14.11.jar
      435 __JVM_LookupDefineClass__
       95 __dynamic_proxy__
       84 jrt:/java.management
       69
org.springframework.boot.autoconfigure.web.embedded.TomcatWebServerFactoryC
ustomizer
       56 org.mariadb.jdbc.client.DataType
       53 jrt:/jdk.jfr
       47 jrt:/java.base
       31
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryC
ustomizer
       31
org.springframework.boot.autoconfigure.security.oauth2.client.OAuth2ClientP
ropertiesMapper
```

```
Top 10 packages:
     5755 org.springframework (78.11% from cache)
     2742 org.hibernate (91.72% from cache)
      935 java.lang (54.12% from cache)
      805 net.bytebuddy (4.97% from cache)
      790 sun.security (99.62% from cache)
      778 org.apache (93.96% from cache)
      751 java.util (98.00% from cache)
      635 com.fasterxml (96.54% from cache)
      417 ch.qos (83.21% from cache)
      372 jdk.internal (93.55% from cache)
--------------------------------------------------------------------------------
```

실제 소요된 실행시간을 확인해 보겠습니다.

케이스 1: 일반 jar 파일을 이용한 실행시간

```
Started CrashCollectorApplication in 75.101 seconds (process running for
83.693)
```

케이스 2: Layered-jar 파일을 이용한 실행시간

```
Started CrashCollectorApplication in 66.292 seconds (process running for
71.196)
```

케이스 3: Layered-jar 파일에 CDS 캐싱을 적용한 후 실행시간

```
Started CrashCollectorApplication in 51.601 seconds (process running for
56.225)
```

위 테스트를 보면 Layered-jar 가 언급되는데요. 기본 Jar 는 압축된 상태의 파일입니다. 이 파일의 압축을 풀어서 계층 형태로 분배하고 이를 Docker 복사하면서 도커내 개별 레이어를 같게 됩니다. 이렇게 되면 라이브러리 영역과 사용자 코드 영역 등으로 분리되면서 신규 도커이미지가 추가 될 때 중복이 제거되어 관리되는 도커이미지의 크기를 줄 일 수 있습니다. 즉, 라이브러리는 100Mb 이지만 실제 코드는 1Mb 라면, 레이어로 분리되어 관리 된다면 수정된 어플리케이션의 신규 도커이미지 변동 부분은 1Mb 인 것입니다. 추가적으로 실행 시간도 개선됩니다. Layered-jar 는 Spring boot 2.3 이후부터 정식 지원이 되었습니다.

Layered jar 를 생성 및 실행하는 방법은 다음과 같습니다.

```
RUN java -Djarmode=layertools -jar /build/target/demo.jar extract
```

```
COPY --from=builder /build/dependencies/ ./unpack/
COPY --from=builder /build/spring-boot-loader/ ./unpack/
COPY --from=builder /build/snapshot-dependencies/ ./unpack/
COPY --from=builder /build/application/ ./unpack/


WORKDIR /home/appuser/unpack
CMD ["java", "org.springframework.boot.loader.launch.JarLauncher"]
```

실행 스테이지에서 위와 같이 복사해서 실행하면 됩니다.

결론

위 적용 결과를 보면 최초 83초에서 56초로 30% 실행 소요시간이 단축되었습니다. GraalVM을 이용했을 때 보다는 단축시간 개선율이 떨어지지만, 간단한 노력으로 30% 향상시킬 수 있었습니다. 오토스케일링이 빈번하여 초기 실행 시간을 단축하고 싶지만 GraalVM 네이티브 이미지를 사용할 수 없는 환경에서는 한번쯤은 고려해볼만한 선택지라고 생각합니다.

자바가 세상에 나온 지 20년이 넘어가지만 기존의 문제를 해결하기 위한 사용자들의 노력이 지속되면서, 저를 포함한 개발자들의 꾸준한 선택을 받는 것 같습니다.

1. https://spring.io/blog/2023/10/16/runtime-efficiency-with-spring
2. https://spring.io/blog/2020/08/14/creating-efficient-docker-images-with-spring-boot-2-3
3. https://www.baeldung.com/spring-boot-docker-images#layered-jars
4. https://github.com/snicoll/cds-log-parser
5. https://github.com/ionutbalosin/faster-jvm-start-up-techniques/blob/main/app-dynamic-cds-hotspot/README.md
6. https://docs.oracle.com/en/java/javase/17/vm/class-data-sharing.html#GUID-7EAA3411-8CF0-4D19-BD05-DF5E1780AA91
7. https://docs.spring.io/spring-framework/reference/integration/class-data-sharing.html
8. https://www.ciokorea.com/news/237479