Effective Java 3/E

Item 69

발표자 : 김형주

예외는 진짜 예외 상황에만

사용하라

Content

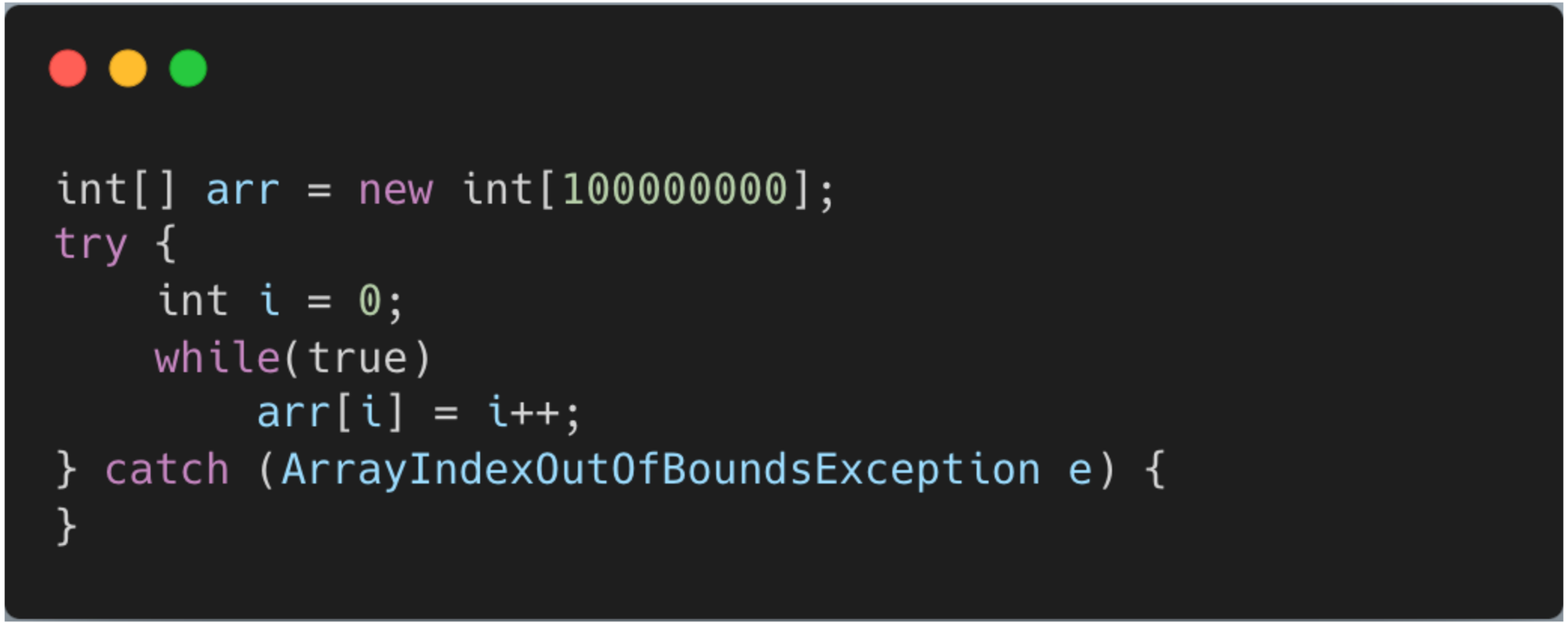
📌 예외 상황에서만 써야하는 이유

📌 백엔드가 배워야할 점

Content

📌 예외 상황에서만 써야하는 이유

📌 백엔드가 배워야할 점



```java
int[] arr = new int[1000000000];
try {
    int i = 0;
    while(true)
        arr[i] = i++;
} catch (ArrayIndexOutOfBoundsException e) {
}
```


이해가
되셨나요?

```
int size = 100000000;
int[] arr = new int[size];
for (int i = 0; i < size; i++) {
    arr[i] = i;
}
```


이번에는
어떠신가요?

```java
int[] arr = new int[100000000];
try {
    int i = 0;
    while(true)
        arr[i] = i++;
} catch (ArrayIndexOutOfBoundsException e) {
}
```

```java
int size = 100000000;
int[] arr = new int[size];
for (int i = 0; i < size; i++) {
    arr[i] = i;
}
```

📌 예외 상황에서만 써야하는 이유

1. 가독성이 떨어진다.

```java
int[] arr = new int[100000000];
try {
    int i = 0;
    while(true)
        arr[i] = i++;
} catch (ArrayIndexOutOfBoundsException e) {
}
```

왜 등장했는가?

```java
int size = 100000000;
int[] arr = new int[size];
for (int i = 0; i < size; i++) {
    arr[i] = i;
}
```

배열에 접근할때 마다

경계를 넘지 않았는지 검사

배열의 경계에 도달하면

자동으로 반복문 종료

```java
int[] arr = new int[100000000];
try {
    int i = 0;
    while(true)
        arr[i] = i++;
} catch (ArrayIndexOutOfBoundsException e) {
}
```

인덱스 조건 없는 접근으로

검사 생략

예외를 통해 나오자

조건 없는 접근으로

검사 생략

를 통해 나오자

📌 예외 상황에서만 써야하는 이유

1. 가독성이 떨어진다.

2. Try - catch 문을 사용하면 JVM이 적용할 수 있는 최적화가 제한된다.

Methods

- Methods are often inlined. This increases the compiler's "horizon" of optimization.
- Static, private, final, and/or "special" invocations are easy to inline.
- Virtual (and interface) invocations are often demoted to "special" invocations, if the class hierarchy permits it. A dependency is registered in case further class
- Virtual (and interface) invocations with a lopsided type profile are compiled with an optimistic check in favor of the historically common type (or two types).
- Depending on the profile, a failure of the optimistic check will either deoptimize or run through a (slow) vtable/itable call.
- On the fast path of an optimistically typed call, inlining is common. The best case is a de facto monomorphic call which is inlined. Such calls, if back-to-back, w
- In the absence of strong profiling information, a virtual (or interface) call site will be compiled in an agnostic state, waiting for the first execution to provide a pro
- An inline cache will flip to a monomorphic state at the first call, and stay in that state as long as the exact receiver type (not a subtype) is repeated every time.

Methods

- Methods are often inlined. This increases the compiler's "horizon" of optimization.
- Static, private, final, and/or "special" invocations are easy to inline.
- Virtual (and interface) invocations are often demoted to "special" invocations, if the class hierarchy permits it. A dependency is registered in case further class
- Virtual (and interface) invocations with a lopsided type profile are compiled with an optimistic check in favor of the historically common type (or two types).
- Depending on the profile, a failure of the optimistic check will either deoptimize or run through a (slow) vtable/itable call.
- On the fast path of an optimistically typed call, inlining is common. The best case is a de facto monomorphic call which is inlined. Such calls, if back-to-back, w
- In the absence of strong profiling information, a virtual (or interface) call site will be compiled in an agnostic state, waiting for the first execution to provide a pro
- An inline cache will flip to a monomorphic state at the first call, and stay in that state as long as the exact receiver type (not a subtype) is repeated every time.

인라이닝 (오버헤드 감소 / 루프 최적화)

: 자주 호출되는 메서드의 경우,

메서드 내용을 기억해놨다가 호출지점에 바로 삽입하는 것

Try-catch 구문에서는

예외 상황도 생각해야 하기 때문에

인라이닝을 하지 않는다.

Deoptimization

Deoptimization is the process of changing an optimized stack frame to an unoptimized one. With respect to compiled methods, it is also the process of throwing away co dozens of times.

- The compiler may stub out an untaken branch and deoptimize if it is ever taken.
- Similarly for low-level safety checks that have historically never failed.
- If a call site or cast encounters an unexpected type, the compiler deoptimizes.
- If a class is loaded that invalidates an earlier class hierarchy analysis, any affected method activations, in any thread, are forced to a safepoint and deoptimized.
- Such indirect deoptimization is mediated by the dependency system. If the compiler makes an unchecked assumption, it must register a checkable dependency.

Deoptimization

Deoptimization is the process of changing an optimized stack frame to an unoptimized one. With respect to compiled methods, it is also the process of throwing away co dozens of times.

- The compiler may stub out an untaken branch and deoptimize if it is ever taken.
- Similarly for low-level safety checks that have historically never failed.
- If a call site or cast encounters an unexpected type, the compiler deoptimizes.
- If a class is loaded that invalidates an earlier class hierarchy analysis, any affected method activations, in any thread, are forced to a safepoint and deoptimized.
- Such indirect deoptimization is mediated by the dependency system. If the compiler makes an unchecked assumption, it must register a checkable dependency.

디옵티마이제이션

: 최적화 상태에서 최적화 해제 상태로 되돌리는 것

예상치 못한 타입 변환과 같은 예외를 직면하면 최적화 해제로 변환

반복적인

디옵티마이제이션은

성능을 떨어트린다.

```java
int i = 0;
int[] arr = new int[100000000];

long start = System.currentTimeMillis();
try {
  while (true) {
    arr[i] = i++;
    }
} catch(ArrayIndexOutOfBoundsException e) {
  long end = System.currentTimeMillis();
  System.out.println("실행 시간(ms): " + (end - start));
  e.printStackTrace();
}
```

96ms

```java
int i = 0;
int[] arr = new int[100000000];

long start = System.currentTimeMillis();
try {
  while (true) {
    arr[i] = i++;
    }
} catch(ArrayIndexOutOfBoundsException e) {
  long end = System.currentTimeMillis();
  System.out.println("실행 시간(ms): " + (end - start));
  e.printStackTrace();
}
```

96ms

```java
int[] arr = new int[100000000];

long start = System.currentTimeMillis();
for (int i = 0; i < 100000000; i++) {
  arr[i] = i;
}
long end = System.currentTimeMillis();
System.out.println("실행 시간(ms): " + (end - start));
```

72ms

📌 예외 상황에서만 써야하는 이유

1. 가독성이 떨어진다.

2. Try - catch 문을 사용하면 JVM이 적용할 수 있는 최적화가 제한된다.

📌 예외 상황에서만 써야하는 이유

1. 가독성이 떨어진다.

2. Try - catch 문을 사용하면 JVM이 적용할 수 있는 최적화가 제한된다.

3. 디버깅이 쉽지 않다.

```java
int[] arr = new int[100000000];
try {
  int i = 0;
  while(true){
    ...
  }

} catch (ArrayIndexOutOfBoundsException e) {
}
```

catch하려는 예외랑 똑같은 예외가 발생하면?

📌 예외 상황에서만 써야하는 이유

1. 가독성이 떨어진다.

2. Try - catch 문을 사용하면 JVM이 적용할 수 있는 최적화가 제한된다.

3. 디버깅이 쉽지 않다.

예외를 흐름제어로

사용하지 마라!

= 예외는 진짜 예외 상황에만 사용하라

Content

📌 예외 상황에서만 써야하는 이유

📌 백엔드가 배워야할 점

예외를 흐름제어로

클라이언트

사용하지 마라!

API 설계시, 예외를 제어 흐름으로

개발자

사용해야 할것 같은 뉘앙스를 주지말자.

특정 상태에서만 호출하는 상태의존적 메서드가 존재하면

상태 검사 메서드도 함께 제공하자

상태를 검사하는 메서드 = 상태 검사 메서드

```
for (Iterator<Foo> i = collections.iterator(); i.hasNext(); ) {
    Foo foo = i.next();
}
```

"다음 값이 존재한다." 라는 상태에 의존하는 메서드

= 상태 의존 메서드

상태 검사 메서드

옵셔널 반환

null와 같은 특수한 값 반환

📌 정리

흐름 제어 (X)

진짜 예외 상황에만 사용하자

흐름 제어를 예외로 사용하게

강요하는 API도 만들지 말자

Item 69

진짜 [illegible]하라