MEGAZONE
Cloud Innovator

Just-in-Time (JiT) vs Ahead-of-Time (AoT)

2022.02
DxLabs 가영권

INDEX

1

JIT란 무엇인가?

JIT란?

JIT 컴파일(Just-In-Time compilation) 또는 동적 번역(Dynamic Translation)은 프로그램을 실제 실행하는 시점에 기계어로 번역하는 컴파일 기법이다.

(동적 컴파일 환경은 실행 과정에서 컴파일을 할 수 있기 위해 만들어졌다.)

- ✔ 빠른 개발 주기로 개발이 가능하다.
- ✔ 실행 속도가 느려진다.(프로그램 실행이 시작될 때 코드를 실행하기 전에 분석 및 컴파일을 수행해야 하기 때문이다.)

JIT 컴파일러는 인터프리트 방식, 정적 컴파일을 혼합한 방식으로 생각할 수 있는데, 실행 시점에서 인터프리트 방식으로 기계어 코드를 생성하면서 그 코드를 캐싱하여, 같은 함수가 여러 번 불릴 때 매번 기계어 코드를 생성하는 것을 방지한다 (재사용시 컴파일을 다시 할 필요가 없다.).

[참고 : 인터프리터도 런타임에 소스코드를 기계어로 변환해서 비슷하다고 볼 수 있으나 **JIT가 컴파일 하는 대상은 소스 코드가 아니라 최적화를 한번 거친 ByteCode이다**.]

또한, JIT는 정적 컴파일러만큼 빠르면서 인터프러터 언어의 빠른 응답속도를 추구하기 위해 사용한다.

하지만, JIT 컴파일러는 코드를 실행하기 위해서는 코드 분석과 컴파일을 해야하기 때문에 초기 실행 속도를 느리게 하는 단점이 있다.

최근에는 자바 가상 머신(JVM)과 .NET, V8(node.js)에서 JIT 컴파일을 지원한다.

Java 코드는 JavaCompiler를 통해 ByteCode로 변환한 다음, 실제 ByteCode를 실행하는 시점에서 자바 가상 머신(JVM)이 바이트코드를 JIT 컴파일을 통해 빠른 속도로 기계어로 변환한다.

이런 기계어 변환은 코드가 실행되는 과정에 실시간으로 일어나며(그래서 Just-In-Time이다), 전체 코드의 필요한 부분만 변환한다.

2

JIT는 어떻게 동작하는가?

JIT 동작 원리

JIT 컴파일러는 전체 프로젝트 파일을 기계어로 번역하지 않고 함수 단위나 파일 단위로 바이트 코드를 기계어로 번역한다.

그래서 실행 중 모든 바이트 코드를 기계어로 변환하는 것이 아니라 호출된 함수나 파일만 번역하게 된다.

JIT 컴파일러를 사용하는 JVM들은 내부적으로 해당 메서드가 얼마나 자주 수행되는지 체크하고, 일정 정도를 넘을 때에만 컴파일을 수행한다.

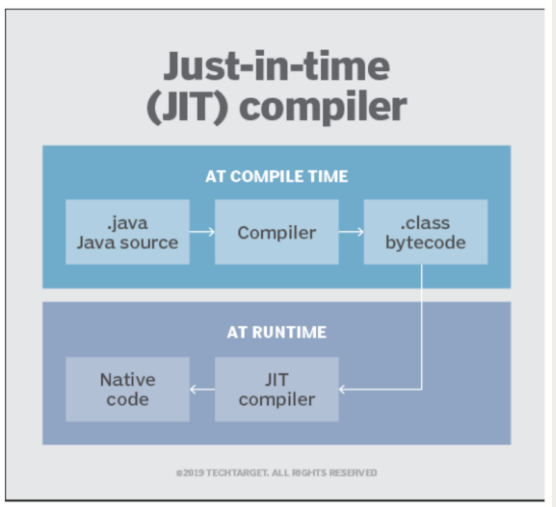
또한, 이미 번역된 기계어 중, 자주 쓰이는 코드를 코드 캐시 공간에 캐싱한 뒤 동일한 코드를 여러 번 호출하는 경우, 다시 컴파일 하는게 아니라 기존에 번역한 코드를 재사용한다. 이렇게 최적화 작업을 거치게 되기 때문에 오버헤드가 최소화 된다.

JIT 컴파일러는 자바에 기본적으로 설정되어 있는 기능이다.

메서드가 컴파일되면 JVM은 해석하지 않고 직접 컴파일 메서드를 호출한다.

이론적으로 컴파일이 프로세서 시간과 메모리 사용을 필요로 하지 않는다면

모든 메서드를 컴파일 했을 때 JAVA 프로그램의 속도가 네이티브 프로그램의 속도에 가까워 질 수 있다.



2

JIT는 어떻게 동작하는가?

JIT 동작 원리

JVM이 처음 시작되면 수천 가지 메서드가 호출된다.

이렇게 모든 메서드를 컴파일하면 결국 프로그램이 매우 우수한 최대 성능을 달성하더라도 시작 시간에 상당한 영향을 줄 수 있다.

각 메서드에 대해 JVM은 사전 정의 된 컴파일 임계 값에서 시작하여 메서드가 호출 될 때마다 감소되는 호출 계수를 정의한다.

호출 계수가 0에 도달하면 메서드에 대한 Just-In-Time 컴파일이 시작된다.

따라서 자주 사용되는 메서드는 JVM이 시작된 직후에 컴파일되며 사용되지 않는 메서드는 훨씬 나중에 컴파일되거나 전혀 컴파일되지 않는다.

JIT 컴파일 임계 값은 JVM이 빨리 시작하고 성능이 향상되도록 도와준다.

임계 값은 시작 시간과 장기간 성능 사이의 최적의 균형을 얻기 위해 선택되었다.

JVM은 실제로 JIT Compiler를 포함하고 있지만, Interpreter 이고 JIT는 Compiler라고 볼 수 있다. 또한 JVM에서는 JIT는 option으로 사용 여부를 선택 할 수 있다.

2

JIT는 어떻게 동작하는가?

JIT 컴파일러의 코드 최적화 방법

JIT 컴파일은 다음 단계로 구성된다.

1. 인라인(Inlining)

- 인라인은 작은 메서드의 트리를 병합하거나 '인라인'하여 그들의 호출 트리를 만든다. 이렇게 하면 자주 호출되는 메서드의 호출 속도가 빨라진다.

2. 지역 최적화(Local optimizations)

- 지역 최적화는 한번에 코드의 작은 부분을 분석하고 성능을 향상시킨다. 주로 코드의 중복된 연산 제거, 예측 가능한 값의 대치 등이다.

3. 제어 흐름 최적화(Control flow optimizations)

- 제어 흐름 최적화는 메서드나 그 내부의 제어 흐름을 분석하고 코드 경로를 재정렬하여 효율성을 향상시킨다.

4. 전역 최적화(Global optimizations)

- 전역 최적화는 전체 메서드에서 즉시 작동한다. 그들은 더 비싼 컴파일 시간을 요구하지만 성능을 크게 향상시킬 수 있다.

5. 네이티브 코드 생성(Native code generation)

- 네이티브 코드 생성은 플랫폼 아키텍처에 따라 다르다. 일반적으로 컴파일러는 이 단계에서 메서드 트리를 네이티브 코드로 변환한다. 일부 작은 최적화가 아키텍처 특성에 따라 수행된다.

3

AOT란 무엇인가?

AOT란?

AOT 컴파일(ahead-of-time compile)은 실행 시점 이전(런타임 이전에)에 기계어로 바꾸는 컴파일러이다.

(정적 컴파일 환경은 실행 전에 컴파일을 할 수 있기 위해 만들어졌다.)

- ✔ 개발 중에 AOT 컴파일을 수행하면 개발 주기가 느려진다. (프로그램을 변경 후 실행하여 결과를 봐야하기 때문이다.)
- ✔ 런타임에 분석 및 컴파일을 위해 일시 중지하지 않고, 보다 예측 가능하게 실행할 수 있는 프로그램을 만든다.
- ✔ 실행 속도가 더 빠르다.

실행 전에 모두 기계어로 변환되어 JIT 컴파일러가 런타임에 컴파일함으로써 발생하는 성능 이슈가 생기지 않고 거의 Native의 성능을 낼 수 있게 된다.

하지만, AOT 컴파일러를 사용하면 실행 전에 전체 파일을 빌드해야 하기 때문에 빌드 속도가 느려진다. (컴파일 시간이 많이 소요)

설치할 때도 JIT 컴파일러를 사용하는 경우 바이트 코드만 받으면 되는 반면, AOT 컴파일러를 사용하면 기계어로 번역하는 작업까지 포함되기 때문에 느리다.

이러한 단점에도 불구하고 AOT 컴파일 방식은 런타임 시점에 프로그램의 동작 예측이 가능하고, 런타임 시점에 분석이나 컴파일을 위해 프로그램을 멈출 필요가 없다. 물론 실행 속도 또한 더 빠르다.

4

JIT 및 AOT 비교

원시 코드를 곧바로 기계어로 컴파일하느냐(AOT),
미리 해석되어 있는 바이트 코드를 기계어로 재차 컴파일 하느냐(JIT)의 차이

5 참고 사이트

JIT

https://medium.com/@lazysoul/jit-just-in-time-16bb63f3ae26

https://selfish-developer.com/entry/JITJust-In-Time-Compilation

https://tjdrnr05571.tistory.com/19

https://bloofer.tistory.com/21

AOT

https://selfish-developer.com/entry/AOTAhead-Of-Time-Compiler?category=692353

https://y-oni.tistory.com/191#toc21

It started!
Let's growing!
Cloud Native