

자바™ 네이티브

상호 작용

프로그래머 가이드 및 사양

1

장

소개

그만큼 JNI(Java™ Native Interface)는 Java 플랫폼의 강력한 기능입니다.

JNI를 사용하는 애플리케이션은 C 및 C++와 같은 프로그래밍 언어로 작성된 네이티브 코드 와 Java 프로그래밍 언어로 작성된 코드를 통합할 수 있습니다. JNI를 통해 프로그래머는 레거시 코드에 대한 투자를 포기하지 않고도 Java 플랫폼의 기능을 활용할 수 있습니다. JNI는 Java 플랫폼의 일부이기 때문에 프로그래머는 상호 운용성 문제를 한 번만 해결할 수 있으며 해당 솔루션이 Java 플랫폼의 모든 구현에서 작동할 것으로 기대할 수 있습니다.

이 책은 JNI에 대한 프로그래밍 가이드이자 참조 매뉴얼입니다.

이 책은 세 부분으로 구성되어 있습니다.

- 2장에서는 간단한 예제를 통해 JNI를 소개합니다. JNI에 익숙하지 않은 초보 사용자를 위한 튜토리얼입니다.
- 3장에서 10장은 다양한 JNI 기능에 대한 광범위한 개요를 제공하는 프로그래머 가이드를 구성합니다. 다양한 JNI 기능을 강조하고 JNI 프로그래밍에서 유용한 것으로 입증된 기술을 제시하기 위해 일련의 짧지만 설명적인 예제를 살펴볼 것입니다. • 11~13장은 모든 JNI 유형 및 기능에 대한 최종 사양을 제공합니다. 이 장들은 또한 참조 설명서 역할을 하도록 구성되어 있습니다.

이 책은 JNI에 대한 다양한 요구를 가진 폭넓은 독자의 관심을 끌기 위해 노력합니다.

자습서 및 프로그래밍 가이드는 초보 프로그래머를 대상으로 하는 반면 숙련된 개발자 및 JNI 구현자는 참조 섹션이 더 유용할 수 있습니다. 대부분의 독자는 JNI를 사용하여 애플리케이션을 작성하는 개발자일 것입니다. 이 책에서 "귀하"라는 용어는 JNI 구현자 또는 JNI를 사용하여 작성된 애플리케이션의 최종 사용자가 아니라 JNI로 프로그래밍하는 개발자를 암시적으로 나타냅니다.

이 책은 독자가 Java, C 및 C++ 프로그래밍 언어에 대한 기본 지식이 있다고 가정합니다. 그렇지 않은 경우 사용할 수 있는 많은 훌륭한 책 중 하나를 참조할 수 있습니다. The Java™ Programming Language, Second Edition, 저서: Ken Arnold 및 James Gosling(Addison-Wesley, 1998), The C Programming Language, Second Edition, by Brian Kernighan과 Dennis Ritchie(프렌티스 헐,

1.1 Java 플랫폼 및 호스트 환경

소개

1988) 및 Bjarne Stroustrup의 C++ 프로그래밍 언어 제3판 (Addison-Wesley, 1997).

이 장의 나머지 부분에서는 JNI의 배경, 역할 및 발전에 대해 소개합니다.

1.1 자바 플랫폼과 호스트 환경

이 책은 Java 프로그래밍 언어와 원시(C, C++ 등) 프로그래밍 언어로 작성된 응용 프로그램을 다루기 때문에 먼저 이러한 언어에 대한 프로그래밍 환경의 정확한 범위를 명확히 하겠습니다.

Java 플랫폼은 Java VM(가상 머신)과 Java API(응용 프로그래밍 인터페이스)로 구성된 프로그래밍 환경입니다.¹ Java 응용 프로그램은 Java 프로그래밍 언어로 작성되고 기계 독립적인 이진 클래스 형식으로 컴파일됩니다. 클래스는 모든 JVM(Java Virtual Machine) 구현에서 실행될 수 있습니다. Java API는 사전 정의된 클래스 세트로 구성됩니다. Java 플랫폼의 모든 구현은 Java 프로그래밍 언어, 가상 머신 및 API를 지원하도록 보장됩니다.

호스트 환경이라는 용어는 호스트 운영 체제, 기본 라이브러리 집합 및 CPU 명령 집합을 나타냅니다. 네이티브 애플리케이션은 C 및 C++와 같은 네이티브 프로그래밍 언어로 작성되고 호스트별 바이너리 코드로 컴파일되어 네이티브 라이브러리와 연결됩니다. 기본 애플리케이션 및 기본 라이브러리는 일반적으로 특정 호스트 환경에 종속됩니다. 예를 들어 하나의 운영 체제용으로 구축된 AC 응용 프로그램은 일반적으로 다른 운영 체제에서는 작동하지 않습니다.

Java 플랫폼은 일반적으로 호스트 환경 위에 배포됩니다. 예를 들어 JRE(Java Runtime Environment)는 Solaris 및 Windows와 같은 기존 운영 체제에서 Java 플랫폼을 지원하는 Sun 제품입니다. Java 플랫폼은 응용 프로그램이 기본 호스트 환경과 독립적으로 의존할 수 있는 일련의 기능을 제공합니다.

1.2 JNI의 역할

Java 플랫폼이 호스트 환경 위에 배포되면 Java 응용 프로그램이 다른 언어로 작성된 기본 코드와 밀접하게 작동하도록 허용하는 것이 바람직하거나 필요할 수 있습니다. 프로그래머들은 전통적으로 C 및 C++로 작성된 애플리케이션을 구축하기 위해 Java 플랫폼을 채택하기 시작했습니다. 때문에

1. 여기에서 사용된 "Java 가상 머신" 또는 "Java VM"이라는 문구는 Java 플랫폼용 가상 머신을 의미합니다. 마찬가지로 "Java API"라는 문구는 Java 플랫폼용 API를 의미합니다.

그러나 레거시 코드에 대한 기존 투자는 Java 애플리케이션이 앞으로 수년 동안 C 및 C++ 코드와 공존할 것입니다.

JNI는 Java 플랫폼을 활용하면서도 다른 언어로 작성된 코드를 계속 활용할 수 있게 해주는 강력한 기능입니다. Java 가상 머신 구현의 일부인 JNI는 Java 애플리케이션이 네이티브 코드를 호출하거나 그 반대의 경우를 허용하는 양방향 인터페이스입니다. 그림 1.1은 JNI의 역할을 보여줍니다.

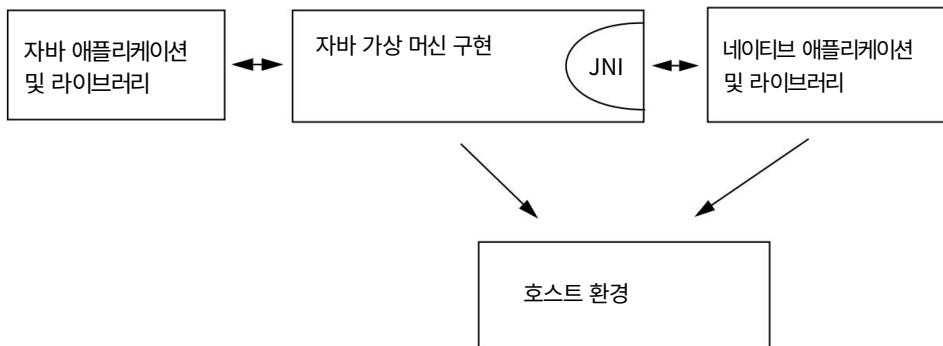


그림 1.1 JNI의 역할

JNI는 Java 애플리케이션을 네이티브 코드와 결합해야 하는 상황을 처리하도록 설계되었습니다. 양방향 인터페이스로서 JNI는 네이티브 라이브러리 와 네이티브 애플리케이션이라는 두 가지 유형의 네이티브 코드를 지원할 수 있습니다.

- JNI를 사용하여 Java 애플리케이션이 네이티브 라이브러리에 구현된 함수를 호출할 수 있도록 하는 네이티브 메서드를 작성할 수 있습니다. Java 응용 프로그램은 Java 프로그래밍 언어로 구현된 메서드를 호출하는 것과 동일한 방식으로 네이티브 메서드를 호출합니다. 그러나 배후에서 기본 메서드는 다른 언어로 구현되고 기본 라이브러리에 상주합니다. • JNI는 Java 가상 머신 구현을 네이티브 애플리케이션에 임베드할 수 있는 호출 인터페이스를 지원합니다. 기본 응용 프로그램은 Java 가상 머신을 구현하는 기본 라이브러리와 연결한 다음 호출 인터페이스를 사용하여 Java 프로그래밍 언어로 작성된 소프트웨어 구성 요소를 실행할 수 있습니다. 예를 들어, C로 작성된 웹 브라우저는 내장된 JVM(Java Virtual Machine) 구현에서 다운로드한 애플리케이션을 실행할 수 있습니다.

1.3 JNI 사용의 의미

애플리케이션이 JNI를 사용하면 Java 플랫폼의 두 가지 이점을 잊을 위험이 있음을 기억하십시오.

첫째, JNI에 의존하는 Java 애플리케이션은 더 이상 여러 호스트 환경에서 쉽게 실행할 수 없습니다. Java 프로그래밍 언어로 작성된 응용 프로그램의 일부가 여러 호스트 환경에 이식 가능하더라도 네이티브 프로그래밍 언어로 작성된 응용 프로그램의 일부를 다시 컴파일해야 합니다.

둘째, Java 프로그래밍 언어는 형식이 안전하고 안전하지만 C 또는 C++와 같은 기본 언어는 그렇지 않습니다. 결과적으로 JNI를 사용하여 애플리케이션을 작성할 때 특별히 주의해야 합니다. 오작동하는 네이티브 메서드는 전체 애플리케이션을 손상시킬 수 있습니다. 이러한 이유로 Java 애플리케이션은 JNI 기능을 호출하기 전에 보안 검사를 받아야 합니다.

일반적으로 네이티브 메서드가 가능한 한 적은 수의 클래스에 정의되도록 애플리케이션을 설계해야 합니다. 이는 네이티브 코드와 애플리케이션의 나머지 부분 사이에 보다 명확한 경리를 수반합니다.

1.4 JNI를 사용하는 경우

JNI를 사용하여 프로젝트를 시작하기 전에 한 걸음 물러서서 더 적합한 대체 솔루션이 있는지 조사해 보는 것이 좋습니다. 마지막 섹션에서 언급한 바와 같이 JNI를 사용하는 애플리케이션은 엄격하게 Java 프로그래밍 언어로 작성된 애플리케이션과 비교할 때 고유한 단점이 있습니다. 예를 들어 Java 프로그램 ming 언어의 형식 안전성 보장이 손실됩니다.

또한 여러 대안적 접근 방식을 통해 Java 애플리케이션의 상호 운용이 가능합니다. 다른 언어로 작성된 코드로 먹었습니다. 예를 들어:

- Java 응용 프로그램은 TCP/IP 연결 또는 기타 IPC(프로세스 간 통신) 메커니즘을 통해 기본 응용 프로그램과 통신할 수 있습니다.
- Java 애플리케이션은 JDBC™를 통해 레거시 데이터베이스에 연결할 수 있습니다.
API.
- Java 응용 프로그램은 다음과 같은 분산 개체 기술을 활용할 수 있습니다.
자바 IDL API로.

이러한 대체 솔루션의 공통적인 특징은 Java 애플리케이션과 원시 코드가 서로 다른 프로세스(경우에 따라 서로 다른 시스템에 있음)에 상주한다는 것입니다. 프로세스 분리는 중요한 이점을 제공합니다. 주소 공간 프로

프로세스가 지원하는 기술은 높은 수준의 오류 격리를 가능하게 합니다. 충돌한 기본 응용 프로그램은 TCP/IP를 통해 통신하는 Java 응용 프로그램을 즉시 종료하지 않습니다.

그러나 경우에 따라 Java 애플리케이션이 동일한 프로세스에 있는 원시 코드와 통신해야 하는 경우가 있습니다. 이때 JNI가 유용해집니다. 예를 들어 다음 시나리오를 고려하십시오.

- Java API는 응용 프로그램에 필요한 특정 호스트 종속 기능을 지원하지 않을 수 있습니다. 예를 들어 애플리케이션은 Java API에서 지원하지 않는 특수 파일 작업을 수행하려고 할 수 있지만 다른 프로세스를 통해 파일을 조작하는 것은 번거롭고 비효율적입니다.
- 기존 기본 라이브러리에 액세스하고 싶지만 다른 프로세스에서 데이터를 복사하고 전송하는 오버헤드에 대해 비용을 지불하고 싶지 않을 수 있습니다.

동일한 프로세스에서 네이티브 라이브러리를 로드하는 것이 훨씬 더 효율적입니다. • 응용 프로그램이 여러 프로세스에 걸쳐 있으면 수용할 수 없는 메모리 사용량이 발생할 수 있습니다. 이는 일반적으로 이러한 프로세스가 동일한 클라이언트 시스템에 상주해야 하는 경우에 해당됩니다. 응용 프로그램을 호스팅하는 기존 프로세스에 기본 라이브러리를 로드하면 새 프로세스를 시작하고 해당 프로세스에 라이브러리를 로드하는 것보다 적은 시스템 리소스가 필요합니다.

- 어셈블리와 같은 저수준 언어로 시간이 중요한 코드의 작은 부분을 구현하고자 할 수 있습니다. 3D 집약적인 응용 프로그램이 대부분의 시간을 그래픽 렌더링에 사용하는 경우 최대 성능을 달성하기 위해 그래픽 라이브러리의 핵심 부분을 어셈블리 코드로 작성해야 할 수도 있습니다.

요약하면 Java 애플리케이션이 네이티브 애플리케이션과 상호 운용되어야 하는 경우 JNI를 사용하십시오. 동일한 프로세스에 상주하는 코드.

1.5 JNI의 진화

네이티브 코드와 상호 운용하기 위한 Java 애플리케이션의 필요성은 Java 플랫폼 초기부터 인식되어 왔습니다. Java 플랫폼의 첫 번째 릴리스인 JDK™(Java Development Kit) 릴리스 1.0에는 Java 응용 프로그램이 C 및 C++와 같은 다른 언어로 작성된 함수를 호출할 수 있도록 하는 기본 메서드 인터페이스가 포함되어 있습니다. 많은 타사 애플리케이션과 Java 클래스 라이브러리(예: java.lang, java.io 및 java.net 포함)의 구현은 네이티브 메서드 인터페이스에 의존하여 다음의 기능에 액세스했습니다. 기본 호스트 환경.

안타깝게도 JDK 릴리스 1.0의 기본 메서드 인터페이스에는 두 가지 주요 문제가 있었습니다.

- 첫째, 기본 코드는 C 구조의 구성원으로 개체의 필드에 액세스합니다.

그러나 JVM(Java Virtual Machine) 사양은 개체가 메모리에 배치되는 방식을 정의하지 않습니다. 주어진 JVM(Java Virtual Machine) 구현이 기본 메소드 인터페이스에서 가정한 것과 다른 방식으로 오브젝트를 배치하는 경우 기본 메소드 라이브러리를 다시 컴파일해야 합니다.

- 둘째, JDK 릴리스 1.0의 기본 메소드 인터페이스는 기본 메소드가 가상 머신의 객체에 대한 직접 포인터를 보유할 수 있기 때문에 보수적인 가비지 수집기에 의존합니다. 고급 가비지 수집 알고리즘을 사용하는 가상 머신 구현은 JDK 릴리스 1.0의 기본 메서드 인터페이스를 지원할 수 없습니다.

JNI는 이러한 문제를 극복하기 위해 설계되었습니다. 다양한 호스트 환경에서 모든 JVM(Java Virtual Machine) 구현에서 지원할 수 있는 인터페이스입니다. JNI 사용:

- 각 가상 머신 구현자는 더 큰 기본 코드 본문을 지원할 수 있습니다. • 개발 도구 공급업체는 다양한 종류의 네이티브 메서드 인터페이스를 처리할 필요가 없습니다.
- 가장 중요한 점은 응용 프로그램 프로그래머가 고유 코드의 한 버전을 작성할 수 있으며 이 버전이 JVM(Java Virtual Machine)의 다른 구현에서 실행된다는 것입니다.

JNI는 JDK 릴리스 1.1에서 처음 지원되었습니다. 그러나 내부적으로 JDK 릴리스 1.1은 여전히 이전 스타일의 네이티브 메서드(JDK 릴리스 1.0에서와 같이)를 사용하여 Java API를 구현합니다. Java 2 SDK 릴리스 1.2(이전의 JDK 릴리스 1.2)에서는 더 이상 그렇지 않습니다. 기본 메서드는 JNI 표준을 준수하도록 다시 작성되었습니다.

JNI는 모든 JVM(Java Virtual Machine) 구현에서 지원하는 기본 인터페이스입니다. JDK 릴리스 1.1부터 JNI로 프로그래밍해야 합니다. 이전 스타일의 기본 메소드 인터페이스는 Java 2 SDK 릴리스 1.2에서 계속 지원되지만 향후 고급 JVM(Java Virtual Machine) 구현에서는 지원되지 않을 것입니다(지원되지 않을 수도 있습니다).

Java 2 SDK 릴리스 1.2에는 많은 JNI 개선 사항이 포함되어 있습니다. 그만큼 향상된 기능은 이전 버전과 호환됩니다. 향후 JNI의 모든 발전은 완전한 바이너리 호환성을 유지할 것입니다.

1.6 예제 프로그램

이 책에는 JNI 기능을 보여주는 수많은 예제 프로그램이 포함되어 있습니다.

예제 프로그램은 일반적으로 다음 형식으로 작성된 여러 코드 세그먼트로 구성됩니다.

Java 프로그래밍 언어 및 C 또는 C++ 네이티브 코드. 때때로 원시 코드는 Solaris 및 Win32의 호스트 특별 기능을 나타냅니다. 또한 JDK 및 Java 2 SDK 릴리스와 함께 제공되는 명령줄 도구(예: javah)를 사용하여 JNI 프로그램을 빌드하는 방법도 보여줍니다.

JNI의 사용은 특정 호스트 환경이나 특정 애플리케이션 개발 도구로 제한되지 않습니다. 이 책은 코드를 작성하고 실행하는 데 사용되는 도구가 아니라 코드 작성에 중점을 둡니다. JDK 및 Java 2 SDK 릴리스와 함께 제공되는 명령줄 도구는 다소 원시적입니다. 타사 도구는 JNI를 사용하는 애플리케이션을 빌드하는 향상된 방법을 제공할 수 있습니다. 선택한 개발 도구와 함께 번들로 제공되는 JNI 관련 문서를 참조하는 것이 좋습니다.

다음 웹 주소에서 이 책에 있는 예제의 소스 코드와 이 책의 최신 업데이트를 다운로드할 수 있습니다.

<http://java.sun.com/docs/books/jni/>

2

장

시작하기

이것

이 장에서는 Java Native Interface를 사용하는 간단한 예를 안내합니다. "Hello World!"를 인쇄하기 위해 C 함수를 호출하는 Java 애플리케이션을 작성할 것입니다.

2.1 개요

그림 2.1은 "Hello World!"를 인쇄하기 위해 C 함수를 호출하는 간단한 Java 애플리케이션을 작성하기 위해 JDK 또는 Java 2 SDK 릴리스를 사용하는 프로세스를 보여줍니다. 프로세스는 다음 단계로 구성됩니다.

1. 네이티브 메서드를 선언하는 클래스 (`HelloWorld.java`)를 만듭니다.
2. `javac`를 사용하여 `HelloWorld` 소스 파일을 컴파일하면 클래스 파일 `HelloWorld.class`가 생성됩니다. `javac` 컴파일러는 JDK 또는 Java 2 SDK 릴리스와 함께 제공됩니다.
3. `javah -jni`를 사용하여 네이티브 메서드 구현을 위한 함수 프로토타입이 포함된 C 헤더 파일 (`HelloWorld.h`)을 생성합니다. `javah` 도구는 JDK 또는 Java 2 SDK 릴리스와 함께 제공됩니다.
4. 네이티브 메서드의 C 구현 (`HelloWorld.c`)을 작성합니다.
5. C 구현을 네이티브 라이브러리로 컴파일하여 `HelloWorld.dll` 또는 `libHelloWorld.so`를 만듭니다. 호스트 환경에서 사용 가능한 C 컴파일러 및 링커를 사용하십시오.
6. Java 런타임 인터프리터를 사용하여 `HelloWorld` 프로그램을 실행합니다. 클래스 파일 (`HelloWorld.class`)과 기본 라이브러리 (`HelloWorld.dll` 또는 `libHelloWorld.so`)는 모두 런타임에 로드됩니다.

이 장의 나머지 부분에서는 이러한 단계를 자세히 설명합니다.

2.1

개요

시작하기

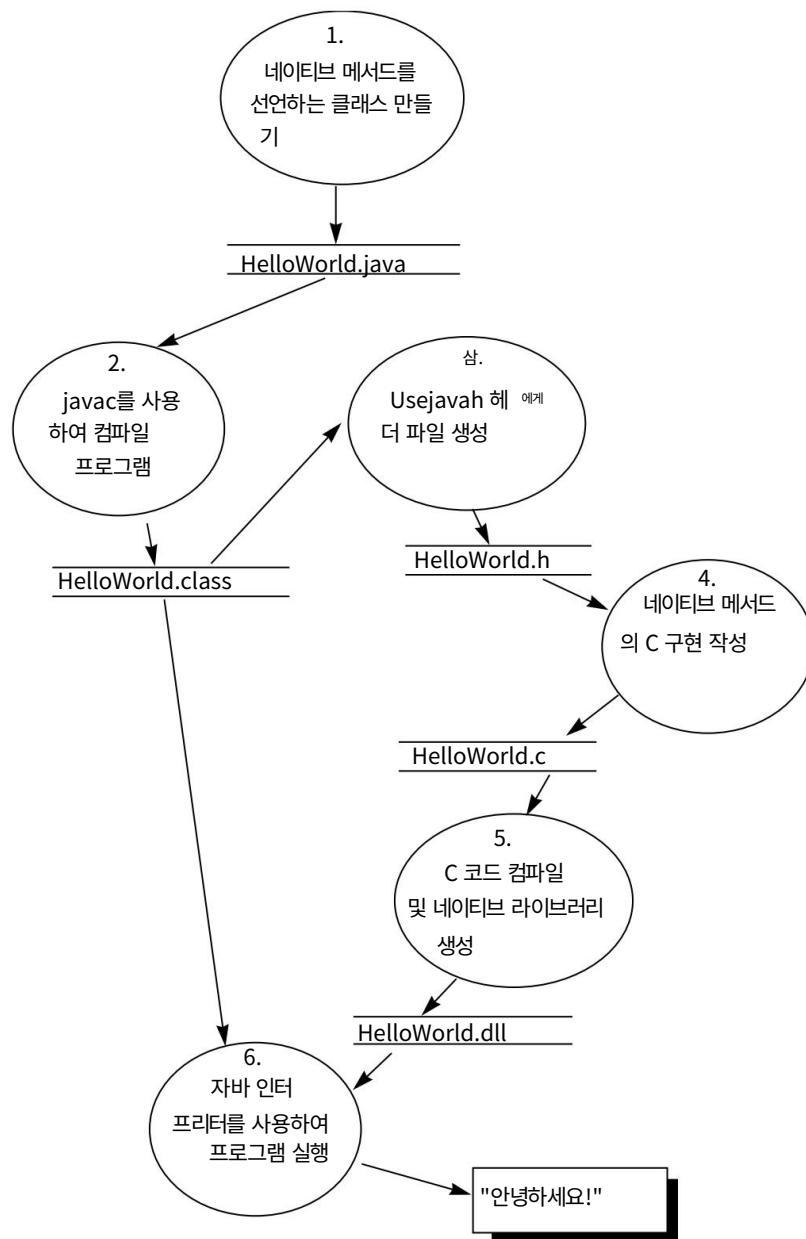


그림 2.1 "Hello World" 프로그램 작성 및 실행 단계

시작하기

네이티브 메서드 선언

2.2

2.2 네이티브 메서드 선언

Java 프로그래밍 언어로 다음 프로그램을 작성하는 것으로 시작합니다.
프로그램은 네이티브 메서드인 print를 포함하는 HelloWorld 라는 클래스를 정의합니다.

```
클래스 HelloWorld { 개인 기
    본 무효 인쇄(); public static void
    main(String[] args) { new HelloWorld().print();

}
정적
{ System.loadLibrary("HelloWorld");
}
}
```

HelloWorld 클래스 정의는 인쇄 네이티브 메서드 의 선언으로 시작됩니다 . 그 다음에는 Hello World 클래스를 인스턴스화 하고 이 인스턴스에 대한 기본 인쇄 메서드를 호출하는 기본 메서드가 옵니다 . 클래스 정의의 마지막 부분은 인쇄 네이티브 메서드의 구현을 포함하는 네이티브 라이브러리를 로드하는 정적 초기화 프로그램입니다.

print 와 같은 네이티브 메서드 선언 과 Java 프로그래밍 언어의 일반 메서드 선언 사이에는 두 가지 차이점이 있습니다.

네이티브 메서드 선언에는 네이티브 한정자가 포함되어야 합니다. 기본 한정자는 이 메서드가 다른 언어로 구현되었음을 나타냅니다. 또한 네이티브 메서드 선언은 클래스 자체에 네이티브 메서드에 대한 구현이 없기 때문에 명령문 종결 기호인 세미콜론으로 끝납니다. 별도의 C 파일에서 인쇄 방법을 구현할 것입니다 .

기본 메소드인 print를 호출하기 전에 인쇄를 구현하는 기본 라이브러리를 로드해야 합니다. 이 경우 HelloWorld 클래스의 정적 초기화 프로그램에서 네이티브 라이브러리를 로드합니다. JVM(Java Virtual Machine)은 HelloWorld 클래스의 메서드를 호출하기 전에 정적 초기화 프로그램을 자동으로 실행하므로 인쇄 네이티브 메서드 가 호출되기 전에 네이티브 라이브러리가 로드되도록 합니다 .

HelloWorld 클래스를 실행할 수 있도록 기본 메서드를 정의합니다 . Hello World.main은 일반 메소드를 호출하는 것과 동일한 방식으로 기본 메소드 인 print를 호출합니다.

System.loadLibrary는 라이브러리 이름을 가져와서 해당 이름에 해당하는 네이티브 라이브러리를 찾은 다음 네이티브 라이브러리를 애플리케이션에 로드합니다. 이 책의 뒷부분에서 정확한 로딩 프로세스에 대해 논의할 것입니다. 지금은 System.loadLibrary("HelloWorld")가 성공하려면 Win32에서 HelloWorld.dll 또는 Solaris에서 libHelloWorld.so 라는 기본 라이브러리를 만들어야 한다는 점만 기억하십시오.

2.3 HelloWorld 클래스 컴파일

시작하기

2.3 HelloWorld 클래스 컴파일

HelloWorld 클래스를 정의한 후 HelloWorld.java라는 파일에 소스 코드를 저장합니다. 그런 다음 JDK 또는 Java 2 SDK 릴리스와 함께 제공되는 javac 컴파일러를 사용하여 소스 파일을 컴파일합니다.

```
javac HelloWorld.java
```

이 명령은 현재 디렉터리에 HelloWorld.class 파일을 생성합니다.

2.4 네이티브 메서드 헤더 파일 만들기

다음으로 javah 도구를 사용하여 C에서 네이티브 메서드를 구현할 때 유용한 JNI 스타일 헤더 파일을 생성합니다. 다음과 같이 Hello World 클래스에서 javah를 실행할 수 있습니다.

```
javah -jni HelloWorld
```

헤더 파일의 이름은 끝에 ".h"가 추가된 클래스 이름입니다. 위에 표시된 명령은 HelloWorld.h라는 파일을 생성합니다. 여기서는 생성된 헤더 파일 전체를 나열하지 않습니다. 헤더 파일의 가장 중요한 부분은 HelloWorld.print 메서드를 구현하는 C 함수인 Java_HelloWorld_print의 함수 프로토타입입니다.

```
JNIEXPORT 무효 JNICALL
Java_HelloWorld_print(JNIEnv *, jobject);
```

지금은 JNIEXPORT 및 JNICALL 매크로를 무시하십시오. 네이티브 메서드의 해당 선언이 인수를 허용하지 않는데도 네이티브 메서드의 C 구현이 두 개의 인수를 허용한다는 사실을 눈치채셨을 것입니다.

모든 네이티브 메서드 구현의 첫 번째 인수는 JNIEnv 인터페이스 포인터입니다. 두 번째 인수는 HelloWorld 객체 자체에 대한 참조입니다 (C++의 "this" 포인터와 비슷함). 이 책의 뒷부분에서 JNIEnv 인터페이스 포인터와 jobject 인수를 사용하는 방법에 대해 설명 하지만 이 간단한 예에서는 두 인수를 모두 무시합니다.

[시작하기](#)[C 소스 컴파일 및 네이티브 라이브러리 만들기](#)

2.6

2.5 네이티브 메서드 구현 작성

`javah`에 의해 생성된 JNI 스타일 헤더 파일은 네이티브 메서드에 대한 C 또는 C++ 구현을 작성하는 데 도움이 됩니다. 작성하는 함수는 생성된 헤더 파일에 지정된 프로토타입을 따라야 합니다. 다음과 같이 C 파일 `HelloWorld.c`에서 `HelloWorld.print` 메서드를 구현할 수 있습니다.

```
#include <jni.h>
#include <stdio.h> #include
"HelloWorld.h"

JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject obj) {

    printf("안녕하세요!\n"); 반품;

}
```

이 네이티브 메서드의 구현은 간단합니다. `printf` 함수를 사용하여 "Hello World!"라는 문자열을 표시합니다. 그런 다음 돌아옵니다. 이전에 언급한 것처럼 `JNIEnv` 포인터와 개체에 대한 참조라는 두 인수가 모두 무시됩니다.

C 프로그램에는 세 개의 헤더 파일이 포함되어 있습니다.

- `jni.h` - 이 헤더 파일은 네이티브 코드가 JNI 함수를 호출하는 데 필요한 정보를 제공합니다. 네이티브 메서드를 작성할 때 항상 이 파일을 C 또는 C++ 소스 파일에 포함해야 합니다.
- `stdio.h` — 위의 코드 스니펫에는 다음을 사용하기 때문에 `stdio.h`도 포함됩니다.
`printf` 함수 .
- `HelloWorld.h` - `javah`를 사용하여 생성한 헤더 파일입니다. 그것은 포함 `Java_HelloWorld_print` 함수의 C/C++ 프로토타입 .

2.6 C 소스 컴파일 및 네이티브 라이브러리 생성

`HelloWorld.java` 파일에서 `HelloWorld` 클래스를 만들 때 프로그램에 네이티브 라이브러리를 로드하는 코드 줄을 포함했음을 기억하십시오.

```
System.loadLibrary("HelloWorld");
```

2.7

프로그램 실행

시작하기

이제 필요한 모든 C 코드가 작성되었으므로 Hello를 컴파일해야 합니다.

World.c를 만들고 이 네이티브 라이브러리를 빌드합니다.

다양한 운영 체제는 네이티브 라이브러리를 구축하는 다양한 방법을 지원합니다.
Solaris에서 다음 명령은 libHello World.so라는 공유 라이브러리를 빌드합니다.

```
cc -G -I/java/include -I/java/include/solaris HelloWorld.c -o
libHelloWorld.so
```

-G 옵션은 일반 Solaris 실행 파일 대신 공유 라이브러리를 생성하도록 C 컴파일러에 지시합니다. 이 책의 페이지 너비 제한 때문에 명령줄을 두 줄로 나눕니다. 한 줄에 명령을 입력하거나 스크립트 파일에 명령을 배치해야 합니다. Win32에서 다음 명령은 Microsoft Visual C++ 컴파일러를 사용하여 DLL(동적 연결 라이브러리) HelloWorld.dll을 빌드합니다.

```
cl -Ic:\java\include -Ic:\java\include\win32 -MD -LD HelloWorld.c
-FeHelloWorld.dll
```

-MD 옵션은 HelloWorld.dll이 Win32 다중 스레드 C 라이브러리와 연결되도록 합니다. -LD 옵션은 일반 Win32 실행 파일 대신 DLL을 생성하도록 C 컴파일러에 지시합니다. 물론 Solaris와 Win32 모두에서 자신의 시스템 설정을 반영하는 포함 경로를 입력해야 합니다.

2.7 프로그램 실행

이제 프로그램을 실행할 준비가 된 두 가지 구성 요소가 있습니다. 클래스 파일 (HelloWorld.class)은 네이티브 메서드를 호출하고 네이티브 라이브러리 (Hello World.dll)는 네이티브 메서드를 구현합니다.

HelloWorld 클래스에는 자체 기본 메서드가 포함되어 있으므로 다음을 실행할 수 있습니다. Solaris 또는 Win32에서 다음과 같이 프로그램하십시오.

자바 HelloWorld

다음 출력이 표시되어야 합니다.

안녕하세요 세계!

프로그램을 실행하려면 기본 라이브러리 경로를 올바르게 설정하는 것이 중요합니다.
네이티브 라이브러리 경로는 Java 가상 머신이 네이티브 라이브러리를 로드할 때 검색하는 디렉터리 목록입니다. 기본 라이브러리 경로가 올바르게 설정되지 않은 경우 다음과 유사한 오류가 표시됩니다.

[시작하기](#)[프로그램 실행](#)

2.7

`java.lang.UnsatisfiedLinkError: HelloWorld.main(HelloWorld.java)의
 java.lang.System.loadLibrary(System.java)의
 java.lang.Runtime.loadLibrary(Runtime.java)의 라이브러리 경로에 HelloWorld가
 없습니다.`

기본 라이브러리가 기본 라이브러리 경로의 디렉토리 중 하나에 있는지 확인하십시오. Solaris 시스템에서 실행 중인 경우 LD_LIBRARY_PATH 환경 변수는 기본 라이브러리 경로를 정의하는 데 사용됩니다. libHelloWorld.so 파일이 포함된 디렉토리의 이름이 포함되어 있는지 확인하십시오 . libHelloWorld.so 파일이 현재 디렉토리에 있는 경우 표준 셸 (sh) 또는 KornShell (ksh) 에서 다음 두 명령을 실행하여 LD_LIBRARY_PATH 환경 변수를 올바르게 설정할 수 있습니다.

`LD_LIBRARY_PATH=.`
`LD_LIBRARY_PATH 내보내기`

C 셸 (csh 또는 tcsh) 에서 동등한 명령은 다음과 같습니다.

`setenv LD_LIBRARY_PATH .`

Windows 95 또는 Windows NT 시스템에서 실행 중인 경우 HelloWorld.dll이 현재 디렉토리나 PATH 환경 변수에 나열된 디렉토리에 있는지 확인하십시오 .

Java 2 SDK 1.2 릴리스에서 네이티브 라이브러리 경로를 지정할 수도 있습니다. java 명령줄을 다음과 같이 시스템 속성으로 지정합니다.

`자바 -Djava.library.path=. 헬로월드`

"-D" 명령줄 옵션은 Java 플랫폼 시스템 속성을 설정합니다 . java.library.path 속성을 "." 로 설정 현재 디렉토리에서 기본 라이브러리를 검색하도록 JVM(Java Virtual Machine)에 지시합니다.

파트 2: 프로그래머 가이드

3 장

기본 유형, 문자열 및 어레이

프로그래머가 Java를 인터페이스할 때 묻는 가장 일반적인 질문 중 하나

네이티브 코드를 사용하는 애플리케이션은 Java 프로그래밍 언어의 데이터 유형이 C 및 C++와 같은 네이티브 프로그래밍 언어의 데이터 유형에 매핑되는 방식입니다. "Hello World!" 이전 장에 제시된 예에서 네이티브 메서드에 인수를 전달하지 않았으며 네이티브 메서드가 결과를 반환하지도 않았습니다. 기본 메서드는 단순히 메시지를 인쇄하고 반환했습니다.

실제로 대부분의 프로그램은 기본 메서드에 인수를 전달하고 기본 메서드에서도 결과를 받아야 합니다. 이 장에서는 Java 프로그래밍 언어로 작성된 코드와 네이티브 메서드를 구현하는 네이티브 코드 간에 데이터 유형을 교환하는 방법에 대해 설명합니다. 정수와 같은 기본 유형과 문자열 및 배열과 같은 일반적인 객체 유형부터 시작하겠습니다. 임의의 객체에 대한 전체 처리는 네이티브 코드가 필드에 액세스하고 메서드를 호출하는 방법을 설명하는 다음 장으로 미루 것입니다.

3.1 간단한 네이티브 메서드

마지막 장의 HelloWorld 프로그램과 크게 다르지 않은 간단한 예제부터 시작하겠습니다. 예제 프로그램인 Prompt.java에는 문자열을 인쇄하고 사용자 입력을 기다린 다음 사용자가 입력한 줄을 반환하는 기본 메서드가 포함되어 있습니다. 이 프로그램의 소스 코드는 다음과 같습니다.

3.1.1 네이티브 메서드 구현을 위한 C 프로토타입

기본 유형, 문자열 및 배열

```
class Prompt { // 프
    룹프트를 인쇄하고 한 줄을 읽는 네이티브 메서드 private native String getLine(String
    prompt);

    public static void main(String args[]) { Prompt p = new
        Prompt(); String input = p.getLine("줄을 입력하세요:
        "); System.out.println("사용자 입력: " + input);

    }
    정적
    { System.loadLibrary("프롬프트");
    }
}
```

Prompt.main은 네이티브 메서드 Prompt.getLine을 호출하여 사용자 입력을 받습니다.
정적 이나셜라이저는 System.loadLibrary 메서드를 호출하여 Prompt라는 네이티브 라이브러리를 로드합니다.

3.1.1 네이티브 메서드 구현을 위한 C 프로토타입

Prompt.getLine 메서드는 다음 C 함수로 구현할 수 있습니다.

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject this, jstring 프롬프트);
```

javadoc 도구(§ 2.4)를 사용하여 위 함수 프로토타입을 포함하는 헤더 파일을 생성할 수 있습니다.
JNIEXPORT 및 JNICALL 매크로(jni.h 헤더 파일에 정의됨)는 이 함수가 기본 라이브러리에서 내보내지고 C 컴파일러가 이 함수에 대한 올바른 호출 규칙으로 코드를 생성하도록 합니다.

C 함수의 이름은 "Java_" 접두사, 클래스 이름 및 메서드 이름을 연결하여 구성됩니다. 섹션 11.3에는 C 함수 이름이 형성되는 방법에 대한 보다 정확한 설명이 포함되어 있습니다.

3.1.2 네이티브 메서드 인수

2.4절에서 간략하게 설명한 것처럼 Java_Prompt_getLine과 같은 기본 메소드 구현은 기본 메소드에서 선언된 인수 외에 두 개의 표준 매개변수를 허용합니다. 첫 번째 매개변수인 JNIEnv 인터페이스 포인터는 함수 테이블에 대한 포인터가 포함된 위치를 가리킵니다. 함수 테이블의 각 항목은 JNI 함수를 가리킵니다. 원시 메소드는 항상 JNI 함수 중 하나를 통해 JVM(Java Virtual Machine)의 데이터 구조에 액세스합니다. 그림 3.1은 JNIEnv 인터페이스 포인터를 보여줍니다.

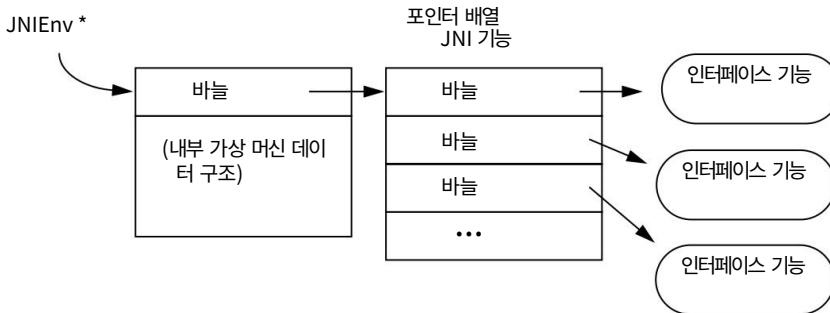


그림 3.1 JNIEnv 인터페이스 포인터

두 번째 인수는 네이티브 메서드가 정적 메서드인지 인스턴스 메서드인지에 따라 다릅니다. 인스턴스 네이티브 메서드에 대한 두 번째 인수는 C++의 `this` 포인터와 유사하게 메서드가 호출되는 개체에 대한 참조입니다. 정적 네이티브 메서드에 대한 두 번째 인수는 메서드가 정의된 클래스에 대한 참조입니다. 예제인 `Java_Prompt_getLine`은 인스턴스 네이티브 메서드를 구현합니다. 따라서 `jobject` 매개 변수는 객체 자체에 대한 참조입니다.

3.1.3 유형 매핑

네이티브 메서드 선언의 인수 유형에는 네이티브 프로그래밍 언어의 해당 유형이 있습니다. JNI는 Java 프로그래밍 언어의 유형에 대응하는 C 및 C++ 유형 세트를 정의합니다.

Java 프로그래밍 언어에는 `int`, `float` 및 `char`과 같은 기본 유형과 클래스, 인스턴스 및 배열과 같은 참조 유형의 두 가지 유형이 있습니다. Java 프로그래밍 언어에서 문자열은 `java.lang.String` 클래스의 인스턴스입니다.

JNI는 기본 유형과 참조 유형을 다르게 취급합니다. 기본 유형의 매핑은 간단합니다. 예를 들어 Java 프로그래밍 언어의 `int` 유형은 C/C++ 유형 `jint` (`jni.h`에서 부호 있는 32비트 정수로 정의됨)에 매핑되는 반면 Java 프로그래밍 언어의 `float` 유형은 C 및 C++에 매핑됩니다. 유형 `jfloat` (`jni.h`에서 32비트 부동 소수점 숫자로 정의됨). 섹션 12.1.1에는 JNI에 정의된 모든 기본 유형의 정의가 포함되어 있습니다.

JNI는 개체를 불투명 참조로 네이티브 메서드에 전달합니다. 불투명 참조는 JVM(Java Virtual Machine)의 내부 데이터 구조를 참조하는 C 포인터 유형입니다. 그러나 내부 데이터 구조의 정확한 레이아웃은 프로그래머에게 숨겨져 있습니다. 네이티브 코드는 다음을 통해 기본 개체를 조작해야 합니다.

JNIEnv 인터페이스 포인터를 통해 사용할 수 있는 적절한 JNI 함수 . 예를 들어 java.lang.String 에 해당하는 JNI 유형은 jstring 입니다 . jstring 참조의 정확한 값은 기본 코드와 관련이 없습니다.

기본 코드는 GetStringUTFChars (§ 3.2.1)와 같은 JNI 함수를 호출하여 문자열의 내용에 액세스합니다.

모든 JNI 참조에는 jobject 유형이 있습니다. 편의성과 향상된 유형 안전성을 위해 JNI는 개념적으로 jobject의 "하위 유형"인 일련의 참조 유형을 정의합니다 . (A 는 A 의 모든 인스턴스가 B 의 인스턴스이기도 한 B 의 하위 유형입니다 .)

이러한 하위 유형은 Java 프로그래밍 언어에서 자주 사용되는 참조 유형에 해당합니다. 예를 들어 jstring은 문자열을 나타냅니다. jobjectArray는 객체의 배열을 나타냅니다. 섹션 12.1.2에는 JNI 참조 유형 및 하위 유형 관계의 전체 목록이 포함되어 있습니다.

3.2 문자열 접근

Java_Prompt_getLine 함수는 프롬프트 인수를 jstring 유형으로 받습니다 . jstring 유형은 JVM (Java Virtual Machine)의 문자열을 나타내며 일반 C 문자열 유형(문자에 대한 포인터, char *)과 다릅니다 . jstring을 일반 C 문자열로 사용할 수 없습니다 . 다음 코드를 실행하면 원하는 결과가 생성되지 않습니다. 실제로 Java 가상 머신이 충돌할 가능성이 높습니다.

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt) {
    /* 오류: jstring을 char* 포인터로 잘못 사용 */ printf("%s", prompt);
    ...
}
```

3.2.1 네이티브 문자열로 변환

네이티브 메서드 코드는 적절한 JNI 함수를 사용하여 jstring 개체를 C/C++ 문자열로 변환해야 합니다. JNI는 유니코드 및 UTF-8 문자열과의 변환을 모두 지원합니다. 유니코드 문자열은 문자를 16비트 값으로 나타내는 반면 UTF-8 문자열(§ 12.3.1)은 7비트 ASCII 문자열과 상향 호환되는 인코딩 체계를 사용합니다. UTF-8 문자열은 ASCII가 아닌 문자를 포함하더라도 NULL로 끝나는 C 문자열처럼 작동합니다. 값이 1에서 127 사이인 모든 7비트 ASCII 문자는 UTF-8 인코딩에서 동일하게 유지됩니다. 가장 높은 비트 세트가 있는 바이트는 멀티바이트 인코딩된 16비트 유니코드 값의 시작을 알립니다.

`Java_Prompt_getLine` 함수는 JNI 함수 `GetStringUTFChars`를 호출 하여 문자열의 내용을 읽습니다. `GetStringUTFChars` 함수는 `JNIEnv` 인터페이스 포인터를 통해 사용할 수 있습니다. 일반적으로 JVM(Java Virtual Machine) 구현에서 유니코드 시퀀스로 표시되는 `jstring` 참조를 UTF-8 형식으로 표시되는 C 문자열로 변환합니다. 원래 문자열에 7비트 ASCII 문자만 포함되어 있고 확신하는 경우 변환된 문자열을 `printf`와 같은 일반 C 라이브러리 함수에 전달할 수 있습니다. (비 ASCII 문자열을 처리하는 방법은 섹션 8.2에서 논의할 것입니다.)

```

JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt) {

    char buf[128]; const
    jbyte *str; str = (*env)->GetStringUTFChars(env, prompt, NULL); if (str == NULL) { NULL 반환; /* OutOfMemoryError가 이미 발생했습니다. */

    }

    printf("%s", str); (*env)->ReleaseStringUTFChars(env, prompt, str);
    /* 여기서는 사용자가 * 127자 이상 입력하지 않는다고 가정합니다. */ scanf("%s", buf);
    return (*env)->NewStringUTF(env, buf);

}

```

`GetStringUTFChars`의 반환 값을 확인하는 것을 잊지 마십시오. JVM(Java Virtual Machine) 구현은 UTF-8 문자열을 보유하기 위해 메모리를 할당해야 하므로 메모리 할당이 실패할 가능성이 있습니다. 이 경우 `GetStringUTFChars`는 NULL을 반환하고 `OutOfMemoryError` 예외를 발생시킵니다. 6장에서 배우겠지만 JNI를 통해 예외를 던지는 것은 Java 프로그래밍 언어에서 예외를 던지는 것과 다릅니다. JNI를 통해 발생한 보류 중인 예외는 네이티브 C 코드의 제어 흐름을 자동으로 변경하지 않습니다. 대신 C 함수의 나머지 문을 건너뛰기 위해 명시적인 `return` 문을 발행해야 합니다. `Java_Prompt_getLine`이 반환된 후 `Prompt.getLine` 네이티브 메서드의 호출자인 `Prompt.main`에서 예외가 발생합니다.

3.2.2 네이티브 문자열 리소스 해제

네이티브 코드가 `GetStringUTFChars`를 통해 얻은 UTF-8 문자열 사용을 마치면 `ReleaseStringUTFChars`를 호출합니다. `ReleaseStringUTFChars`를 호출하면 네이티브 메서드에 더 이상 UTF-8 문자열이 필요하지 않음을 나타냅니다.

3.2.3 새 문자열 구성

기본 유형, 문자열 및 배열

GetStringUTFChars에 의해 반환됨 ; 따라서 UTF-8 문자열이 차지하는 메모리를 해제할 수 있습니다. ReleaseStringUTFChars를 호출하지 못하면 메모리 누수가 발생하여 궁극적으로 메모리 고갈로 이어질 수 있습니다.

3.2.3 새 문자열 구성

JNI 함수 NewStringUTF를 호출하여 원시 메소드에서 새 java.lang.String 인스턴스를 생성할 수 있습니다 . NewStringUTF 함수는 UTF-8 형식의 C 문자열을 사용하여 java.lang.String 인스턴스를 구성합니다 . 새로 구성된 java.lang.String 인스턴스는 주어진 UTF-8 C 문자열과 동일한 유니코드 문자 시퀀스를 나타냅니다.

가상 머신이 java.lang.String 인스턴스를 구성하는 데 필요한 메모리를 할당할 수 없는 경우 NewStringUTF는 OutOfMemoryError 예외를 발생시키고 NULL을 반환합니다 . 이 예제에서는 네이티브 메서드가 즉시 반환되기 때문에 반환 값을 확인할 필요가 없습니다. NewStringUTF가 실패하면 네이티브 메서드 호출을 실행한 Prompt.main 메서드에서 OutOfMemoryError 예외가 발생합니다 . NewStringUTF가 성공하면 새로 생성된 java.lang.String 인스턴스에 대한 JNI 참조를 반환합니다 . 새 인스턴스는 Prompt.getLine에 의해 반환된 다음 Prompt.main의 로컬 변수 입력에 할당됩니다 .

3.2.4 기타 JNI 문자열 함수

JNI는 이전에 소개된 GetStringUTFChars, ReleaseStringUTFChars 및 NewStringUTF 함수 외에도 여러 다른 문자열 관련 함수를 지원합니다 .

GetStringChars 및 ReleaseStringChars는 유니코드 형식으로 표시되는 문자열 문자를 가져옵니다. 이러한 함수는 예를 들어 운영 체제가 고유 문자열 형식으로 유니코드를 지원하는 경우에 유용합니다.

UTF-8 문자열은 항상 '\0' 문자로 끝나는 반면 Uni 코드 문자열은 그렇지 않습니다. jstring 참조에서 유니코드 문자 수를 찾기 위해 JNI 프로그래머는 GetStringLength을 호출할 수 있습니다. UTF-8 형식으로 jstring을 나타내는 데 필요한 바이트 수를 확인하기 위해 JNI 프로그래머는 GetStringUTFChars의 결과에서 ANSI C 함수 strlen을 호출하거나 jstring 참조에서 JNI 함수 GetStringUTFLength를 직접 호출할 수 있습니다.

GetStringChars 및 GetStringUTFChars에 대한 세 번째 인수에는 추가 설명이 필요합니다.

```
const jchar *
GetStringChars(JNIEnv *env, jstring str, jboolean *isCopy);
```

GetStringChars에서 반환할 때 반환된 문자열이 원래 java.lang.String 인스턴스에 있는 문자의 복사본인 경우 isCopy가 가리키는 메모리 위치는 JNI_TRUE로 설정됩니다. 반환된 문자열이 원래 java.lang.String 인스턴스의 문자에 대한 직접 포인터인 경우 isCopy가 가리키는 메모리 위치는 JNI_FALSE로 설정됩니다. isCopy가 가리키는 위치가 JNI_FALSE로 설정된 경우 네이티브 코드는 반환된 문자열의 내용을 수정하면 안 됩니다. 이 규칙을 위반하면 원래 java.lang.String 인스턴스도 수정됩니다. 이것은 java.lang.String 인스턴스가 불변이라는 불변성을 깨뜨립니다.

자바 가상 머신이 java.lang.String 인스턴스에 있는 문자의 복사본을 반환하는지 원본에 대한 직접 포인터를 반환하는지 상관하지 않기 때문에 대부분 isCopy 인수로 NULL을 전달합니다.

일반적으로 가상 머신이 지정된 java.lang.String 인스턴스의 문자를 복사할지 여부를 예측할 수 없습니다. 따라서 프로그래머는 GetStringChars와 같은 기능이 java.lang.String 인스턴스의 문자 수에 비례하여 시간과 공간이 소요될 수 있다고 가정해야 합니다. 일반적인 JVM(Java Virtual Machine) 구현에서 가비지 수집기는 힙에서 개체를 재배치합니다. java.lang.String 인스턴스에 대한 직접 포인터가 네이티브 코드로 다시 전달되면 가비지 수집기는 더 이상 java.lang.String 인스턴스를 재배치할 수 없습니다. 달리 말하면 가상 머신은 java.lang.String 인스턴스를 고정해야 합니다. 과도한 고정은 메모리 조각화로 이어지기 때문에 가상 머신 구현은 재량에 따라 각 개별 GetStringChars 호출에 대해 문자를 복사하거나 인스턴스를 고정하도록 결정할 수 있습니다.

GetStringChars에서 반환된 문자열 요소에 더 이상 액세스할 필요가 없으면 ReleaseStringChars를 호출하는 것을 잊지 마십시오. ReleaseStringChars 호출은 GetStringChars가 *isCopy를 JNI_TRUE 또는 JNI_FALSE로 설정했는지 여부에 관계없이 필요합니다. ReleaseStringChars는 GetStringChars가 복사본을 반환했는지 여부에 따라 복사본을 해제하거나 인스턴스를 고정 해제합니다.

3.2.5 Java 2 SDK 릴리스 1.2의 새로운 JNI 문자열 함수

가상 머신이 java.lang.String 인스턴스의 문자에 대한 직접 포인터를 반환할 수 있는 가능성을 높이기 위해 Java 2 SDK 릴리스 1.2에서는 새로운 함수 쌍인 Get/ReleaseStringCritical을 도입했습니다. 표면적으로 둘 다 가능한 경우 문자에 대한 포인터를 반환한다는 점에서 Get/ReleaseStringChars 함수와 유사해 보입니다. 그렇지 않으면 복사본이 만들어집니다. 그러나 이러한 기능을 사용할 수 있는 방법에는 상당한 제한이 있습니다.

이 함수 쌍 내부의 코드를 "중요 영역"에서 실행되는 것으로 취급해야 합니다. 임계 영역 내에서 네이티브 코드는 임의의 JNI 함수 또는 현재 스레드를 차단하고 기다릴 수 있는 네이티브 함수를 호출해서는 안 됩니다.

3.2.5 Java 2 SDK 릴리스 1.2의 새로운 JNI 문자열 함수

기본 유형, 문자열 및 배열

Java 가상 머신에서 실행 중인 다른 스레드. 예를 들어, 현재 스레드는 다른 스레드가 쓰고 있는 I/O 스트림에서 입력을 기다리면 안 됩니다.

이러한 제한으로 인해 네이티브 코드가 GetStringCritical을 통해 얻은 문자열 요소에 대한 직접 포인터를 보유하고 있을 때 가상 머신이 가비지 수집을 비활성화할 수 있습니다. 가비지 수집이 비활성화되면 가비지 수집을 트리거하는 다른 모든 스레드도 차단됩니다. Get/ReleaseStringCritical 쌍 사이의 네이티브 코드는 차단 호출을 실행하거나 JVM(Java Virtual Machine)에서 새 개체를 할당하면 안 됩니다. 그렇지 않으면 가상 머신이 교착 상태에 빠질 수 있습니다. 다음 시나리오를 고려하십시오.

- 다른 스레드에 의해 트리거된 가비지 수집은 현재 스레드가 차단 호출을 완료하고 가비지 수집을 다시 활성화할 때까지 진행할 수 없습니다. • 한편, 블로킹 호출은 가비지 수집을 수행하기 위해 대기 중인 다른 스레드가 이미 보유하고 있는 잠금을 획득해야 하기 때문에 현재 스레드는 진행할 수 없습니다.

여러 쌍의 GetStringCritical 및 Release StringCritical 함수를 겹치는 것이 안전합니다 . 예를 들어:

```
jchar *s1, *s2; s1 =
(*env)->GetStringCritical(env, jstr1); if (s1 == NULL) { ... /* 오류
처리 */
}

s2 = (*env)->GetStringCritical(env, jstr2); if (s2 == NULL)
{ (*env)->ReleaseStringCritical(env, jstr1, s1); ... /* 오류 처리 */

}

...
/* s1 및 s2 사용 */ (*env)-
>ReleaseStringCritical(env, jstr1, s1); (*env)-
>ReleaseStringCritical(env, jstr2, s2);
```

Get /ReleaseStringCritical 쌍은 스택 순서에 엄격하게 중첩될 필요가 없습니다. VM이 내부적으로 다른 for mat의 배열을 나타내는 경우 GetStringCritical이 여전히 버퍼를 할당하고 배열의 복사본을 만들 수 있기 때문에 가능한 메모리 부족 상황에 대해 NULL에 대한 반환 값을 확인하는 것을 잊지 말아야 합니다 . 예를 들어, JVM(Java Virtual Machine)은 배열을 연속적으로 저장하지 않을 수 있습니다. 이 경우 GetStringCritical은 네이티브 코드에 연속적인 문자 배열을 반환하기 위해 jstring 인스턴스의 모든 문자를 복사해야 합니다 .

교착 상태를 방지하려면 네이티브 코드가 GetStringCritical 호출을 실행한 후 해당 ReleaseStringCritical 호출을 수행하기 전에 임의의 JNI 함수를 호출하지 않도록 해야 합니다 . 유일한 JNI 기능

"중요 영역"에서 허용되는 것은 중첩된 Get/ReleaseStringCritical 및 Get/ReleasePrimitiveArrayCritical (§ 3.3.2) 호출입니다.

JNI는 GetStringUTFCritical 및 ReleaseStringUTF Critical 기능을 지원하지 않습니다. 이러한 기능은 가상 머신 구현이 거의 확실하게 문자열을 내부적으로 유니코드 형식으로 나타내기 때문에 가상 머신이 문자열을 복사해야 할 가능성이 높습니다.

Java 2 SDK 릴리스 1.2에 대한 다른 추가 사항은 GetStringRegion 및 GetStringUTFRegion입니다. 이러한 함수는 문자열 요소를 미리 할당된 버퍼에 복사합니다. Prompt.getLine 메서드는 다음과 같이 GetStringUTFRegion을 사용하여 다시 구현할 수 있습니다.

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring 프롬프트) {
    /* 프롬프트 문자열과 사용자 입력이 128보다 작다고 가정합니다.
       문자 */ char
    outbuf[128], inbuf[128]; int len = (*env)->GetStringLength(env, 프롬프트); (*env)->GetStringUTFRegion(env, prompt, 0, len, outbuf); printf("%s", outbuf); scanf("%s", inbuf); return (*env)->NewStringUTF(env, inbuf);
}
```

GetStringUTFRegion 함수는 유니코드 문자 수로 계산되는 시작 인덱스와 길이를 사용합니다. 이 함수는 경계 검사도 수행하고 필요한 경우 StringIndexOutOfBoundsException을 발생시킵니다. 위의 코드에서는 문자열 참조 자체에서 길이를 얻었으므로 인덱스 오버플로가 없을 것이라고 확신합니다. (그러나 위의 코드에는 프롬프트 문자열이 128자 미만인지 확인하는 데 필요한 검사가 없습니다.)

이 코드는 GetStringUTFChars를 사용하는 것보다 다소 간단합니다. GetStringUTFRegion은 메모리 할당을 수행하지 않기 때문에 가능한 메모리 부족 조건을 확인할 필요가 없습니다. (다시 말 하지만, 위의 코드에는 사용자 입력이 128자 미만인지 확인하는 데 필요한 검사가 없습니다.)

3.2.6 JNI 문자열 함수 요약

표 3.1에는 모든 문자열 관련 JNI 함수가 요약되어 있습니다. Java 2 SDK 1.2 릴리스에는 특정 문자열 작업의 성능을 향상시키는 여러 가지 새로운 기능이 추가되었습니다. 추가된 기능은 성능 향상을 가져오는 것 외에 새로운 작업을 지원하지 않습니다.

3.2.6 JNI 문자열 함수 요약

기본 유형, 문자열 및 배열

표 3.1 JNI 문자열 함수 요약

JNI 함수	설명	부터
GetStringChars ReleaseStringChars	mat에 대한 유니코드 문자열의 내용에 대한 포인터를 얻거나 해제합니다. 문자열의 복사본을 반환할 수 있습니다.	JDK1.1
GetStringUTFChars ReleaseStringUTFChars	UTF-8 형식의 문자열 내용에 대한 포인터를 얻거나 해제합니다. 문자열의 복사본을 반환할 수 있습니다.	JDK1.1
GetStringLength	문자열의 유니코드 문자 수를 반환합니다.	JDK1.1
GetStringUTF길이	UTF-8 형식의 문자열을 나타내는 데 필요한 바이트 수(후행 0 제외)를 반환합니다.	JDK1.1
새 문자열	지정된 유니코드 C 문자열과 동일한 문자 시퀀스를 포함하는 java.lang.String 인스턴스를 만듭니다 .	JDK1.1
NewStringUTF	지정된 UTF-8 인코딩된 C 문자열과 동일한 문자 시퀀스를 포함하는 java.lang.String 인스턴스를 만듭니다 .	JDK1.1
GetStringCritical ReleaseStringCritical	유니코드 형식의 문자열 내용에 대한 포인터를 가져옵니다. 문자열의 복사본을 반환할 수 있습니다. 네이티브 코드는 한 쌍의 Get/ReleaseStringCritical 호출 사이를 차단해서는 안 됩니다.	자바 2 SDK1.2
GetStringRegion SetStringRegion	문자열의 내용을 유니코드 형식으로 미리 할당된 C 버퍼로 또는 그 버퍼에서 복사합니다.	자바 2 SDK1.2
GetStringUTFRegion SetStringUTFRegion	문자열 내용을 UTF-8 형식으로 미리 할당된 C 버퍼로 복사하거나 버퍼에서 복사합니다.	자바 2 SDK1.2

3.2.7 문자열 함수 중에서 선택

그림 3.2는 프로그래머가 JDK 릴리스 1.1 및 Java 2 SDK 릴리스 1.2의 문자열 관련 함수 중에서 선택할 수 있는 방법을 보여줍니다.

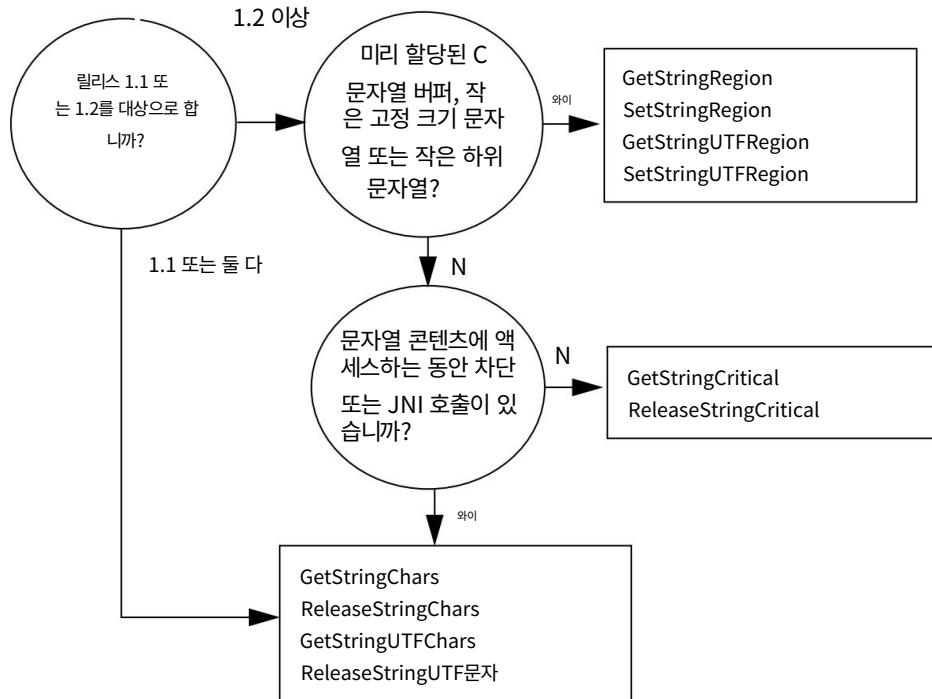


그림 3.2 JNI 문자열 함수 중에서 선택

1.1 또는 1.1 및 1.2 릴리스를 모두 대상으로 하는 경우 다른 선택의 여지가 없습니다.
Get/ReleaseStringChars 및 Get/ReleaseStringUTFChars 보다 .
Java 2 SDK 릴리스 1.2 이상에서 프로그래밍 중이고 문자열의 내용을 이미 할당된 C 버퍼에 복사하려면 GetString Region 또는 GetStringUTFRegion을 사용하십시오.

작은 고정 크기 문자열의 경우 Get/SetStringRegion 및 Get/SetStringUTFRegion이 거의 항상 선호되는 함수입니다. 왜냐하면 C 버퍼가 C 스택에 매우 저렴하게 할당될 수 있기 때문입니다. 문자열에서 적은 수의 문자를 복사하는 오버헤드는 무시할 수 있습니다.

Get/SetStringRegion 및 Get/SetStringUTFRegion의 장점 중 하나는 메모리 할당을 수행하지 않으므로 예기치 않은 오류가 발생하지 않는다는 것입니다.

3.2.7 문자열 함수 중에서 선택

기본 유형, 문자열 및 배열

메모리 부족 예외. 인덱스 오버플로우가 발생하지 않도록 하려면 예외 검사가 필요하지 않습니다.

Get/SetStringRegion 및 Get/SetStringUTFRegion 의 또 다른 장점은 시작 인덱스와 문자 수를 지정할 수 있다는 것입니다. 이러한 함수는 네이티브 코드가 긴 문자열의 문자 하위 집합에만 액세스하면 되는 경우에 적합합니다.

GetStringCritical은 매우 주의해서 사용해야 합니다(§ 3.2.5). GetStringCritical을 통해 얻은 포인터를 유지하는 동안 네이티브 코드가 JVM(Java Virtual Machine)에 새 개체를 할당하거나 시스템을 교착 상태에 빠뜨릴 수 있는 다른 차단 호출을 수행하지 않는지 확인해야 합니다.

다음은 GetStringCritical 사용 시 미묘한 문제를 보여주는 예입니다 . 다음 코드는 문자열의 내용을 가져오고 fprintf 함수를 호출하여 파일 핸들 fd 에 문자를 기록합니다 .

```
/* 이것은 안전하지 않습니다! */ const
char *c_str = (*env)->GetStringCritical(env, j_str, 0); if (c_str == NULL) { ... /* 오류 처리 */

}

fprintf(fd, "%s\n", c_str); (*env)-
>ReleaseStringCritical(env, j_str, c_str);
```

위 코드의 문제점은 현재 스레드에 의해 가비지 수집이 비활성화된 경우 파일 핸들에 쓰는 것이 항상 안전하지 않다는 것입니다. 예를 들어 다른 스레드 T가 fd 파일 핸들에서 읽기를 기다리고 있다고 가정합니다 . 또한 운영 체제 버퍼링이 스레드 T가 fd 에서 보류 중인 모든 데이터 읽기를 완료할 때까지 fprintf 호출이 대기 하는 방식으로 설정되어 있다고 가정해 보겠습니다 . 교착 상태에 대한 가능한 시나리오를 구성했습니다. 스레드 T가 파일 핸들에서 읽기 위한 버퍼 역할을 할 충분한 메모리를 할당할 수 없는 경우 가비지 수집을 요청해야 합니다. 가비지 수집 요청은 현재 스레드가 ReleaseStringCritical 을 실행할 때까지 차단되며 fprintf 호출이 반환될 때까지 발생할 수 없습니다 . 그러나 fprintf 호출은 스레드 T가 파일 핸들에서 읽기를 완료하기를 기다리고 있습니다 .

다음 코드는 위의 예와 유사하지만 교착 상태가 거의 없는 것이 거의 확실합니다.

```
/* 이 코드 세그먼트는 정상입니다. */ const char
*c_str = (*env)->GetStringCritical(env, j_str, 0); if (c_str == NULL) { ... /* 오류 처리 */

}

DrawString(c_str); (*env)-
>ReleaseStringCritical(env, j_str, c_str);
```

`DrawString` 은 문자열을 화면에 직접 쓰는 시스템 호출입니다.
 스크린 디스플레이 드라이버가 동일한 가상 머신에서 실행되는 Java 애플리케이션이 아닌 한
`DrawString` 기능은 가비지 수집이 발생하기를 무한정 기다리지 않습니다.

요약하면 한 쌍의 `Get/ReleaseStringCritical` 호출 간에 가능한 모든 차단 동작을 고려해야 합니다.

3.3 배열에 접근하기

JNI는 기본 배열과 객체 배열을 다르게 취급합니다. 기본 배열에는 `int` 및 `boolean`과 같은 기본 유형의 요소가 포함됩니다. 객체 배열에는 클래스 인스턴스 및 기타 배열과 같은 참조 유형의 요소가 포함됩니다. 예를 들어 Java 프로그래밍 언어로 작성된 다음 코드 세그먼트에서:

```
int[] iarr; float[]
파르;
Object[] 노; int[][] arr2;
```

`iarr` 및 `farr`은 기본 배열인 반면 `oarr` 및 `arr2`는 객체 배열입니다.

기본 메서드에서 기본 배열에 액세스하려면 문자열 액세스에 사용되는 것과 유사한 JNI 함수를 사용해야 합니다. 간단한 예를 살펴보겠습니다.

다음 프로그램은 `int` 배열의 내용을 더하는 네이티브 메서드 `sumArray`를 호출합니다.

```
클래스 IntArray { 개인 네이
  티브 int sumArray(int[] arr); 공개 정적 무효 메인(문자열[] 인수)
  {
    IntArray p = new IntArray(); int arr[] = 새로
    운 int[10]; for (int i = 0; i < 10; i++) { arr[i]
    = i;

    }
    정수 합계 = p.sumArray(arr);
    System.out.println("합계 = " + 합계);
  }
  정적
  { System.loadLibrary("IntArray");
  }
}
```

3.3.1 C에서 배열에 액세스하기

기본 유형, 문자열 및 배열

3.3.1 C에서 배열에 접근하기

배열은 jarray 참조 유형 및 jintArray와 같은 "하위 유형"으로 표시됩니다. jstring이 C 문자열 유형이 아닌 것처럼 jarray도 C 배열 유형이 아닙니다. jarray 참조를 통해 간접적으로 Java_IntArray_sumArray 기본 메서드를 구현할 수 없습니다. 다음 C 코드는 불법이며 원하는 결과를 생성하지 않습니다.

```
/* 이 프로그램은 불법입니다! */ JNIEXPORT jint
JNICALL Java_IntArray_sumArray(JNIEnv *
*env, jobject obj, jintArray arr) {

    정수 i, 합계 = 0; for (i =
0; i < 10; i++) { sum += arr[i];

    }
}
```

대신 기본 배열 요소에 액세스하려면 적절한 JNI 함수를 사용해야 합니다.
수정된 다음 예와 같이

```
JNIEXPORT 진트 JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr) {

    진트버프[10]; 진트 i,
    합계 = 0; (*env)->GetIntArrayRegion(env, arr, 0, 10, buf); for (i = 0; i < 10; i++) { sum +=
buf[i];

    }
    반환 합계;
}
```

3.3.2 기본 유형의 배열에 액세스

이전 예제에서는 GetIntArrayRegion 함수를 사용하여 정수 배열의 모든 요소를 C 버퍼 (buf)에 복사합니다. 세 번째 인수는 요소의 시작 인덱스이고 네 번째 인수는 복사할 요소의 수입니다. 요소가 C 버퍼에 있으면 네이티브 코드에서 액세스할 수 있습니다.

예제에서 10이 배열의 길이라는 것을 알고 있으므로 인덱스 오버플로가 있을 수 없기 때문에 예외 검사가 필요하지 않습니다.

JNI는 네이티브 코드가 int 유형의 배열 요소를 수정할 수 있도록 하는 해당 SetIntArrayRegion 함수를 지원합니다. 다른 기본 유형(예: boolean, short 및 float)의 배열도 지원됩니다.

JNI 는 네이티브 코드가 기본 배열의 요소에 대한 직접 포인터를 얻을 수 있도록 하는 Get/Release<Type>ArrayElements 함수(예: Get/ReleaseIntArrayElements 포함) 제품군을 지원합니다. 기본 가비지 수집기가 고정을 지원하지 않을 수 있으므로 가상 머신은 원래 기본 배열의 복사본에 대한 포인터를 반환할 수 있습니다. 다음과 같이 GetIntArrayElements를 사용하여 섹션 3.3.1의 네이티브 메서드 구현을 다시 작성할 수 있습니다.

```
JNIEXPORT 진트 JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr) {

    진트 *카; 진트 i,
    합계 = 0; carr = (*env)->GetIntArrayElements(env, arr, NULL); if (carr == NULL) { 0 반환; /* 예외 발생 */ }

    }
    for (i=0; i<10; i++) { sum += carr[i]; }

    }
    (*env)->ReleaseIntArrayElements(env, arr, carr, 0); 반환 합계;

}
```

GetArrayLength 함수는 프리미티브 또는 객체 배열의 요소 수를 반환합니다. 배열의 고정 길이는 배열이 처음 할당될 때 결정됩니다.

Java 2 SDK 릴리스 1.2에는 Get/ReleasePrimitiveArrayCritical 함수가 도입되었습니다. 이러한 기능을 통해 가상 머신은 기본 코드가 기본 배열의 내용에 액세스하는 동안 가비지 수집을 비활성화 할 수 있습니다. 프로그래머는 Get/ReleaseStringCritical 함수(§ 3.2.4)를 사용할 때와 동일한 주의를 기울여야 합니다. Get/ReleasePrimitiveArrayCritical 함수 쌍 사이에서 네이티브 코드는 임의의 JNI 함수를 호출하거나 애플리케이션을 교착 상태에 빠뜨릴 수 있는 차단 작업을 수행하면 안 됩니다.

3.3.3 JNI 기본 배열 함수 요약

표 3.2는 기본 배열과 관련된 모든 JNI 함수를 요약한 것입니다. Java 2 SDK 릴리스 1.2에는 특정 배열 작업의 성능을 향상시키는 여러 가지 새로운 기능이 추가되었습니다. 추가된 기능은 성능 향상을 가져오는 것 외에 새로운 작업을 지원하지 않습니다.

표 3.2 JNI 기본 배열 함수 요약

JNI 함수	설명	부터
Get<Type>ArrayRegion Set<Type>ArrayRegion	프리미티브 배열의 내용을 미리 할당된 C 버퍼로 또는 그로부터 복사합니다.	JDK1.1
Get<Type>ArrayElements Release<Type>ArrayElements	프리미티브 배열의 내용에 대한 포인터를 얻습니다. 사본을 반환할 수 있습니다. 정렬.	JDK1.1
GetArrayLength	배열의 요소 수를 반환합니다.	JDK1.1
새로운<Type>배열	주어진 길이로 배열을 생성합니다.	JDK1.1
GetPrimitiveArrayCritical ReleasePrimitiveArrayCritical	프리미티브 배열의 내용에 대한 포인터를 얻거나 해제합니다. 가비지 수집을 비활성화하거나 어레이의 복사본을 반환할 수 있습니다.	자바 2 SDK1.2

3.3.4 기본 배열 함수 중에서 선택

그림 3.3은 프로그래머가 JDK 릴리스 1.1 및 Java 2 SDK 릴리스 1.2의 기본 배열에 액세스하기 위해 JNI 함수 중에서 선택하는 방법을 보여줍니다.

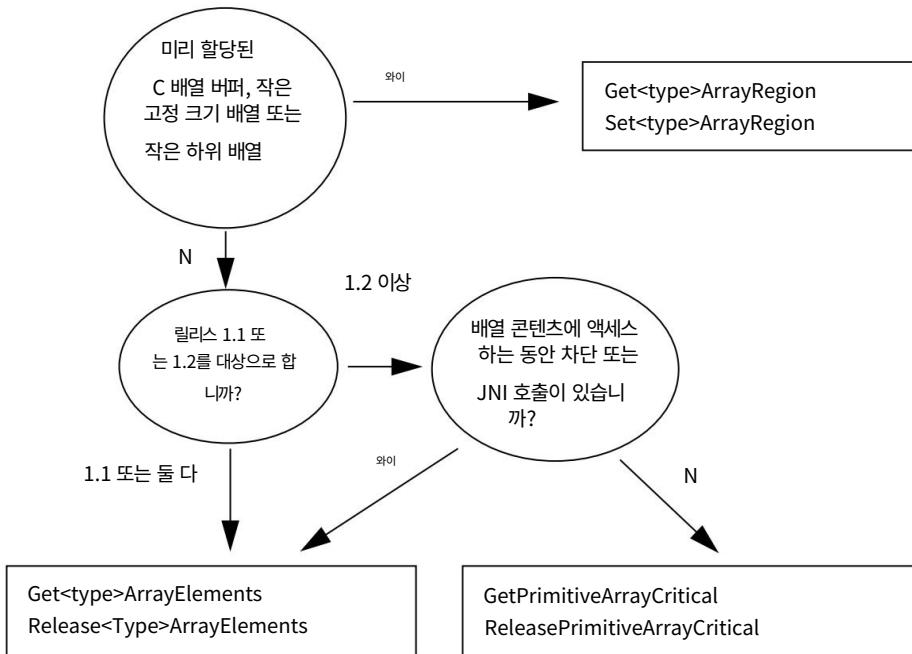


그림 3.3 기본 배열 함수 중에서 선택

사전 할당된 C 버퍼로 복사하거나 복사해야 하는 경우 Get/Set<Type>ArrayRegion 함수 계열을 사용하십시오. 이러한 함수는 경계 검사를 수행하고 필요할 때 ArrayIndexOutOfBoundsException 예외를 발생시킵니다. 섹션 3.3.1의 네이티브 메서드 구현은 GetIntArray Region을 사용하여 jarray 참조에서 10개의 요소를 복사합니다.

작은 고정 크기 배열의 경우 Get/Set<Type>ArrayRegion은 C 버퍼가 C 스택에 매우 저렴하게 할당될 수 있기 때문에 거의 항상 선호되는 함수입니다. 적은 수의 배열 요소를 복사하는 오버헤드는 무시할 수 있습니다.

Get /Set<Type>ArrayRegion 함수를 사용하면 시작 인덱스와 요소 수를 지정할 수 있으므로 기본 코드가 큰 배열의 요소 하위 집합에만 액세스해야 하는 경우 선호되는 함수입니다.

미리 할당된 C 버퍼가 없고 기본 배열의 크기가 결정되지 않았으며 기본 코드가 배열 요소에 대한 포인터를 보유하는 동안 차단 호출을 실행하지 않는 경우 Java 2 SDK 릴리스 1.2에서 Get/ReleasePrimitiveArrayCritical 기능을 사용하십시오. Get/ReleaseStringCritical 함수 와 마찬가지로 교착 상태를 피하려면 Get/ReleasePrimitiveArrayCritical 함수를 매우 주의해서 사용해야 합니다.

3.3.5 객체 배열 액세스

기본 유형, 문자열 및 배열

`Get/Release<type>ArrayElements` 함수 계열을 사용하는 것이 항상 안전합니다. 가상 머신은 배열 요소에 대한 직접 포인터를 반환하거나 배열 요소의 복사본을 보유하는 버퍼를 반환합니다.

3.3.5 객체 배열 접근

JNI는 개체 배열에 액세스하기 위한 별도의 함수 쌍을 제공합니다.

`GetObjectArrayElement`는 지정된 인덱스에 있는 요소를 반환하는 반면 `SetObjectArrayElement`는 지정된 인덱스에 있는 요소를 업데이트합니다. 기본 배열 유형의 상황과 달리 모든 개체 요소를 가져오거나 한 번에 여러 개체 요소를 복사할 수 없습니다.

문자열과 배열은 참조 유형입니다. `Get/SetObjectArray Element`를 사용하여 문자열 배열 및 배열 배열에 액세스합니다.

다음 예제에서는 네이티브 메서드를 호출하여 2차원 `int`의 배열을 출력한 다음 배열의 내용을 인쇄합니다.

```
클래스 ObjectArrayTest { 개인 정적
    네이티브 int[][] initInt2DArray(int 크기); public static void main(String[] args) { int[]
        [] i2arr = initInt2DArray(3); for (int i = 0; i < 3; i++) {

            for (int j = 0; j < 3; j++) { System.out.print(
                " " + i2arr[i][j]);
            }
            System.out.println();
        }
    }
    정적
    { System.loadLibrary("ObjectArrayTest");
    }
}
```

정적 네이티브 메서드 `initInt2DArray`는 주어진 크기의 2차원 배열을 만듭니다. 2차원 배열을 할당하고 초기화하는 네이티브 메서드는 다음과 같이 작성할 수 있습니다.

```

JNIEXPORT jobjectArray JNICALL Java_ObjectArrayTest_initInt2DArray(JNIEnv *env, jclass cls, int 크기)

{
    jobjectArray 결과; 정수 i;
    jclass intArrCls = (*env)->FindClass(env, "[I");
    if (intArrCls == NULL) {

        NULL을 반환합니다. /* 예외 발생 */
    }
    result = (*env)->NewObjectArray(env, 크기, intArrCls,
                                     없는);
    if (결과 == NULL) { NULL 반환;
        * 메모리 부족 오류 발생 */
    }
    for (i = 0; i < 크기; i++) {
        진트 tmp[256]; /* 충분히 큰지 확인하세요! */ 정수 j; jintArray iarr = (*env)->NewIntArray(env, 크기);
        if (iarr == NULL) { NULL 반환; /* 메모리 부족 오류 발생 */
    }

    }
    for (j = 0; j < 크기; j++) { tmp[j] = i + j;

    }
    (*env)->SetIntArrayRegion(env, iarr, 0, 크기, tmp); (*env)->SetObjectArrayElement(env, result, i, iarr); (*env)->DeleteLocalRef(env, iarr);
}
반환 결과;
}

```

newInt2DArray 메서드는 먼저 JNI 함수 FindClass를 호출하여 2차원 int 배열의 요소 클래스 참조를 가져옵니다. FindClass에 대한 "[I" 인수는 Java 프로그래밍 언어의 int[] 유형에 해당하는 JNI 클래스 설명자 (§ 12.3.2)입니다. FindClass는 NULL을 반환하고 클래스 로딩에 실패하면 예외를 발생시킵니다(예: 클래스 파일 누락 또는 메모리 부족 조건으로 인해).

다음으로 NewObjectArray 함수는 요소 유형이 intArrCls 클래스 참조로 표시되는 배열을 할당합니다. NewObjectArray 함수는 첫 번째 차원만 할당하고 두 번째 차원을 구성하는 배열 요소를 채우는 작업은 여전히 남아 있습니다. JVM(Java Virtual Machine)에는 다차원 배열을 위한 특별한 데이터 구조가 없습니다. 2차원 배열은 단순히 배열의 배열입니다.

3.3.5 객체 배열 액세스

기본 유형, 문자열 및 배열

두 번째 차원을 생성하는 코드는 매우 간단합니다. NewIntArray는 개별 배열 요소를 할당하고 SetIntArrayRegion은 tmp[] 버퍼의 내용을 새로 할당된 1차원 배열에 복사합니다.

SetObjectArrayElement 호출을 완료한 후 i번째 1차원 배열의 j번째 요소의 값은 $i+j$ 입니다.

ObjectArrayTest.main 메서드를 실행하면 다음 출력이 생성됩니다.

```
0 1 2 1  
2 3 2  
3 4
```

루프 끝의 DeleteLocalRef 호출은 가상 머신이 iarr 와 같은 JNI 참조를 보유하는 데 사용되는 메모리가 부족하지 않도록 합니다 .

5.2.1절에서는 DeleteLocalRef를 호출해야 하는 시기와 이유에 대해 자세히 설명합니다 .

4

장

필드 및 방법

이제 JNI가 원시 코드에 원시 유형 및 참조에 액세스하도록 허용하는 방법을 알고 있습니다.

문자열 및 배열과 같은 `erence` 유형, 다음 단계는 임의 개체의 필드 및 메서드와 상호 작용하는 방법을 배우는 것입니다. 여기에는 필드에 액세스하는 것 외에도 일반적으로 네이티브 코드에서 콜백을 수행하는 것으로 알려진 네이티브 코드에서 Java 프로그래밍 언어로 구현된 메서드를 호출하는 것이 포함됩니다.

필드 액세스 및 메서드 콜백을 지원하는 JNI 함수를 소개하면서 시작하겠습니다. 이 장의 뒷부분에서 간단하지만 효과적인 캐싱 기술을 사용하여 이러한 작업을 보다 효율적으로 만드는 방법에 대해 설명합니다. 마지막 섹션에서는 네이티브 메서드 호출과 네이티브 코드에서 필드 액세스 및 메서드 호출의 성능 특성에 대해 설명합니다.

4.1 필드 접근

Java 프로그래밍 언어는 두 종류의 필드를 지원합니다. 클래스의 각 인스턴스에는 클래스의 인스턴스 필드에 대한 고유한 복사본이 있는 반면 클래스의 모든 인스턴스는 클래스의 정적 필드를 공유합니다.

JNI는 네이티브 코드가 개체의 인스턴스 필드와 클래스의 정적 필드를 가져오고 설정하는 데 사용할 수 있는 함수를 제공합니다. 네이티브 메서드 구현에서 인스턴스 필드에 액세스하는 방법을 보여주는 예제 프로그램을 먼저 살펴보겠습니다.

4.1 필드 액세스

필드 및 방법

```

클래스 InstanceFieldAccess { 개인 문자열
    s;

    비공개 기본 무효 accessField(); public static void
    main(String args[]) { InstanceFieldAccess c = new
        InstanceFieldAccess(); cs = "abc"; c.accessField(); System.out.println("자
        바에서:"); System.out.println(" cs ='" + cs + "'");

    }
    정적
    { System.loadLibrary("InstanceFieldAccess");
    }
}

```

InstanceFieldAccess 클래스는 인스턴스 필드를 정의합니다 . 기본 메서드 는 개체를 만들고 인스턴스 필드를 설정한 다음 네이티브 메서드 InstanceFieldAccess.accessField를 호출합니다. 곧 보게 되겠지만 기본 메서드는 인스턴스 필드의 기존 값을 인쇄한 다음 필드를 새 값으로 설정합니다. 프로그램은 네이티브 메서드가 반환된 후 필드 값을 다시 인쇄하여 필드 값이 실제로 변경되었음을 보여줍니다.

다음은 InstanceFieldAccess.accessField 기본 메서드 의 구현입니다 .

```

JNIEXPORT 무효 JNICALL
Java_InstanceFieldAccess_accessField(JNIEnv *env, jobject obj) {

    jfieldID fid; /* 필드 ID 저장 */ jstring jstr; const char *str;

    /* obj의 클래스에 대한 참조 얻기 */ jclass cls = (*env)->GetObjectClass(env, obj);

    printf("C에서:\n");

    /* cls에서 인스턴스 필드 s를 찾습니다. */ fid = (*env)->GetFieldID(env,
    cls, "s",
                                         "Ljava/lang/String;");
    if (fid == NULL) { 반환; /* 필
        드 찾기 실패 */
    }
}

```

```

/* 인스턴스 필드 s 읽기 */ jstr = (*env)-
>GetObjectField(env, obj, fid); str = (*env)->GetStringUTFChars(env,
jstr, NULL); if (str == NULL) { 반환; /* 메모리 부족 */

}

printf(" cs = \"%s\"\n", str); (*env)-
>ReleaseStringUTFChars(env, jstr, str);

/* 새 문자열을 만들고 인스턴스 필드를 덮어씁니다. */ jstr = (*env)->NewStringUTF(env, "123");
if (jstr == NULL) { 반환; /* 메모리 부족 */

}

(*env)->SetObjectField(env, obj, fid, jstr);
}

```

InstanceFieldAccess 네이티브 라이브러리 와 함께 InstanceFieldAccess 클래스를 실행하면 다음 출력이 생성됩니다.

C에서:
 cs = "abc"
 자바: cs =
 "123"

4.1.1 인스턴스 필드 접근 절차

인스턴스 필드에 액세스하기 위해 기본 메서드는 2단계 프로세스를 따릅니다. 먼저 GetFieldID를 호출하여 클래스 참조, 필드 이름 및 필드 설명자에서 필드 ID를 가져옵니다 .

```
fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");
```

예제 코드는 네이티브 메서드 구현에 두 번째 인수로 전달되는 인스턴스 참조 obj 에서 GetObjectClass를 호출하여 클래스 참조 cls를 얻습니다.

필드 ID를 얻은 후에는 개체 참조 및 필드 ID를 적절한 인스턴스 필드 액세스 기능에 전달할 수 있습니다.

```
jstr = (*env)->GetObjectField(env, obj, fid);
```

문자열과 배열은 특별한 종류의 객체이기 때문에 GetObject Field를 사용하여 문자열인 인스턴스 필드에 액세스합니다. Get/SetObjectField 외에도 JNI는 기본 유형의 인스턴스 필드에 액세스하기 위한 GetIntField 및 SetFloatField 와 같은 다른 기능도 지원합니다 .

4.1.2 필드 설명자

필드 및 방법

4.1.2 필드 설명자

이전 섹션에서 특별히 인코딩된 C 문자열 "Ljava/lang/String;"을 사용했음을 알 수 있습니다. Java 프로그래밍 언어의 필드 유형을 나타냅니다. 이러한 C 문자열을 JNI 필드 설명자라고 합니다.

문자열의 내용은 선언된 필드 유형에 따라 결정됩니다. 예를 들어 "I"로 int 필드, "F"로 부동 필드, "D"로 이중 필드, "Z"로 부울 필드 등을 나타냅니다.

`java.lang.String`과 같은 참조 유형의 설명자는 문자 L로 시작하고 그 뒤에 JNI 클래스 설명자(§ 3.3.5)가 오고 세미콜론으로 끝납니다. ". ." 정규화된 클래스 이름의 구분 기호는 JNI 클래스 설명자에서 "/"로 변경됩니다. 따라서 다음과 같이 `java.lang.String` 유형의 필드에 대한 필드 설명자를 구성합니다.

```
"Ljava/lang/문자열;"
```

배열 유형에 대한 설명자는 "[" 문자와 배열의 구성요소 유형 설명자로 구성됩니다. 예를 들어 "[I]"는 int[] 필드 유형에 대한 설명자입니다. 섹션 12.3.3에는 Java 프로그래밍 언어의 필드 설명자 및 일치 유형에 대한 세부 정보가 포함되어 있습니다.

`javap` 도구(JDK 또는 Java 2 SDK 릴리스와 함께 제공됨)를 사용하여 클래스 파일에서 필드 설명자를 생성할 수 있습니다. 일반적으로 `javap`은 주어진 클래스의 메소드 및 필드 유형을 인쇄합니다. -s 옵션(및 전용 멤버 노출을 위한 -p 옵션)을 지정하면 `javap`은 JNI 설명자를 대신 인쇄합니다.

```
javap -s -p InstanceFieldAccess
```

그러면 필드 s에 대한 JNI 설명자를 포함하는 출력이 제공됩니다.

```
...
s Ljava/lang/String;
...
```

`javap` 도구를 사용하면 JNI 설명자 문자열을 직접 파생할 때 발생할 수 있는 실수를 제거하는 데 도움이 됩니다.

4.1.3 정적 필드 액세스

정적 필드에 액세스하는 것은 인스턴스 필드에 액세스하는 것과 유사합니다. `InstanceFieldAccess` 예제의 사소한 변형을 살펴보겠습니다.

```

클래스 StaticFieldAccess { 개인 정적 int
    si;

    비공개 기본 무효 accessField(); 공개 정적 무효 메인(문자
    열 args[]) {
        StaticFieldAccess c = new StaticFieldAccess(); StaticFieldAccess.si
        = 100; c.accessField(); System.out.println("자바에서:");
        System.out.println(" StaticFieldAccess.si =
            " + 시);
    }
    정적
    { System.loadLibrary("StaticFieldAccess");
    }
}

```

StaticFieldAccess 클래스에는 정적 정수 필드 si가 포함되어 있습니다. StaticFieldAccess.main 메서드는 개체를 만들고 정적 필드를 초기화한 다음 네이티브 메서드 StaticFieldAccess.accessField를 호출합니다. 곧 보게 되겠지만 기본 메서드는 정적 필드의 기존 값을 출력한 다음 필드를 새 값으로 설정합니다. 필드가 실제로 변경되었는지 확인하기 위해 프로그램은 기본 메서드가 반환된 후 정적 필드 값을 다시 인쇄합니다.

다음은 StaticFieldAccess.accessField 기본 메서드의 구현입니다 .

```

JNIEXPORT 무효 JNICALL
Java_StaticFieldAccess_accessField(JNIEnv *env, jobject obj) {

    jfieldID fid; /* 필드 ID 저장 */ jint si;

    /* obj의 클래스에 대한 참조 얻기 */ jclass cls = (*env)->GetObjectClass(env, obj);

    printf("C에서:\n");

    /* cls에서 정적 필드 si를 찾습니다. */ fid = (*env)->GetStaticFieldID(env, cls, "si", "I"); if (fid == NULL) { 반환; /* 필드를 찾을 수 없음 */

    }

    /* 정적 필드 si에 액세스 */ si = (*env)->GetStaticIntField(env, cls, fid); printf(" StaticFieldAccess.si =
        %d\n", si); (*env)->SetStaticIntField(env, cls, fid, 200);

}

```

4.2

호출 방법

필드 및 방법

네이티브 라이브러리로 프로그램을 실행하면 다음과 같은 결과가 생성됩니다.

C:

```
StaticFieldAccess.si = 100 Java:  
StaticFieldAccess.si = 200
```

정적 필드에 액세스하는 방법과 인스턴스 필드에 액세스하는 방법에는 두 가지 차이점이 있습니다.

1. 인스턴스 필드의 경우 GetFieldID 와 달리 정적 필드의 경우 GetStaticFieldID 를 호출합니다 . GetStaticFieldID 및 GetFieldID는 동일한 반환 유형 jfieldID를 갖습니다.
2. 정적 필드 ID를 얻은 후에는 개체 참조 가 아닌 클래스 참조를 적절한 정적 필드 액세스 기능에 전달 합니다.

4.2 호출 방법

Java 프로그래밍 언어에는 여러 종류의 메소드가 있습니다. 인스턴스 메서드는 클래스의 특정 인스턴스에서 호출되어야 하는 반면 정적 메서드는 모든 인스턴스와 독립적으로 호출될 수 있습니다. 생성자에 대한 논의는 다음 섹션으로 미루겠습니다 .

JNI는 네이티브 코드에서 콜백을 수행할 수 있는 완전한 기능 세트를 지원합니다. 아래 예제 프로그램에는 Java 프로그래밍 환경에서 구현된 인스턴스 메소드를 차례로 호출하는 기본 메소드가 포함되어 있습니다.

게이지.

```
클래스 InstanceMethodCall { 개인 기본  
무효 기본 방법(); private void callback()  
{ System.out.println("In Java");  
  
}  
public static void main(String args[]) { InstanceMethodCall  
c = new InstanceMethodCall(); c.nativeMethod();  
  
}  
정적  
{ System.loadLibrary("InstanceMethodCall");  
}  
}
```

네이티브 메서드의 구현은 다음과 같습니다.

```
JNIEXPORT 무효 JNICALL
Java_InstanceMethodCall_nativeMethod(JNIEnv *env, jobject obj) {

    jclass cls = (*env)->GetObjectClass(env, obj); jmethodID mid = (*env)-
>GetMethodID(env, cls, "콜백", "()V"); if (mid == NULL) { 반환; /* 메서드를 찾
을 수 없음 */

}

printf("C에서\n"); (*env)-
>CallVoidMethod(env, obj, mid);
}
```

위의 프로그램을 실행하면 다음과 같은 결과가 생성됩니다.

```
C에서
자바에서
```

4.2.1 인스턴스 메서드 호출

Java_InstanceMethodCall_nativeMethod 구현은 인스턴스 메서드를 호출하는 데 필요한 두 단계를 보여줍니다.

- 네이티브 메서드는 먼저 JNI 함수 GetMethodID를 호출합니다. GetMethodID는 주어진 클래스에서 메서드 조회를 수행합니다. 조회는 메서드의 이름 및 형식 설명자를 기반으로 합니다. 메서드가 없으면 GetMethodID는 NULL을 반환합니다. 이 시점에서 네이티브 메서드에서 즉시 반환하면 호출한 코드에서 NoSuchMethodError가 발생합니다.

InstanceMethodCall.nativeMethod.

- 네이티브 메서드는 CallVoidMethod를 호출합니다. CallVoidMethod는 반환 유형이 void인 인스턴스 메서드를 호출합니다. 개체, 메서드 ID 및 실제 인수(위의 예에는 없음)를 CallVoidMethod에 전달합니다.

CallVoidMethod 함수 외에도 JNI는 다른 반환 유형이 있는 메서드 호출 함수도 지원합니다. 예를 들어 콜백한 메서드가 int 유형의 값을 반환한 경우 네이티브 메서드는 CallIntMethod를 사용합니다.

마찬가지로 CallObjectMethod를 사용하여 java.lang.String 인스턴스 및 배열을 포함하는 객체를 반환하는 메서드를 호출 할 수 있습니다 .

Call<Type>Method 함수군을 사용하여 인터페이스 메소드를 호출할 수도 있습니다. 인터페이스 유형에서 메소드 ID를 파생시켜야 합니다. 다음과 같은

4.2.2 메서드 설명자 구성

필드 및 방법

예를 들어 코드 세그먼트는 java.lang.Thread 인스턴스에서 Runnable.run 메서드를 호출합니다.

```
작업객체 thd = ...; /* java.lang.Thread 인스턴스 */ jmethodID mid; jclass
runnableIntf = (*env)->FindClass(env, "java/lang/Runnable"); if (runnableIntf
== NULL) {

    ...
    /* 오류 처리 */
}

mid = (*env)->GetMethodID(env, runnableIntf, "run", "()V"); if (mid == NULL) { ...
    /* 오
    류 처리 */
}

(*env)->CallVoidMethod(env, thd, mid); ... /* 가능한 예
외 확인 */
```

섹션 3.3.5에서 FindClass 함수가 참조를 반환하는 것을 보았습니다.
명명된 클래스에. 여기서는 명명된 인터페이스에 대한 참조를 얻기 위해 이를 사용하기도 합니다.

4.2.2 메서드 설명자 구성

JNI는 설명자 문자열을 사용하여 필드 유형을 나타내는 방법과 유사한 방식으로 메서드 유형을 나타냅니다. 메서드 설명자는 메서드의 인수 형식과 반환 형식을 결합합니다. 인수 유형이 먼저 나타나고 한 쌍의 괄호로 둘러싸여 있습니다. 인수 유형은 메소드 선언에 나타나는 순서대로 나열됩니다. 여러 인수 유형 사이에는 구분 기호가 없습니다. 메서드가 인수를 사용하지 않으면 빈 괄호 쌍으로 표시됩니다. 인수 유형에 대한 오른쪽 닫는 괄호 바로 뒤에 메서드의 반환 유형을 배치합니다.

예를 들어 "(I)V"는 int 형식의 인수 하나를 사용하고 반환 형식이 void인 메서드를 나타냅니다. "()D"는 인수를 사용하지 않고 double을 반환하는 메서드를 나타냅니다. "int f(void)" 와 같은 C 함수 프로토타입이 "(V)I"가 유효한 메서드 설명자라고 잘못 생각하게 만들지 마십시오. 대신 "(I)"를 사용하십시오.

메서드 설명자는 클래스 설명자(§ 12.3.2)를 포함할 수 있습니다. 예를 들어, 방법은 다음과 같습니다.

네이티브 개인 문자열 getLine(String);

다음 설명자가 있습니다.

"(Ljava/lang/String;)Ljava/lang/String;"

배열 유형에 대한 설명자는 "[" 문자로 시작하고 뒤에
배열 요소 유형의 설명자. 예를 들어 다음과 같은 메서드 설명자는 다음과 같습니다.

```
public static void main(String[] args);
```

다음과 같다:

"([Ljava/lang/String;)V"

섹션 12.3.4는 JNI 메서드 설명자를 형성하는 방법에 대한 완전한 설명을 제공합니다. javap 도구를 사용하여 JNI 메서드 설명자를 출력 할 수 있습니다 . 예를 들어 다음을 실행합니다.

```
javap -s -p InstanceMethodCall
```

다음 출력을 얻습니다.

```
...
비공개 콜백()V 공개 정적 메인
([Ljava/lang/String;)V 비공개 네이티브 nativeMethod()V
...

```

-s 플래그는 Java 프로그래밍 언어에 표시되는 유형이 아니라 JNI 설명자 문자열을 출력하도록 javap 에 알립니다 . -p 플래그 는 javap가 출력에 클래스의 전용 멤버에 대한 정보를 포함하도록 합니다 .

4.2.3 정적 메서드 호출

앞의 예제는 네이티브 코드가 인스턴스 메서드를 호출하는 방법을 보여줍니다.

마찬가지로 다음 단계에 따라 네이티브 코드에서 정적 메서드에 대한 콜백을 수행할 수 있습니다.

- Get 이 아닌 GetStaticMethodID를 사용하여 메서드 ID를 얻습니다.
MethodID.
- 클래스, 메서드 ID 및 인수를 CallStaticVoidMethod, CallStaticBooleanMethod 등 의 정적 메서드 호출 함수 계열 중 하나로 전달합니다 .

정적 메서드를 호출할 수 있는 함수와 인스턴스 메서드를 호출할 수 있는 함수 사이에는 주요 차이점이 있습니다. 전자는 클래스 참조를 두 번째 인수로 사용하는 반면 후자는 개체 참조를 두 번째 인수로 사용합니다. 예를 들어 클래스 참조를 CallStaticVoidMethod 에 전달 하지만 객체 참조를 CallVoidMethod에 전달합니다.

4.2.3 정적 메서드 호출

필드 및 방법

Java 프로그래밍 언어 수준에서 Cls.f 또는 obj.f (여기서 obj는 Cls의 인스턴스임)의 두 가지 대체 구문을 사용하여 Cls 클래스에서 정적 메서드 f를 호출할 수 있습니다. (그러나 후자는 권장되는 프로그래밍 스타일입니다.) JNI에서는 네이티브 코드에서 정적 메서드 호출을 실행할 때 항상 클래스 참조를 지정해야 합니다.

네이티브에서 정적 메서드로 콜백하는 예를 살펴보겠습니다.
암호. 이전 InstanceMethodCall 예제의 약간 변형입니다.

```
클래스 StaticMethodCall { 개인 기본 무효
    기본 방법(); 개인 정적 무효 콜백() {

        System.out.println("자바에서");
    }
    공개 정적 무효 메인(문자열 args[]) {
        StaticMethodCall c = 새로운 StaticMethodCall(); c.nativeMethod();

    }
    정적
    { System.loadLibrary("StaticMethodCall");
    }
}
```

네이티브 메서드의 구현은 다음과 같습니다.

```
JNIEXPORT 무효 JNICALL
Java_StaticMethodCall_nativeMethod(JNIEnv *env, jobject obj) {

    jclass cls = (*env)->GetObjectClass(env, obj); jmethodID mid = (*env)->GetStaticMethodID(env, cls, "콜백", "()V"); if (mid == NULL) { 반환; /* 메서드를 찾을 수 없음 */

    }
    printf("C에서\n"); (*env)->CallStaticVoidMethod(env, cls, mid);
}
```

obj와 달리 cls(굵게 강조 표시됨)를 CallStaticVoidMethod에 전달해야 합니다. 위의 프로그램을 실행하면 다음과 같은 예상 출력이 생성됩니다.

C에서
자바에서

4.2.4 수퍼클래스의 인스턴스 메서드 호출

슈퍼클래스에 정의되었지만 개체가 속한 클래스에서 재정의된 인스턴스 메서드를 호출할 수 있습니다. JNI는 이를 위해 일련의 `CallNonvirtual<Type>Method` 함수를 제공합니다. 슈퍼클래스에 정의된 인스턴스 메서드를 호출하려면 다음을 수행합니다.

- `GetStaticMethodID` 가 아닌 `GetMethodID`를 사용하여 수퍼클래스에 대한 참조에서 메서드 ID를 얻습니다. • 개체, 슈퍼클래스, 메서드 ID 및 인수를 `CallNonvirtualVoidMethod`, `CallNonvirtualBooleanMethod` 등과 같은 비가상 호출 함수 계열 중 하나로 전달합니다.

수퍼클래스의 인스턴스 메서드를 호출해야 하는 경우는 비교적 드뭅니다. 이 기능은 Java 프로그래밍 언어에서 다음 구성을 사용하여 재정의된 슈퍼클래스 메서드(예: `f`)를 호출하는 것과 유사합니다.

```
슈퍼.에프();
```

`CallNonvirtualVoidMethod`은 생성자를 호출하는 데 사용할 수도 있습니다. 다음 섹션에서 설명합니다.

4.3 생성자 호출

JNI에서 생성자는 인스턴스 메소드 호출에 사용되는 것과 유사한 단계에 따라 호출될 수 있습니다. 생성자의 메서드 ID를 얻으려면 메서드 설명자에서 메서드 이름으로 "<init>"를 전달하고 반환 유형으로 "V"를 전달합니다.

그런 다음 메서드 ID를 `NewObject`와 같은 JNI 함수에 전달하여 생성자를 호출할 수 있습니다. 다음 코드는 C 버퍼에 저장된 유니코드 문자에서 `java.lang.String` 객체를 구성하는 JNI 함수 `NewString`과 동등한 기능을 구현합니다.

```
jstring
MyNewString(JNIEnv *env, jchar *chars, jint len) {

    jclass 스트링클래스;
    jmethodID cid; jcharArray
    elemArr; jstring 결과;

    stringClass = (*env)->FindClass(env, "java/lang/String"); if (문자열 클래스 == NULL)
    { NULL 반환; /* 예외 발생 */

}
```

```

/* String(char[]) 생성자에 대한 메서드 ID 가져오기 */ cid = (*env)->GetMethodID(env,
stringClass, "<init>", "([C)V");

if (cid == NULL) { 반환
    NULL; /* 예외 발생 */
}

/* 문자열 문자를 보유하는 char[] 생성 */ elemArr = (*env)->NewCharArray(env, len);
if (elemArr == NULL) { NULL 반환; /* 예외 발생 */

}

(*env)->SetCharArrayRegion(env, elemArr, 0, len, chars);

/* java.lang.String 객체 생성 */ result = (*env)->NewObject(env,
stringClass, cid, elemArr);

/* 무료 로컬 참조 */ (*env)-
>DeleteLocalRef(env, elemArr); (*env)-
>DeleteLocalRef(env, stringClass); 반환 결과;

}

```

이 함수는 신중하게 설명할 만큼 충분히 복잡합니다. 먼저 Find Class는 java.lang.String 클래스에 대한 참조를 반환합니다. 다음으로 GetMethodID는 문자열 생성자 String(char[] chars)의 메서드 ID를 반환합니다. 그런 다음 NewCharArray를 호출하여 모든 문자열 요소를 포함하는 문자 배열을 할당합니다. JNI 함수 NewObject는 메서드 ID로 지정된 생성자를 호출합니다. NewObject 함수는 구성할 클래스에 대한 참조, 생성자의 메서드 ID 및 생성자에 전달해야 하는 인수를 인수로 사용합니다.

DeleteLocalRef 호출을 통해 가상 머신은 로컬 참조 elemArr 및 stringClass에서 사용하는 리소스를 해제할 수 있습니다. 섹션 5.2.1에서는 DeleteLocalRef를 호출해야 하는 시기와 이유에 대해 자세히 설명합니다.

문자열은 객체입니다. 이 예는 요점을 더 강조합니다. 그러나 예는 또한 질문으로 이어집니다. 다른 JNI 함수를 사용하여 동등한 기능을 구현할 수 있는 경우 JNI가 NewString과 같은 내장 함수를 제공하는 이유는 무엇입니까? 내장 문자열 함수가 네이티브 코드에서 java.lang.String API를 호출하는 것보다 훨씬 더 효율적이기 때문입니다. 문자열은 가장 자주 사용되는 객체 유형이며 JNI에서 특별히 지원해야 합니다.

CallNonvirtualVoid 메서드 함수를 사용하여 생성자를 호출하는 것도 가능합니다. 이 경우 네이티브 코드는 먼저 AllocObject 함수를 호출하여 초기화되지 않은 개체를 만들어야 합니다. 위의 단일 NewObject 호출:

```
result = (*env)->NewObject(env, stringClass, cid, elemArr);
```

CallNonvirtualVoid 가 뒤따르는 AllocObject 호출로 대체될 수 있습니다.
메서드 호출:

```
result = (*env)->AllocObject(env, stringClass); if (result) { (*env)->CallNonvirtualVoidMethod(env, result, stringClass, cid, elemArr); / * 가능한 예외를 확인해야 합니다. */ if ((*env)->ExceptionCheck(env)) { (*env)->DeleteLocalRef(env, result); 결과 = NULL; } }
```

AllocObject는 초기화되지 않은 개체를 생성하며 생성자가 각 개체에서 최대 한 번만 호출되도록 주의해서 사용해야 합니다. 네이티브 코드는 동일한 개체에서 생성자를 여러 번 호출하면 안 됩니다.

때때로 초기화되지 않은 개체를 먼저 할당하고 나중에 생성자를 호출하는 것이 유용할 수 있습니다. 그러나 대부분의 경우 NewObject를 사용해야 하며 오류가 발생하기 쉬운 AllocObject/CallNonvirtualVoid 메서드 쌍을 피해야 합니다.

4.4 캐싱 필드 및 메서드 ID

필드 및 메서드 ID를 얻으려면 필드 또는 메서드의 이름과 설명자를 기반으로 기호 조회가 필요합니다. 기호 조회는 상대적으로 비용이 많이 듭니다. 이 섹션에서는 이 오버헤드를 줄이는 데 사용할 수 있는 기술을 소개합니다.

아이디어는 필드 및 메서드 ID를 계산하고 나중에 반복적으로 사용할 수 있도록 캐시하는 것입니다. 캐싱이 필드 또는 메서드 ID의 사용 지점에서 수행되는지 또는 필드 또는 메서드를 정의하는 클래스의 정적 초기화 프로그램에서 수행되는지 여부에 따라 필드 및 메서드 ID를 캐시하는 두 가지 방법이 있습니다.

4.4.1 사용 시점의 캐싱

필드 및 메서드 ID는 네이티브 코드가 필드 값에 액세스하거나 메서드 콜백을 수행하는 지점에서 캐시될 수 있습니다. Java_InstanceFieldAccess_accessField 함수의 다음 구현은 InstanceFieldAccess.accessField 메서드를 호출할 때마다 다시 계산할 필요가 없도록 정적 변수에 필드 ID를 캐시합니다.

4.4.1 사용 지점에서 캐싱

필드 및 방법

```
JNIEXPORT 무효 JNICALL
Java_InstanceFieldAccess_accessField(JNIEnv *env, jobject obj) {

    정적 jfieldID fid_s = NULL; /* s에 대한 캐시된 필드 ID */

    jclass cls = (*env)->GetObjectClass(env, obj); jstring jstr; const char
    *str;

    if (fid_s == NULL) { fid_s =
        (*env)->GetFieldID(env, cls, "s",
                           "Ljava/lang/String;");
        if (fid_s == NULL) { 반환; /* 0
                           미 발생한 예외 */
    }
}

printf("C에서:\n");

jstr = (*env)->GetObjectField(env, obj, fid_s); str = (*env)-
>GetStringUTFChars(env, jstr, NULL); if (str == NULL) { 반환; /* 메모리 부족
*/
}

printf(" cs = \"%s\"\n", str); (*env)-
>ReleaseStringUTFChars(env, jstr, str);

jstr = (*env)->NewStringUTF(env, "123"); if (jstr == NULL)
{ 반환; /* 메모리 부족 */

}
(*env)->SetObjectField(env, obj, fid_s, jstr);
}
```

강조 표시된 정적 변수 fid_s는 InstanceFieldAccess.s에 대해 미리 계산된 필드 ID를 저장합니다. 정적 변수는 NULL로 초기화됩니다. InstanceFieldAccess.accessField 메서드가 처음 호출되면 필드 ID를 계산하고 나중에 사용할 수 있도록 정적 변수에 캐시합니다.

위의 코드에서 명백한 경쟁 조건이 있음을 알 수 있습니다. 여러 스레드가 동시에 InstanceFieldAccess.accessField 메서드를 호출하고 동일한 필드 ID를 동시에 계산할 수 있습니다. 한 스레드가 다른 스레드에서 계산한 정적 변수 fid_s를 덮어쓸 수 있습니다. 운 좋게도 이 경합 상태는 여러 스레드에서 중복 작업으로 이어지지만 그렇지 않으면 덜 해롭습니다. 동일한 클래스의 동일한 필드에 대해 여러 스레드에서 계산된 필드 ID는 반드시 동일해야 합니다.

동일한 아이디어에 따라 메서드 ID를 캐시할 수도 있습니다.
이전 MyNewString 예제의 java.lang.String 생성자 :

```
jstring
MyNewString(JNIEnv *env, jchar *chars, jint len) {

    jclass 스트링클래스;
    jcharArray elemArr; 정적
    jmethodID cid = NULL; jstring 결과;

    stringClass = (*env)->FindClass(env, "java/lang/String"); if (문자열 클래스 == NULL)
    { NULL 반환; /* 예외 발생 */

    }

    /* cid는 정적 변수임에 유의하십시오. */ if (cid == NULL) {

        /* String 생성자에 대한 메서드 ID 가져오기 */ cid = (*env)->GetMethodID(env,
        stringClass, "<init>", "([C)V");

        if (cid == NULL) { 반환
            NULL; /* 예외 발생 */
        }
    }

    /* 문자열 문자를 보유하는 char[] 생성 */ elemArr = (*env)->NewCharArray(env, len);
    if (elemArr == NULL) { NULL 반환; /* 예외 발생 */

    }

    (*env)->SetCharArrayRegion(env, elemArr, 0, len, chars);

    /* java.lang.String 객체 생성 */ result = (*env)->NewObject(env,
    stringClass, cid, elemArr);

    /* 무료 로컬 참조 */ (*env)-
    >DeleteLocalRef(env, elemArr); (*env)-
    >DeleteLocalRef(env, stringClass); 반환 결과;

}
```

MyNewString이 처음 호출될 때 java.lang.String 생성자 의 메서드 ID를 계산합니다 . 강조 표시된 정적 변수 cid는 결과를 캐시합니다.

4.4.2 정의 클래스의 이나셜라이저에서 캐싱

필드 및 방법

4.4.2 정의 클래스 초기화 프로그램의 캐싱

사용 지점에서 필드 또는 메소드 ID를 캐시할 때 ID가 이미 캐시되었는지 여부를 감지하기 위한 검사를 도입해야 합니다. 이 접근 방식은 ID가 이미 캐싱된 경우 "빠른 경로"에 약간의 성능 영향을 미칠 뿐만 아니라 캐싱 및 확인이 중복될 수 있습니다. 예를 들어 여러 네이티브 메서드가 모두 동일한 필드에 액세스해야 하는 경우 모두 해당 필드 ID를 계산하고 캐시하기 위한 검사가 필요합니다.

많은 상황에서 응용 프로그램이 네이티브 메서드를 호출할 기회를 갖기 전에 네이티브 메서드에 필요한 필드 및 메서드 ID를 초기화하는 것이 더 편리합니다. 가상 머신은 해당 클래스의 메서드를 호출하기 전에 항상 클래스의 정적 초기화 프로그램을 실행합니다. 따라서 필드 또는 메서드 ID를 계산하고 캐싱하기에 적합한 위치는 필드 또는 메서드를 정의하는 클래스의 정적 초기화 프로그램입니다.

예를 들어 `InstanceMethodCall.callback`에 대한 메서드 ID를 캐시하기 위해 `InstanceMethodCall` 클래스의 정적 초기화 프로그램에서 호출되는 새로운 네이티브 메서드 `initIDs`를 도입합니다.

```
클래스 InstanceMethodCall { 개인 정적 기
    본 무효 initIDs(); 비공개 네이티브 무효 네이티브 메소드();
    private void callback() { System.out.println("In Java");

}

public static void main(String args[]) { InstanceMethodCall
    c = new InstanceMethodCall(); c.nativeMethod();

}

정적
{ System.loadLibrary("InstanceMethodCall"); 초기 ID();

}

}
```

섹션 4.2의 원래 코드와 비교하여 위의 프로그램에는 두 개의 추가 라인이 포함되어 있습니다 (굵은 글꼴로 강조 표시됨). `initID`의 구현은 단순히 `InstanceMethodCall.callback`에 대한 메서드 ID를 계산하고 캐시합니다.

```
jmethodID MID_InstanceMethodCall_callback;
JNIEXPORT 무효 JNICALL
Java_InstanceMethodCall_initIDs(JNIEnv *env, jclass cls {

    MID_InstanceMethodCall_callback = (*env)-
        >GetMethodID(env, cls, "콜백", "()V");
}
```

가상 머신은 정적 이니셜라이저를 실행하고 InstanceMethodCall 클래스에서 다른 메서드(예: nativeMethod 또는 main)를 실행하기 전에 initIDs 메서드를 차례로 호출합니다. 메서드 ID가 이미 전역 변수에 캐시되어 있으므로 InstanceMethodCall.nativeMethod의 기본 구현은 더 이상 기호 조회를 수행할 필요가 없습니다.

```
JNIEXPORT 무효 JNICALL
Java_InstanceMethodCall_nativeMethod(JNIEnv *env, jobject obj) {
    printf("C에서\n"); (*env)->CallVoidMethod(env, obj,
                                                MID_InstanceMethodCall_callback);
}
```

4.4.3 캐싱 ID에 대한 두 가지 접근 방식 간의 비교

JNI 프로그래머가 필드 또는 메서드를 정의하는 클래스의 소스를 제어할 수 없는 경우 사용 시점에 ID를 캐싱하는 것이 합리적인 솔루션입니다.

예를 들어 MyNewString 예제에서는 java.lang.String 생성자의 메서드 ID를 미리 계산하고 캐시하기 위해 사용자 정의 initIDs 네이티브 메서드를 java.lang.String 클래스에 주입할 수 있습니다.

사용 시점의 캐싱은 정의 클래스의 정적 초기화 캐싱과 비교할 때 여러 가지 단점이 있습니다.

- 앞서 설명한 바와 같이 사용 지점에서 캐싱하려면 실행 빠른 경로에서 확인이 필요하며 중복 확인 및 동일한 필드 또는 메서드 ID의 초기화가 필요할 수도 있습니다.
- 메서드 및 필드 ID는 클래스가 언로드될 때까지만 유효합니다. 사용 시점에 필드 및 메서드 ID를 캐시하는 경우 네이티브 코드가 여전히 캐시된 ID의 값에 의존하는 한 정의 클래스가 언로드 및 다시 로드되지 않도록 해야 합니다. (다음 장에서는 JNI를 사용하여 해당 클래스에 대한 참조를 생성하여 클래스가 언로드되지 않도록 하는 방법을 보여줍니다.) 반면 정의 클래스의 정적 초기화 프로그램에서 캐싱이 수행되면 캐시된 ID는 클래스가 언로드되고 나중에 다시 로드될 때 자동으로 다시 계산됩니다.

따라서 가능한 경우 필드 및 메서드 ID를 캐시에 저장하는 것이 좋습니다.
정의 클래스의 정적 초기화 프로그램.

4.5 JNI 필드 및 메서드 작업의 성능

성능 향상을 위해 필드 및 메서드 ID를 캐시하는 방법을 배운 후 JNI를 사용하여 필드에 액세스하고 메서드를 호출하는 성능 특성은 무엇입니까? 네이티브 코드에서 콜백을 수행하는 비용(네이티브/Java 콜백)은 네이티브 메서드를 호출하는 비용(Java/네이티브 호출) 및 일반 메서드를 호출하는 비용(Java/Java 호출)과 어떻게 비교됩니까?)?

이 질문에 대한 대답은 기본 가상 머신이 얼마나 효율적으로 JNI를 구현하는지에 달려 있습니다. 따라서 다양한 가상 머신 구현에 적용되도록 보장되는 성능 특성에 대한 정확한 설명을 제공하는 것은 불가능합니다. 대신 네이티브 메서드 호출과 JNI 필드 및 메서드 작업의 고유 비용을 분석하고 JNI 프로그래머와 구현자를 위한 일반적인 성능 지침을 제공합니다.

Java/네이티브 호출 비용과 Java/Java 호출 비용을 비교하는 것으로 시작하겠습니다. Java/네이티브 호출은 다음과 같은 이유로 Java/Java 호출보다 잠재적으로 느립니다.

- 기본 메서드는 Java 가상 머신 내에서 Java/Java 호출에 사용되는 호출 규칙과 다른 호출 규칙을 따를 가능성이 높습니다. 결과적으로 가상 머신은 네이티브 메서드 진입점으로 점프하기 전에 인수를 구축하고 스택 프레임을 설정하기 위해 추가 작업을 수행해야 합니다.
- 가상 머신이 메서드 호출을 인라인하는 것은 일반적입니다. 인라인 자바/기본 호출은 Java/Java 호출을 인라인하는 것보다 훨씬 어렵습니다.

우리는 일반적인 가상 머신이 Java/Java 호출을 실행하는 것보다 약 2~3배 느리게 Java/네이티브 호출을 실행할 수 있다고 추정합니다. Java/Java 호출은 몇 주기만 걸리기 때문에 기본 메서드가 사소한 작업을 수행하지 않는 한 추가 오버헤드는 무시할 수 있습니다. Java/Java 호출과 비슷하거나 동일한 Java/네이티브 호출 성능으로 가상 머신 구현을 구축하는 것도 가능합니다. (예를 들어 이러한 가상 머신 구현은 JNI 호출 규칙을 내부 Java/Java 호출 규칙으로 채택할 수 있습니다.)

네이티브/자바 콜백의 성능 특성은 기술적으로 자바/네이티브 호출과 유사합니다. 이론적으로 기본/Java 콜백의 오버헤드는 Java/Java 호출의 2~3배 이내일 수 있습니다. 그러나 실제로는 기본/Java 콜백이 상대적으로 드뭅니다. 가상 머신 구현은 일반적으로 콜백의 성능을 최적화하지 않습니다. 이 글을 쓰는 시점에서 많은 프로덕션 가상 머신 구현은 네이티브/자바 콜백의 오버헤드가 자바/자바 호출보다 10배나 높을 수 있습니다.

JNI를 사용한 필드 액세스의 오버헤드는 JNIEnv를 통한 호출 비용에 있습니다. 개체를 직접 역참조하는 대신 네이티브 코드는 개체를 역참조하는 C 함수 호출을 수행해야 합니다. 함수 호출은 가상 머신 구현에 의해 유지되는 내부 객체 표현에서 네이티브 코드를 분리하기 때문에 필요합니다. 함수 호출이 몇 주기만 걸리기 때문에 머리 위의 JNI 필드 액세스는 일반적으로 무시할 수 있습니다.

5 장

로컬 및 글로벌 참조

그만큼 JNI는 인스턴스 및 배열 유형(예: jobject, jclass, jstring 및 jarray)을 불투명 참조로 노출합니다. 네이티브 코드는 불투명 참조 포인터의 내용을 직접 검사하지 않습니다. 대신 JNI 함수를 사용하여 불투명 참조가 가리키는 데이터 구조에 액세스합니다. 불투명 참조만 처리하면 특정 JVM(Java Virtual Machine) 구현에 종속되는 내부 개체 레이아웃에 대해 걱정할 필요가 없습니다. 그러나 JNI에서 다양한 종류의 참조에 대해 자세히 알아볼 필요가 있습니다.

- JNI는 세 가지 종류의 불투명 참조(로컬 참조, 전역 참조 및 약한 전역 참조)를 지원합니다.
- 로컬 및 전역 참조는 서로 다른 수명을 가집니다. 로컬 참조는 자동으로 해제되는 반면 전역 및 약한 전역 참조는 프로그래머가 해제할 때까지 유효한 상태로 유지됩니다.
- 로컬 또는 전역 참조는 참조된 개체가 가비지 수집되지 않도록 합니다. 반면에 약한 전역 참조는 참조된 개체를 가비지 수집할 수 있도록 합니다.
- 모든 참조가 모든 컨텍스트에서 사용될 수 있는 것은 아닙니다. 예를 들어 참조를 만든 기본 메서드가 반환된 후에 로컬 참조를 사용하는 것은 불법입니다.

이 장에서는 이러한 문제에 대해 자세히 설명합니다. 안정적이고 공간 효율적인 코드를 작성하려면 JNI 참조를 올바르게 관리하는 것이 중요합니다.

5.1 로컬 및 전역 참조

로컬 및 글로벌 참조는 무엇이며 어떻게 다릅니까? 일련의 예제를 사용하여 로컬 및 글로벌 참조를 설명합니다.

5.1.1 로컬 참조

로컬 및 글로벌 참조

5.1.1.1 로컬 참조

대부분의 JNI 함수는 로컬 참조를 만듭니다. 예를 들어 JNI 함수 New Object는 새 인스턴스를 만들고 해당 인스턴스에 대한 로컬 참조를 반환합니다.

로컬 참조는 이를 생성하는 네이티브 메서드의 동적 컨텍스트 내에서만, 그리고 네이티브 메서드의 해당 호출 내에서만 유효합니다.

네이티브 메서드 실행 중에 생성된 모든 로컬 참조는 네이티브 메서드가 반환되면 해제됩니다.

정적 변수에 로컬 참조를 저장하고 후속 호출에서 동일한 참조를 사용할 것으로 예상되는 네이티브 메서드를 작성해서는 안 됩니다. 예를 들어, 섹션 4.4.1에 있는 MyNewString 함수의 수정된 버전인 다음 프로그램은 로컬 참조를 잘못 사용합니다.

```
/* 이 코드는 불법입니다 */ jstring
MyNewString(JNIEnv *env, jchar *chars, jint len) {
    정적 jclass 문자열 클래스 = NULL; jmethodID cid;
    jcharArray elemArr; jstring 결과;

    if (stringClass == NULL) { stringClass
        = (*env)->FindClass(env,
                           "자바/언어/문자열");
        if (문자열 클래스 == NULL) { NULL 반환;
            * 예외 발생 *
        }
    }
    /* 여기에서 캐시된 stringClass를 사용하는 것은 잘못되었습니다. 유효하지 않을 수
     있기 때문입니다. */ cid = (*env)->GetMethodID(env, stringClass,
    "<init>", "[C)V");
    ...
    elemArr = (*env)->NewCharArray(env, len);
    ...
    result = (*env)->NewObject(env, stringClass, cid, elemArr); (*env)->DeleteLocalRef(env,
    elemArr); 반환 결과;
}
```

여기서 논의와 직접적인 관련이 없는 줄은 생략했습니다.

정적 변수에 stringClass를 캐싱하는 목적은 다음과 같은 함수 호출을 반복적으로 수행하는 오버헤드를 제거하는 것일 수 있습니다.

```
FindClass(env, "java/lang/String");
```

이것은 FindClass가 java.lang.String 클래스 객체에 대한 로컬 참조를 반환하기 때문에 올바른 접근 방식이 아닙니다. 이것이 왜 문제인지 알아보기 위해 Cf의 네이티브 메서드 구현이 MyNewString을 호출한다고 가정합니다.

```
JNIEXPORT jstring JNICALL
Java_C_f(JNIEnv *env, jobject this) {

    문자 *c_str = ...;
    ...
    return MyNewString(c_str);
}
```

네이티브 메서드 Cf가 반환된 후 가상 머신은 Java_C_f 실행 중에 생성된 모든 로컬 참조를 해제합니다. 이러한 해제된 로컬 참조에는 stringClass 변수에 저장된 클래스 객체에 대한 로컬 참조가 포함됩니다.

향후 MyNewString 호출은 유효하지 않은 로컬 참조를 사용하려고 시도하므로 메모리 손상이나 시스템 충돌이 발생할 수 있습니다. 예를 들어 다음과 같은 코드 세그먼트는 Cf를 두 번 연속으로 호출하고 MyNewString이 잘못된 로컬 참조를 만나도록 합니다.

```
...
...    = CF(); // 첫 번째 호출은 아마도 괜찮을 것입니다.
...    = CF(); // 유효하지 않은 로컬 참조를 사용합니다.
...
```

로컬 참조를 무효화하는 방법에는 두 가지가 있습니다. 앞에서 설명한 것처럼 가상 머신은 네이티브 메서드가 반환된 후 네이티브 메서드 실행 중에 생성된 모든 로컬 참조를 자동으로 해제합니다. 또한 프로그래머는 DeleteLocalRef와 같은 JNI 함수를 사용하여 로컬 참조의 수명을 명시적으로 관리할 수 있습니다.

네이티브 메서드가 반환된 후 가상 머신이 로컬 참조를 자동으로 해제하는 경우 로컬 참조를 명시적으로 삭제하려는 이유는 무엇입니까? 로컬 참조는 로컬 참조가 무효화될 때까지 참조된 객체가 가비지 수집되지 않도록 합니다. 예를 들어 MyNewString에서 DeleteLocalRef 호출을 사용하면 중간 배열 객체인 elemArr이 즉시 가비지 수집될 수 있습니다. 그렇지 않으면 가상 머신은 MyNewString(위의 Cf와 같은)을 호출하는 네이티브 메서드가 반환된 후에만 elemArr 객체를 해제할 수 있습니다.

로컬 참조는 소멸되기 전에 여러 네이티브 함수를 통해 전달될 수 있습니다. 예를 들어 MyNewString은 NewObject에서 만든 문자열 참조를 반환합니다. 그런 다음 MyNewString에서 반환된 로컬 참조를 해제할지 여부를 결정하는 것은 MyNewString 호출자에게 달려 있습니다. Java_C_f 예제에서 Cf는 네이티브 메서드 호출의 결과로 MyNewString의 결과를 반환합니다. 가상 머신이 Java_C_f에서 로컬 참조를 수신한 후

5.1.2 전역 참조

로컬 및 글로벌 참조

함수에서 기본 문자열 개체를 Cf 호출자에게 전달한 다음 원래 JNI 함수 NewObject에 의해 생성된 로컬 참조를 삭제합니다.

로컬 참조는 참조를 생성하는 스레드에서만 유효합니다. 한 스레드에서 생성된 로컬 참조는 다른 스레드에서 사용할 수 없습니다. 네이티브 메서드가 전역 변수에 로컬 참조를 저장하고 다른 스레드가 로컬 참조를 사용하도록 기대하는 것은 프로그래밍 오류입니다.

5.1.2 전역 참조

네이티브 메서드의 여러 호출에서 전역 참조를 사용할 수 있습니다. 전역 참조는 여러 스레드에서 사용할 수 있으며 프로그래머가 해제할 때까지 유효합니다. 로컬 참조와 마찬가지로 전역 참조는 참조된 개체가 가비지 수집되지 않도록 합니다.

대부분의 JNI 함수에 의해 생성되는 로컬 참조와 달리 전역 참조는 하나의 JNI 함수인 NewGlobalRef에 의해 생성됩니다. 다음 버전의 MyNewString은 전역 참조를 사용하는 방법을 보여줍니다. 아래 코드와 마지막 섹션에서 실수로 로컬 참조를 캐시한 코드의 차이점을 강조합니다.

```
/* 이 코드는 정상입니다 */ jstring
MyNewString(JNIEnv *env, jchar *chars, jint len) {
    정적 jclass 문자열 클래스 = NULL;
    ...
    if (stringClass == NULL) { jclass
        localRefCls = (*env)->FindClass(env, "java/lang/String"); if (localRefCls == NULL)
        { 반환 NULL; /* 예외 발생 */
        }
        /* 전역 참조 생성 */ stringClass = (*env)->NewGlobalRef(env, localRefCls);

        /* 로컬 참조는 더 이상 유용하지 않습니다. */ (*env)->DeleteLocalRef(env,
        localRefCls);

        /* 전역 참조가 성공적으로 생성되었습니까? */ if (문자열 클래스 == NULL) {
            NULL을 반환합니다. /* 메모리 부족 예외 발생 */
        }
    }
    ...
}
```

수정된 버전은 FindClass에서 반환된 로컬 참조를 NewGlobalRef로 전달하여 java.lang.String 클래스 자체에 대한 전역 참조를 만듭니다. 두 경우 모두 로컬 참조 localRefCls를 삭제해야 하기 때문에 localRefCls를 삭제 한 후 NewGlobalRef가 문자열 클래스를 성공적으로 생성했는지 확인합니다.

5.1.3 약한 전역 참조

약한 글로벌 참조는 Java 2 SDK 릴리스 1.2의 새로운 기능입니다. NewGlobalWeakRef를 사용하여 생성되고 DeleteGlobalWeakRef를 사용하여 해제됩니다. 전역 참조와 마찬가지로 약한 전역 참조는 네이티브 메서드 호출과 다른 스레드 간에 유효합니다. 전역 참조와 달리 약한 전역 참조는 기본 객체가 가비지 수집되는 것을 막지 않습니다.

MyNewString 예제는 java.lang.String 클래스에 대한 전역 참조를 캐시하는 방법을 보여줍니다. MyNewString 예제는 약한 전역 참조를 대신 사용하여 캐시된 java.lang.String 클래스를 저장할 수 있습니다. java.lang.String은 시스템 클래스이고 가비지 수집되지 않기 때문에 전역 참조를 사용하는 약한 전역 참조를 사용하는 상관 없습니다.

약한 전역 참조는 네이티브 코드에 의해 캐시된 참조가 기본 객체가 가비지 수집되는 것을 막지 않아야 할 때 더 유용합니다.

예를 들어 네이티브 메서드 mypkg.MyCls.f가 클래스 mypkg.MyCls2에 대한 참조를 캐시해야 한다고 가정합니다. 약한 전역 참조에 클래스를 캐싱하면 mypkg.MyCls2 클래스를 계속 언로드할 수 있습니다.

```
JNIEXPORT 무효 JNICALL
Java_mypkg_MyCls_f(JNIEnv *env, jobject self) {
    정적 jclass myCls2 = NULL; if (myCls2 ==
    NULL) { jclass myCls2Local = (*env)-
        >FindClass(env, "mypkg/MyCls2");
        if (myCls2Local == NULL) { 반환; /* 클래스를 찾을 수 없습니다
    */
    }
    myCls2 = NewWeakGlobalRef(env, myCls2Local); if (myCls2 ==
    NULL) { 반환; /* 메모리 부족 */
    }
}
} ... /* myCls2 사용 */
}
```

MyCls와 MyCls2의 수명이 같다고 가정합니다. (예를 들어, 동일한 클래스 로더에 의해 로드될 수 있습니다.) 따라서 우리는 다음과 같은 경우를 고려하지 않습니다.

5.1.4

참조 비교

로컬 및 글로벌 참조

MyCls2 는 언로드되었다가 나중에 다시 로드되는 반면 MyCls 및 해당 네이티브 메서드 구현 Java_mypkg_MyCls 는 계속 사용됩니다. 그런 일이 발생할 수 있다면 캐시된 약한 참조가 여전히 라이브 클래스 객체를 가리키는지 아니면 이미 가버지 수집된 클래스 객체를 가리키는지 확인해야 합니다. 다음 섹션에서는 약한 글로벌 참조에 대해 이러한 검사를 수행하는 방법을 설명합니다.

5.1.4 참조 비교

두 개의 로컬, 전역 또는 약한 전역 참조가 주어지면 IsSameObject 함수를 사용하여 동일한 객체를 참조하는지 여부를 확인할 수 있습니다. 예를 들어:

```
(*env)->IsSameObject(env, obj1, obj2)
```

obj1 과 obj2가 동일한 객체를 참조 하면 JNI_TRUE (또는 1) 를 반환하고 그렇지 않으면 JNI_FALSE (또는 0)를 반환합니다.

JNI의 NULL 참조는 JVM(Java Virtual Machine)의 널 객체를 참조 합니다. obj가 로컬 또는 전역 참조 인 경우 둘 중 하나를 사용할 수 있습니다.

```
(*env)->IsSameObject(env, obj, NULL)
```

또는

객체 == NULL

obj가 null 객체를 참조하는지 확인합니다.

약한 전역 참조에 대한 규칙은 다소 다릅니다. NULL 약한 참조는 null 객체를 참조합니다. 그러나 IsSameObject는 약한 전역 참조에 대해 특별한 용도가 있습니다. IsSameObject를 사용하여 NULL이 아닌 약한 전역 참조가 여전히 활성 개체를 가리키는지 여부를 확인할 수 있습니다. wobj가 NULL이 아닌 약한 전역 참조 라고 가정합니다. 다음 호출:

```
(*env)->IsSameObject(env, wobj, NULL)
```

wobj가 이미 수집된 개체를 참조하는 경우 JNI_TRUE를 반환하고 wobj가 여전히 활성 개체를 참조하는 경우 JNI_FALSE를 반환합니다.

5.2 참조 해제

각 JNI 참조는 참조된 개체가 차지하는 메모리 외에 자체적으로 일정량의 메모리를 사용합니다. JNI 프로그래머는 주어진 시간에 프로그램이 사용할 참조 수를 알고 있어야 합니다. 특히, 지역 참조 횟수의 상한선을 알고 있어야 합니다.

이러한 로컬 참조는 결국 가상 머신에 의해 자동으로 해제되더라도 프로그램이 실행되는 동안 언제든지 생성할 수 있습니다.

과도한 참조 생성은 일시적이지만 메모리 고갈로 이어질 수 있습니다.

5.2.1 로컬 참조 해제

대부분의 경우 네이티브 메서드를 구현할 때 로컬 참조를 해제하는 것에 대해 걱정할 필요가 없습니다. 원시 메소드가 호출자에게 리턴되면 JVM(Java Virtual Machine)이 이를 해제합니다. 그러나 JNI 프로그래머가 과도한 메모리 사용을 피하기 위해 로컬 참조를 명시적으로 해제해야 하는 경우가 있습니다. 다음 상황을 고려하십시오.

- 단일 네이티브 메서드 호출에서 많은 수의 로컬 참조를 생성해야 합니다. 이로 인해 내부 JNI 로컬 참조 테이블이 오버플로될 수 있습니다. 필요하지 않은 로컬 참조는 즉시 삭제하는 것이 좋습니다. 예를 들어 다음 프로그램 세그먼트에서 네이티브 코드는 잠재적으로 큰 문자열 배열을 반복합니다. 각 반복 후 네이티브 코드는 다음과 같이 문자열 요소에 대한 로컬 참조를 명시적으로 해제해야 합니다.

```
for (i = 0; i < len; i++) {
    jstring jstr = (*env)->GetObjectArrayElement(env, arr, i); /* 프로세스 jstr */ (*env)->DeleteLocalRef(env, jstr);
}
```

- 알 수 없는 컨텍스트에서 호출되는 유ти리티 함수를 작성하려고 합니다. 섹션 4.3에 표시된 MyNewString 예제는 Delete LocalRef를 사용하여 유ти리티 기능에서 즉시 로컬 참조를 삭제하는 방법을 보여 줍니다. 그렇지 않으면 MyNewString 함수를 호출할 때마다 할당된 상태로 남아 있는 두 개의 로컬 참조가 있습니다. • 네이티브 메서드가 전혀 반환되지 않습니다. 예를 들어 네이티브 메서드는 무한 이벤트 디스패치 루프에 들어갈 수 있습니다. 무한정 누적되어 메모리 누수가 발생하지 않도록 루프 내부에서 생성된 로컬 참조를 해제하는 것이 중요합니다.

- 네이티브 메서드는 대형 객체에 액세스하여 객체에 대한 로컬 참조를 생성합니다. 그런 다음 네이티브 메서드는 호출자에게 반환하기 전에 추가 계산을 수행합니다. 큰 객체에 대한 로컬 참조는 개체가 네이티브 메서드의 나머지 부분에서 더 이상 사용되지 않는 경우에도 네이티브 메서드가 반환될 때까지 개체가 가비지 수집되지 않도록 합니다. 예를 들어, 다음 프로그램 세그먼트에서 미리 DeleteLocalRef에 대한 명시적 호출이 있기 때문에 가비지 수집기는

5.2.2 Java 2 SDK 릴리스 1.2에서 로컬 참조 관리 및 글로벌 참조

실행이 함수 내부에 있을 때 lref 가 참조하는 객체
계산:

```
/* 네이티브 메서드 구현 */
JNIEXPORT void JNICALL Java_pkg_Cls_func(JNIEnv *env, jobject this) {
    lref = ... /* 큰 Java 객체 */ /* lref의 마지막 사용 */
    ...
    (*env)->DeleteLocalRef(env, lref);

    길이 계산(); /* 다소 시간이 걸릴 수 있음 */ /* 모든 로컬 참조가 해제됨 */ return;
}
```

5.2.2 Java 2 SDK 릴리스 1.2에서 로컬 참조 관리

Java 2 SDK 릴리스 1.2는 로컬 참조의 수명을 관리하기 위한 추가 기능 세트를 제공합니다. 이러한 함수는 ConfirmLocalCapacity, New LocalRef, PushLocalFrame 및 PopLocalFrame입니다.

JNI 사양에 따르면 가상 머신은 각 네이티브 메서드가 최소 16개의 로컬 참조를 생성할 수 있도록 자동으로 보장합니다. 경험에 따르면 이는 JVM(Java Virtual Machine)의 개체와의 복잡한 상호 작용을 포함하지 않는 대부분의 기본 메서드에 충분한 용량을 제공합니다. 그러나 추가 로컬 참조를 생성해야 하는 경우 네이티브 메서드는 충분한 수의 로컬 참조를 위한 공간을 사용할 수 있도록 보장LocalCapacity 호출을 실행할 수 있습니다. 예를 들어, 위의 이전 예제를 약간 변형하면 충분한 메모리를 사용할 수 있는 경우 루프 실행 중에 생성된 모든 로컬 참조에 대해 충분한 용량을 예약합니다.

```
/* 생성할 로컬 참조의 수는 배열의 길이와 같습니다. */

if ((*env)->EnsureLocalCapacity(env, len)) < 0) {
    ... /* 메모리 부족 */
}
for (i = 0; i < len; i++) {
    jstring jstr = (*env)->GetObjectArrayElement(env, arr, i); /* 프로세스 jstr */ /*
    DeleteLocalRef는 더 이상 필요하지 않음 */
}
```

물론 위의 버전은 이전 버전보다 더 많은 메모리를 소비할 가능성이 높습니다.
로컬 참조를 즉시 삭제하는 ous 버전입니다.

또는 Push/PopLocalFrame 함수를 사용하여 프로그래머가 로컬 참조의 중첩된 범위를 만들 수 있습니다. 예를 들어, 동일한 예를 다음과 같이 다시 작성할 수도 있습니다.

```
#define N_REFS ... /* 로컬 참조의 최대 수
                   각 반복에서 사용됨 */
for (i = 0; i < len; i++) { if ((*env)->PushLocalFrame(env, N_REFS) < 0) {
    ... /* 메모리 부족 */
}
jstr = (*env)->GetObjectArrayElement(env, arr, i); ... /* 프로세스 jstr */
(*env)->PopLocalFrame(env, NULL);
}
```

PushLocalFrame은 특정 수의 로컬 참조에 대한 새 범위를 만듭니다.

PopLocalFrame은 최상위 범위를 파괴하여 해당 범위의 모든 로컬 참조를 해제합니다. 범위.

Push/PopLocalFrame 함수를 사용하는 이점은 실행 중에 생성될 수 있는 모든 단일 로컬 참조에 대해 걱정 할 필요 없이 로컬 참조의 수명을 관리할 수 있다는 것입니다. 위의 예에서 jstr을 처리하는 계산이 추가 로컬 참조를 생성하는 경우 이러한 로컬 참조는 PopLocalFrame이 반환된 후 해제됩니다.

NewLocalRef 함수는 로컬 참조를 반환할 것으로 예상되는 유ти리티 함수를 작성할 때 유용합니다. 섹션 5.3에서 New LocalRef 기능의 사용을 시연할 것입니다.

네이티브 코드는 기본 용량인 16 또는 PushLocalFrame 또는 MakeLocalCapacity 호출에 예약된 용량을 초과하는 로컬 참조를 생성할 수 있습니다. 가상 머신 구현은 로컬 참조에 필요한 메모리 할당을 시도합니다. 그러나 메모리를 사용할 수 있다는 보장은 없습니다.

메모리 할당에 실패하면 가상 머신이 종료됩니다. 이러한 예기치 않은 가상 머신 종료를 방지하려면 로컬 참조 및 무료로컬 참조를 위한 충분한 메모리를 즉시 예약해야 합니다.

Java 2 SDK 릴리스 1.2는 명령줄 옵션 -verbose:jni를 지원합니다.

이 옵션이 활성화되면 가상 머신 구현은 예약된 용량을 초과하는 과도한 로컬 참조 생성을 보고합니다.

5.2.3 전역 참조 해제

네이티브 코드에서 전역 참조에 더 이상 액세스할 필요가 없으면 DeleteGlobalRef를 호출해야 합니다. 이 함수를 호출하지 못하면 JVM(Java Virtual Machine)은 객체가 시스템의 다른 곳에서 더 이상 사용되지 않는 경우에도 해당 객체를 가비지 수집하지 않습니다.

네이티브 코드에서 더 이상 약한 전역 참조에 액세스할 필요가 없으면 `DeleteWeakGlobalRef`를 호출해야 합니다. 이 함수를 호출하는 데 실패하면 JVM(Java Virtual Machine)은 여전히 기본 개체를 가비지 수집할 수 있지만 약한 전역 참조 자체에서 소비한 메모리를 회수할 수 없습니다.

5.3 레퍼런스 관리 규칙

이제 이전 섹션에서 다른 내용을 기반으로 네이티브 코드에서 JNI 참조를 관리하기 위한 규칙을 살펴볼 준비가 되었습니다. 목표는 불필요한 메모리 사용 및 개체 보존을 제거하는 것입니다.

일반적으로 네이티브 코드에는 두 가지 종류가 있습니다.
임의의 컨텍스트에서 사용되는 기본 메서드 및 유ти리티 함수를 나타냅니다.

네이티브 메서드를 직접 구현하는 함수를 작성할 때 루프에서 과도한 로컬 참조 생성 및 반환하지 않는 네이티브 메서드로 인한 불필요한 로컬 참조 생성에 주의해야 합니다. 기본 메서드가 반환된 후 삭제하기 위해 가상 머신에 사용 중인 로컬 참조를 최대 16개까지 남겨둘 수 있습니다. 전역 및 약한 전역 참조는 기본 메서드가 반환된 후 자동으로 해제되지 않기 때문에 기본 메서드 호출로 인해 전역 또는 약한 전역 참조가 누적되지 않아야 합니다.

네이티브 유ти리티 함수를 작성할 때 함수 전체의 실행 경로에서 로컬 참조가 누출되지 않도록 주의해야 합니다. 유ти리티 함수는 예기치 않은 컨텍스트에서 반복적으로 호출될 수 있으므로 불필요한 참조 생성으로 인해 메모리 오버플로가 발생할 수 있습니다.

- 원시 타입을 반환하는 유ти리티 함수 호출 시 추가적인 로컬, 글로벌, 약한 글로벌 참조가 누적되는 부작용이 없어야 함
ences.
- 참조 유형을 반환하는 유ти리티 함수가 호출될 때 결과로 반환되는 참조 외에 추가 로컬, 전역 또는 약한 전역 참조를 누적해서는 안 됩니다.

첫 번째 호출에서만 이러한 참조를 생성하기 때문에 유ти리티 함수가 캐싱 목적으로 일부 전역 또는 약한 전역 참조를 생성하는 것은 허용됩니다.

유ти리티 함수가 참조를 반환하는 경우 반환된 참조의 종류를 함수 사양의 일부로 만들어야 합니다. 어떤 때는 로컬 참조를 반환하고 다른 때는 전역 참조를 반환하면 안 됩니다. 호출자는 자신의 JNI 참조를 올바르게 관리하기 위해 유ти리티 함수가 반환하는 참조 유형을 알아야 합니다. 예를 들어 다음 코드는 유ти리티 함수 `GetInfoString`을 반복적으로 호출합니다. 반환된 참조 유형을 알아야 합니다.

`GetInfoString`은 각 반복 후 반환된 JNI 참조를 적절하게 해제할 수 있도록 합니다.

```
while (JNI_TRUE) { jstring
    infoString = GetInfoString(정보); ... /* 프로세스 infoString */

    ??? /* GetInfoString이 반환한 참조 유형에 따라 DeleteLocalRef, DeleteGlobalRef 또는 DeleteWeakGlobalRef를 호출해야 합니다. */

}
```

Java 2 SDK 릴리스 1.2에서 `NewLocalRef` 함수는 때때로 유틸리티 함수가 항상 로컬 참조를 반환하도록 하는 데 유용합니다. 설명을 위해 `MyNewString` 함수에 또 다른(다소 인위적인) 변경을 가해 보겠습니다. 다음 버전은 전역 참조에서 자주 요청되는 문자열(예: "CommonString")을 캐시합니다.

```
jstring
MyNewString(JNIEnv *env, jchar *chars, jint len) {

    정적 jstring 결과;

    /* wstrcmp는 두 개의 유니코드 문자열을 비교합니다. */ if
    (wstrcmp("CommonString", chars, len) == 0) {
        /* 글로벌 ref 캐싱 "CommonString" 참조 */ static jstring cachedString = NULL;
        if (cachedString == NULL) { /* 처음으로 cachedString 생성 */ jstring
            cachedStringLocal = /* 전역 참조에 결과 캐시 */ cachedString = (*env)-
                >NewGlobalRef(env, cachedStringLocal);
            ...
        }
        return (*env)->NewLocalRef(env, cachedString);
    }

    ... /* 문자열을 로컬 참조로 생성하고 결과를 로컬 참조로 저장 */ return result;
}
```

일반 경로는 문자열을 로컬 참조로 반환합니다. 앞에서 설명한 것처럼 캐시된 문자열을 전역 참조에 저장해야 여러 네이티브 메서드 호출 및 여러 스레드에서 액세스할 수 있습니다. 강조 표시된 줄은 캐시된 전역 객체와 동일한 객체를 참조하는 새로운 로컬 참조를 생성합니다.

참조. 호출자와의 계약의 일부로 MyNewString은 항상 로컬 참조를 반환합니다.

Push /PopLocalFrame 함수는 로컬 참조의 수명을 관리하는 데 특히 편리합니다. 네이티브 함수 진입 시 PushLocalFrame을 호출한 경우 네이티브 함수가 반환되기 전에 PopLocalFrame을 호출하면 네이티브 함수 실행 중에 생성된 모든 로컬 참조가 해제됩니다. Push /PopLocalFrame 함수는 효율적입니다. 이를 사용하는 것이 좋습니다.

함수 시작 시 PushLocalFrame을 호출하는 경우 모든 함수 종료 경로에서 Pop LocalFrame 을 호출해야 합니다. 예를 들어 다음 함수에는 PushLocalFrame에 대한 호출이 한 번 있지만 PopLocalFrame에 대한 여러 호출이 필요합니다 .

```
jobject f(JNIEnv *env, ...) {
    작업 결과; if ((*env)->PushLocalFrame(env, 10) < 0) {
        /* 프레임이 푸시되지 않음, PopLocalFrame 필요 없음 */ return NULL;
    }
    ...
    결과 = ...; if (...) { /* 반환 전에 로컬 프레임 을 팝하는 것을 기억하십시오. */ result = (*env)->PopLocalFrame(env, result); } 반환 결과;
    ...
    result = (*env)->PopLocalFrame(env, result); /* 정상 반환 */ 반환 결과;
}
}
```

PopLocalFrame 호출을 제대로 배치하지 못하면 가상 머신 충돌과 같은 정의되지 않은 동작이 발생할 수 있습니다.

위의 예는 또한 PopLocalFrame에 대한 두 번째 인수를 지정하는 것이 때때로 유용한 이유를 보여줍니다 . 결과 로컬 참조는 처음에 PushLocalFrame에 의해 생성된 새 프레임에 생성됩니다 . PopLocalFrame은 최상위 프레임을 팝하기 전에 두 번째 인수 결과를 이전 프레임의 새 로컬 참조로 변환합니다.

6 장

예외

우리는 네이티브 코드가 다음을 확인하는 수많은 상황에 직면했습니다.

JNI 함수 호출 후 발생할 수 있는 오류. 이 장에서는 네이티브 코드가 이러한 오류 조건을 감지하고 복구하는 방법을 살펴봅니다.

네이티브 코드에서 발생하는 임의의 오류가 아닌 JNI 함수 호출을 실행한 결과 발생하는 오류에 중점을 둘 것입니다. 네이티브 메서드가 운영 체제를 호출하는 경우 시스템 호출에서 가능한 오류를 확인하는 문서화된 방법을 따릅니다. 반면에 네이티브 메서드가 Java API 메서드에 대한 콜백을 발행하는 경우 메서드 실행에서 발생한 가능한 예외를 적절하게 확인하고 복구하기 위해 이 장에 설명된 단계를 따라야 합니다.

6.1 개요

일련의 예제를 통해 JNI 예외 처리 기능을 소개합니다.

6.1.1 네이티브 코드에서 캐싱 및 예외 발생

아래 프로그램은 예외를 발생시키는 네이티브 메서드를 선언하는 방법을 보여줍니다. CatchThrow 클래스는 doit 네이티브 메서드를 선언하고 IllegalArgumentException을 발생시킬도록 지정합니다.

```
class CatchThrow { private
    native void doit() throws
        IllegalArgumentException; 개인 무효 콜백()이
    NullPointerException을 발생시킵니다 {
        throw new NullPointerException("CatchThrow.callback");
    }
}
```

6.1.1 네이티브 코드에서 캐싱 및 예외 발생

예외

```

public static void main(String args[]) { CatchThrow c = new
    CatchThrow(); 시도 {c.doit(); } catch (Exception e)
    { System.out.println("In Java:\n\t" + e);

}

}
정적
{ System.loadLibrary("CatchThrow");
}
}

```

CatchThrow.main 메서드 는 다음과 같이 구현된 네이티브 메서드 doit를 호출합니다.

JNIEXPORT 무효 JNICALL

```

Java_CatchThrow_doit(JNIEnv *env, jobject obj) {

    jthrowable 예외; jclass
    cls = (*env)->GetObjectClass(env, obj); jmethodID mid = (*env)-
    >GetMethodID(env, cls, "콜백", "()V"); if (mid == NULL) { 반환;

}

(*env)->CallVoidMethod(env, obj, mid); exc = (*env)-
>ExceptionOccurred(env); if (exc) { /* 예외에 대해 디버그 메
시지를 인쇄하고 지우고 새 예외를 발생시키는 것을 제외하고는 예외
에 대해 많은 작업을 수행하지 않습니다. */ jclass newExcCls; (*env)-
>ExceptionDescribe(env); (*env)->ExceptionClear(env); newExcCls =
(*env)->FindClass(env, "java/lang/IllegalArgumentException"); if
(newExcCls == NULL) {

/* 예외 클래스를 찾을 수 없습니다. 포기합니다. */ 반환;

}
(*env)->ThrowNew(env, newExcCls, "C 코드에서 발생");
}
}
```

예외

유ти리티 기능

6.1.2

네이티브 라이브러리로 프로그램을 실행하면 다음과 같은 결과가 생성됩니다.

```
java.lang.NullPointerException: at
    CatchThrow.callback(CatchThrow.java) at
    CatchThrow.doit(Native Method) at
    CatchThrow.main(CatchThrow.java)
Java:
    java.lang.IllegalArgumentException: C 코드에서 발생
```

콜백 메서드는 NullPointerException을 발생시킵니다. CallVoid 메서드가 컨트롤을 네이티브 메서드로 반환하면 네이티브 코드는 JNI 함수 ExceptionOccurred를 호출하여 이 예외를 감지합니다. 이 예에서 예외가 감지되면 네이티브 코드는 ExceptionDescribe를 호출하여 예외에 대한 설명 메시지를 출력하고 Exception Clear를 사용하여 예외를 지우고 대신 IllegalArgumentException을 발생시킵니다.

JNI를 통해 발생하는 보류 중인 예외(예 : ThrowNew 호출)는 네이티브 메서드 실행을 즉시 방해하지 않습니다. 이는 Java 프로그래밍 언어에서 예외가 작동하는 방식과 다릅니다. Java 프로그래밍 언어에서 예외가 발생하면 가상 머신은 예외 유형과 일치하는 가장 가까운 주변 try/catch 문으로 제어 흐름을 자동으로 전송합니다 . 그런 다음 가상 머신은 보류 중인 예외를 지우고 예외 처리기를 실행합니다. 반대로 JNI 프로그래머는 예외가 발생한 후 제어 흐름을 명시적으로 구현해야 합니다.

6.1.2 유ти리티 기능

예외를 throw하려면 먼저 예외 클래스를 찾은 다음 ThrowNew 함수를 호출해야 합니다. 작업을 단순화하기 위해 명명된 예외를 발생시키는 유ти리티 함수를 작성할 수 있습니다.

```
무효의
JNU_ThrowByName(JNIEnv *env, const char *name, const char *msg) {
    jclass cls = (*env)->FindClass(env, name); /* cls가 NULL이면 예
    외가 이미 발생한 것입니다. */ if (cls != NULL) { (*env)->ThrowNew(env, cls, msg);

} /* 로컬 참조 해제 */ (*env)->DeleteLocalRef(env, cls);
}
```

이 책에서 JNU 접두어는 JNI 유ти리티를 나타냅니다 . JNU_ThrowByName은 먼저 FindClass 함수를 사용하여 예외 클래스를 찾습니다 . FindClass 가 실패하는 경우

6.2	적절한 예외 처리	예외
(NULL 반환), 가상 머신에서 예외(예: NoClassDefFoundError)가 발생했을 것입니다. 이 경우 JNU_ThrowByName은 다른 예외 발생을 시도하지 않습니다. FindClass가 성공하면 ThrowNew 를 호출하여 명명된 예외를 throw합니다 . JNU_ThrowByName이 반환될 때 보류 중인 예외가 반드시 name 인수로 지정되는 것은 아니지만 보류 중인 예외가 있음을 보장합니다 . 이 함수에서 생성된 예외 클래스에 대한 로컬 참조를 삭제해야 합니다. DeleteLocalRef에 NULL을 전달하는 것은 no op이며 FindClass가 실패하고 NULL을 반환하는 경우 적절한 조치입니다.		

6.2 적절한 예외 처리

JNI 프로그래머는 가능한 예외 조건을 예측하고 이러한 경우를 확인하고 처리하는 코드를 작성해야 합니다. 적절한 예외 처리는 때때로 지루하지만 강력한 애플리케이션을 생성하기 위해 필요합니다.

6.2.1 예외 확인

오류가 발생했는지 여부를 확인하는 방법에는 두 가지가 있습니다.

1. 대부분의 JNI 함수는 고유한 반환 값(예: NULL)을 사용하여 오류가 발생했음을 나타냅니다. 오류 반환 값은 또한 현재 스레드에 보류 중인 예외가 있음을 의미합니다. (반환 값의 인코딩 오류 조건은 C에서 일반적입니다.)

다음 예에서는 GetFieldID에서 반환된 NULL 값을 사용하여 오류를 확인하는 방법을 보여줍니다. 이 예제는 여러 인스턴스 필드 (핸들, 길이 및 너비)를 정의하는 Window 클래스와 이러한 필드의 필드 ID를 캐시하는 네이티브 메서드의 두 부분으로 구성됩니다. 이러한 필드가 Window 클래스에 존재하더라도 가상 머신이 필드 ID를 나타내는 데 필요한 메모리를 할당하지 못할 수 있으므로 GetFieldID에서 반환된 가능한 오류를 확인해야 합니다.

```
/* Java 프로그래밍 언어의 클래스 */ public class Window { long handle;
```

```
정수 길이; 정수 너비; 정적 네이티브 무효 초기화 ID(); 정적 { initIDs();
```

```
}
```

예외

예외 확인

6.2.1

```
/* Window.initIDs를 구현하는 C 코드 */ jfieldID FID_Window_handle;
jfieldID FID_Window_length; jfieldID FID_Window_width;

JNIEXPORT void JNICALL Java_Window_initIDs(JNIEnv *env, jclass classWindow) {

    FID_Window_handle =
        (*env)->GetFieldID(env, classWindow, "handle", "J"); if (FID_Window_handle
        == NULL) { /* 중요한 체크. */
            반품; /* 오류가 발생했습니다. */
        }
    FID_Window_length =
        (*env)->GetFieldID(env, classWindow, "길이", "I"); if (FID_Window_length ==
        NULL) { /* 중요한 체크. */
            반품; /* 오류가 발생했습니다. */
        }
    FID_Window_width =
        (*env)->GetFieldID(env, classWindow, "너비", "I"); /* 검사가 필요하지 않습니다.
        우리는 어쨌든 돌아올 것입니다 */
}
```

2. 반환 값이 오류가 발생했음을 표시할 수 없는 JNI 함수를 사용하는 경우 네이티브 코드는 발생 한 예외에 의존하여 오류 검사를 수행해야 합니다. 현재 스레드에서 보류 중인 예외를 확인하는 JNI 함수는 `ExceptionOccurred`입니다. (`ExceptionCheck`는 Java 2 SDK 릴리스 1.2에도 추가되었습니다.) 예를 들어 JNI 함수 `CallIntMethod`는 반환 값에서 오류 조건을 인코딩할 수 없습니다. `NULL` 및 `-1`과 같은 일반적인 오류 조건 반환 값 선택은 호출된 메서드에서 반환된 을 바른 값일 수 있으므로 작동하지 않습니다. 바닥 메서드가 분수 값의 정수 부분을 반환하는 `Fraction` 클래스 와 이 메서드를 호출하는 일부 네이티브 코드를 고려하십시오 .

```
public class Fraction { // 생략된 생
    성자와 같은 세부 사항 int over, under; public int floor()
    { return Math.floor((double)over/under);

    }
}
```

6.2.2 예외 처리

예외

```

/* Fraction.floor를 호출하는 네이티브 코드. 메서드 ID MID_Fraction_floor가 다른 곳에서 초
기화되었다고 가정합니다. */
무효 f(JNIEnv *env, jobject fraction) {

    jint floor = (*env)->CallIntMethod(env, fraction,
                                         MID_Fraction_floor); /* 중요:
    예외가 발생했는지 확인 */ if ((*env)->ExceptionCheck(env)) { return;

}

} ... /* 바닥 사용 */
}

```

JNI 함수가 고유한 오류 코드를 반환하는 경우 네이티브 코드는 예를 들어 `ExceptionCheck`를 호출하여 명시적으로 예외를 확인할 수 있습니다. 그러나 고유한 오류 반환 값을 대신 확인하는 것이 더 효율적입니다. JNI 함수가 오류 값을 반환하는 경우 현재 스레드의 후속 `ExceptionCheck` 호출은 `JNI_TRUE`를 반환하도록 보장됩니다.

6.2.2 예외 처리

네이티브 코드는 두 가지 방법으로 보류 중인 예외를 처리할 수 있습니다.

- 네이티브 메서드 구현은 즉시 반환하도록 선택할 수 있습니다.
호출자에서 처리할 예외입니다.
- 네이티브 코드는 `ExceptionClear`을 호출하여 예외를 지운 다음
자체 예외 처리 코드를 실행합니다.

후속 JNI 함수를 호출하기 전에 보류 중인 예외를 확인, 처리 및 자우는 것이 매우 중요합니다. 보류 중인 예외(명시적으로 지우지 않은 예외 포함)가 있는 대부분의 JNI 함수를 호출하면 예기치 않은 결과가 발생할 수 있습니다. 현재 스레드에 보류 중인 예외가 있는 경우 소수의 JNI 함수만 안전하게 호출할 수 있습니다. 섹션 11.8.2는 이러한 JNI 함수의 전체 목록을 지정합니다. 일반적으로 보류 중인 예외가 있는 경우 예외를 처리하도록 설계된 JNI 함수와 JNI를 통해 노출되는 다양한 가상 머신 리소스를 해제하는 JNI 함수를 호출할 수 있습니다.

예외가 발생할 때 리소스를 해제할 수 있어야 하는 경우가 많습니다. 다음 예제에서 네이티브 메서드는 먼저 `GetStringChars` 호출을 실행하여 문자열의 내용을 가져옵니다. 후속 작업이 실패하면 `ReleaseStringChars`를 호출합니다.

```
JNIEXPORT 무효 JNICALL
Java_pkg_Cls_f(JNIEnv *env, jclass cls, jstring jstr) {
    const jchar *cstr = (*env)->GetStringChars(env, jstr); if (c_str == NULL) { 반환;
}

...
if (...) {/* 예외 발생 */ (*env)->ReleaseStringChars(env,
    jstr, cstr); 빙품;

}
...
/* 정상 반환 */ (*env)->ReleaseStringChars(env, jstr, cstr);
}
```

보류 중인 예외가 있을 때 `ReleaseStringChars`에 대한 첫 번째 호출이 실행됩니다. 기본 메서드 구현은 문자열 리소스를 해제하고 먼저 예외를 지우지 않고 즉시 반환합니다.

6.2.3 유ти리티 기능의 예외

유ти리티 함수를 작성하는 프로그래머는 예외가 호출자 네이티브 메서드로 전파되도록 특별한 주의를 기울여야 합니다. 특히 다음 두 가지 문제를 강조합니다.

- 바람직하게는 유ти리티 함수는 예외가 발생했음을 나타내는 특수 반환 값을 제공해야 합니다. 이는 보류 중인 예외를 확인하는 호출자의 작업을 단순화합니다. • 또한 유ти리티 함수는 예외 처리 코드에서 로컬 참조를 관리하기 위한 규칙(§ 5.3)을 따라야 합니다.

설명을 위해 인스턴스 메서드의 이름과 설명자를 기반으로 콜백을 수행하는 유ти리티 함수를 소개하겠습니다.

```
jvalue
JNU_CallMethodByName(JNIEnv *env,
                      jboolean *hasException, jobject
                      obj, const char *name, const
                      char *descriptor, ...)

{
    va_list 인수; j클래스
    클래스; jmethodID
    중간;
```

6.2.3 유틸리티 기능의 예외

예외

```

jvalue 결과; if ((*env)-
>EnsureLocalCapacity(env, 2) == JNI_OK) {
    clazz = (*env)->GetObjectClass(env, obj); mid = (*env)-
>GetMethodID(env, clazz, 이름, 기술자);

    if (mid) { const
        char *p = 설명자; /* 반환 유형을 찾기 위해
        인수 유형을 건너뜁니다. */ while (*p != ')') p++; /* 건너뛰기 ')' */ p++;
        va_start(인수, 설명자); switch (*p) { case 'V': (*env)-
>CallVoidMethodV(env, obj, mid, args); 부서지다; 케이스 '[': 케이스
'L': result.l = (*env)->CallObjectMethodV(env, obj, mid, args);

        부서지다;
        케이스 'Z':
        result.z = (*env)->CallBooleanMethodV(env, obj, mid,
                                                args);
        부서지다;
        케이스 'B':
        result.b = (*env)->CallByteMethodV(
                                         env, obj, mid, args);
        부서지다;
        케이스 'C':
        result.c = (*env)->CallCharMethodV(
                                         env, obj, mid, args);
        부서지다;
        케이스 'S':
        result.s = (*env)->CallShortMethodV(
                                         env, obj, mid, args);
        부서지다;
        케이스 'I':
        result.i = (*env)->CallIntMethodV(env, obj, mid,
                                             args);
        부서지다;
        케이스 'J':
        result.j = (*env)->CallLongMethodV(
                                         env, obj, mid, args);
        부서지다;
        케이스 'F':
        result.f = (*env)->CallFloatMethodV(
                                         env, obj, mid, args);
    }
}

```

```

        env, obj, mid, args);
부서지다;
케이스 'D':
    result.d = (*env)->CallDoubleMethodV(env, obj, mid,
                                             args);
부서지다;
기본값: (*env)-
        >FatalError(env, "잘못된 설명자");
}
va_end(인수);
}
(*env)->DeleteLocalRef(env, clazz);
}
if (hasException)
    {*hasException = (*env)->ExceptionCheck(env);
}
반환 결과;
}

```

JNU_CallMethodByName은 다른 인수 중에서 jbool ean에 대한 포인터를 사용합니다 . jboolean은 모든 것이 성공하면 JNI_FALSE로 설정되고 이 함수를 실행하는 동안 예외가 발생하면 JNI_TRUE로 설정됩니다.

이는 JNU_CallMethodByName 호출자에게 가능한 예외를 확인하는 확실한 방법을 제공합니다.

JNU_CallMethodByName은 먼저 두 개의 로컬 참조를 생성할 수 있는지 확인합니다. 하나는 클래스 참조이고 다른 하나는 메서드 호출에서 반환된 결과입니다. 그런 다음 개체에서 클래스 참조를 가져오고 메서드 ID를 찾습니다. 반환 유형에 따라 switch 문은 해당 JNI 메서드 호출 함수로 디스패치합니다. 콜백이 반환된 후 hasException이 NULL이 아니면 ExceptionCheck를 호출하여 보류 중인 예외를 확인합니다.

ExceptionCheck 기능은 Java 2 SDK 릴리스 1.2의 새로운 기능입니다. ExceptionOccurred 함수와 유사합니다. 차이점은 ExceptionCheck는 예외 개체에 대한 참조를 반환하지 않지만 보류 중인 예외가 있을 때 JNI_TRUE를 반환하고 보류 중인 예외가 없을 때 JNI_FALSE를 반환한다는 것입니다.

ExceptionCheck는 네이티브 코드가 예외가 발생했는지 여부만 알면 되지만 예외 개체에 대한 참조는 얻을 필요가 없을 때 로컬 참조 관리를 단순화합니다. 이전 코드는 JDK 릴리스 1.1에서 다음과 같이 다시 작성해야 합니다.

```

if (hasException)
    { jthrowable exc = (*env)->ExceptionOccurred(env); *hasException =
예외 != NULL; (*env)->DeleteLocalRef(env, exc);
}

```

6.2.3 유틸리티 기능의 예외

예외

예외 개체에 대한 로컬 참조를 삭제하려면 추가 DeleteLocalRef 호출이 필요합니다.

JNU_CallMethodByName 함수를 사용하여 구현을 다시 작성할 수 있습니다.
다음과 같이 섹션 4.2의 InstanceMethodCall.nativeMethod 의 설정 :

```
JNIEXPORT 무효 JNICALL  
Java_InstanceMethodCall_nativeMethod(JNIEnv *env, jobject obj) {  
  
    printf("C에서\n");  
    JNU_CallMethodByName(env, NULL, obj, "콜백", "()V");  
}
```

JNU_CallMethodByName 호출 후 네이티브 메서드가 즉시 반환되기 때문에 예외를 확인할 필요가 없습니다.

7 장

호출 인터페이스

이것 이 장에서는 네이티브 애플리케이션에 JVM(Java Virtual Machine)을 내장하는 방법을 설명합니다. JVM(Java Virtual Machine) 구현은 일반적으로 기본 라이브러리로 제공됩니다. 네이티브 애플리케이션은 이 라이브러리에 연결하고 호출 인터페이스를 사용하여 JVM(Java Virtual Machine)을 로드할 수 있습니다. 실제로 JDK 또는 Java 2 SDK 릴리스의 표준 실행기 명령 (java) 은 Java 가상 머신과 연결된 단순한 C 프로그램에 지나지 않습니다. 런처는 명령줄 인수를 구문 분석하고 가상 머신을 로드하며 호출 인터페이스를 통해 Java 애플리케이션을 실행합니다.

7.1 자바 가상 머신 생성

호출 인터페이스를 설명하기 위해 Java 가상 머신을 로드하고 다음과 같이 정의된 Prog.main 메소드를 호출하는 C 프로그램을 살펴보겠습니다 .

```
public class Prog { public  
    static void main(String[] args) { System.out.println("Hello  
        World " + args[0]);  
    }  
}
```

다음 C 프로그램 invoke.c는 JVM(Java Virtual Machine)을 로드하고 Prog.main을 호출합니다.

7.1

자바 가상 머신 생성

호출 인터페이스

```
#포함 <jni.h>

#define PATH_SEPARATOR ':' /* Solaris에서 ':'로 정의 */ #define USER_CLASSPATH ":" /* 여기서
Prog.class는 */

기본() {
    JNIEnv *env;
    자바 VM *jvm; 진트
    레스; j클래스 클래
    스; jmethodID 종
    간; jstring jstr; jclass
    스트링클래스;
    jobjectArray 인수;
}

#ifndef JNI_VERSION_1_2
    JavaVMInitArgs vm_args;
    JavaVMOption 옵션[1];
    options[0].optionString = "-Djava.class.path=USER_CLASSPATH";
    vm_args.version = 0x00010002; vm_args.options = 옵션;
    vm_args.nOptions = 1; vm_args.ignoreUnrecognized =
    JNI_TRUE; /* Java VM 생성 */ res = JNI_CreateJavaVM(&jvm,
    (void**)&env, &vm_args); #else JDK1_1InitArgs vm_args;
    문자 클래스 경로[1024]; vm_args.version = 0x00010001;
    JNI_GetDefaultJavaVMInitArgs(&vm_args); /* 기본 시스템 클래스 경로에
    USER_CLASSPATH 추가 */ sprintf(classpath, "%s%c%s", vm_args.classpath,
    PATH_SEPARATOR, USER_CLASSPATH); vm_args.classpath = 클래스 경로; /*
    Java VM 생성 */ res = JNI_CreateJavaVM(&jvm, &env, &vm_args);
}

#endif /* JNI_VERSION_1_2 */

if (res < 0)
    { fprintf(stderr, "자바 VM을 생성할 수 없습니다\n"); 출구(1);

}
cls = (*env)->FindClass(env, "Prog"); if (cls == NULL)
{ goto destroy;

}
```

```

mid = (*env)->GetStaticMethodID(env, cls, "main",
                                  "[Ljava/lang/String;)V");
if (mid == NULL) { goto
    destroy;
}
jstr = (*env)->NewStringUTF(env, " from C!"); if (jstr == NULL)
{ goto destroy;

}
stringClass = (*env)->FindClass(env, "java/lang/String"); args = (*env)->NewObjectArray(env, 1, stringClass, jstr); if (args == NULL) { goto destroy;

}

(*env)->CallStaticVoidMethod(env, cls, mid, args);

파괴: if
    ((*env)->ExceptionOccurred(env)) { (*env)->ExceptionDescribe(env);
    }
    (*jvm)->DestroyJavaVM(jvm);
}

```

이 코드는 JDK 릴리스 1.1의 가상 머신 구현에 특정한 초기화 구조 `JDK1_1InitArgs`를 조건부로 컴파일합니다. Java 2 SDK 릴리스 1.2는 여전히 `JDK1_1InitArgs`를 지원 하지만 `JavaVMInitArgs`라는 범용 가상 머신 초기화 구조를 도입했습니다 . 상수 `JNI_VERSION_1_2`는 Java 2 SDK 릴리스 1.2에 정의되어 있지만 JDK 릴리스 1.1에는 정의되어 있지 않습니다.

1.1 릴리스를 대상으로 하는 경우 C 코드는 기본 가상 머신 설정을 가져오기 위해 `JNI_GetDefaultJavaVMInitArgs`를 호출하는 것으로 시작합니다 . `JNI_GetDefaultJavaVMInitArgs`는 `vm_args` 매개변수에서 힙 크기, 스택 크기, 기본 클래스 경로 등과 같은 값을 반환합니다 . 그런 다음 `Prog.class`가 있는 디렉토리를 `vm_args.classpath`에 추가합니다 .

1.2 릴리스를 대상으로 하는 경우 C 코드는 `JavaVMInitArgs` 구조를 생성합니다. 가상 머신 초기화 인수는 `JavaVMOption` 배열에 저장됩니다. 공통 옵션(예: `-Djava.class.path=.`) 과 Java 명령 줄 옵션에 직접 해당하는 구현 관련 옵션(예: `-Xmx64m`)을 모두 설정할 수 있습니다 . `ignoreUnrecognized` 필드를 `JNI_TRUE`로 설정하면 인식되지 않는 구현 관련 옵션을 무시하도록 가상 머신에 지시합니다.

가상 머신 초기화 구조를 설정한 후 C 프로그램은 `JNI_CreateJavaVM`을 호출하여 Java 가상 머신을 로드하고 초기화합니다. `JNI_CreateJavaVM` 함수는 두 개의 반환 값을 채웁니다.

7.2 자바 가상 머신과 네이티브 애플리케이션 연결

호출 인터페이스

- 새로 생성된 JVM(Java Virtual Machine)에 대한 인터페이스 포인터 jvm . • 현재 스레드에 대한 JNIEnv 인터페이스 포인터 env . 그 원주민을 기억하십시오
코드는 env 인터페이스 포인터를 통해 JNI 함수에 액세스합니다 .

JNI_CreateJavaVM 함수가 성공적으로 반환 되면 현재 네이티브 스레드가 JVM(Java Virtual Machine)으로 부트스트랩된 것입니다. 이 시점에서 네이티브 메서드처럼 실행되고 있습니다. 따라서 무엇보다도 JNI 호출을 실행하여 Prog.main 메서드를 호출할 수 있습니다.

결국 프로그램은 DestroyJavaVM 함수를 호출하여 Java 가상 머신을 언로드합니다. (안타깝게도 JDK 릴리스 1.1 또는 Java 2 SDK 릴리스 1.2에서는 JVM(Java Virtual Machine) 구현을 언로드할 수 없습니다. DestroyJavaVM은 이러한 릴리스에서 항상 오류 코드를 반환합니다.)

위의 프로그램을 실행하면 다음이 생성됩니다.

C의 Hello World!

7.2 네이티브 애플리케이션과 자바 가상 머신 연결

호출 인터페이스를 사용하려면 invoke.c 와 같은 프로그램을 JVM (Java Virtual Machine) 구현과 링크해야 합니다. JVM(Java Virtual Machine)과 연결하는 방법은 네이티브 애플리케이션이 특정 가상 머신 구현으로만 배포되도록 의도되었는지 아니면 다른 공급업체의 다양한 가상 머신 구현과 함께 작동하도록 설계되었는지에 따라 다릅니다.

7.2.1 알려진 Java 가상 머신과의 연결

특정 가상 머신 구현을 통해서만 기본 애플리케이션을 배포하도록 결정할 수 있습니다. 이 경우 네이티브 애플리케이션을 가상 머신을 구현하는 네이티브 라이브러리와 연결할 수 있습니다. 예를 들어 Solaris 용 JDK 1.1 릴리스에서는 다음 명령을 사용하여 invoke.c를 컴파일 및 링크할 수 있습니다 .

```
cc -I<jni.h 디렉토리> -L<libjava.so 디렉토리> -lthread -ljava invoke.c
```

-lthread 옵션은 기본 스레드 지원(§ 8.1.5) 과 함께 JVM(Java Virtual Machine) 구현을 사용함을 나타냅니다. -ljava 옵션은 libjava.so가 JVM (Java Virtual Machine)을 구현하는 Solaris 공유 라이브러리임을 지정합니다 .

Microsoft Visual C++ 컴파일러가 있는 Win32에서 다음 명령줄은
JDK 1.1 릴리스와 동일한 프로그램을 컴파일하고 링크하는 것은 다음과 같습니다.

```
cl -I<jni.h 디렉토리> -MD invoke.c -link <javai.lib 디렉토리>\javai.lib
```

물론 컴퓨터의 JDK 설치에 해당하는 올바른 포함 및 라이브러리 디렉토리를 제공해야 합니다.
-MD 옵션은 네이티브 애플리케이션이 JDK 1.1 및 Java 2 SDK 1.2 릴리스의 Java 가상 머신 구현에
서 사용되는 것과 동일한 C 라이브러리인 Win32 다중 스레드 C 라이브러리와 연결되도록 합니다. cl 명
령은 가상 머신에 구현된 JNI_CreateJavaVM 과 같은 호출 인터페이스 기능에 대한 연결 정보에 대
해 Win32에서 JDK 릴리스 1.1과 함께 제공되는 javai.lib 파일을 참조합니다 . 런타임에 사용되는 실
제 JDK 1.1 가상 머신 구현은 javai.dll이라는 별도의 동적 링크 라이브러리 파일에 포함되어 있습니
다. 대조적으로 동일한 Solaris 공유 라이브러리 (.so 파일)가 링크 시간과 실행 시간에 모두 사용됩니
다.

Java 2 SDK 릴리스 1.2에서는 가상 머신 라이브러리 이름이 Solaris에서는 libjvm.so 로,
Win32에서는 jvm.lib 및 jvm.dll 로 변경되었습니다 . 일반적으로 벤더마다 가상 머신 구현의 이름을
다르게 지정할 수 있습니다.

컴파일 및 연결이 완료되면 명령줄에서 결과 실행 파일을 실행할 수 있습니다. 시스템이 공유 라이
브러리 또는 동적 링크 라이브러리를 찾을 수 없다는 오류가 발생할 수 있습니다. Solaris에서 시스템
이 공유 라이브러리 libjava.so (또는 Java 2 SDK 릴리스 1.2의 lib jvm.so)를 찾을 수 없다는 오
류 메시지가 표시되면 가상 머신 라이브러리가 포함된 디렉토리를 LD_LIBRARY_PATH 변수에 추가해
야 합니다. Win32 시스템에서 오류 메시지는 동적 링크 라이브러리 javai.dll (또는 Java 2 SDK 릴리
스 1.2의 jvm.dll)을 찾을 수 없음을 나타낼 수 있습니다. 이 경우 DLL이 포함된 디렉터리를 PATH 환
경 변수에 추가합니다.

7.2.2 알 수 없는 자바 가상 머신과의 연결

애플리케이션이 다른 공급업체의 가상 머신 구현과 함께 작동하도록 의도된 경우 가상 머신을 구현하는
하나의 특정 라이브러리와 기본 애플리케이션을 연결할 수 없습니다. JNI는 JVM(Java Virtual
Machine)을 구현하는 기본 라이브러리의 이름을 지정하지 않기 때문에 다른 이름으로 제공되는
JVM(Java Virtual Machine) 구현으로 작업할 준비가 되어 있어야 합니다. 예를 들어 Win32에서 가
상 머신은 JDK 릴리스 1.1에서는 javai.dll 로, Java 2 SDK 릴리스 1.2에서는 jvm.dll 로 제공됩니다 .

7.2.2 알 수 없는 Java 가상 머신과 연결

호출 인터페이스

해결책은 런타임 동적 연결을 사용하여 애플리케이션에 필요한 특정 가상 머신 라이브러리를 로드하는 것입니다. 그러면 가상 머신 라이브러리의 이름을 애플리케이션별 방식으로 쉽게 구성할 수 있습니다. 예를 들어 다음 Win32 코드는 가상 머신 라이브러리의 경로가 지정된 JNI_CreateJavaVM에 대한 함수 진입점을 찾습니다.

```
/* Win32 버전 */ void
*JNU_FindCreateJavaVM(char *vmlibpath) {

    헌트 hVM = LoadLibrary(vmlibpath); if (hVM == NULL) { 반
    환 NULL;

    }
    GetProcAddress(hVM, "JNI_CreateJavaVM")를 반환합니다.
}
```

LoadLibrary 및 GetProcAddress는 Win32에서 동적 연결을 위한 API 함수입니다. LoadLibrary는 구현하는 기본 라이브러리의 이름(예: "jvm") 또는 경로(예: "C:\jdk1.2\jre\bin\classic\jvm.dll")를 허용할 수 있지만 JNU_FindCreateJavaVM에 네이티브 라이브러리의 절대 경로를 전달하는 것이 좋습니다. LoadL 라이브러리를 사용하여 jvm.dll을 검색하면 응용 프로그램이 PATH 환경 변수 추가와 같은 구성 변경에 영향을 받을 수 있습니다.

Solaris 버전도 비슷합니다.

```
/* 솔라리스 버전 */ void
*JNU_FindCreateJavaVM(char *vmlibpath) {

    무효 *libVM = dlopen(vmlibpath, RTLD_LAZY); if (libVM == NULL)
    { 반환 NULL;

    }
    return dlsym(libVM, "JNI_CreateJavaVM");
}
```

dlopen 및 dlsym 함수는 Solaris에서 공유 라이브러리를 동적으로 연결하는 것을 지원합니다.

7.3 네이티브 스레드 연결

C로 작성된 웹 서버와 같은 다중 스레드 응용 프로그램이 있다고 가정합니다. HTTP 요청이 들어오면 서버는 HTTP 요청을 동시에 처리하기 위해 여러 기본 스레드를 생성합니다. 그림 7.1과 같이 여러 스레드가 동시에 JVM(Java Virtual Machine)에서 작업을 수행할 수 있도록 이 서버에 JVM(Java Virtual Machine)을 내장하려고 합니다.

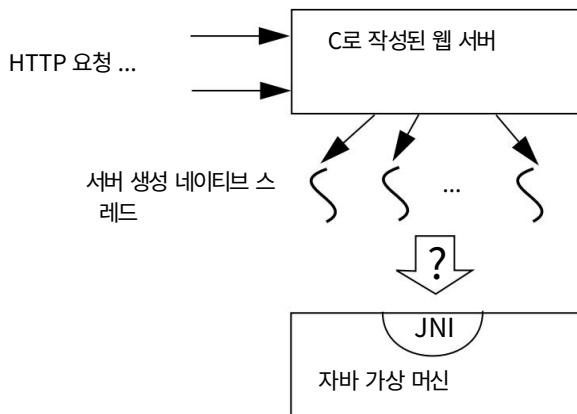


그림 7.1 웹 서버에 JVM(Java Virtual Machine) 내장

서버 생성 네이티브 메서드는 Java 가상 머신보다 수명이 짧을 수 있습니다. 따라서 이미 실행 중인 Java 가상 머신에 네이티브 스레드를 연결하고 연결된 네이티브 스레드에서 JNI 호출을 수행한 다음 연결된 다른 스레드를 방해하지 않고 네이티브 스레드를 가상 머신에서 분리하는 방법이 필요합니다.

다음 예제 attach.c는 호출 인터페이스를 사용하여 기본 스레드를 가상 머신에 연결하는 방법을 보여줍니다. 이 프로그램은 Win32 스레드 API를 사용하여 작성되었습니다. Solaris 및 기타 운영 체제용으로 유사한 버전을 작성할 수 있습니다.

7.3 네이티브 스레드 연결

호출 인터페이스

```

/* 참고: 이 프로그램은 Win32에서만 작동합니다. */ #include
<windows.h> #include <jni.h> JavaVM *jvm; /* 가상 머신 인스턴스
*/
#define PATH_SEPARATOR ';' #define
USER_CLASSPATH "." /* 여기서 Prog.class는 */

무효 thread_fun(공허 *arg) {

    진트레스; jclass
    스 클래스;
    jmethodID 중간;
    jstring jstr; jclass 스
    트링클래스; jobjectArray 인수;

    JNIEnv *env;
    char buf[100]; int
    threadNum = (int)arg;
    /* 세 번째 인수로 NULL을 전달 */ #ifdef JNI_VERSION_1_2
    res = (*jvm)->AttachCurrentThread(jvm, (void**)&env, NULL); #else res = (*jvm)-
    >AttachCurrentThread(jvm, &env, NULL); #endif if (res < 0) { fprintf(stderr, "첨부 실패\n"); 반
    품;

}

cls = (*env)->FindClass(env, "Prog"); if (cls == NULL)
{ goto detach;

}

mid = (*env)->GetStaticMethodID(env, cls, "main",
                                  "[Ljava/lang/String;)V");
if (mid == NULL) { goto
    detach;
}
sprintf(buf, " from Thread %d", threadNum); jstr = (*env)-
>NewStringUTF(env, buf); if (jstr == NULL) { goto detach;

}

stringClass = (*env)->FindClass(env, "java/lang/String"); args = (*env)-
>NewObjectArray(env, 1, stringClass, jstr); if (args == NULL) { goto detach;

}

}

```

```

(*env)->CallStaticVoidMethod(env, cls, mid, args);

분리: if
    ((*env)->ExceptionOccurred(env)) { (*env)-
        >ExceptionDescribe(env);
    }
    (*jvm)->DetachCurrentThread(jvm);
}

기본() {
    JNIEnv *env; 정
    수 i; 진트레스;

#endif JNI_VERSION_1_2
    JavaVMInitArgs vm_args;
    JavaVMOption 옵션[1];
    options[0].optionString = "-"
        Djava.class.path=" USER_CLASSPATH;
    vm_args.version = 0x00010002; vm_args.options = 옵션;
    vm_args.nOptions = 1; vm_args.ignoreUnrecognized =
    TRUE; /* Java VM 생성 */ res = JNI_CreateJavaVM(&jvm,
    (void**)&env, &vm_args); #else JDK1_1InitArgs vm_args;
    문자 클래스 경로[1024]; vm_args.version = 0x00010001;
    JNI_GetDefaultJavaVMInitArgs(&vm_args); /* 기본 시스템 클래스 경로에
    USER_CLASSPATH 추가 */ sprintf(classpath, "%s%c%s", vm_args.classpath,
    PATH_SEPARATOR, USER_CLASSPATH); vm_args.classpath = 클래스 경로; /*
    Java VM 생성 */ res = JNI_CreateJavaVM(&jvm, &env, &vm_args);

#endif /* JNI_VERSION_1_2 */

경우 (res < 0) {
    fprintf(stderr, "Java VM을 생성할 수 없습니다\n"); 출구(1);

}
for (i = 0; i < 5; i++)
    /* 모든 스레드에 스레드 번호를 전달합니다. */ _beginthread(thread_fun, 0,
    (void *)i);
수면(1000); /* 스레드 시작 대기 */ (*jvm)->DestroyJavaVM(jvm);

}

```

7.3 네이티브 스레드 연결

호출 인터페이스

attach.c 프로그램은 invoke.c의 변형입니다. 기본 스레드에서 Prog.main을 호출하는 대신 네이티브 코드는 5개의 스레드를 시작합니다. 스레드를 생성하면 스레드가 시작될 때까지 기다린 다음 DestroyJavaVM을 호출합니다. 생성된 각 스레드는 자신을 JVM(Java Virtual Machine)에 연결하고 Prog.main 메서드를 호출한 다음 가상 머신이 종료되기 전에 최종적으로 가상 머신에서 분리됩니다.

DestroyJavaVM은 5개의 스레드가 모두 종료된 후 반환됩니다. 이 함수는 JDK 릴리스 1.1 및 Java 2 SDK 릴리스 1.2에서 완전히 구현되지 않았기 때문에 지금은 DestroyJavaVM의 반환 값을 무시합니다.

JNI_AttachCurrentThread는 세 번째 인수로 NULL을 사용합니다. Java 2 SDK 릴리스 1.2에는 JNI_ThreadAttachArgs 구조가 도입되었습니다. 연결하려는 스레드 그룹과 같은 추가 인수를 지정할 수 있습니다. JNI_ThreadAttachArgs 구조의 세부 사항은 섹션 13.2의 JNI_AttachCurrentThread 사양의 일부로 설명됩니다.

프로그램이 DetachCurrentThread 함수를 실행하면 현재 스레드에 속한 모든 로컬 참조가 해제됩니다.

프로그램을 실행하면 다음과 같은 출력이 생성됩니다.

```
스레드 1의 Hello World  
스레드 0의 Hello World  
스레드 4의 Hello World  
스레드 2의 Hello World  
스레드 3의 Hello World
```

출력의 정확한 순서는 스레드 스케줄링의 임의 요인에 따라 달라질 수 있습니다.

8 장

추가 JNI 기능

네이티브 메서드 작성에 사용되는 JNI 기능과

기본 애플리케이션에 Java 가상 머신 구현을 내장합니다. 이 장에서는 나머지 JNI 기능을 소개합니다.

8.1 JNI와 쓰레드

JVM(Java Virtual Machine)은 동일한 주소 공간에서 동시에 실행되는 다중 제어 스레드를 지원합니다. 이 동시성은 단일 스레드 환경에서 가질 수 없는 복잡성을 가져옵니다. 여러 스레드가 동일한 객체, 동일한 파일 디스크립터(즉, 동일한 공유 리소스)에 동시에 액세스할 수 있습니다.

이 섹션을 최대한 활용하려면 다중 스레드 프로그래밍의 개념을 잘 알고 있어야 합니다. 여러 스레드를 활용하는 Java 애플리케이션을 작성하는 방법과 공유 리소스의 액세스를 동기화하는 방법을 알아야 합니다. Java 프로그래밍 언어의 다중 스레드 프로그래밍에 대한 좋은 참고 자료는 Doug Lea(Addison-Wesley, 1997)의 Java™의 동시 프로그래밍, 설계 원칙 및 패턴입니다.

8.1.1 제약

멀티스레드 환경에서 실행할 네이티브 메서드를 작성할 때 염두에 두어야 할 특정 제약 조건이 있습니다. 이러한 제약 조건 내에서 이해하고 프로그래밍하면 지정된 기본 메서드를 동시에 실행하는 스레드 수에 관계없이 기본 메서드가 안전하게 실행됩니다. 예를 들어:

- JNIEnv 포인터는 연결된 스레드에서만 유효합니다. 이 포인터를 한 스레드에서 다른 스레드로 전달하거나 캐시하여 여러 스레드에서 사용하면 안 됩니다. JVM(Java Virtual Machine)은 동일한 스레드에서 연속적으로 호출할 때 동일한 JNIEnv 포인터를 기본 메서드에 전달하지만 다른 스레드에서 해당 기본 메서드를 호출할 때는 다른 JNIEnv 포인터를 전달합니다.

8.1.2 모니터 입력 및 종료

추가 JNI 기능

한 스레드의 JNIEnv 포인터를 캐싱하고 다른 스레드에서 포인터를 사용하는 일반적인 실수를 피하십시오. • 로컬 참조는 참조를 생성한 스레드에서만 유효합니다. 한 스레드에서 다른 스레드로 로컬 참조를 전달하면 안 됩니다. 여러 스레드가 동일한 참조를 사용할 가능성이 있을 때마다 항상 로컬 참조를 전역 참조로 변환해야 합니다.

8.1.2 모니터 입력 및 종료

모니터는 Java 플랫폼의 기본 동기화 메커니즘입니다. 각 개체는 모니터와 동적으로 연결될 수 있습니다. JNI를 사용하면 이러한 모니터를 사용하여 동기화할 수 있으므로 Java 프로그래밍 언어의 동기화된 블록에 해당하는 기능을 구현할 수 있습니다.

```
동기화됨(obj) {
    ...
    // 동기화된 블록
}
```

JVM(Java Virtual Machine)은 스레드가 블록의 명령문을 실행하기 전에 개체 obj 와 연결된 모니터를 획득하도록 보장합니다 . 이렇게 하면 주어진 시간에 동기화된 블록 내에서 모니터를 보유하고 실행하는 최대 하나의 스레드가 있을 수 있습니다. 다른 스레드가 모니터를 종료할 때까지 스레드가 차단됩니다.

기본 코드는 JNI 함수를 사용하여 JNI 참조에서 동등한 동기화를 수행할 수 있습니다. MonitorEnter 함수를 사용하여 모니터에 들어가고 MonitorExit 함수를 사용하여 모니터를 종료할 수 있습니다.

```
if ((*env)->MonitorEnter(env, obj) != JNI_OK) {
    ... /* 오류 처리 */
}
...
/* 동기화 블록 */
if ((*env)->MonitorExit(env, obj) != JNI_OK) { ... /* 오류 처리 */
};
```

위의 코드를 실행하면 스레드는 동기화된 블록 내부의 코드를 실행하기 전에 먼저 obj 와 연결된 모니터에 들어가야 합니다. Monitor Enter 작업은 jobject를 인수로 사용하고 다른 스레드가 jobject 와 연결된 모니터에 이미 진입한 경우 차단합니다. 현재 스레드가 모니터를 소유하지 않을 때 MonitorExit를 호출하면 오류가 발생하고 IllegalMonitorStateException이 발생합니다 . 위의 코드에는 일치하는 MonitorEnter 및 MonitorExit 호출 쌍이 포함되어 있지만 여전히 가능한 오류를 확인해야 합니다. 예를 들어 다음과 같은 경우 모니터 작업이 실패할 수 있습니다.

추가 JNI 기능

대기 모니터링 및 알림

8.1.3

스레드 구현은 모니터 작업을 수행하는 데 필요한 리소스를 할당할 수 없습니다.

MonitorEnter 및 MonitorExit는 jclass, jstring 및 jarray에서 작동합니다.
 jobject 참조의 특별한 종류인 유형.
 적절한 수의 MonitorEnter 호출을 일치시키는 것을 기억하십시오.
 특히 오류 및 예외를 처리하는 코드에서 MonitorExit 호출:

```
if ((*env)->MonitorEnter(env, obj) != JNI_OK) ...;
...
if ((*env)->ExceptionOccurred(env)) { ... /* 예외 처리
   */ /* 여기에서 MonitorExit를 호출하는 것을 기억하세요
   */
   if ((*env)->MonitorExit(env, obj) != JNI_OK) ...;

}
... /* 일반 실행 경로. if ((*env)->MonitorExit(env, obj) != JNI_OK) ...;
```

MonitorExit를 호출하지 못하면 교착 상태가 발생할 가능성이 높습니다. 위의 C 코드 세그먼트를 이 섹션의 시작 부분에 있는 코드 세그먼트와 비교하면 JNI보다 Java 프로그래밍 언어로 프로그래밍하는 것이 얼마나 쉬운지 알 수 있습니다. 따라서 Java 프로그래밍 언어에서 동기화 구조를 표현하는 것이 바람직합니다. 예를 들어 정적 네이티브 메서드가 정의 클래스와 연결된 모니터에 들어가야 하는 경우 네이티브 코드에서 JNI 수준 모니터 동기화를 수행하는 대신 정적 동기화 네이티브 메서드를 정의해야 합니다.

8.1.3 모니터 대기 및 알림

Java API에는 스레드 동기화에 유용한 몇 가지 다른 메서드가 포함되어 있습니다. Object.wait, Object.notify 및 Object.notifyAll입니다. 모니터 대기 및 알림 작업은 모니터 시작 및 종료 작업만큼 성능이 중요하지 않기 때문에 이러한 메서드에 직접 대응하는 JNI 함수는 제공되지 않습니다. 네이티브 코드는 대신 JNI 메서드 호출 메커니즘을 사용하여 Java API에서 해당 메서드를 호출할 수 있습니다.

8.1.4 임의 컨텍스트에서 JNIEnv 포인터 얻기

추가 JNI 기능

```

/* 미리 계산된 메서드 ID */ static jmethodID
MID_Object_wait; 정적 jmethodID
MID_Object_notify; 정적 jmethodID
MID_Object_notifyAll;

무효의
JNU_MonitorWait(JNIEnv *env, jobject 객체, jlong 시간 초과) {

    (*env)->CallVoidMethod(env, object, MID_Object_wait, timeout);

}

무효의
JNU_MonitorNotify(JNIEnv *env, jobject 객체) {

    (*env)->CallVoidMethod(env, object, MID_Object_notify);

}

무효의
JNU_MonitorNotifyAll(JNIEnv *env, jobject 객체) {

    (*env)->CallVoidMethod(env, object, MID_Object_notifyAll);

}

```

Object.wait, Object.notify 및 Object.notifyAll에 대한 메서드 ID가 다른 곳에서 계산되었으며 전역 변수에 캐시되어 있다고 가정합니다. Java 프로그래밍 언어와 마찬가지로 jobject 인수와 연결된 모니터를 보유하고 있을 때만 위의 모니터 관련 함수를 호출할 수 있습니다.

8.1.4 임의의 컨텍스트에서 JNIEnv 포인터 얻기

앞에서 JNIEnv 포인터는 연결된 스레드에서만 유효하다고 설명했습니다.

네이티브 메서드는 가상 머신에서 첫 번째 인수로 JNIEnv 포인터를 받기 때문에 일반적으로 네이티브 메서드에는 문제가 되지 않습니다. 그러나 경우에 따라 현재 스레드에 속하는 JNIEnv 인터페이스 포인터를 얻기 위해 가상 머신에서 직접 호출되지 않은 원시 코드 조각이 필요할 수 있습니다. 예를 들어 네이티브 코드 조각은 운영 체제에서 호출하는 "콜백" 함수에 속할 수 있으며 이 경우 JNIEnv 포인터를 인수로 사용할 수 없습니다.

다음을 호출하여 현재 스레드에 대한 JNIEnv 포인터를 얻을 수 있습니다.
호출 인터페이스의 AttachCurrentThread 함수:

추가 JNI 기능

스레드 모델 일치 8.1.5

```

자바 VM *jvm; /* 이미 설정 */

에프()
{
    JNIEnv *env;
    (*jvm)->AttachCurrentThread(jvm, (void **)&env, NULL); ... /* 환경 사용 */
}

```

현재 스레드가 이미 가상 머신에 연결된 경우 Attach CurrentThread는 현재 스레드에 속하는 JNIEnv 인터페이스 포인터를 반환합니다.

JavaVM 포인터를 얻는 방법에는 여러 가지가 있습니다. 가상 머신이 생성될 때 기록하거나, JNI_GetCreatedJavaVM을 사용하여 생성된 가상 머신을 쿼리하거나, 일반 네이티브 메서드 내에서 JNI 함수 GetJavaVM을 호출하거나, JNI_OnLoad 처리기를 정의합니다. JNIEnv 포인터 와 달리 JavaVM 포인터는 전역 변수에 캐시될 수 있도록 여러 스레드에서 유효한 상태로 유지됩니다.

Java 2 SDK 릴리스 1.2는 새로운 호출 인터페이스 함수인 GetEnv를 제공 하여 현재 스레드가 가상 머신에 연결되어 있는지 여부를 확인할 수 있고 연결되어 있는 경우 현재 스레드에 속하는 JNIEnv 포인터를 반환할 수 있습니다.

GetEnv 및 AttachCurrentThread는 현재 스레드가 이미 가상 머신에 연결된 경우 기능적으로 동일합니다.

8.1.5 스레드 모델 일치

여러 스레드에서 실행될 네이티브 코드가 전역 리소스에 액세스한다고 가정합니다.

네이티브 코드는 JNI 함수 MonitorEnter 및 MonitorExit를 사용해야 합니까, 아니면 호스트 환경에서 네이티브 스레드 동기화 프리미티브(예: Solaris의 mutex_lock)를 사용해야 합니까? 마찬가지로 네이티브 코드가 새 스레드를 생성해야 하는 경우 java.lang.Thread 객체를 생성하고 JNI를 통해 Thread.start 의 콜백을 수행하거나 호스트 환경에서 네이티브 스레드 생성 프리미티브를 사용해야 합니다(Solaris의 thr_create 와 같은)?

대답은 JVM(Java Virtual Machine) 구현이 기본 코드에서 사용하는 것과 일치하는 스레드 모델을 지원하는 경우 이러한 모든 접근 방식이 작동한다는 것입니다.

스레드 모델은 시스템 호출에서 예약, 컨텍스트 전환, 동기화 및 차단과 같은 필수 스레드 작업을 시스템이 구현하는 방법을 나타냅니다. 기본 스레드 모델에서 운영 체제는 모든 필수 스레드 작업을 관리합니다. 반면 사용자 스레드 모델에서는 애플리케이션 코드가 스레드 작업을 구현합니다. 예를 들어 Solaris에서 JDK 및 Java 2 SDK 릴리스와 함께 제공되는 "그린 스레드" 모델은 ANSI C 함수 setjmp 및 longjmp를 사용하여 컨텍스트 스위치를 구현합니다.

8.1.5 스레드 모델 일치

추가 JNI 기능

많은 최신 운영 체제(예: Solaris 및 Win32)는 기본 스레드 모델을 지원합니다. 불행하게도 일부 운영 체제는 여전히 기본 스레드 지원이 부족합니다. 대신 이러한 운영 체제에는 하나 이상의 사용자 스레드 패키지가 있을 수 있습니다.

엄격하게 Java 프로그래밍 언어로 애플리케이션을 작성하는 경우 가상 머신 구현의 기본 스레드 모델에 대해 걱정할 필요가 없습니다. Java 플랫폼은 필요한 스레드 프리미티브 세트를 지원하는 모든 호스트 환경으로 포팅될 수 있습니다. 대부분의 기본 및 사용자 스레드 패키지는 JVM(Java Virtual Machine)을 구현하는 데 필요한 스레드 프리미티브를 제공합니다.

반면에 JNI 프로그래머는 스레드 모델에 주의를 기울여야 합니다. 네이티브 코드를 사용하는 애플리케이션은 Java 가상 구현과 네이티브 코드의 스레딩 및 동기화 개념이 다를 경우 제대로 작동하지 않을 수 있습니다. 예를 들어, 자체 스레드 모델의 동기화 작업에서 네이티브 메서드가 차단될 수 있지만 다른 스레드 모델에서 실행되는 JVM(Java Virtual Machine)은 네이티브 메서드를 실행하는 스레드가 차단되었음을 인식하지 못할 수 있습니다. 다른 스레드가 예약되지 않기 때문에 애플리케이션이 교착 상태에 빠집니다.

원시 코드가 Java 가상 머신과 동일한 스레드 모델을 사용하는 경우 스레드 모델이 일치합니다. JVM(Java Virtual Machine) 구현이 기본 스레드 지원을 사용하는 경우 기본 코드는 호스트 환경에서 스레드 관련 프리미티브를 자유롭게 호출할 수 있습니다. JVM(Java Virtual Machine) 구현이 사용자 스레드 패키지를 기반으로 하는 경우 기본 코드는 동일한 사용자 스레드 패키지와 연결하거나 스레드 작업에 전혀 의존하지 않아야 합니다. 후자는 생각보다 달성하기 어려울 수 있습니다. 대부분의 C 라이브러리 호출(예: I/O 및 메모리 할당 함수)은 아래에서 스레드 동기화를 수행합니다. 네이티브 코드가 순수한 계산을 수행하고 라이브러리 호출을 수행하지 않는 한 스레드 프리미티브를 간접적으로 사용할 가능성이 높습니다.

대부분의 가상 머신 구현은 JNI 네이티브 코드에 대한 특정 스레드 모델만 지원합니다. 기본 스레드를 지원하는 구현이 가장 유연하므로 사용 가능한 경우 일반적으로 지정된 호스트 환경에서 기본 스레드가 선호됩니다. 특정 사용자 스레드 패키지에 의존하는 가상 머신 구현은 작동할 수 있는 기본 코드 유형에 따라 심각하게 제한될 수 있습니다.

일부 가상 머신 구현은 다양한 스레드 모델을 지원할 수 있습니다. 보다 유연한 유형의 가상 머신 구현을 통해 가상 머신의 내부 사용을 위한 사용자 지정 스레드 모델 구현을 제공할 수 있으므로 가상 머신 구현이 네이티브 코드와 함께 작동할 수 있습니다. 네이티브 코드가 필요할 가능성이 있는 프로젝트를 시작하기 전에 스레드 모델 제한에 대한 가상 머신 구현과 함께 제공되는 설명서를 참조해야 합니다.

추가 JNI 기능

기본 문자열에서 jstring 만들기

8.2.1

8.2 국제화된 코드 작성

여러 로케일에서 제대로 작동하는 코드를 작성하려면 특별한 주의를 기울여야 합니다. JNI는 프로그래머에게 Java 플랫폼의 국제화 기능에 대한 완전한 액세스를 제공합니다. 많은 로케일에서 파일 이름과 메시지에 ASCII가 아닌 문자가 포함될 수 있으므로 문자열 변환을 예로 사용합니다.

JVM(Java Virtual Machine)은 유니코드 형식의 문자열을 나타냅니다. 일부 기본 플랫폼(예: Windows NT)도 유니코드 지원을 제공하지만 대부분은 로캘별 인코딩으로 문자열을 나타냅니다.

UTF-8이 플랫폼의 기본 인코딩이 아닌 한 GetStringUTFChars 및 GetStringUTFRegion 함수를 사용하여 jstring 과 로케일 특정 문자열 사이를 변환 하지 마십시오 . UTF-8 문자열은 JNI 함수에 전달될 이름 및 설명자(예: GetMethodID에 대한 인수)를 나타낼 때 유용하지만 파일 이름과 같은 로케일 관련 문자열을 나타내는 데는 적합하지 않습니다.

8.2.1 네이티브 문자열에서 jstring 만들기

`String(byte[] bytes)` 생성자를 사용하여 네이티브 문자열을 jstring으로 변환합니다. 다음 유ти리티 함수는 로캘별 기본 C 문자열에서 jstring 을 만듭니다.

```
jstring JNU_NewStringNative(JNIEnv *env, const char *str) {
    jstring 결과; jbyteArray
    바이트 = 0; 정수 렌; if ((*env)->EnsureLocalCapacity(env, 2)
    < 0) { return NULL; /* 메모리 부족 오류 */ }

    }
    길이 = strlen(str); 바이트 =
    (*env)->NewByteArray(env, len); if (bytes != NULL)
    { (*env)->SetByteArrayRegion(env, bytes, 0, len, (jbyte
        *)str); result = (*env)->NewObject(env, Class_java_lang_String,
        MID_String_init, bytes);
    (*env)->DeleteLocalRef(env, 바이트); 반환 결과;

    } /* 그렇지 않으면 통과 */ return NULL;
}
```

8.2.2 jstring을 네이티브 문자열로 변환

추가 JNI 기능

이 함수는 바이트 배열을 만들고 네이티브 C 문자열을 바이트 배열로 복사한 다음 마지막으로 String (byte[] bytes) 생성자를 호출하여 결과 jstring 객체를 만듭니다 . Class_java_lang_String 은 java.lang.String 클래스에 대한 전역 참조이고 MID_String_init 는 문자열 생성자의 메서드 ID입니다. 이것은 유ти리티 기능이므로 문자를 저장하기 위해 임시로 생성된 바이트 배열에 대한 로컬 참조를 삭제해야 합니다.

JDK 릴리스 1.1과 함께 이 기능을 사용해야 하는 경우에는 EnsureLocalCapacity 에 대한 호출을 삭제하십시오.

8.2.2 jstring을 네이티브 문자열로 변환하기

String.getBytes 메서드를 사용하여 jstring을 적절한 기본 인코딩으로 변환합니다. 다음 유ти리티 함수는 jstring 을 로케일별 기본 C 문자열로 변환합니다.

```
char *JNU_GetStringNativeChars(JNIEnv *env, jstring jstr) {
    jbyteArray 바이트 = 0;
    jthrowable 예외; 문자 * 결과 =
    0; if ((*env)->EnsureLocalCapacity(env, 2) < 0) {
        0을 반환합니다. /* 메모리 부족 오류 */
    }
    바이트 = (*env)->CallObjectMethod(env, jstr,
                                         MID_String_getBytes);
    exc = (*env)->ExceptionOccurred(env); 경우 (!exc) {

        jint len = (*env)->GetArrayLength(env, 바이트); 결과 = (문자
        *)malloc(len + 1); 경우 (결과 == 0) {

            JNU_ThrowByName(env, "java/lang/OutOfMemoryError",
                           0);
            (*env)->DeleteLocalRef(env, 바이트); 0을 반환합니다.

        }
        (*env)->GetByteArrayRegion(env, 바이트, 0, len,
                                    (jbyte *)결과); 결과[길이]
        = 0; /* NULL 종료 */ } else { (*env)->DeleteLocalRef(env, exc);

    }
    (*env)->DeleteLocalRef(env, 바이트); 반환 결과;
}
```

이 함수는 `java.lang.String` 참조를 `String.getBytes` 메서드에 전달한 다음 바이트 배열의 요소를 새로 할당된 C 배열에 복사합니다. `MID_String_getBytes`는 `String.getBytes` 메서드의 미리 계산된 메서드 ID입니다. 이것은 유ти리티 함수이기 때문에 바이트 배열과 예외 객체에 대한 로컬 참조를 삭제해야 합니다. 예외 객체에 대한 JNI 참조를 삭제해도 보류 중인 예외가 자워지지 않는다는 점에 유의하십시오.

다시 한 번, JDK 릴리스 1.1에서 이 기능을 사용해야 하는 경우 `MakeLocalCapacity`에 대한 호출을 삭제하십시오.

8.3 네이티브 메서드 등록

애플리케이션이 네이티브 메서드를 실행하기 전에 네이티브 메서드 구현이 포함된 네이티브 라이브러리를 로드한 다음 네이티브 메서드 구현에 연결하는 2단계 프로세스를 거칩니다.

1. `System.loadLibrary`는 명명된 기본 라이브러리를 찾아 로드합니다. 예를 들어 `System.loadLibrary("foo")`로 인해 `foo.dll`이 Win32에 로드될 수 있습니다.
2. 가상 머신은 로드된 기본 라이브러리 중 하나에서 기본 메서드 구현을 찾습니다. 예를 들어, `Foo.g` 네이티브 메서드 호출은 `foo.dll`에 상주할 수 있는 네이티브 함수 `Java_Foo_g`를 찾아 연결해야 합니다.

이 섹션에서는 두 번째 단계를 수행하는 다른 방법을 소개합니다.

이미 로드된 네이티브 라이브러리에서 네이티브 메서드를 검색하기 위해 가상 머신에 의존하는 대신 JNI 프로그래머는 함수 포인터를 클래스 참조, 메서드 이름 및 메서드 설명자와 함께 등록하여 네이티브 메서드를 수동으로 연결할 수 있습니다.

```
JNINativeMethod nm;
nm.name = "g"; /* 서명 필드
에 할당된 메서드 설명자 */ nm.signature = "()V"; nm.fnPtr = g_impl; (*env)-
>RegisterNatives(env, cls, &nm, 1);
```

위의 코드는 네이티브 함수 `g_impl`을 `Foo.g` 네이티브 메서드의 구현으로 등록합니다.

```
void JNICALL g_impl(JNIEnv *env, jobject self);
```

기본 함수 `g_impl`은 함수 포인터만 포함되기 때문에 JNI 명명 규칙을 따를 필요가 없으며 내보낼 필요도 없습니다.

8.4

핸들러 로드 및 언로드

추가 JNI 기능

라이브러리에서(따라서 JNIEXPORT를 사용하여 함수를 선언할 필요가 없습니다). 그러나 네이티브 함수 g_impl은 여전히 JNICALL 호출 규칙을 따라야 합니다.

RegisterNatives 함수는 여러 가지 용도로 유용합니다.

- 때때로 가상 머신이 이러한 항목을 느리게 연결하도록 하는 것과는 반대로 많은 수의 네이티브 메서드 구현을 열심히 등록하는 것이 더 편리하고 효율적입니다.
- 메서드에서 RegisterNatives를 여러 번 호출할 수 있습니다.
런타임 시 업데이트되는 네이티브 메서드 구현입니다.
- RegisterNatives는 네이티브 애플리케이션이 가상 머신 구현을 내장하고 네이티브 애플리케이션에 정의된 네이티브 메서드 구현과 연결해야 할 때 특히 유용합니다. 가상 머신은 애플리케이션 자체가 아닌 기본 라이브러리에서만 검색하기 때문에 이 기본 메서드 구현을 자동으로 찾을 수 없습니다.

8.4 핸들러 로드 및 언로드

로드 및 언로드 핸들러를 사용하면 네이티브 라이브러리가 두 가지 함수를 내보낼 수 있습니다. 하나는 System.loadLibrary가 네이티브 라이브러리를 로드할 때 호출되고 다른 하나는 가상 머신이 네이티브 라이브러리를 언로드할 때 호출됩니다. 이 기능은 Java 2 SDK 릴리스 1.2에 추가되었습니다.

8.4.1 JNI_OnLoad 핸들러

System.loadLibrary가 네이티브 라이브러리를 로드 하면 가상 머신은 네이티브 라이브러리에서 내보낸 다음 항목을 검색합니다.

```
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *jvm, void *reserved);
```

JNI_Onload 구현에서 모든 JNI 함수를 호출할 수 있습니다. JNI_OnLoad 처리기의 일반적인 용도는 다음 예제와 같이 JavaVM 포인터, 클래스 참조 또는 메서드 및 필드 ID를 캐싱하는 것입니다.

```
JavaVM *cached_jvm; jclass  
스 클래스_C; jmethodID  
MID_C_g;
```

추가 JNI 기능

JNI_OnLoad 핸들러 8.4.1 _

```

JNIEXPORTEXPORT 진트 JNICALL
JNI_OnLoad(JavaVM *jvm, 무효 *예약됨) {

    JNIEnv *env; j클래스
    클래스; cached_jvm
    = jvm; /* JavaVM 포인터 캐시 */

    if ((*jvm)->GetEnv(jvm, (void **)&env, JNI_VERSION_1_2)) {
        JNI_ERR을 반환합니다. /* 지원하지 않는 JNI 버전 */
    }
    cls = (*env)->FindClass(env, "C"); if (cls == NULL) { JNI_ERR
    반환;

    }
    /* 약한 전역 참조를 사용하여 C 클래스를 언로드할 수 있도록 함 */
    Class_C = (*env)->NewWeakGlobalRef(env, cls); if (Class_C == NULL)
    { JNI_ERR 반환;

    }
    /* 메서드 ID 계산 및 캐시 */
    MID_C_g = (*env)->GetMethodID(env, cls, "g", "()V"); if (MID_C_g == NULL) { JNI_ERR 반
    환;

    }
    JNI_VERSION_1_2를 반환합니다.
}

```

JNI_OnLoad 함수는 먼저 전역 변수 cached_jvm에서 JavaVM 포인터를 캐시합니다. 그런 다음 GetEnv를 호출하여 JNIEnv 포인터를 얻습니다. 마지막으로 C 클래스를 로드하고 클래스 참조를 캐시하고 Cg에 대한 메서드 ID를 계산합니다.

JNI_OnLoad 함수는 오류 시 JNI_ERR (§ 12.4)을 반환하고 그렇지 않으면 기본 라이브러리에 필요한 JNIEnv 버전 JNI_VERSION_1_2를 반환합니다.

다음 섹션에서 왜 우리가 약한 전역에서 C 클래스를 캐시하는지 설명할 것입니다.
전역 참조 대신 참조.

캐시된 JavaVM 인터페이스 포인터가 주어지면 네이티브 코드가 현재 스레드(§ 8.1.4)에 대한 JNIEnv 인터페이스 포인터를 얻을 수 있도록 하는 유ти리티 함수를 구현하는 것은 간단합니다.

```

JNIEnv *JNU_GetEnv() {

    JNIEnv *env;
    (*cached_jvm)->GetEnv(cached_jvm, (무효 **)&env,
                           JNI_VERSION_1_2);
    반환 환경;
}

```

8.4.2 JNI_OnUnload 핸들러

추가 JNI 기능

8.4.2 JNI_OnUnload 핸들러

직관적으로 가상 머신은 JNI 네이티브 라이브러리를 언로드할 때 JNI_OnUnload 핸들러를 호출합니다. 그러나 이것은 충분히 정확하지 않습니다. 가상 머신은 언제 네이티브 라이브러리를 언로드할 수 있다고 결정합니까? JNI_OnUnload 핸들러를 실행하는 스레드는 무엇입니까?

네이티브 라이브러리 언로드 규칙은 다음과 같습니다.

- 가상 머신은 각 기본 라이브러리를 System.loadLibrary 호출을 실행하는 클래스 C 의 클래스 로더 L 과 연결합니다 .
 - 가상 머신은 JNI_OnUnload 핸들러를 호출하고 클래스 로더 L이 더 이상 라이브 객체가 아니라고 판단한 후 네이티브 라이브러리를 언로드합니다.
- 클래스 로더는 자신이 정의하는 모든 클래스를 참조하기 때문에 C 도 언로드할 수 있음을 의미합니다.
- JNI_OnUnload 처리기는 종료자에서 실행되며 java.lang.System.runFinalization 에 의해 동기적으로 호출되거나 가상 머신에 의해 비동기적으로 호출됩니다.

다음은 마지막 섹션에서 JNI_OnLoad 핸들러 가 할당한 리소스를 정리하는 JNI_OnUnload 핸들러의 정의입니다 .

```
JNIEXPORT 무효 JNICALL
JNI_OnUnload(JavaVM *jvm, 무효 *예약됨) {

    JNIEnv *env; if
    ((*jvm)->GetEnv(jvm, (void **)&env, JNI_VERSION_1_2)) {
        반품;
    }
    (*env)->DeleteWeakGlobalRef(env, Class_C); 반품;

}
```

JNI_OnUnload 함수는 JNI_OnLoad 처리기에서 생성된 C 클래스에 대한 약한 전역 참조를 삭제합니다 . 정의 클래스 C 를 언로드할 때 가상 머신이 C의 메서드 ID를 나타내는 데 필요한 리소스를 자동으로 회수하기 때문에 메서드 ID MID_C_g 를 삭제할 필요가 없습니다.

이제 전역 참조 대신 약한 전역 참조에 C 클래스를 캐시하는 이유를 설명할 준비가 되었습니다 . 전역 참조는 C를 활성 상태로 유지하고 C의 클래스 로더를 활성 상태로 유지합니다 . 네이티브 라이브러리가 C의 클래스 로더 L 과 연결되어 있는 경우 네이티브 라이브러리는 언로드되지 않고 JNI_OnUnload가 호출되지 않습니다.

JNI_OnUnload 처리기는 종료자에서 실행됩니다 . 반대로 JNI_OnLoad 핸들러는 System.loadLibrary 호출을 시작하는 스레드에서 실행됩니다 . JNI_OnUnload는 알 수 없는 스레드 컨텍스트에서 실행되므로 가능한 교착 상태를 방지하기 위해

추가 JNI 기능

반사 지원

8.5

JNI_OnUnload에서 복잡한 동기화 및 잠금 작업을 피해야 합니다. JNI_OnUnload 처리기는 일반적으로 기본 라이브러리에서 할당한 리소스 해제와 같은 간단한 작업을 수행합니다.

JNI_OnUnload 핸들러는 라이브러리를 로드한 클래스 로더와 해당 클래스 로더가 정의한 모든 클래스가 더 이상 활성 상태가 아닐 때 실행됩니다. JNI_OnUnload 핸들러는 어떤 식으로든 이러한 클래스를 사용해서는 안 됩니다. 위의 JNI_OnUnload 정의에서 Class_C가 여전히 유효한 클래스를 참조한다고 가정하는 작업을 수행하면 안 됩니다. 예제의 DeleteWeakGlobalRef 호출은 약한 전역 참조 자체에 대한 메모리를 해제하지만 참조된 클래스 C를 조작하지는 않습니다.

그래도.

요약하면 JNI_OnUnload 핸들러를 작성할 때 주의해야 합니다.

교착 상태를 유발할 수 있는 복잡한 잠금 작업을 피하십시오. JNI_OnUnload 핸들러가 호출될 때 클래스가 언로드되었음을 명심하십시오.

8.5 반사 지원

리플렉션은 일반적으로 런타임에 언어 수준 구성을 조작하는 것을 말합니다.

예를 들어 리플렉션을 사용하면 런타임에 임의의 클래스 개체의 이름과 클래스에 정의된 필드 및 메서드 집합을 검색할 수 있습니다. 리플렉션 지원은 java.lang.Object 및 java.lang.Class 클래스의 일부 메서드와 java.lang.reflect 패키지를 통해 Java 프로그래밍 언어 수준에서 제공됩니다. 항상 해당 Java API를 호출하여 반사 작업을 수행할 수 있지만 JNI는 네이티브 코드에서 빈번한 반사 작업을 보다 효율적이고 편리하게 수행할 수 있도록 다음과 같은 기능을 제공합니다.

- GetSuperclass는 주어진 클래스 참조의 슈퍼클래스를 반환합니다. • IsAssignableFrom

은 다른 클래스의 인스턴스가 필요할 때 한 클래스의 인스턴스를 사용할 수 있는지 확인합니다.

- GetObjectClass는 주어진 jobject 참조의 클래스를 반환합니다. • IsInstanceOf는

jobject 참조가 주어진 객체의 인스턴스인지 확인합니다.
수업.

- FromReflectedField 및 ToReflectedField는 네이티브 코드가

필드 ID와 java.lang.reflect.Field 객체 사이의 vert. Java 2 SDK 릴리스 1.2에 새로 추가되었습니다.

- FromReflectedMethod 및 ToReflectedMethod는 네이티브 코드가 메소드 ID,

java.lang.reflect.Method 객체 및 java.lang.reflect.Constructor 객체 간에 변환할 수 있도록 합니다. Java 2 SDK 릴리스 1.2에 새로 추가되었습니다.

8.6 C++에서 JNI 프로그래밍

JNI는 C++ 프로그래머를 위한 약간 더 간단한 인터페이스를 제공합니다. jni.h 파일에는 C++ 프로그래머가 다음과 같이 작성할 수 있도록 일련의 정의가 포함되어 있습니다.

```
jclass cls = env->FindClass("java/lang/String");
```

C 대신:

```
jclass cls = (*env)->FindClass(env, "java/lang/String");
```

env에 대한 추가 간접 참조 수준과 FindClass에 대한 env 인수는 프로그래머에게 숨겨져 있습니다. C++ 컴파일러는 C++ 멤버 함수 호출을 해당하는 C 함수에 인라인합니다. 결과 코드는 완전히 동일합니다.

C 또는 C++에서 JNI를 사용할 때 본질적인 성능 차이는 없습니다.

또한 jni.h 파일은 더미 C++ 클래스 집합을 정의하여 서로 다른 jobject 하위 유형 간의 하위 유형 지정 관계를 적용합니다.

```
// C++에서 정의된 JNI 참조 유형 class _jobject {};
// _jclass : 공개 _jobject {};
// 클래스 _jstring : 공개 _jobject {};

...
typedef _jobject* 작업 객체; typedef
_jclass* jclass; typedef _jstring*
jstring;
...
```

C++ 컴파일러는 예를 들어 jobject를 GetMethodID에 전달하는 경우 컴파일 시간에 감지할 수 있습니다.

```
// 오류: jobject를 jclass로 전달: jobject obj = env-
>NewObject(...); jmethodID mid = env-
>GetMethodID(obj, "foo", "()V");
```

GetMethodID는 jclass 참조를 기대 하므로 C++ 컴파일러는 오류 메시지를 표시합니다. JNI에 대한 C 유형 정의에서 jclass는 jobject와 동일합니다.

```
typedef jobject jclass;
```

따라서 C 컴파일러는 실수로 jclass 대신 jobject를 전달했음을 감지할 수 없습니다.

C++에서 추가된 유형 계층 구조는 때때로 추가 캐스팅을 필요로 합니다.
C에서는 문자열 배열에서 문자열을 가져오고 그 결과를 jstring에 할당할 수 있습니다.

```
jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);
```

그러나 C++에서는 명시적 변환을 삽입해야 합니다.

```
jstring jstr = (jstring)env->GetObjectArrayElement(arr, i);
```


9 장

기존 네이티브 활용 도서관

JNI의 애플리케이션 중 하나는 코드를 활용하는 네이티브 메서드를 작성하는 것입니다.

기존 네이티브 라이브러리에서. 이 장에서 다루는 일반적인 접근 방식은 기본 함수 집합을 래핑하는 클래스 라이브러리를 생성하는 것입니다 .

이 장에서는 먼저 래퍼 클래스를 작성하는 가장 간단한 방법인 일대일 매핑에 대해 설명합니다. 그런 다음 래퍼 클래스 작성 작업을 단순화하는 공유 스텝 기술을 소개합니다 .

일대일 매핑과 공유 스텝은 둘 다 네이티브 함수를 래핑하는 기술입니다. 이 장의 끝에서 피어 클래스를 사용하여 기본 데이터 구조를 래핑하는 방법에 대해서도 설명합니다 .

이 장에서 설명하는 접근 방식은 네이티브 메서드를 사용하여 네이티브 라이브러리를 직접 노출하므로 네이티브 라이브러리에 종속된 네이티브 메서드를 호출하는 애플리케이션을 만드는 단점이 있습니다. 이러한 응용 프로그램은 기본 라이브러리를 제공하는 운영 체제에서만 실행할 수 있습니다. 선호되는 접근 방식은 운영 체제 독립적인 네이티브 메서드를 선언하는 것입니다. 이러한 네이티브 메서드를 구현하는 네이티브 함수만 네이티브 라이브러리를 직접 사용하므로 해당 네이티브 함수로 포팅할 필요가 없습니다. 네이티브 메서드 선언을 포함하여 애플리케이션을 이식할 필요가 없습니다.

9.1 일대일 매핑

간단한 예부터 시작하겠습니다. 표준 C 라이브러리에서 atol 함수를 노출하는 래퍼 클래스를 작성한다고 가정합니다 .

긴 atol(const char *str);

atol 함수는 문자열을 구문 분석하고 문자열이 나타내는 10진수 값을 반환합니다 . Java API의 일부인 Integer.parseInt 메서드가 동등한 기능을 제공하기 때문에 실제로는 이러한 네이티브 메서드를 정의할 이유가 거의 없습니다 . 예를 들어 atol("100")을 평가하면 정수 값이 100이 됩니다 . 래퍼 클래스를 다음과 같이 정의합니다.

9.1

일대일 매핑

기존 네이티브 라이브러리 활용

```
public class C { public
    static native int atol(String str);
    ...
}
```

C++에서 JNI 프로그래밍을 설명하기 위해 이 장에서는 C++(§ 8.6)를 사용하여 네이티브 메서드를 구현합니다. C.atol 네이티브 메서드의 C++ 구현은 다음과 같습니다.

JNIEXPORT 진트 JNICALL

```
Java_C_atol(JNIEnv *env, jclass cls, jstring str) {

    const char *cstr = env->GetStringUTFChars(str, 0); if (cstr == NULL) { 0 반
    환; /* 메모리 부족 */

}

정수 결과 = atol(cstr); env-
>ReleaseStringUTFChars(str, cstr); 반환 결과;

}
```

구현은 매우 간단합니다. 우리는 GetStringUTFChars를 사용 하여 10진수는 ASCII 문자이므로 유니코드 문자열을 변환합니다.

이제 구조 포인터를 C 함수에 전달하는 것과 관련된 보다 복잡한 예를 살펴보겠습니다. Win32 API에서 CreateFile 함수를 노출하는 래퍼 클래스를 작성한다고 가정합니다.

```
typedef 무효 * 핸들; typedef 긴
DWORD; typedef 구조체 {...}
SECURITY_ATTRIBUTES;

HANDLE CreateFile( const
    char *fileName, // 파일 이름 DWORD desiredAccess, // 액세스
    (읽기-쓰기) 모드 DWORD shareMode, // 공유 SECURITY_ATTRIBUTES *attrs, // 보안
    속성 DWORD creationDistribution, // DWORD 속성 HANDLEAttributesset/분별자 할 짜영
    이 있는 파일

);
```

CreateFile 함수는 플랫폼 독립적인 Java 파일 API에서 사용할 수 없는 여러 Win32 관련 기능을 지원합니다. 예를 들어 파일 만들기 기능을 사용하여 특수 액세스 모드 및 파일 속성을 지정하고 Win32 명명된 파이프를 열고 직렬 포트 통신을 처리할 수 있습니다.

이 책에서는 CreateFile 함수에 대한 자세한 내용을 다루지 않습니다.
초점은 CreateFile이 Win32라는 래퍼 클래스에 정의된 네이티브 메서드에 매핑되는 방법에 있습니다.

```
public class Win32 { public
    static native int CreateFile( String fileName, // 파일
        이름 int desiredAccess, // 액세스(읽기-쓰기) 모드 jint shareMode, //
        공유 모드 int[] secAttrs, // 보안 속성 int creationDistribution, // int
        flagsAndAttributes를 생성하는 방법, // 파일 속성을 설정하는 templateFile); // 속성이 있는
        ...
    }

    char 포인터 유형에서 String으로의 매핑은 명백합니다. 기본 Win32 type long (DWORD) 을 Java
    프로그래밍 언어의 int에 매핑합니다. 불투명한 32비트 포인터 유형인 Win32 유형 HANDLE도 int에 매핑됩니다.

    필드가 메모리에 배치되는 방식의 잠재적 차이 때문에 C 구조를 Java 프로그래밍 언어의 클래스에 매핑하지 않습니다. 대신 배열을 사용하여 C 구조 SECURITY_ATTRIBUTES의 내용을 저장합니다. 호출자는 null을 secAttrs로 전달하여 기본 Win32 보안 특성을 지정할 수도 있습니다.
```

SECURITY_ATTRIBUTES 구조의 내용이나 이를 int 배열로 인코딩하는 방법에 대해서는 논의하지 않습니다.

위 네이티브 메서드의 C++ 구현은 다음과 같습니다.

```
JNIEXPORT jint JNICALL Java_Win32_CreateFile( JNIEnv *env,
    jclass cls, jstring fileName, // 파일 이름 jint
    desiredAccess, // 액세스(읽기-쓰기) 모드 jint
    shareMode, // 공유 모드 jintArray secAttrs, // 보안 속성
    creationDistribution, // jint flagsAndAttributes를 설정하는 방법, // jint templateFile) //
    attr이 있는 파일. 복사하다
```

```
{
    진트 결과 = 0; 진트
    *cSecAttrs = NULL; if (secAttrs)
    { cSecAttrs = env-
        >GetIntArrayElements(secAttrs, 0); if (cSecAttrs == NULL) { 0 반환; /* 메모
        리 부족 */
    }
}
```

```

char *cFileName = JNU_GetStringNativeChars(env, fileName); if (cFileName) { /* 실제 Win32 함수 호출 */ result = (jint)CreateFile(cFileName, desiredAccess, shareMode,
(Security_ATTRIBUTES *)cSecAttrs,
creationDistribution, flagsAndAttributes,
(HANDLE)템플릿파일);
free(cFileName);

} /* 실패, 메모리 부족 예외 발생 */ if (secAttrs) { env->ReleaseIntArrayElements(secAttrs,
cSecAttrs, 0);

}
반환 결과;
}

```

먼저 int 배열에 저장된 보안 속성을 jint 배열로 변환합니다. secAttrs 인수가 NULL 참조인 경우 보안 특성으로 NULL을 Win32 CreateFile 함수에 전달합니다. 다음으로 유틸리티 함수 JNU_GetStringNativeChars (§ 8.2.2)를 호출하여 로캘별 C 문자열로 표시된 파일 이름을 가져옵니다. 보안 특성과 파일 이름을 변환했으면 변환 결과와 나머지 인수를 Win32 CreateFile 함수에 전달합니다.

우리는 예외를 확인하고 가상 머신 리소스(예: cSecAttrs)를 해제하도록 주의를 기울입니다.

C.atol 및 Win32.CreateFile 예제는 래퍼 클래스 및 네이티브 메서드 작성에 대한 일반적인 접근 방식을 보여줍니다. 각 네이티브 함수(예: CreateFile)는 단일 네이티브 스텝 함수(예: Java_Win32_CreateFile)에 매핑되고 단일 네이티브 메서드 정의(예: Win32.CreateFile)에 차례로 매핑됩니다. 일대일 매핑에서 스텝 함수는 두 가지 용도로 사용됩니다.

1. 스텝은 네이티브 함수의 인수 전달 규칙을 JVM(Java Virtual Machine)에서 예상하는 대로 조정합니다. 가상 머신은 기본 메서드 구현이 지정된 명명 규칙을 따르고 두 개의 추가 인수(JNIEnv 포인터 및 "this" 포인터)를 수락할 것으로 예상합니다.
2. 스텝은 Java 프로그래밍 언어 유형과 원시 유형 사이를 변환합니다.
예를 들어 Java_Win32_CreateFile 함수는 jstring 파일 이름을 로캘별 C 문자열로 변환합니다.

9.2 공유 스텝

일대일 매핑 방식에서는 래핑하려는 각 네이티브 함수에 대해 하나의 스텝 함수를 작성해야 합니다. 이것은 많은 수의 기본 함수에 대한 래퍼 클래스를 작성하는 작업에 직면할 때 지루해집니다. 이 섹션에서는 공유 스텝의 개념을 소개하고 공유 스텝을 사용하여 래퍼 클래스 작성 작업을 단순화하는 방법을 보여줍니다.

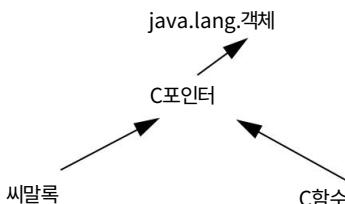
공유 스텝은 다른 네이티브 함수로 디스패치하는 네이티브 메서드입니다. 공유 스텝은 인수 유형을 호출자가 제공한 것에서 기본 함수가 허용하는 것으로 변환하는 역할을 합니다.

우리는 곧 공유 스텝 클래스 CFunction을 소개할 것이지만 먼저 C.atol 메서드의 구현을 단순화할 수 있는 방법을 보여드리겠습니다.

```
public class C { private
    static CFunction c_atol = new
        CFunction("msvcrt.dll", // 네이티브 라이브러리 이름 // C 함수 이름 // 호출 규칙
                  " atol",
                  "C");
    public static int atol(String str) { return c_atol.callInt(new
        Object[] {str});
    }
    ...
}
```

C.atol은 더 이상 기본 메서드가 아니므로 더 이상 스텝 함수가 필요하지 않습니다. 대신 C.atol은 CFunction 클래스를 사용하여 정의됩니다. CFunction 클래스는 내부적으로 공유 스텝을 구현합니다. 정적 변수 C.c_atol은 msvcrt.dll 라이브러리(Win32의 다중 스레드 C 라이브러리)의 C 함수 atol에 해당하는 CFunction 개체를 저장합니다. CFunction 생성자 호출은 또한 atol이 C 호출 규칙(§ 11.4)을 따르도록 지정 합니다. c_atol 필드가 초기화 되면 C.atol 메서드에 대한 호출은 공유 스텝인 c_atol.callInt를 통해 다시 전달하기만 하면 됩니다.

CFunction 클래스는 우리가 곧 구축하고 사용할 클래스 계층 구조에 속합니다.



CFunction 클래스의 인스턴스는 C 함수에 대한 포인터를 나타냅니다. CFunction은 임의의 C 포인터를 나타내는 CPointer의 하위 클래스입니다.

```
공개 클래스 CFunction 확장 CPointer { 공개 CFunction(String
    lib, // 네이티브 라이브러리 이름
    String fname, // C 함수 이름
    String conv) { // 호출 규칙
        ...
    }
    public native int callInt(Object[] args);
    ...
}
```

callInt 메소드는 java.lang.Object의 배열을 인수로 사용합니다. 배열에 있는 요소의 유형을 검사하고 변환하고(예: jstring에서 char *로) 기본 C 함수에 대한 인수로 전달합니다.

그런 다음 callInt 메서드는 기본 C 함수의 결과를 int로 반환 합니다. CFunction 클래스는 다른 반환 유형이 있는 C 함수를 처리하기 위해 callFloat 또는 callDouble과 같은 메서드를 정의할 수 있습니다.

CPointer 클래스는 다음과 같이 정의됩니다.

```
public abstract class CPointer { public native
    void copyIn( int bOff, // C 포인터로부터의 오
        프셋 int[] buf, // 소스 대상 off, // 스스로의 오프셋 int len); // 복사
        할 요소 수 public native void copyOut(...);

    ...
}
```

CPointer는 C 포인터에 대한 임의의 액세스를 지원하는 추상 클래스입니다. 예를 들어 copyIn 메서드는 int 배열의 여러 요소를 C 포인터가 가리키는 위치로 복사합니다. 이 방법은 주소 공간에서 임의의 메모리 위치를 손상시키는 데 쉽게 사용될 수 있으므로 주의해서 사용해야 합니다. CPointer.copyIn과 같은 기본 메서드는 C에서 직접 포인터를 조작하는 것만큼 안전하지 않습니다.

Cmalloc은 할당된 메모리 블록을 가리키는 CPointer의 하위 클래스입니다.
malloc을 사용하는 C 힙에서 :

```
공개 클래스 CMalloc 확장 CPointer {
    public CMalloc(int size)이 OutOfMemoryError { ... }를 던집니다. public native void
    free();
    ...
}
```

CMalloc 생성자는 C 힙에 지정된 크기의 메모리 블록을 할당합니다. CMalloc.free 메서드는 메모리 블록을 해제합니다.

CFunction 및 CMalloc 클래스를 사용하여 다음과 같이 Win32.CreateFile을 다시 구현할 수 있습니다.

```
공개 클래스 Win32 { 개인 정적
    CFunction c_CreateFile =
        new CFunction ("kernel32.dll", // 네이티브 라이브러리 이름 // 네이티브 함수 // 호출 규칙
                      "CreateFileA",
                      "JNI");

    public static int CreateFile( String
                                fileName, // 파일 이름 int desiredAccess, // 액세스(읽기-쓰
                                기) 모드 int shareMode, // 공유 모드 int[] 속성 attrs, // CreationDistribution, // int
                                flagsAndAttributes 생성 방법, // 파일 속성 attrs templateFile) // attr이 있는 파일. 복

    {

        CMalloc cSecAttrs = null; if (secAttrs !
        = null) {
            cSecAttrs = 새로운 CMalloc(secAttrs.length * 4); cSecAttrs.copyIn(0,
            secAttrs, 0, secAttrs.length);
        }
        try
        { return c_CreateFile.callInt(new Object[] { fileName, new
                                                Integer(desiredAccess), new
                                                Integer(shareMode), cSecAttrs, new
                                                Integer(creationDistribution), new
                                                Integer(flagsAndAttributes), new
                                                Integer(templateFile)});}

    } 마지막으로 { if
        (secAttrs != null) { cSecAttrs.free();

        }
    }
}
...
}
```

정적 변수에 CFunction 객체를 캐시합니다. Win32 API 생성 파일은 kernel32.dll에서 CreateFileA로 내보내집니다. 내보낸 또 다른 항목인 CreateFileW는 유니코드 문자열을 파일 이름 인수로 사용합니다. 이 함수는

9.3

일대일 매핑 대 공유 스텝

기준 네이티브 라이브러리 활용

표준 Win32 호출 규칙 (stdcall) 인 JNI 호출 규칙을 낮춥니다 .

Win32.CreateFile 구현은 먼저 보안 특성을 임시로 보유할 수 있을 만큼 충분히 큰 C 힙에 메모리 블록을 할당합니다 . 그런 다음 배열의 모든 인수를 패키징하고 공유 디스패처를 통해 기본 C 함수 CreateFileA를 호출합니다. 마지막으로 Win32.CreateFile 메서드는 보안 특성을 유지하는 데 사용되는 C 메모리 블록을 해제합니다. c_CreateFile.callInt 호출이 예외를 발생시키더라도 일시적으로 C 메모리가 해제되도록 하기 위해 finally 절에서 cSecAttrs.free를 호출합니다 .

9.3 일대일 매핑 대 공유 스텝

일대일 매핑 및 공유 스텝은 네이티브 라이브러리에 대한 래퍼 클래스를 빌드하는 두 가지 방법입니다. 각각의 장점이 있습니다.

공유 스텝의 주요 이점은 프로그래머가 네이티브 코드로 많은 수의 스텝 함수를 작성할 필요가 없다는 것입니다. CFunction 과 같은 공유 스텝 구현을 사용할 수 있게 되면 프로그래머는 한 줄의 기본 코드를 작성하지 않고도 래퍼 클래스를 빌드할 수 있습니다.

그러나 공유 스텝은 주의해서 사용해야 합니다. 공유 스텝을 통해 프로그래머는 기본적으로 Java 프로그래밍 언어로 C 코드를 작성합니다. 이는 Java 프로그래밍 언어의 유형 안전성을 무효화합니다. 공유 스텝을 잘못 사용하면 메모리 손상 및 애플리케이션 충돌이 발생할 수 있습니다.

일대일 매핑의 장점은 일반적으로 JVM(Java Virtual Machine)과 원시 코드 간에 전송되는 데이터 유형을 변환하는 데 더 효율적이라는 것입니다. 반면 공유 스텝은 기껏해야 미리 결정된 인수 유형 집합만 처리할 수 있으며 이러한 인수 유형에 대해서도 최적의 성능을 달성할 수 없습니다. CFunction.callInt 의 호출자는 항상 각 int 인수에 대해 Integer 객체를 생성해야 합니다 . 이는 공유 스텝 체계에 공간 및 시간 오버헤드를 모두 추가합니다.

실제로는 성능, 휴대성 및 단기 생산성의 균형을 맞춰야 합니다. 공유 스텝은 약간의 성능 저하를 허용할 수 있는 본질적으로 이식할 수 없는 네이티브 코드를 활용하는 데 적합할 수 있지만 일대일 매핑은 최고의 성능이 필요하거나 이식성이 중요한 경우에 사용해야 합니다.

9.4 공유 스텝의 구현

우리는 지금까지 CFunction, CPointer 및 CMalloc 클래스를 블랙 박스로 처리했습니다. 이 섹션에서는 기본 JNI 기능을 사용하여 구현하는 방법에 대해 설명합니다.

9.4.1 CPointer 클래스

CPointer 클래스는 CFunction 과 CMalloc 의 상위 클래스이기 때문에 먼저 살펴봅니다 . 추상 클래스 CPointer에는 기본 C 포인터를 저장하는 64비트 필드인 peer가 포함되어 있습니다.

```
공개 추상 클래스 CPointer {보호된 긴 피어; public
    native void copyIn(int bOff, int[] buf, int
    off, int len);

    공개 기본 무효 copyOut(...);
    ...
}
```

copyIn 과 같은 네이티브 메서드의 C++ 구현은 간단합니다.

```
JNIEXPORT 무효 JNICALL
Java_CPointer_copyIn__I_3III(JNIEnv *env, jobject self, jint boff, jintArray arr,
    jint 까짐, jint len)
{
    긴 피어 = env->GetLongField(self, FID_CPointer_peer); env->GetIntArrayRegion(arr,
    off, len, (jint *)peer + boff);
}
```

FID_CPointer_peer 는 CPointer.peer 의 미리 계산된 필드 ID입니다 . 네이티브 메서드 구현은 긴 이를 인코딩 체계(§ 11.3)를 사용하여 CPointer 클래스의 다른 배열 유형에 대해 오버로드된 copyIn 네이티브 메서드 구현과의 충돌을 해결합니다 .

9.4.2 CMalloc 클래스

CMalloc 클래스는 C 메모리 블록을 할당하고 해제하는 데 사용되는 두 가지 기본 메서드를 추가합니다 .

```
공개 클래스 CMalloc 확장 CPointer {
    비공개 정적 네이티브 long malloc(int 크기); 공개 CMalloc(정수 크기)이
    OutOfMemoryError {
        피어 = malloc(크기); if (peer
        == 0) { throw new
            OutOfMemoryError();
        }
    }
    공개 기본 무효 free();
    ...
}
```

9.4.3 C 함수 클래스

기존 네이티브 라이브러리 활용

CMalloc 생성자는 네이티브 메서드 CMalloc.malloc을 호출하고 CMalloc.malloc이 C 힙에서 새로 할당된 메모리 블록을 반환하지 못하는 경우 OutOfMemoryError를 발생시킵니다. 다음과 같이 CMalloc.malloc 및 CMalloc.free 메서드를 구현할 수 있습니다.

```
JNIEXPORT jlong JNICALL Java_CMalloc_malloc(JNIEnv *env, jclass cls, jint 크기) {
    return (jlong)malloc(크기);
}

JNIEXPORT 무효 JNICALL Java_CMalloc_free(JNIEnv *env, jobject self) {
    긴 피어 = env->GetLongField(self, FID_CPointer_peer); free((무효 *)피어);
}
```

9.4.3 CFunction 클래스

CFunction 클래스를 구현하려면 운영 체제 및 CPU별 어셈블리 코드에서 동적 연결 지원을 사용해야 합니다. 아래에 제시된 구현은 특히 Win32/Intel x86 환경을 대상으로 합니다. CFunction 클래스 구현의 원리를 이해하면 동일한 단계를 따라 다른 플랫폼에서 구현할 수 있습니다.

CFunction 클래스는 다음과 같이 정의됩니다.

```
공개 클래스 CFunction 확장 CPointer { 개인 정적 최종 int CONV_C
= 0; 개인 정적 최종 int CONV_JNI = 1; 개인 정수 변환; private
네이티브 long find(String lib, String fname);
```

```
공공 C함수(문자열 lib, // 네이티브 라이브러리 이름 String
fname, // C 함수 이름 String conv) { // 호출 규칙 if
(conv.equals("C")) { conv = CONV_C; } 그렇지 않으면
(conv.equals("JNI")) { conv = CONV_JNI; } else { throw new
IllegalArgumentException( "잘못된 호출 규칙"); }

}

피어 = 찾기(lib, fname);
}
```

```
public native int callInt(Object[] args);
...
}
```

CFunction 클래스는 C 함수의 호출 규칙을 저장하는 데 사용되는 전용 필드 conv를 선언합니다. CFunction.find 네이티브 메서드는 다음 과 같이 구현됩니다.

```
JNIEXPORT jlong JNICALL Java_CFunction_find(JNIEnv *env, jobject self, jstring lib, jstring fun)

{
    목표 *핸들; 목표 *기
    능; char *libname;
    char *편네임;

    if ((libname = JNU_GetStringNativeChars(env, lib))) {
        if ((funname = JNU_GetStringNativeChars(env, fun))) { if ((handle =
            LoadLibrary(libname))) { if (!func = GetProcAddress(handle, funname)))
            { JNU_ThrowByName(env, "java/lang/ UnsatisfiedLinkError",
                funname);

        } } 또 다른 {
            JNU_ThrowByName(env,
                "java/lang/UnsatisfiedLinkError", libname);

        }
        무료(편네임);
    }
    무료(libname);
}
return (jlong)func;
}
```

CFunction.find는 라이브러리 이름과 함수 이름을 로캘별 C 문자열로 변환한 다음 Win32 API 함수 LoadLibrary 및 GetProcAddress를 호출하여 명명된 네이티브 라이브러리에서 C 함수를 찾습니다.

다음과 같이 구현된 callInt 메서드는 기본 C 함수로 다시 전달하는 주요 작업을 수행합니다.

9.4.3 C 함수 클래스

기존 네이티브 라이브러리 활용

```

JNIEXPORT 진트 JNICALL
Java_CFunction_callInt(JNIEnv *env, jobject self, jobjectArray arr)

{ #define MAX_NARGS 32
    진노; int nargs,
    nwords; jboolean
    is_string[MAX_NARGS]; word_t 인수
    [MAX_NARGS];

    nargs = env->GetArrayLength(arr); if (nargs >
    MAX_NARGS) {
        JNU_ThrowByName(env,
                      "java/lang/IllegalArgumentException", "너무 많은 인수");

        0을 반환합니다.
    }

    // 인수 변환 (nwords = 0; nwords
    < nargs; nwords++) { is_string[nwords] = JNI_FALSE; jobject arg =
    env->GetObjectArrayElement(arr, nwords);

    if (arg == NULL)
        { args[nwords].p = NULL; } else if
    (env->IsInstanceOf(arg, Class_Integer)) { args[nwords].i = env->GetIntField(arg,
    FID_Integer_value); } else if (env->IsInstanceOf(arg, Class_Float))
        { args[nwords].f = env->GetFloatField(arg, FID_Float_value); } 그
    렇지 않으면 (env->IsInstanceOf(arg, Class_CPointer)) {

        args[nwords].p = (void *) env-
            >GetLongField(arg, FID_CPointer_peer); } 그렇지 않으면 (env-
            >IsInstanceOf(arg, Class_String)) {
        문자 * cstr =
            JNU_GetStringNativeChars(env, (jstring)arg); if ((args[nwords].p
            = cstr) == NULL) { goto cleanup; // 오류 발생

        }
        is_string[nwords] = JNI_TRUE; } 또 다른 {

        JNU_ThrowByName(env,
                      "java/lang/IllegalArgumentException", "인식할 수 없는 인
                      수 유형"); 고토 정리;

    }
    env->DeleteLocalRef(arg);
}

```

```
무효 * 기능 =
    (void *)env->GetLongField(self, FID_CPointer_peer); int conv = env
>GetIntField(self, FID_CFunction_conv);

// 이제 제어를 func로 넘깁니다. ires =
asm_dispatch(func, nwords, args, conv);

cleanup: //
    우리가 생성한 모든 기본 문자열을 해제합니다. for (int i = 0; i < nwords; i++)
    { if (is_string[i]) { free(args[i].p);
        }

    }
반성;
```

적절한 클래스 참조 및 필드 ID를 캐싱하기 위해 여려 전역 변수를 설정했다고 가정합니다. 예를 들어 전역 변수 FID_CPointer_peer는 CPointer.peer 의 필드 ID를 캐시 하고 전역 변수 Class_String은 java.lang.String 클래스 개체에 대한 전역 참조입니다. word_t 유형은 기계어를 나타내며 다음과 같이 정의됩니다.

```
typedef union { jint i;  
    jfloat f; 무효 *p; }  
    워드_티;
```

Java_CFunction_callInt 함수는 인수 배열을 반복합니다 . 각 요소의 유형을 확인합니다.

- 요소가 null 참조이면 C func에 대한 NULL 포인터로 전달됩니다.
합니다.
 - 요소가 java.lang.Integer 클래스의 인스턴스인 경우 정수
값을 가져와서 C 함수에 전달합니다.
 - 요소가 java.lang.Float 클래스의 인스턴스인 경우 부동 소수점
값을 가져와서 C 함수에 전달합니다.
 - 요소가 CPointer 클래스의 인스턴스인 경우 피어 포인터를 가져옵니다.
C 함수에 전달됩니다.
 - 인수가 java.lang.String의 인스턴스인 경우 로캘별 C 문자열로 변환되어 C 함수에 전달됩니다.
그렇지 않으면 IllegalArgumentException 이 발생합니다.

9.4.3 C 함수 클래스

기존 네이티브 라이브러리 활용

인수 변환 중에 발생할 수 있는 오류를 주의 깊게 확인하고 Java_CFunction_callInt 함수에서 반환하기 전에 C 문자열에 할당된 모든 임시 저장소를 해제합니다.

임시 버퍼 args에서 C 함수로 인수를 전송하는 코드는 C 스택을 직접 조작해야 합니다. 인라인 어셈블리로 작성되었습니다.

```
int asm_dispatch(void *func, // C 함수에 대한 포인터 int nwords, // 인수 배열의 단어 수
                 word_t *args, // 인수 데이터의 시작 int conv) // 호출 규칙 0: C //
```

```
1: JNI
{
    __asm
    {
        mov esi, args
        mov edx, nwords // 워
        // 주소 -> 바이트 주소 shl edx, 2 sub edx, 4 jc
        args_done

        // 마지막 인수를 먼저 푸시 args_loop: mov eax,
        // DWORD PTR [esi+edx] 푸시 eax sub edx, 4 jge
        // SHORT args_loop args_done: call func

        // 호출 규칙 확인 mov edx, conv 또는 edx, edx
        jnz jni_call

        // 인수 팝 mov edx, nwords
        // shl edx, 2 add esp, edx
        // jni_call: // 완료, eax에 값 반환
    }
}
```

어셈블리 루틴은 인수를 C 스택에 복사한 다음 패치를 C 함수 func에 다시 배포합니다. func가 반환된 후 asm_dispatch 루틴은

func의 호출 규칙을 확인합니다. func가 C 호출 규칙을 따르는 경우 asm_dispatch는 func에 전달된 인수를 팝합니다. func가 JNI 호출 규칙을 따르는 경우 asm_dispatch는 인수를 팝하지 않습니다. func는 반환하기 전에 인수를 팝합니다.

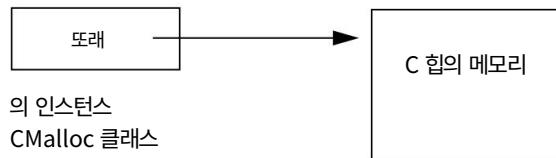
9.5 피어 클래스

일대일 매핑과 공유 스텝은 모두 네이티브 함수 래핑 문제를 해결합니다. 또한 공유 스텝 구현을 구성하는 과정에서 기본 데이터 구조를 래핑하는 문제에 직면했습니다. CPointer 클래스의 정의를 살펴보겠습니다.

```
공개 추상 클래스 CPointer {보호된 긴 피어; public native
    void copyIn(int bOff, int[] buf, int off, int len);
```

```
    공개 기본 무효 copyOut(...);
    ...
}
```

네이티브 데이터 구조(이 경우 C 주소 공간의 메모리 조각)를 참조하는 64비트 피어 필드를 포함합니다. CPointer의 하위 클래스는 피어 필드에 특정 의미를 할당합니다. 예를 들어 CMalloc 클래스는 피어 필드를 사용하여 C 힙의 메모리 청크를 가리킵니다.



CPointer 및 CMalloc과 같은 기본 데이터 구조에 직접 해당하는 클래스를 피어 클래스라고 합니다. 예를 들어 다음과 같은 다양한 기본 데이터 구조에 대한 피어 클래스를 구성할 수 있습니다.

- 파일 설명자 • 소켓 설
- 명자 • 창 또는 기타 그래픽
- 사용자 인터페이스 구성 요소

9.5.1 Java 플랫폼의 피어 클래스

기존 네이티브 라이브러리 활용

9.5.1 자바 플랫폼의 피어 클래스

현재 JDK 및 Java 2 SDK 릴리스(1.1 및 1.2)는 피어 클래스를 내부적으로 사용하여 `java.io`, `java.net` 및 `java.awt` 패키지를 구현합니다. 예를 들어 `java.io.FileDescriptor` 클래스의 인스턴스에는 기본 파일 설명자 를 재전송하는 전용 필드 `fd`가 포함되어 있습니다.

```
// java.io.FileDescriptor 클래스 구현 public final class FileDescriptor { private
int fd;

...
}
```

Java 플랫폼 API에서 지원하지 않는 파일 작업을 수행하려고 한다고 가정합니다. JNI를 사용하여 `java.io.FileDescriptor` 인스턴스의 기본 기본 파일 디스크립터를 찾고 싶을 수 있습니다. JNI를 사용하면 이름과 유형을 알고 있는 한 비공개 필드에 액세스할 수 있습니다. 그런 다음 해당 파일 설명자에서 직접 기본 파일 작업을 수행할 수 있다고 생각할 수 있습니다. 그러나 이 접근 방식에는 몇 가지 문제가 있습니다.

- 첫째, `fd`라는 전용 필드에 기본 파일 설명자를 저장하는 하나의 `java.io.FileDescriptor` 구현에 의존하고 있습니다. 그러나 Sun의 향후 구현 또는 `java.io.FileDescriptor` 클래스의 타사 구현이 여전히 기본 파일 설명자에 대해 동일한 개인 필드 이름 `fd`를 사용할 것이라는 보장은 없습니다. 피어 필드의 이름을 가정하는 네이티브 코드는 Java 플랫폼의 다른 구현에서 작동하지 않을 수 있습니다.
- 둘째, 기본 파일 설명자에서 직접 수행하는 작업은 피어 클래스의 내부 일관성을 방해할 수 있습니다. 예를 들어 `java.io.FileDescriptor` 인스턴스는 기본 기본 파일 설명자가 담았는지 여부를 나타내는 내부 상태를 유지합니다. 네이티브 코드를 사용하여 피어 클래스를 우회하고 기본 파일 설명자를 담는 경우 `java.io.FileDescriptor` 인스턴스에서 유지 관리되는 상태는 더 이상 네이티브 파일 설명자의 실제 상태와 일치하지 않습니다. 피어 클래스 구현은 일반적으로 기본 데이터 구조에 대한 배타적 액세스 권한이 있다고 가정합니다.

이러한 문제를 극복하는 유일한 방법은 기본 데이터 구조를 래핑하는 고유한 피어 클래스를 정의하는 것입니다. 위의 경우 필요한 작업 집합을 지원하는 고유한 파일 설명자 피어 클래스를 정의할 수 있습니다. 이 접근법

기존 네이티브 라이브러리 활용

기본 데이터 구조 해제

9.5.2

Java API 클래스를 구현하기 위해 자신의 피어 클래스를 사용할 수 없습니다. 예를 들어 자신의 파일 설명자 인스턴스를 `java.io.FileDescriptor` 인스턴스를 예상하는 메서드에 전달할 수 없습니다. 그러나 Java API에서 표준 인터페이스를 구현하는 고유한 피어 클래스를 쉽게 정의할 수 있습니다. 이는 클래스 대신 인터페이스를 기반으로 API를 설계해야 한다는 강력한 주장입니다.

9.5.2 네이티브 데이터 구조 해제

피어 클래스는 Java 프로그래밍 언어로 정의됩니다. 따라서 피어 클래스의 인스턴스는 자동으로 가비지 수집됩니다. 그러나 기본 기본 데이터 구조도 해제되는지 확인해야 합니다.

`CMalloc` 클래스에는 명시적으로 해제하기 위한 `free` 메서드가 포함되어 있음을 기억하십시오. `malloc'ed C` 메모리 :

```
공개 클래스 CMalloc 확장 CPointer { 공개 기본 무효 free();  
...  
}
```

`CMalloc` 클래스의 인스턴스에서 `free`를 호출하는 것을 기억해야 합니다. 그렇지 않으면 `CMalloc` 인스턴스가 가비지 수집될 수 있지만 해당 `malloc'ed C` 메모리는 절대 회수되지 않습니다.

일부 프로그래머는 `CMalloc`과 같은 피어 클래스에 파일라이저를 넣는 것을 좋아합니다.

```
공개 클래스 CMalloc 확장 CPointer {  
    공개 기본 동기화 무효 free(); 보호된 무효 finalize() { free();  
    }  
    ...  
}
```

가상 머신은 `CMalloc`의 인스턴스를 가비지 수집하기 전에 `finalize` 메서드를 호출합니다. `free`를 호출하는 것을 잊은 경우에도 `finalize` 메서드는 `malloc`된 C 메모리를 해제합니다.

여러 번 호출될 가능성을 고려하여 `CMalloc.free` 네이티브 메서드 구현을 약간 변경해야 합니다. 또한 스레드 경합 상태를 방지하려면 `CMalloc.free`를 동기화된 메서드로 만들어야 합니다.

9.5.2

기본 데이터 구조 해제

기존 네이티브 라이브러리 활용

```
JNIEXPORT 무효 JNICALL
Java_CMalloc_free(JNIEnv *env, jobject self) {

    긴 피어 = env->GetLongField(self, FID_CPointer_peer); if (피어 == 0) { 반환; /* 오류가
아님, 이전에 해제됨 */

    }
    free((무효 *)피어); 동료 = 0;
    env->SetLongField(self,
    FID_CPointer_peer, peer);
}
```

다음 두 문을 사용하여 피어 필드를 설정합니다 .

```
동료 = 0; env-
>SetLongField(self, FID_CPointer_peer, peer);
```

하나의 진술 대신:

```
env->SetLongField(self, FID_CPointer_peer, 0);
```

C++ 컴파일러는 리터럴 0을 64비트 정수가 아닌 32비트 정수로 간주하기 때문입니다. 일부 C++ 컴파일러에서는 64비트 정수 리터럴을 지정할 수 있지만 64비트 리터럴을 사용하면 이식성이 떨어집니다.

finalize 메서드를 정의하는 것은 적절한 안전 장치이지만 기본 데이터 구조를 해제하는 유일한 수단으로 종료자에 의존해서는 안 됩니다. 그 이유는 네이티브 데이터 구조가 피어 인스턴스보다 훨씬 더 많은 리소스를 소비할 수 있기 때문입니다. JVM(Java Virtual Machine)은 피어 클래스의 인스턴스를 가비지 수집 및 종료 하지 않아 네이티브 대용 항목을 해제할 수 있을 만큼 빠릅니다.

파이널라이저를 정의하면 성능에도 영향을 미칩니다. 일반적으로 종료자가 없는 클래스 인스턴스를 만들고 회수하는 것보다 종료자가 있는 클래스의 인스턴스를 만들고 회수하는 것이 더 느립니다.

피어 클래스의 기본 데이터 구조를 수동으로 해제할 수 있는지 항상 확인할 수 있는 경우 파이널라이저를 정의할 필요가 없습니다. 그러나 모든 실행 경로에서 기본 데이터 구조를 해제해야 합니다. 그렇지 않으면 리소스 누수가 발생했을 수 있습니다. 피어 인스턴스를 사용하는 과정에서 발생할 수 있는 예외에 특히 주의하십시오. finally 절에서 항상 기본 데이터 구조를 해제합니다 .

```
CMalloc ctr = new CMalloc(10); try { ... // cptr
사용 } finally { cptr.free();
```

```
}
```

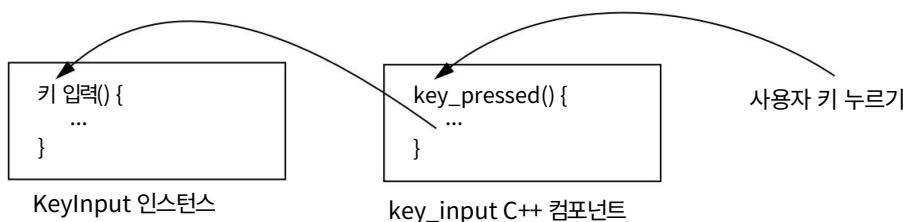
finally 절은 try 블록 내에서 예외가 발생하더라도 ctr이 해제되도록 합니다.

9.5.3 피어 인스턴스에 대한 백포인터

우리는 피어 클래스가 일반적으로 기본 네이티브 데이터 구조를 참조하는 전용 필드를 포함한다는 것을 보여주었습니다. 어떤 경우에는 네이티브 데이터 구조에서 피어 클래스의 인스턴스에 대한 참조도 포함하는 것이 바람직합니다. 예를 들어 네이티브 코드가 피어 클래스의 인스턴스 메서드에 대한 콜백을 시작해야 하는 경우에 이런 일이 발생합니다.

KeyInput이라는 가상의 사용자 인터페이스 구성 요소를 구축한다고 가정합니다. KeyInput의 기본 C++ 구성 요소인 key_input은 사용자가 키를 누를 때 운영 체제에서 key_pressed C++ 함수 호출로 이벤트를 수신합니다. key_input C++ 구성 요소는 KeyInput 인스턴스에서 keyPressed 메서드를 호출하여 운영 체제 이벤트를 KeyInput 인스턴스에 보고합니다.

아래 그림의 화살표는 키 누름 이벤트가 사용자 키 누름에 의해 시작되고 key_input C++ 구성 요소에서 키 입력 피어 인스턴스로 전파되는 방식을 나타냅니다.



KeyInput 피어 클래스는 다음과 같이 정의됩니다.

```

클래스 KeyInput { 프라이
    빗 롱 피어; 프라이빗 네이티브
    롱 create(); private 네이티브 void destroy(롱
        피어); 공개 KeyInput() { 피어 = 생성();}

    }

    공개 파괴() { 파괴(피어);

    }

    private void keyPressed(int key) { ... /* 키 이벤트 처리 */
    }
}
    
```

9.5.3 피어 인스턴스에 대한 백포인터

기존 네이티브 라이브러리 활용

네이티브 메서드 만들기 구현은 C++ 구조 key_input의 인스턴스를 할당합니다. C++ 구조는 C++ 클래스와 비슷하지만 유일한 차이점은 모든 멤버가 기본적으로 개인이 아닌 공용이라는 것입니다. 이 예제에서는 주로 Java 프로그래밍 언어의 클래스와의 혼동을 피하기 위해 C++ 클래스 대신 C++ 구조를 사용합니다.

```
// C++ 구조, KeyInput의 네이티브 대응 struct key_input { jobject back_ptr; // 피어
인스턴스에 대한 백 포인터 int key_pressed(int key); // 운영체제에서 호출

};

JNIEXPORT jlong JNICALL Java_KeyInput_create(JNIEnv *env, jobject self){

    key_input *cpp_obj = new key_input(); cpp_obj->back_ptr
    = env->NewGlobalRef(self); 반환 (jlong)cpp_obj;

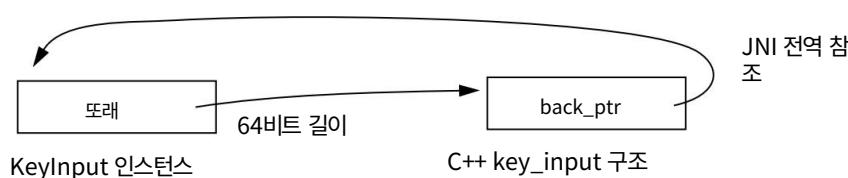
}

JNIEXPORT void JNICALL Java_KeyInput_destroy(JNIEnv *env, jobject 자체, jlong 피어){

    key_input *cpp_obj = (key_input*)피어; env-
    >DeleteGlobalRef(cpp_obj->back_ptr); cpp_obj 삭제; 반품;

}
```

네이티브 생성 메서드는 C++ 구조를 할당하고 back_ptr 필드를 KeyInput 피어 인스턴스에 대한 전역 참조로 초기화합니다. destroy native 메서드는 피어 인스턴스에 대한 전역 참조와 피어 인스턴스가 참조하는 C++ 구조를 삭제합니다. KeyInput 생성자는 피어 인스턴스와 네이티브 카운터 파트 간의 링크를 설정하기 위해 네이티브 생성 메서드를 호출합니다.



사용자가 키를 누르면 운영 체제는 C++ 멤버 함수 `key_input::key_pressed`를 호출합니다. 이 멤버 함수는 `KeyInput` 피어 인스턴스에서 `keyPressed` 메서드에 대한 콜백을 발행하여 이벤트에 응답합니다.

```
// 성공 시 0, 실패 시 -1 반환 int key_input::key_pressed(int
key) {

    jboolean has_exception;
    JNIEnv *env = JNU_GetEnv();
    JNU_CallMethodByName(env,
        &has_exception,
        java_peer,
        "keyPressed",
        "()V",
        key);

    if (has_exception) { env-
        >ExceptionClear(); -1 반환; } 그렇
        지 않으면 { 0을 반환합니다.

    }
}
```

`key_press` 멤버 함수는 콜백 후 모든 예외를 치우고 -1 반환 코드를 사용하여 운영 체제에 오류 조건을 반환합니다. `JNU_CallMethodByName` 및 `JNU_GetEnv` 유ти리티 함수의 정의는 각각 섹션 6.2.3 및 8.4.1을 참조하십시오.

이 섹션을 마치기 전에 마지막 문제에 대해 논의해 보겠습니다. 당신이 잠재적인 메모리 누수를 방지하기 위해 `KeyInput` 클래스에 `finalize` 메서드를 추가합니다.

```
클래스 키 입력 {
    ...
    공개 동기화된 파괴() { if(피어 != 0) { 파괴(피어); 동
        료 = 0;

    }
}
void finalize() 보호 { destroy();

    }
}
```

`destroy` 메서드는 피어 필드가 0인지 확인하고 오버로드된 `destroy` 네이티브 메서드를 호출한 후 피어 필드를 0으로 설정합니다. 경쟁 조건을 피하기 위해 동기화된 방법으로 정의됩니다.

9.5.3 피어 인스턴스에 대한 백포인터

기존 네이티브 라이브러리 활용

그러나 위의 코드는 예상대로 작동하지 않습니다. 명시적으로 `destroy`를 호출하지 않는 한 가상 머신은 `KeyInput` 인스턴스를 가비지 수집하지 않습니다. `KeyInput` 생성자는 `KeyInput` 인스턴스에 대한 JNI 전역 참조를 만듭니다. 전역 참조는 `KeyInput` 인스턴스가 가비지 수집되는 것을 방지합니다. 전역 참조 대신 약한 전역 참조를 사용하여 이 문제를 극복할 수 있습니다.

```
JNIEXPORT jlong JNICALL Java_KeyInput_create(JNIEnv *env, jobject self) {
    key_input *cpp_obj = new key_input(); cpp_obj->back_ptr = env->NewWeakGlobalRef(self); 반환 (jlong)cpp_obj;
}

JNIEXPORT 무효 JNICALL Java_KeyInput_destroy(JNIEnv *env, jobject 자체, jlong 피어) {
    key_input *cpp_obj = (key_input*)피어; env->DeleteWeakGlobalRef(cpp_obj->back_ptr); cpp_obj 삭제; 반품;
}
```

10 장

함정과 함정

에게 이전 장에서 다른 중요한 기술을 강조하지만, 이 장에서는 JNI 프로그래머가 일반적으로 저지르는 여러 가지 실수를 다룹니다. 여기에 설명된 각 실수는 실제 프로젝트에서 발생했습니다.

10.1 오류 검사

네이티브 메서드를 작성할 때 가장 흔한 실수는 오류 조건이 발생했는지 확인하는 것을 잊는 것입니다. Java 프로그래밍 언어와 달리 네이티브 언어는 표준 예외 메커니즘을 제공하지 않습니다. JNI는 특정 기본 예외 메커니즘(예: C++ 예외)에 의존하지 않습니다. 결과적으로 프로그래머는 예외를 발생시킬 수 있는 모든 JNI 함수 호출 후에 명시적 검사를 수행해야 합니다. 모든 JNI 함수가 예외를 발생시키는 것은 아니지만 대부분 가능합니다. 예외 검사는 번거롭지만 네이티브 메서드를 사용하는 애플리케이션이 견고하다는 것을 확인하는 데 필요합니다.

오류 검사의 지루함은 네이티브 코드를 JNI(§ 10.5)를 사용해야 하는 응용 프로그램의 잘 정의된 하위 집합으로 제한해야 할 필요성을 크게 강조합니다.

10.2 JNI 함수에 유효하지 않은 인수 전달

JNI 함수는 유효하지 않은 인수를 감지하거나 복구하려고 시도하지 않습니다. 참조를 예상하는 JNI 함수에 NULL 또는 (jobject)0xFFFFFFFF를 전달하면 결과 동작이 정의되지 않습니다. 실제로 이로 인해 잘못된 결과가 발생하거나 가상 머신이 충돌할 수 있습니다. Java 2 SDK 릴리스 1.2는 명령줄 옵션 -Xcheck:jni를 제공합니다. 이 옵션은 JNI 함수에 잘못된 인수를 전달하는 원시 코드의 전부는 아니지만 많은 경우를 감지하고 보고하도록 가상 머신에 지시합니다. 인수의 유효성을 확인하면 상당한 양의 오버헤드가 발생하므로 기본적으로 활성화되지 않습니다.

10.3 jclass 와 jobject 의 혼동

합정과 합정

인수의 유효성을 검사하지 않는 것은 C 및 C++ 라이브러리에서 일반적인 관행입니다. 라이브러리 를 사용하는 코드는 라이브러리 함수에 전달된 모든 인수가 유효한지 확인해야 합니다. 그러나 Java 프로그래밍 언어에 익숙하다면 JNI 프로그래밍의 안전성 부족이라는 특정 측면에 적응해야 할 수도 있습니다.

10.3 jclass 와 jobject 의 혼동

인스턴스 참조(jobject 유형의 값)와 클래스 참조(jclass 유형의 값) 간의 차이점은 JNI를 처음 사용 할 때 혼란스러울 수 있습니다.

인스턴스 참조는 `java.lang.Object` 또는 해당 하위 클래스 중 하나의 배열 및 인스턴스에 해당 합니다 . 클래스 참조는 클래스 유형을 나타내는 `java.lang.Class` 인스턴스에 해당합니다.

`jclass`를 사용하는 `GetFieldID` 와 같은 작업은 클래스에서 필드 설명자를 가져오기 때문에 클래스 작업입니다. 반대로 `jobject`를 받는 `GetIntField`는 인스턴스에서 필드 값을 가져오기 때문에 인스턴스 작업입니다. `jobject` 와 인스턴스 작업의 연결 및 `jclass` 와 클래스 작업의 연결은 모든 JNI 함수에 서 일관되므로 클래스 작업이 인스턴스 작업과 구별된다는 점을 기억하기 쉽습니다.

10.4 jboolean 인수 자르기

`jboolean`은 0 에서 255까지의 값을 저장할 수 있는 8비트 부호 없는 C 유형입니다. 값 0은 상수 `JNI_FALSE`에 해당하고 1에서 255까지의 값은 `JNI_TRUE`에 해당합니다. 그러나 하위 8비트가 0인 255보다 큰 32비트 또는 16비트 값은 문제가 됩니다.

유형이 `jboolean`인 인수 조건을 사용하는 함수 `print`를 정의했다고 가정합니다 .

```
무효 인쇄(jboolean 조건) {
```

```
    /* C 컴파일러는 조건을 하위 8비트로 자르는 코드를 생성합니다. */ if (조건)
        { printf("true\n"); } else { printf("거짓\n"); }
```

```
}
```

이전 정의에는 잘못된 것이 없습니다. 그러나 다음
순진해 보이는 인쇄 호출은 다소 예상치 못한 결과를 생성합니다.

```
정수 n = 256; /* 하위 8비트가 모두 0인 값 0x100 */ print(n);
```

0이 아닌 값(256)을 전달하여 true를 나타낼 것으로 예상하여 인쇄했습니다. 그러나 하위 8비트 이외의 모든 비트가 잘리기 때문에 인수는 0으로 평가됩니다. 프로그램은 예상과 달리 "거짓"을 인쇄합니다.

int 와 같은 정수 유형을 jboolean 으로 강제할 때 좋은 경험 법칙은 항상 정수 유형에 대한 조건을 평가하여 강제 변환 중에 부주의한 오류를 방지하는 것입니다. 인쇄 호출을 다음과 같이 다시 작성 할 수 있습니다.

```
n = 256; 인  
쇄(n ? JNI_TRUE : JNI_FALSE);
```

10.5 자바 애플리케이션과 네이티브 코드 사이의 경계

네이티브 코드로 지원되는 Java 애플리케이션을 설계할 때 자주 묻는 질문은 "네이티브 코드에 무엇을 얼마나 많이 포함해야 합니까?"입니다. 네이티브 코드와 Java 프로그래밍 언어로 작성된 나머지 애플리케이션 사이의 경계는 애플리케이션에 따라 다르지만 일반적으로 적용 가능한 몇 가지 원칙이 있습니다.

- 경계를 단순하게 유지하십시오. JVM(Java Virtual Machine)과 네이티브 코드 사이를 오가는 복잡한 제어 흐름은 디버깅 및 유지 관리가 어려울 수 있습니다. 이러한 제어 흐름은 고성능 가상 머신 구현에 의해 수행되는 최적화에도 방해가 됩니다. 예를 들어 가상 머신 구현에서는 C 및 C++에서 정의된 네이티브 메서드를 인라인하는 것보다 Java 프로그래밍 언어로 정의된 메서드를 인라인 하는 것이 훨씬 쉽습니다.
- 네이티브 코드 측의 코드를 최소한으로 유지합니다. 그렇게 해야 할 강력한 이유가 있습니다. 네이티브 코드는 이식 가능하거나 형식이 안전하지 않습니다. 네이티브 코드의 오류 검사는 지루합니다 (§ 10.1). 그러한 부분을 최소한으로 유지하는 것이 좋은 소프트웨어 엔지니어링입니다.
- 네이티브 코드를 격리된 상태로 유지합니다. 실제로 이것은 모든 기본 메서드가 동일한 패키지 또는 동일한 클래스에 있으며 나머지 응용 프로그램과 격리되어 있음을 의미할 수 있습니다. 기본 메서드를 포함하는 패키지 또는 클래스는 기본적으로 응용 프로그램의 "포팅 계층"이 됩니다.

JNI는 클래스 로딩, 객체 생성, 필드 액세스, 메서드 호출, 스레드 동기화 등과 같은 가상 머신 기능에 대한 액세스를 제공합니다.

앞으로, Java 가상 머신 기능과의 복잡한 상호 작용을 네이티브 코드로 표현하고 싶은 유혹이 들 수도 있지만 실제로는 Java 프로그래밍 언어로 동일한 작업을 수행하는 것이 더 간단합니다. 다음 예는 "네이티브 코드의 Java 프로그래밍"이 나쁜 습관인 이유를 보여줍니다. Java 프로그래밍 언어로 작성된 새 스레드를 생성하는 간단한 명령문을 고려하십시오.

```
new JobThread().start();
```

동일한 문을 네이티브 코드로 작성할 수도 있습니다.

```
/* 다음 변수가 미리 계산되고 캐시된다고 가정합니다. Class_JobThread: 클래스
 *          "JobThread"
 *          MID_Thread_init: 생성자의 메소드 ID MID_Thread_start:
 *          Thread.start()의 메소드 ID
 */
aThreadObject =
    (*env)->NewObject(env, Class_JobThread, MID_Thread_init);
if (aThreadObject == NULL) { ... /* 메모리
    부족 */
}
(*env)->CallVoidMethod(env, aThreadObject, MID_Thread_start); if ((*env)-
>ExceptionOccurred(env)) { ... /* 스레드가 시작되지 않음 */
}
```

네이티브 코드는 오류 검사에 필요한 코드 줄을 생략했음에도 불구하고 Java 프로그래밍 언어로 작성된 동급 코드보다 훨씬 더 복잡합니다.

JVM(Java Virtual Machine)을 조작하는 원시 코드의 복잡한 세그먼트를 작성하는 것보다 Java 프로그래밍 언어에서 보조 메소드를 정의하고 원시 코드가 보조 메소드에 대한 콜백을 발행하도록 하는 것이 종종 선호됩니다.

10.6 ID와 참조의 혼동

JNI는 개체를 참조로 노출합니다. 클래스, 문자열 및 배열은 특별한 유형의 참조입니다. JNI는 메서드와 필드를 ID로 노출합니다. ID는 참조가 아닙니다. 클래스 참조를 "클래스 ID" 또는 메소드 ID를 "메소드 참조"라고 부르지 마십시오.

참조는 네이티브 코드로 명시적으로 관리할 수 있는 가상 머신 리소스입니다. 예를 들어 JNI 함수 `DeleteLocalRef`를 사용 하면 네이티브 코드에서 로컬 참조를 삭제할 수 있습니다. 반대로 필드 및 메서드 ID는 가상 머신에서 관리하며 정의 클래스가 언로드될 때까지 유효한 상태로 유지됩니다. 네이티브 코드는 가상 머신이 정의 클래스를 언로드하기 전에 필드 또는 메서드 ID를 명시적으로 삭제할 수 없습니다.

네이티브 코드는 동일한 개체를 참조하는 여러 참조를 만들 수 있습니다. 예를 들어 전역 및 로컬 참조는 동일한 개체를 참조할 수 있습니다. 대조적으로 고유한 필드 또는 메서드 ID는 필드 또는 메서드의 동일한 정의에 대해 파생됩니다. 클래스 A가 메서드 f를 정의하고 클래스 B 가 A에서 f를 상속하는 경우 다음 코드의 두 Get MethodID 호출은 항상 동일한 결과를 반환합니다.

```
jmethodID MID_A_f = (*env)->GetMethodID(env, A, "f", "()V");
jmethodID MID_B_f = (*env)->GetMethodID(env, B, "f", "()V");
```

10.7 캐싱 필드 및 메서드 ID

네이티브 코드는 필드 또는 메서드의 이름과 형식 설명자를 문자열로 지정하여 가상 머신에서 필드 또는 메서드 ID를 가져옵니다(§ 4.1, § 4.2). 이름 및 유형 문자열을 사용하는 필드 및 메서드 조회가 느립니다. ID를 캐시하는 것은 종종 보상을 합니다. 필드 및 메서드 ID 캐시 실패는 네이티브 코드에서 일반적인 성능 문제입니다.

경우에 따라 ID 캐싱은 성능 향상 이상입니다. 캐시된 ID는 기본 코드에서 올바른 필드 또는 메서드에 액세스하도록 하기 위해 필요할 수 있습니다.

다음 예는 필드 ID 캐시 실패가 미묘한 버그로 이어질 수 있는 방법을 보여줍니다.

```
클래스 C { 개
    인 int i; 네이티브 무효
    f();
}
```

네이티브 메서드 f 가 C 의 인스턴스에서 필드 i 의 값을 가져와야 한다고 가정합니다. ID를 캐시하지 않는 간단한 구현은 다음 세 단계로 이를 달성합니다. 1) 개체의 클래스를 가져옵니다. 2) 클래스 참조에서 i 에 대한 필드 ID를 찾습니다. 3) 개체 참조 및 필드 ID를 기반으로 필드 값에 액세스합니다.

10.7 캐싱 필드 및 메서드 ID

함정과 함정

```
// 캐시된 필드 ID가 없습니다.
JNIEXPORT void JNICALL Java_C_f(JNIEnv *env, jobject this) { jclass cls =
    (*env)->GetObjectClass(env, this); ... /* 오류 검사 */ jfieldID fid =
    (*env)->GetFieldID(env, cls, "i", "I"); ... /* 오류 검사 */ ival = (*env)-
>GetIntField(env, this, fid); ... /* ival은 이제 this.i의 값을 가집니다 */
}

}
```

코드는 다른 클래스 D를 C 의 하위 클래스로 정의하고 D 에서 "i" 라는 개인 필드를 선언할 때까지 잘 작동합니다.

```
// ID 캐싱이 없을 때의 문제 class D extends C { private int i;

    디() { 예
        프(); // C에서 상속
    }
}
```

D의 생성자가 Cf를 호출 할 때 네이티브 메서드는 D 의 인스턴스 를 this 인수 로 받고 cls 는 D 클래스를 참조하며 fid는 Di를 나타냅니다 . 네이티브 메서드의 끝에서 ival 에는 Ci 대신 Di 값이 포함됩니다 . 네이티브 메서드 Cf를 구현할 때 예상한 것과 다름

해결책은 D 가 아닌 C 에 대한 클래스 참조가 있다고 확신할 때 필드 ID를 계산하고 캐시하는 것입니다. 이 캐시된 ID의 후속 액세스는 항상 올바른 필드 Ci를 참조합니다. 다음은 수정된 버전입니다.

```
// 정적 이나셜라이저에서 ID를 캐시하는 버전 class C { private int i; 네이티브 무
효 f(); 개인 정적 기본 무효 초기화 ID(); 정적 { initIDs(); // 초기화 네이티브 메서드
    호출
}
}
```

수정된 네이티브 코드는 다음과 같습니다.

```

정적 jfieldID FID_C_i;

JNIEXPORT 무효 JNICALL
Java_C_initIDs(JNIEnv *env, jclass cls) {
    /* 기본 메서드에 필요한 C의 모든 필드/메서드에 대한 ID를 가져옵니다. */
    FID_C_i = (*env)->GetFieldID(env, cls, "i", "I");

}

JNIEXPORT 무효 JNICALL
Java_C_f(JNIEnv *env, jobject this) { ival = (*env)-
    >GetIntField(env, this, FID_C_i); ... /* ival은 항상 Di가 아니라 Ci입니다. */

}

```

필드 ID는 C의 정적 초기화 프로그램에서 계산 및 캐시됩니다. 이렇게 하면 Ci의 필드 ID가 캐시되고 따라서 네이티브 메서드 구현 Java_C_f가 이 개체의 실제 클래스와 관계없이 Ci의 값을 읽게 됩니다.

일부 메서드 호출에도 캐싱이 필요할 수 있습니다. 위의 예를 약간 변경하여 클래스 C 와 D 가 각각 개인 메서드 g에 대한 고유한 정의를 가지도록 하면 f는 실수로 Dg를 호출하는 것을 방지하기 위해 Cg의 메서드 ID를 캐시해야 합니다. 올바른 가상 메서드 호출을 위해 캐싱이 필요하지 않습니다. 정의에 따라 가상 메서드는 메서드가 호출되는 인스턴스에 동적으로 바인딩됩니다. 따라서 JNU_CallMethodByName 유ти리티 함수(§ 6.2.3)를 안전하게 사용하여 가상 메서드를 호출할 수 있습니다. 그러나 이전 예제는 유사한 JNU_GetFieldByName 유ти리티 함수를 정의하지 않는 이유를 알려줍니다.

10.8 유니코드 문자열 종료

GetStringChars 또는 GetStringCritical에서 가져온 유니코드 문자열은 NULL로 끝나지 않습니다. GetStringLength를 호출하여 문자열에서 16비트 유니코드 문자의 수를 찾으십시오. Windows NT와 같은 일부 운영 체제에서는 두 개의 후행 0바이트 값이 유니코드 문자열을 종료할 것으로 예상합니다. 유니코드 문자열을 예상하는 Windows NT API에 GetStringChars의 결과를 전달할 수 없습니다. 문자열의 또 다른 복사본을 만들고 두 개의 후행 0바이트 값을 삽입해야 합니다.

10.9 액세스 제어 규칙 위반

JNI는 mod 사용을 통해 Java 프로그래밍 언어 수준에서 표현할 수 있는 클래스, 필드 및 메서드 액세스 제어 제한을 적용하지 않습니다.

10.10 국제화 무시

합정과 합정

`private` 및 `final` 과 같은 식별자 . Java 프로그래밍 언어 수준에서 그렇게 하면 `IllegalAccessException`이 발생하더라도 개체의 필드에 액세스하거나 수정하는 네이티브 코드를 작성할 수 있습니다 . JNI의 관용성은 기본 코드가 힙의 모든 메모리 위치에 액세스하고 수정할 수 있다는 점을 감안할 때 의식적인 설계 결정이었습니다.

소스 언어 수준 액세스 검사를 우회하는 네이티브 코드는 프로그램 실행에 바람직하지 않은 영향을 미칠 수 있습니다. 예를 들어 JIT(Just-In-Time) 컴파일러가 필드에 대한 액세스를 인라인한 후 기본 메서드가 최종 필드를 수정하는 경우 불일치가 발생할 수 있습니다 . 마찬가지로 네이티브 메서드는 `java.lang.String` 또는 `java.lang.Integer` 인스턴스의 필드와 같은 불변 객체를 수정해서는 안 됩니다 . 그렇게 하면 Java 플랫폼 구현에서 불변성이 손상될 수 있습니다.

10.10 국제화 무시

JVM(Java Virtual Machine)의 문자열은 유니코드 문자로 구성되는 반면 기본 문자열은 일반적으로 로케일별 인코딩입니다. `JNU_NewStringNative` (§ 8.2.1) 및 `JNU_GetStringNativeChars` (§ 8.2.2) 와 같은 유ти리티 함수를 사용하여 유니코드 `jstring` 과 기본 호스트 환경의 로케일별 기본 문자열 간에 변환합니다 . 일반적으로 국제화되는 메시지 문자열과 파일 이름에 특별한 주의를 기울이십시오. 네이티브 메서드가 파일 이름을 `jstring` 으로 가져오는 경우 파일 이름은 C 라이브러리 루틴으로 전달되기 전에 네이티브 문자열로 변환되어야 합니다.

다음 기본 메서드인 `MyFile.open`은 파일을 열고 파일 설명자를 결과로 반환합니다.

```

JNIEXPORT jint JNICALL Java_MyFile_open(JNIEnv *env, jobject self, jstring 이름,
                                         jint 모드)
{
    jint 결과; char
    *cname = JNU_GetStringNativeChars(env, name); if (cname == NULL)
    { 0 반환;

    }
    result = open(cname, 모드); 무료
    (cname); 반환 결과;
}

```

개방형 시스템 호출은 파일 이름이 로케일 특정 인코딩에 있을 것으로 예상하기 때문에 JNU_GetStringNativeChars 함수를 사용하여 jstring 인수를 변환합니다.

10.11 가상 머신 리소스 유지

기본 메서드에서 흔히 저지르는 실수는 가상 머신 리소스를 해제하는 것을 잊는 것입니다. 프로그래머는 오류가 있을 때만 실행되는 코드 경로에 특히 주의해야 합니다. 섹션 6.2.2의 예제를 약간 수정한 다음 코드 세그먼트는 ReleaseStringChars 호출을 누락합니다.

```
JNIEXPORT 무효 JNICALL
Java_pkg_Cls_f(JNIEnv *env, jclass cls, jstring jstr) {
    const jchar *cstr = (*env)->GetStringChars(env, jstr, NULL); if (cstr == NULL) { 반환;
}

...
if (...) { /* 예외 발생 */
/* ReleaseStringChars 호출을 놓침 */ return;
}

...
/* 정상 반환 */ (*env)->ReleaseStringChars(env, jstr, cstr);
}
```

ReleaseStringChars 함수 호출을 잊으면 jstring 개체가 무기한 고정되어 메모리 조각화가 발생하거나 C 복사본이 무기한 유지되어 메모리 누수가 발생할 수 있습니다.

GetStringChars가 문자열의 복사본을 만들었는지 여부에 관계없이 해당하는 ReleaseStringChars 호출이 있어야 합니다. 다음 코드는 가상 머신 리소스를 제대로 해제하지 못합니다.

10.12 과도한 로컬 참조 생성

함정과 함정

```
/* isCopy 인수는 여기서 잘못 사용되었습니다! */ JNIEXPORT 무효
JNIEXPORT Java_pckg_Cls_f(JNIEnv *env, jclass cls, jstring jstr) {

    jboolean isCopy; const
    jchar *cstr = (*env)->GetStringChars(env, jstr, &isCopy);

    if (cstr == NULL) { 반환;

    } ... /* cstr 사용 */ /* 이것은
잘못된 것입니다. 항상 ReleaseStringChars를 호출해야 합니다. */ if (isCopy) { (*env)-
>ReleaseStringChars(env, jstr, cstr);

    }
}
```

ReleaseStringChars에 대한 호출은 isCopy가 있는 경우에도 여전히 필요합니다.
가상 머신이 jstring 요소를 고정 해제하도록 JNI_FALSE 합니다.

10.12 과도한 로컬 참조 생성

과도한 로컬 참조 생성은 프로그램이 불필요하게 메모리를 유지하게 합니다. 불필요한 로컬 참조는 참조된
개체와 참조 자체 모두에 대해 메모리를 낭비합니다.

오래 실행되는 네이티브 메서드, 루프에서 생성된 로컬 참조 및 유ти리티 함수에 특히 주의하십시오.
Java 2 SDK 릴리스 1.2의 새로운 Push/PopLocalFrame 기능을 활용하여 로컬 참조를 보다 효과적으로
관리하십시오.
이 문제에 대한 자세한 설명은 섹션 5.2.1 및 섹션 5.2.2를 참조하십시오.

Java 2 SDK 1.2에서 -verbose:jni 옵션을 지정하여 과도한 로컬 참조 생성을 감지하고 보고하도록
가상 머신에 요청할 수 있습니다. 이 옵션을 사용하여 클래스 Foo를 실행한다고 가정합니다.

```
% java -verbose:jni Foo
```

출력에는 다음이 포함됩니다.

```
***경고: JNI 로컬 참조 생성이 용량을 초과했습니다.
(작성: 17, 제한: 16).
Baz.g(네이티브 방식) Bar.f(컴파일 방
식) Foo.main(컴파일 방식)
```

Baz.g에 대한 기본 메서드 구현이 관리에 실패할 가능성이 있습니다.
로컬 참조가 제대로 이루어집니다.

10.13 유효하지 않은 로컬 참조 사용

로컬 참조는 네이티브 메서드의 단일 호출 내에서만 유효합니다.
네이티브 메서드 호출에서 생성된 로컬 참조는 메서드를 구현하는 네이티브 함수가 반환된 후 자동으로 해제됩니다. 네이티브 코드는 전역 변수에 로컬 참조를 저장해서는 안 되며 나중에 네이티브 메서드를 호출할 때 이를 사용할 것으로 예상해야 합니다.

로컬 참조는 참조가 생성된 스레드 내에서만 유효합니다.
한 스레드에서 다른 스레드로 로컬 참조를 전달하면 안 됩니다. 스레드 간에 참조를 전달해야 하는 경우 전역 참조를 만듭니다.

10.14 여러 스레드에서 JNIEnv 사용

모든 네이티브 메서드의 첫 번째 인수로 전달되는 JNIEnv 포인터는 연결된 스레드에서만 사용할 수 있습니다. 한 스레드에서 얻은 JNIEnv 인터페이스 포인터를 캐시하고 다른 스레드에서 해당 포인터를 사용하는 것은 잘못된 것입니다.

섹션 8.1.4에서는 현재 스레드에 대한 JNIEnv 인터페이스 포인터를 얻을 수 있는 방법을 설명합니다.

10.15 일치하지 않는 스레드 모델

JNI는 호스트 네이티브 코드와 JVM(Java Virtual Machine) 구현이 동일한 스레드 모델(§ 8.1.5)을 공유하는 경우에만 작동합니다. 예를 들어, 프로그래머는 사용자 스레드 패키지를 사용하여 구현된 임베디드 Java 가상 머신에 기본 플랫폼 스레드를 연결할 수 없습니다.

Solaris에서 Sun은 Green 스레드로 알려진 사용자 스레드 패키지를 기반으로 하는 가상 머신 구현을 제공합니다. 네이티브 코드가 Solaris 네이티브 스레드 지원에 의존하는 경우 Green-thread 기반 JVM(Java Virtual Machine) 구현에서는 작동하지 않습니다. Solaris 네이티브 스레드와 함께 작동하도록 설계된 가상 머신 구현이 필요합니다. Solaris JDK 릴리스 1.1의 기본 스레드 지원에는 별도의 다운로드가 필요합니다. 기본 스레드 지원은 Solaris Java 2 SDK 릴리스 1.2와 함께 번들로 제공됩니다.

Win32에서 Sun의 가상 머신 구현은 다음을 통해 기본 스레드를 지원합니다.
기본이며 기본 Win32 응용 프로그램에 쉽게 포함할 수 있습니다.

파트 3: 사양

11

장

JNI 설계 개요

이것

장에서는 JNI 디자인에 대한 개요를 제공합니다. 필요한 경우 기본 기술 동기도 제공합니다. 디자인 개요는 JNIEnv 인터페이스 포인터, 로컬 및 전역 참조, 필드 및 메서드 ID와 같은 주요 JNI 개념에 대한 사양 역할을 합니다. 기술적 동기는 독자가 다양한 디자인 절충점을 이해하도록 돕는 것을 목표로 합니다. 경우에 따라 특정 기능을 구현하는 방법에 대해 논의할 것입니다. 이러한 논의의 목적은 실질적인 구현 전략을 제시하는 것이 아니라 미묘한 의미론적 문제를 명확히 하는 것입니다.

서로 다른 언어를 연결하는 프로그래밍 인터페이스의 개념은 새로운 것이 아닙니다. 예를 들어 C 프로그램은 일반적으로 FORTRAN 및 어셈블리와 같은 언어로 작성된 함수를 호출할 수 있습니다. 마찬가지로 LISP 및 Smalltalk와 같은 프로그래밍 언어의 구현은 다양한 외부 함수 인터페이스를 지원합니다.

JNI는 다른 언어에서 지원하는 상호 운용성 메커니즘에서 다루는 것과 유사한 문제를 해결합니다. 그러나 JNI와 많은 다른 언어에서 사용되는 상호 운용성 메커니즘 사이에는 상당한 차이가 있습니다. JNI는 Java 가상 머신의 특정 구현을 위해 설계되지 않았습니다. 오히려 JVM(Java Virtual Machine)의 모든 구현에서 지원할 수 있는 기본 인터페이스입니다. JNI 설계 목표를 설명하면서 이에 대해 더 자세히 설명하겠습니다.

11.1 설계 목표

JNI 디자인의 가장 중요한 목표는 주어진 호스트 환경에서 서로 다른 JVM(Java Virtual Machine) 구현 간에 바이너리 호환성을 제공하는 것입니다. 동일한 기본 라이브러리 바이너리는 재컴파일할 필요 없이 지정된 호스트 환경의 다른 가상 머신 구현에서 실행됩니다.

이 목표를 달성하기 위해 JNI 디자인은 JVM 구현의 내부 세부사항에 대해 어떤 가정도 할 수 없습니다. Java 가상 머신 구현 기술이 빠르게 발전하고 있기 때문에 우리는

11.2 네이티브 라이브러리 로드

JNI 설계 개요

향후 고급 구현 기술을 방해할 수 있는 제약 조건을 도입하지 마십시오.

JNI 설계의 두 번째 목표는 효율성입니다. 시간이 중요한 코드를 지원하기 위해 JNI는 가능한 한 적은 오버헤드를 부과합니다. 그러나 우리의 첫 번째 목표인 구현 독립성의 필요성은 때때로 우리가 그렇지 않은 경우보다 약간 덜 효율적인 설계를 선택하도록 요구한다는 것을 알게 될 것입니다. 우리는 효율성과 구현 독립성 사이에서 타협점을 찾습니다.

마지막으로 JNI는 기능적으로 완전해야 합니다. 네이티브 메서드와 애플리케이션이 유용한 작업을 수행할 수 있도록 충분한 Java 가상 머신 기능을 제공해야 합니다.

주어진 JVM 구현에서 지원하는 유일한 기본 프로그래밍 인터페이스가 되는 것이 JNI의 목표는 아닙니다. 표준 인터페이스는 기본 코드 라이브러리를 다른 JVM(Java Virtual Machine) 구현에 로드하려는 프로그래머에게 유용합니다. 그러나 어떤 경우에는 낮은 수준의 구현 관련 인터페이스가 더 높은 성능을 달성할 수 있습니다. 다른 경우에는 프로그래머가 더 높은 수준의 인터페이스를 사용하여 소프트웨어 구성 요소를 구축할 수 있습니다.

11.2 네이티브 라이브러리 로드

애플리케이션이 네이티브 메서드를 호출하려면 먼저 가상 머신이 네이티브 메서드 구현이 포함된 네이티브 라이브러리를 찾아 로드해야 합니다.

11.2.1 클래스 로더

네이티브 라이브러리는 클래스 로더에 의해 위치합니다. 클래스 로더는 예를 들어 클래스 파일 로드, 클래스 및 인터페이스 정의, 소프트웨어 구성 요소 간의 네임스페이스 분리 제공, 여러 클래스 및 인터페이스 간의 기호 참조 해결, 마지막으로 기본 라이브러리 찾기를 포함하여 JVM(Java Virtual Machine)에서 많은 용도로 사용됩니다. 클래스 로더에 대한 기본적인 이해가 있다고 가정하므로 JVM(Java Virtual Machine)에서 클래스를 로드하고 링크하는 방법에 대한 세부 정보는 다루지 않습니다. 클래스 로더에 대한 자세한 내용은 Sheng Liang과 Gilad Bracha가 OOPSLA(Object Oriented Programming Systems, Languages, and Applications)에 관한 ACM 컨퍼런스 절차에 게시한 Java 가상 머신의 동적 클래스 로딩 문서에서 클래스 로더에 대한 자세한 내용을 확인할 수 있습니다. 1998.

클래스 로더는 동일한 가상 머신의 인스턴스 내에서 여러 구성 요소(예: 다른 웹 사이트에서 다운로드한 애플릿)를 실행하는 데 필요한 네임스페이스 분리를 제공합니다. 클래스 로더는 클래스 또는 인터페이스 이름을 JVM(Java Virtual Machine)에서 개체로 표시되는 실제 클래스 또는 인터페이스 유형에 매핑하여 별도의 네임스페이스를 유지합니다. 각 클래스 또는 인터페이스 유형은 정의하는 로더, 처음에 클래스 파일을 읽고

클래스 또는 인터페이스 객체를 정의합니다. 두 개의 클래스 또는 인터페이스 유형은 동일한 이름과 동일한 정의 로더를 갖는 경우에만 동일합니다. 예를 들어, 그림 11.1에서 클래스 로더 L1 과 L2 는 각각 C라는 이름의 클래스를 정의합니다. C 라는 이 두 클래스는 동일하지 않습니다. 실제로, 그들은 별개의 반환 유형을 가진 두 개의 서로 다른 f 메서드를 포함합니다.

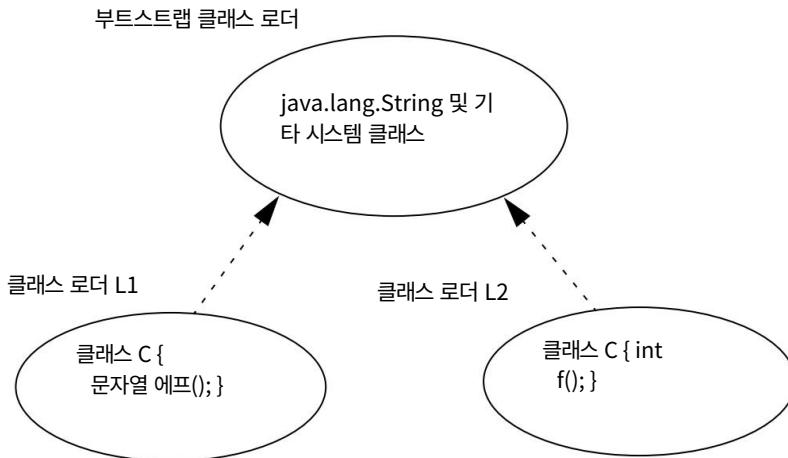


그림 11.1 서로 다른 클래스 로더에 의해 로드된 동일한 이름의 두 클래스

위 그림에서 점선은 클래스 로더 간의 위임 관계를 나타냅니다. 클래스 로더는 다른 클래스 로더에게 대신 클래스 또는 인터페이스를 로드하도록 요청할 수 있습니다. 예를 들어, L1 과 L2 모두 시스템 클래스 `java.lang.String`을 로드하기 위해 부트스트랩 클래스 로더에 위임합니다. 위임을 통해 모든 클래스 로더 간에 시스템 클래스를 공유할 수 있습니다. 예를 들어 애플리케이션과 시스템 코드가 `java.lang.String` 유형에 대해 서로 다른 개념을 갖고 있는 경우 유형 안전성이 위반될 수 있기 때문에 이는 필요합니다.

11.2.2 클래스 로더와 네이티브 라이브러리

이제 두 클래스 C 의 메서드 f가 기본 메서드라고 가정합니다. 가상 머신은 "C_f" 라는 이름을 사용하여 두 Cf 방법 모두에 대한 기본 구현을 찾습니다. 각 C 클래스가 올바른 기본 기능과 연결되도록 하기 위해 각 클래스 로더는 그림 11.2와 같이 고유한 기본 라이브러리 세트를 유지해야 합니다.

11.2.3 네이티브 라이브러리 찾기

JNI 설계 개요

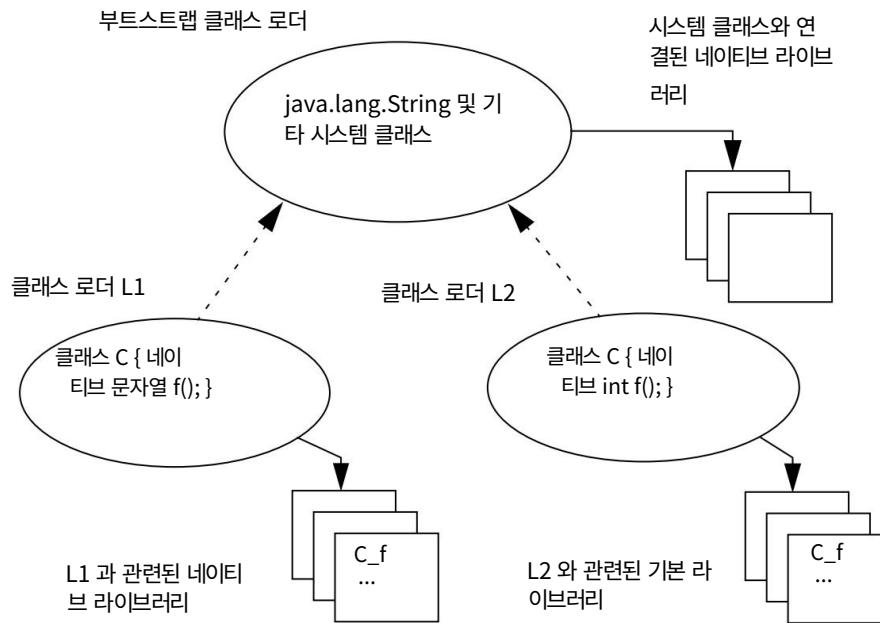


그림 11.2 네이티브 라이브러리를 클래스 로더와 연결

각 클래스 로더는 기본 라이브러리 세트를 유지하기 때문에 프로그래머는 클래스가 동일한 정의로더를 가지고 있는 한 여러 클래스에 필요한 모든 기본 메서드를 저장하기 위해 단일 라이브러리를 사용할 수 있습니다.

기본 라이브러리는 다음과 같은 경우 가상 머신에 의해 자동으로 언로드됩니다.
해당 클래스 로더는 가비지 수집됩니다(§ 11.2.5).

11.2.3 네이티브 라이브러리 찾기

네이티브 라이브러리는 `System.loadLibrary` 메서드에 의해 로드됩니다. 다음 예제에서 `Cls` 클래스의 정적 초기화 프로그램은 네이티브 메서드 `f`가 정의된 플랫폼별 네이티브 라이브러리를 로드합니다.

```

파키지 패키지; 클래
스 Cls { 네이티브 더
    블 f(int i, String s); 정적
    { System.loadLibrary("mypkg");
}
}
  
```

`System.loadLibrary`에 대한 인수는 프로그래머가 선택한 라이브러리 이름입니다. 소프트웨어 개발자는 이를 충돌 가능성을 최소화하는 네이티브 라이브러리 이름을 선택할 책임이 있습니다. 가상 머신은 라이브러리 이름을 기본 라이브러리 이름으로 변환하기 위해 표준이지만 호스트 환경에 특정한 규칙을 따릅니다. 예를 들어 Solaris 운영 체제는 이를 mypkg를 `libmypkg.so`로 변환하는 반면 Win32 운영 체제는 동일한 mypkg 이름을 `mypkg.dll`로 변환합니다.

JVM(Java Virtual Machine)이 시작되면 애플리케이션 클래스의 기본 라이브러리를 찾는 데 사용할 디렉토리 목록을 구성합니다. 목록의 내용은 호스트 환경과 가상 머신 구현에 따라 다릅니다. 예를 들어 Win32 JDK 또는 Java 2 SDK 릴리스에서 디렉토리 목록은 Windows 시스템 디렉토리, 현재 작업 디렉토리 및 PATH 환경 변수의 항목으로 구성 됩니다 . Solaris JDK 또는 Java 2 SDK 릴리스에서 디렉토리 목록은 `LD_LIBRARY_PATH` 환경 변수의 항목으로 구성됩니다.

`System.loadLibrary`는 명명된 네이티브 라이브러리를 로드하지 못하면 `UnsatisfiedLinkError`를 발생시킵니다 . `System.loadLibrary`에 대한 이전 호출이 이미 동일한 기본 라이브러리를 로드한 경우 `System.loadLibrary`는 자동으로 완료됩니다. 기본 운영 체제가 동적 연결을 지원하지 않는 경우 모든 기본 메서드는 가상 머신과 미리 연결되어야 합니다. 이 경우 가상 머신은 실제로 라이브러리를 로드하지 않고 `System.loadLibrary` 호출을 완료합니다.

가상 머신은 각 클래스 로더에 대해 로드된 기본 라이브러리 목록을 내부적으로 유지합니다. 새로 로드된 기본 라이브러리와 연관되어야 하는 클래스 로더를 판별하려면 다음 세 단계를 따르십시오.

1. `System.loadLibrary` 의 즉각적인 호출자를 결정합니다 . 2. 호출자
- 를 정의하는 클래스를 식별합니다. 3. 호출자 클래스의 정의 로더를 얻습니다.

다음 예제에서 네이티브 라이브러리 `foo`는 C의 정의 로더 와 연결됩니다 .

```
클래스 C { 정
    적
        { System.loadLibrary("foo");
        }
    }
```

Java 2 SDK 릴리스 1.2 에는 프로그래머가 지정된 클래스 로더에 특정한 사용자 지정 라이브러리 로드 정책을 지정할 수 있는 새로운 `ClassLoader.findLibrary` 메서드가 도입되었습니다.

`ClassLoader.findLibrary` 메서드는 플랫폼 독립적인 라이브러리 이름(예: `mypkg`)을 인수로 사용 하며 다음 을 수행합니다.

11.2.4 A 유형 안전 제한

JNI 설계 개요

- null을 반환하여 가상 머신이 기본 라이브러리를 따르도록 지시합니다.
검색 경로,
- 또는 라이브러리 파일의 호스트 환경 종속 절대 경로를 반환합니다(예:
"c:\\mylibs\\mypkg.dll"로).

ClassLoader.findLibrary는 일반적으로 Java 2 SDK 릴리스 1.2인 System.mapLibraryName에 추가된 다른 메서드와 함께 사용됩니다. System.mapLibraryName은 플랫폼 독립적 라이브러리 이름(예: mypkg)을 플랫폼 종속 라이브러리 파일 이름(예: mypkg.dll)에 매핑합니다.

java.library.path 속성을 설정하여 Java 2 SDK 릴리스 1.2에서 기본 라이브러리 검색 경로를 재정의할 수 있습니다. 예를 들어, 다음 명령줄은 c:\\mylibs 디렉토리에서 기본 라이브러리를 로드해야 하는 Foo 프로그램을 시작합니다.

```
java -Djava.library.path=c:\\mylibs Foo
```

11.2.4 A 유형 안전 제한

가상 머신은 지정된 JNI 기본 라이브러리가 둘 이상의 클래스 로더에 의해 로드되는 것을 허용하지 않습니다. 여러 클래스 로더에서 동일한 기본 라이브러리를 로드하려고 하면 UnsatisfiedLinkError가 발생합니다. 이 제한의 목적은 클래스 로더를 기반으로 하는 네임스페이스 분리가 네이티브 라이브러리에서 유지되도록 하는 것입니다. 이 제한이 없으면 네이티브 메서드를 통해 다른 클래스 로더의 클래스와 인터페이스를 잘못 혼합하기가 훨씬 쉬워집니다. 전역 참조에서 자체 정의 클래스 Foo를 캐시하는 기본 메서드 Foo.f를 고려하십시오.

```
JNIEXPORT 무효 JNICALL
Java_Foo_f(JNIEnv *env, jobject self) {
    정적 jclass cachedFooClass; /* 캐시된 클래스 Foo */ if (cachedFooClass == NULL)
    { jclass fooClass = (*env)->FindClass(env, "Foo"); if (fooClass == NULL) { 반환; /
     * 오류 */
    }

    }
    cachedFooClass = (*env)->NewGlobalRef(env, fooClass); if (cachedFooClass
    == NULL) { 반환; /* 오류 */
    }

    }
    assert((*env)->IsInstanceOf(env, self, cachedFooClass)); ... /* cachedFooClass 사용
    */
}
```

`Foo.f`는 인스턴스 메서드이고 `self`는 `Foo`의 인스턴스를 참조하기 때문에 어설션이 성공할 것으로 예상합니다. 그러나 두 개의 다른 `Foo` 클래스가 클래스 로더 L1 및 L2에 의해 로드되고 두 `Foo` 클래스가 모두 `Foo.f`의 이전 구현과 연결되어 있는 경우 어설션이 실패할 수 있습니다. `f` 메서드가 먼저 호출되는 `Foo` 클래스에 대해 `cachedFooClass` 전역 참조가 생성됩니다. 나중에 다른 `Foo` 클래스의 `f` 메서드를 호출 하면 어설션이 실패합니다.

JDK 릴리스 1.1은 클래스 로더 간에 기본 라이브러리 분리를 제대로 적용하지 않았습니다. 즉, 서로 다른 클래스 로더의 두 클래스가 동일한 기본 메서드와 연결될 수 있습니다. 이전 예제에서 볼 수 있듯이 JDK 릴리스 1.1의 접근 방식은 다음 두 가지 문제를 야기합니다.

- 클래스가 클래스에 의해 로드된 기본 라이브러리와 실수로 링크될 수 있음
다른 클래스 로더에서 같은 이름으로.
- 기본 메서드는 서로 다른 클래스 로더의 클래스를 쉽게 혼합할 수 있습니다. 이는 클래스 로더가 제공하는 네임스페이스 분리를 깨고 유형 안전성 문제를 일으킵니다.

11.2.5 네이티브 라이브러리 언로드

가상 머신은 네이티브 라이브러리와 연결된 클래스 로더를 가비지 수집한 후 네이티브 라이브러리를 언로드합니다. 클래스는 정의 로더를 참조하기 때문에 이는 가상 머신이 정적 초기화 프로그램이 `System.loadLibrary`라고 하는 클래스를 언로드하고 기본 라이브러리(§ 11.2.2)를 로드했음을 의미합니다.

11.3 네이티브 메서드 연결

가상 머신은 각 원시 메소드를 처음으로 호출하기 전에 연결을 시도합니다. 네이티브 메서드 `f`가 연결될 수 있는 가장 빠른 시간은 메서드 `g`의 첫 번째 호출이며, 여기서 `g`의 메서드 본문에서 `f`로의 참조가 있습니다. 가상 머신 구현은 네이티브 메서드를 너무 일찍 연결하려고 시도해서는 안 됩니다.

그렇게 하면 네이티브 메서드를 구현하는 네이티브 라이브러리가 로드되지 않았을 수 있으므로 예기치 않은 연결 오류가 발생할 수 있습니다.

기본 메서드 연결에는 다음 단계가 포함됩니다.

- 네이티브 메서드를 정의하는 클래스의 클래스 로더를 결정합니다. • 찾을 이 클래스 로더와 연관된 기본 라이브러리 세트 검색
- 네이티브 메소드를 구현하는 네이티브 함수.
- 네이티브 메서드에 대한 모든 향후 호출이 네이티브 함수로 직접 점프하도록 내부 데이터 구조를 설정합니다.

11.3 기본 메서드 연결

JNI 설계 개요

가상 머신은 다음 구성 요소를 연결하여 네이티브 메서드의 이름에서 네이티브 함수의 이름을 추론합니다.

- 접두사 "Java_" • 인코딩

된 정규화된 클래스 이름 • 밑줄 ("_") 구분 기호

- 인코딩된 메서드 이름

- 오버로드된 기본 메서드의 경우 두 개의 밑줄 ("__") 다음에
인코딩된 인수 설명자

가상 머신은 정의 로더와 연결된 모든 네이티브 라이브러리를 반복하여 적절한 이름을 가진 네이티브 함수를 검색합니다. 각 기본 라이브러리에 대해 가상 머신은 먼저 짧은 이름, 즉 인수 설명자가 없는 이름을 찾습니다. 그런 다음 인수 설명자가 있는 이름인 긴 이름을 찾습니다. 프로그래머는 네이티브 메서드가 다른 네이티브 메서드로 오버로드되는 경우에만 긴 이름을 사용해야 합니다. 그러나 네이티브 메서드가 네이티브가 아닌 메서드로 오버로드되는 경우에는 문제가 되지 않습니다. 후자는 기본 라이브러리에 상주하지 않습니다.

다음 예에서 네이티브 메서드 g는 연결할 필요가 없습니다.

다른 메서드 g는 기본 메서드가 아니기 때문에 긴 이름을 사용합니다.

```
클래스 CLS1 {
    int g(int i) { ... } // 일반 메서드 네이티브 int g(double d);
}
```

JNI는 모든 유니코드 문자가 유효한 C 함수 이름으로 변환되도록 간단한 이름 인코딩 체계를 채택합니다. 밑줄 ("_") 문자는 정규화된 클래스 이름의 구성 요소를 구분합니다. 이름이나 유형 설명자는 숫자로 시작하지 않으므로 아래 그림과 같이 이스케이프 시퀀스에 _0, ..., _9를 사용할 수 있습니다.

이스케이프 시퀀스	의미
_XXXX	유니코드 문자 XXXX
1	문자 ""
_2	문자 ";" 디스크립터에서
_삼	디스크립터의 문자 "["

JNI 설계 개요

JNIEnv 인터페이스 포인터 11.5.1 의 구성

인코딩된 네이티브 메서드 이름과 일치하는 네이티브 함수가 여러 네이티브 라이브러리에 있는 경우 가장 먼저 로드되는 네이티브 라이브러리의 함수가 네이티브 메서드와 연결됩니다. 네이티브 메서드 이름과 일치하는 함수가 없으면 `UnsatisfiedLinkError`가 발생합니다.

프로그래머는 또한 JNI 함수 `RegisterNatives`를 호출하여 클래스와 관련된 기본 메서드를 등록 할 수 있습니다. `RegisterNatives` 함수는 정적으로 연결된 함수에 특히 유용합니다.

11.4 호출 규칙

호출 규칙은 네이티브 함수가 인수를 받고 결과를 반환하는 방법을 결정합니다. 다양한 모국어 사이 또는 동일한 언어의 서로 다른 구현 사이에는 표준 호출 규칙이 없습니다. 예를 들어 C++ 컴파일러마다 서로 다른 호출 규칙을 따르는 코드를 생성하는 것이 일반적입니다.

불가능하지는 않더라도 Java 가상 머신이 다양한 기본 호출 규칙과 상호 운용되도록 요구하는 것은 어렵습니다. JNI를 사용하려면 지정된 호스트 환경에서 지정된 표준 호출 규칙으로 네이티브 메서드를 작성해야 합니다. 예를 들어 JNI는 UNIX의 C 호출 규칙과 Win32의 stdcall 규칙을 따릅니다.

프로그래머가 다른 호출 규칙을 따르는 함수를 호출해야 하는 경우 JNI 호출 규칙을 적절한 모국어의 규칙에 맞게 조정하는 스텝 루틴을 작성해야 합니다.

11.5 JNIEnv 인터페이스 포인터

네이티브 코드는 JNIEnv 인터페이스 포인터를 통해 내보낸 다양한 함수를 호출하여 가상 머신 기능에 액세스합니다.

11.5.1 JNIEnv 인터페이스 포인터의 구성

JNIEnv 인터페이스 포인터는 함수 테이블에 대한 포인터를 포함하는 스레드 로컬 데이터에 대한 포인터입니다. 모든 인터페이스 기능은 테이블에서 미리 정의된 오프셋에 있습니다. JNIEnv 인터페이스는 C++ + 가상 함수 테이블처럼 구성 되며

11.5.1 JNIEnv 인터페이스 포인터의 구성

JNI 설계 개요

Microsoft COM 인터페이스와도 같습니다. 그림 11.3은 JNIEnv 인터페이스 포인터 세트를 보여줍니다.

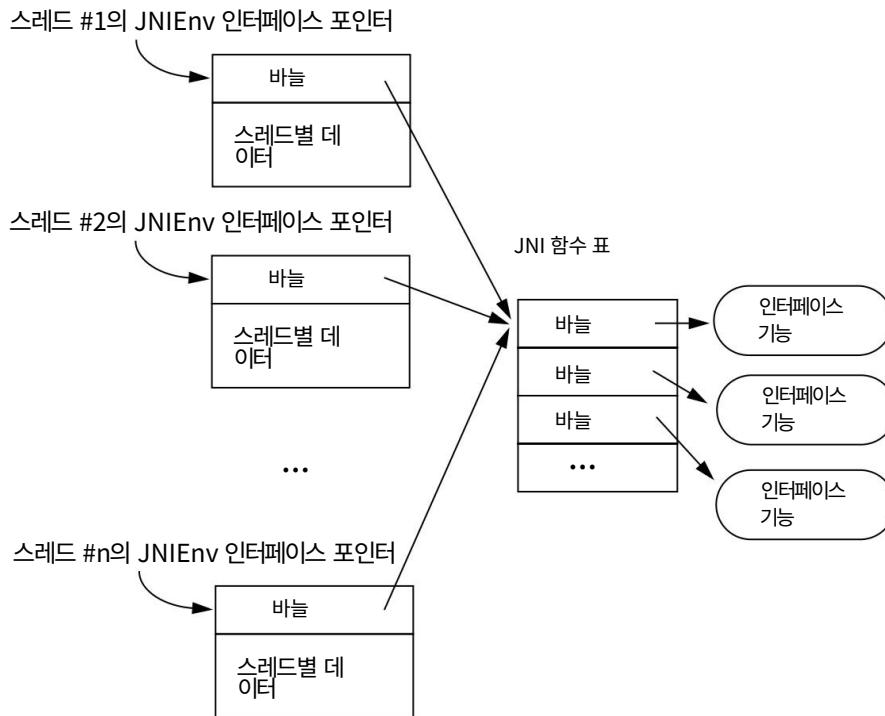


그림 11.3 스레드 로컬 JNIEnv 인터페이스 포인터

네이티브 메서드를 구현하는 함수는 JNIEnv 인터페이스 포인터를 첫 번째 인수로 받습니다. 가상 머신은 동일한 스레드에서 호출된 네이티브 메서드 구현 함수에 동일한 인터페이스 포인터를 전달하도록 보장됩니다. 그러나 네이티브 메서드는 다른 스레드에서 호출할 수 있으므로 다른 JNIEnv 인터페이스 포인터가 전달될 수 있습니다. 인터페이스 포인터는 스레드 로컬이지만 이중 간접 JNI 함수 테이블은 여러 스레드 간에 공유됩니다.

JNIEnv 인터페이스 포인터가 스레드 로컬 구조를 참조하는 이유는 일부 플랫폼이 스레드 로컬 스토리지 액세스를 효율적으로 지원하지 않기 때문입니다.

스레드 로컬 포인터를 전달함으로써 가상 머신 내부의 JNI 구현은 그렇지 않으면 수행해야 하는 많은 스레드 로컬 스토리지 액세스 작업을 피할 수 있습니다.

JNINv 인터페이스 포인터는 스레드 로컬이므로 네이티브 코드는 다른 스레드의 한 스레드에 속하는 JNINv 인터페이스 포인터를 사용하면 안 됩니다. 네이티브 코드는 JNINv 포인터를 스레드 수명 동안 고유하게 유지되는 스레드 ID로 사용할 수 있습니다.

11.5.2 인터페이스 포인터의 이점

유선 함수 항목과 달리 인터페이스 포인터를 사용하면 다음과 같은 몇 가지 이점이 있습니다.

- 가장 중요한 점은 JNI 함수 테이블이 각 네이티브 메서드에 인수로 전달되기 때문에 네이티브 라이브러리가 JVM(Java Virtual Machine)의 특정 구현과 연결될 필요가 없다는 것입니다. 벤더마다 가상 머신 구현의 이름을 다르게 지정할 수 있기 때문에 이는 매우 중요합니다. 각 기본 라이브러리를 독립적으로 유지하는 것은 동일한 기본 라이브러리 바이너리가 지정된 호스트 환경에서 다른 공급업체의 가상 머신 구현과 함께 작동하기 위한 전제 조건입니다. • 둘째, 유선 함수 항목을 사용하지 않음으로써 가상 머신 구현이 여러 버전의 JNI 함수 테이블을 제공하도록 선택할 수 있습니다. 예를 들어, 가상 머신 구현은 두 개의 JNI 함수 테이블을 지원할 수 있습니다. 하나는 철저한 불법 인수 검사를 수행하고 디버깅에 적합합니다. 다른 하나는 JNI 사양에서 요구하는 최소한의 검사를 수행하므로 더 효율적입니다. Java 2 SDK 릴리스 1.2는 선택적으로 JNI 기능에 대한 추가 검사를 설정하는 -Xcheck:jni 옵션을 지원합니다.
- 마지막으로 여러 JNI 함수 테이블을 사용하면 향후 여러 버전의 JNINv 유사 인터페이스를 지원할 수 있습니다. 아직 그렇게 해야 할 필요성을 예측하지는 못하지만 Java 플랫폼의 향후 버전은 1.1 및 1.2 릴리스에서 JNINv 인터페이스가 가리키는 것 외에도 새로운 JNI 함수 테이블을 지원할 수 있습니다. Java 2 SDK 릴리스 1.2에는 JNI_Onload 함수가 도입되었습니다. 이 함수는 네이티브 라이브러리에 필요한 JNI 함수 테이블의 버전을 나타내기 위해 네이티브 라이브러리에서 정의할 수 있습니다. Java 가상 머신의 향후 구현은 여러 버전의 JNI 함수 테이블을 동시에 지원하고 필요에 따라 개별 기본 라이브러리에 옮바른 버전을 전달할 수 있습니다.

11.6 데이터 전달

정수, 문자 등과 같은 기본 데이터 유형은 JVM(Java Virtual Machine)과 기본 코드 간에 복사됩니다. 반면 객체는 전달됩니다.

11.6.1 글로벌 및 로컬 참조

JNI 설계 개요

참고로. 각 참조에는 기본 개체에 대한 직접 포인터가 포함됩니다.

개체에 대한 포인터는 네이티브 코드에서 직접 사용되지 않습니다. 네이티브 코드의 관점에서 참조는 불투명합니다.

개체에 대한 직접적인 포인터 대신 참조를 전달하면 가상 머신이 보다 유연한 방식으로 개체를 관리할 수 있습니다. 그림 11.4는 이러한 유연성 중 하나를 보여줍니다. 네이티브 코드가 참조를 보유하고 있는 동안 가상 머신은 메모리의 한 영역에서 다른 영역으로 개체를 복사하는 가비지 수집을 수행할 수 있습니다. 가상 머신은 개체가 이동하더라도 참조가 여전히 유효하도록 참조 내용을 자동으로 업데이트할 수 있습니다.

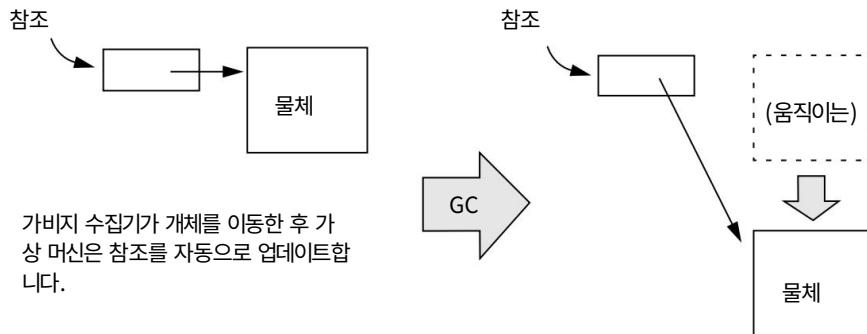


그림 11.4 네이티브 코드에 참조가 있는 동안 개체 재배치

11.6.1 글로벌 및 로컬 참조

JNI는 네이티브 코드에 대해 로컬 참조와 전역 참조라는 두 가지 개체 참조를 만듭니다. 로컬 참조는 네이티브 메서드 호출 기간 동안 유효합니다.

네이티브 메서드가 반환된 후 자동으로 해제됩니다. 전역 참조는 명시적으로 해제될 때까지 유효한 상태로 유지됩니다.

개체는 로컬 참조로 네이티브 메서드에 전달됩니다. 대부분의 JNI 함수는 로컬 참조를 반환합니다. JNI를 사용하면 프로그래머가 로컬 참조에서 전역 참조를 만들 수 있습니다. 개체를 인수로 사용하는 JNI 함수는 전역 및 로컬 참조를 모두 허용합니다. 네이티브 메서드는 가상 머신에 대한 로컬 또는 전역 참조를 결과로 반환할 수 있습니다.

로컬 참조는 참조가 생성된 스레드에서만 유효합니다. 네이티브 코드는 한 스레드에서 다른 스레드로 로컬 참조를 전달하면 안 됩니다.

JNI의 NULL 참조는 JVM (Java Virtual Machine)의 널 객체를 참조 합니다 . 값이 NULL 이 아닌 로컬 또는 전역 참조는 null 개체를 참조하지 않습니다 .

11.6.2 로컬 참조 구현

로컬 참조를 구현하기 위해 JVM(Java Virtual Machine)은 가상 머신에서 원시 메소드로 제어가 전환될 때마다 레지스트리를 작성합니다. 레지스트리는 이동할 수 없는 로컬 참조를 개체 포인터에 맵핑합니다. 레지스터의 개체는 가비지 수집할 수 없습니다. JNI 함수 호출의 결과로 반환되는 개체를 포함하여 네이티브 메서드에 전달된 모든 개체는 레지스트리에 자동으로 추가됩니다. 네이티브 메서드가 반환된 후 레지스트리가 삭제되어 해당 항목이 가비지 수집될 수 있습니다. 그림 11.5는 로컬 참조 레지스트리 를 만들고 삭제하는 방법을 보여줍니다. 기본 메소드에 해당하는 JVM(Java Virtual Machine) 프레임에는 로컬 참조 레지스트리에 대한 포인터가 포함되어 있습니다. 메서드 Df는 네이티브 메서드 Cg를 호출합니다. Cg는 C 함수 Java_C_g에 의해 구현됩니다. 가상 머신은 Java_C_g를 입력하기 전에 로컬 참조 레지스트리를 생성하고 Java_C_g를 반환된 후 로컬 참조 레지스트리를 삭제합니다.

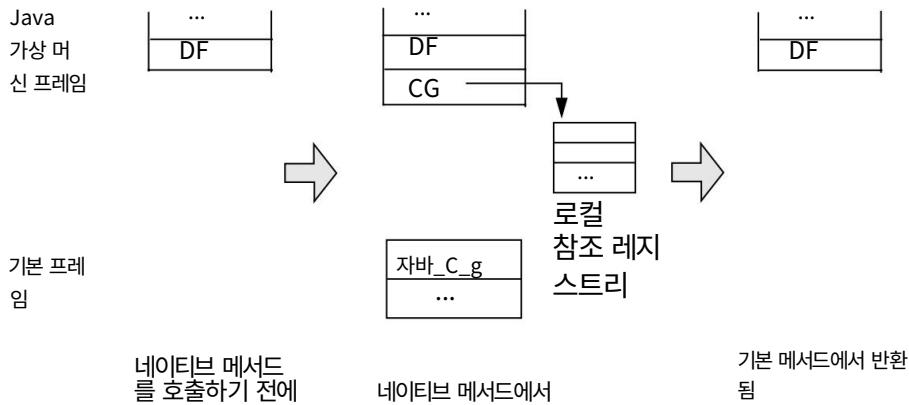


그림 11.5 로컬 참조 레지스트리 생성 및 삭제

스택, 테이블, 연결 목록 또는 해시 테이블을 사용하는 것과 같이 레지스트리를 구현하는 다양한 방법이 있습니다. 레지스트리에서 중복 항목을 방지하기 위해 참조 카운팅을 사용할 수 있지만 JNI 구현은 중복 항목을 감지하고 축소할 의무가 없습니다.

기본 스택을 보수적으로 스캔하여 로컬 참조를 충실히 구현할 수 없습니다. 네이티브 코드는 로컬 참조를 전역 또는 C 힙 데이터에 저장할 수 있습니다. 구조.

11.6.3 약한 전역 참조

JNI 설계 개요

11.6.3 약한 전역 참조

Java 2 SDK 릴리스 1.2에서는 새로운 종류의 전역 참조인 약한 전역 참조를 도입했습니다. 일반 전역 참조와 달리 약한 전역 참조를 사용하면 참조된 개체를 가비지 수집할 수 있습니다. 기본 개체가 가비지 수집된 후 약한 전역 참조가 지워집니다. 네이티브 코드는 참조를 NULL과 비교하기 위해 `IsSameObject`를 사용하여 약한 전역 참조가 지워졌는지 여부를 테스트할 수 있습니다.

11.7 객체 접근

JNI는 객체에 대한 참조를 위한 풍부한 접근자 함수 세트를 제공합니다. 이는 가상 머신이 내부적으로 개체를 나타내는 방식에 관계없이 동일한 기본 메서드 구현이 작동함을 의미합니다. 이는 모든 가상 머신 구현에서 JNI를 지원할 수 있도록 하는 중요한 설계 결정입니다.

불투명 참조를 통해 액세서 함수를 사용하는 오버헤드는 C 데이터 구조에 대한 직접 액세스보다 높습니다. 우리는 대부분의 경우 네이티브 메서드가 추가 함수 호출 비용을 무색하게 하는 중요한 작업을 수행한다고 믿습니다.

11.7.1 기본 배열 액세스

그러나 정수 배열 및 문자열과 같은 큰 개체의 기본 데이터 유형 값에 반복적으로 액세스하는 경우 함수 호출 오버헤드는 허용되지 않습니다.

벡터 및 행렬 계산을 수행하는 데 사용되는 기본 방법을 고려하십시오.

정수 배열을 반복하고 함수 호출로 모든 요소를 검색하는 것은 매우 비효율적입니다.

한 가지 솔루션은 "고정" 개념을 도입하여 네이티브 메서드가 가상 머신에 어레이의 내용을 이동하지 않도록 요청할 수 있도록 합니다. 그런 다음 네이티브 메서드는 요소에 대한 직접 포인터를 받습니다. 그러나 이 접근 방식에는 두 가지 의미가 있습니다.

- 가비지 수집기는 고정을 지원해야 합니다. 많은 구현에서 피닝은 가비지 수집 알고리즘을 복잡하게 하고 메모리 조각화로 이어지기 때문에 바람직하지 않습니다.
- 가상 머신은 기본 배열을 메모리에 연속적으로 배치해야 합니다.
이것이 대부분의 기본 배열에 대한 자연스러운 구현이지만 부울 배열은 압축되거나 압축되지 않은 상태로 구현될 수 있습니다. 압축된 부울 배열은 각 요소에 대해 1비트를 사용하는 반면 압축 해제된 부울 배열은 일반적으로 각 요소에 대해 1바이트를 사용합니다. 따라서 정확한 레이아웃에 의존하는 네이티브 코드는

부울 배열은 이식할 수 없습니다.

JNI는 위의 두 가지 문제를 모두 해결하는 절충안을 채택합니다.

첫째, JNI는 기본 배열의 세그먼트와 기본 메모리 버퍼 간에 기본 배열 요소를 복사하는 함수 세트(예: `GetIntArrayRegion` 및 `SetIntArrayRegion`)를 제공합니다. 네이티브 메서드가 큰 배열의 적은 수의 요소에만 액세스해야 하거나 네이티브 메서드가 어쨌든 배열의 복사본을 만들어야 하는 경우 이러한 함수를 사용하십시오.

둘째, 프로그래머는 다른 함수 집합(예: `GetInt ArrayElements`)을 사용하여 배열 요소의 고정된 버전을 얻으려고 시도할 수 있습니다. 그러나 가상 머신 구현에 따라 이러한 기능으로 인해 스토리지 할당 및 복사가 발생할 수 있습니다. 이러한 기능이 실제로 어레이를 복사하는지 여부는 다음과 같이 가상 머신 구현에 따라 다릅니다.

- 가비지 수집기가 고정을 지원하고 배열의 레이아웃이 동일한 유형의 기본 배열과 동일한 경우 복사가 필요하지 않습니다. • 그렇지 않으면 어레이가 이동 불가능한 메모리 블록(예: C 힙)에 복사되고 필요한 형식 변환이 수행됩니다. 복사본에 대한 포인터가 반환됩니다.

네이티브 코드는 세 번째 함수 집합(예: `ReleaseIntArrayElements`)을 호출 하여 네이티브 코드가 더 이상 배열 요소에 액세스할 필요가 없음을 가상 머신에 알립니다. 그런 일이 발생하면 가상 머신은 어레이를 고정 해제하거나 원래 어레이를 이동할 수 없는 복사본과 조정하고 복사본을 해제합니다.

이 접근 방식은 유연성을 제공합니다. 가비지 수집기 알고리즘은 각 어레이의 복사 또는 고정에 대해 별도의 결정을 내릴 수 있습니다. 특정 구현 체계에서 가비지 수집기는 작은 배열을 복사하지만 큰 배열은 고정할 수 있습니다.

마지막으로 Java 2 SDK 릴리스 1.2에는 `GetPrimitiveArrayCritical` 및 `ReleasePrimitiveArrayCritical`이라는 두 가지 새로운 기능이 도입되었습니다. 이러한 함수는 예를 들어 `GetIntArrayElements` 및 `ReleaseIntArrayElements`와 유사한 방식으로 사용할 수 있습니다. 그러나 `GetPrimitiveArrayCritical`를 사용하여 배열 요소에 대한 포인터를 얻은 후 `ReleasePrimitiveArrayCritical`를 사용하여 포인터를 해제하기 전에 네이티브 코드에 상당한 제한이 있습니다. "중요 영역" 내에서 네이티브 코드는 무한정 실행되어서는 안 되며, 임의의 JNI 함수를 호출해서는 안 되며, 현재 스레드가 차단되고 가상 머신의 다른 스레드를 기다릴 수 있는 작업을 수행해서는 안 됩니다. 이러한 제한 사항이 주어지면 가상 머신은 네이티브 코드에 배열 요소에 대한 직접 액세스 권한을 부여하면서 일시적으로 가비지 수집을 비활성화할 수 있습니다. 고정 지원이 필요하지 않기 때문에 `GetPrimitive`

11.7.2 필드 및 메소드

JNI 설계 개요

ArrayCritical은 예를 들어 GetIntArrayElements보다 기본 배열 요소에 대한 직접 포인터를 반환할 가능성 이 더 높습니다.

JNI 구현은 여러 스레드에서 실행되는 기본 메서드가 동일한 배열에 동시에 액세스할 수 있도록 해야 합니다. 예를 들어 JNI는 고정된 각 배열에 대한 내부 카운터를 유지하여 한 스레드가 다른 스레드에서도 고정된 배열 을 고정 해제하지 않도록 할 수 있습니다. JNI는 기본 메서드의 베타적 액세스를 위해 기본 배열을 잠글 필요가 없습니다. 서로 다른 스레드에서 배열을 동시에 업데이트하는 것은 허용되지만 이로 인해 결정적이지 않은 결과가 발생합니다.

11.7.2 필드 및 메소드

JNI를 사용하면 네이티브 코드가 필드에 액세스하고 Java 프로그래밍 언어에 정의된 메서드를 호출할 수 있습니다. JNI는 기호 이름과 유형 설명자로 메소드와 필드를 식별합니다. 2단계 프로세스는 이름과 설명자에서 필드 또는 메서드를 찾는 비용을 고려합니다. 예를 들어 클래스 `cls`에서 정수 인스턴스 필드 `i`를 읽으려면 기본 코드가 먼저 다음과 같이 필드 ID를 얻습니다.

```
jfieldID fid = env->GetFieldID(env, cls, "i", "I");
```

네이티브 코드는 다음과 같이 필드 조회 비용 없이 필드 ID를 반복적으로 사용할 수 있습니다.

```
진트 값 = env->GetIntField(env, obj, fid);
```

필드 또는 메서드 ID는 가상 머신이 클래스를 언로드할 때까지 유효한 상태로 유지됩니다.

또는 해당 필드 또는 메소드를 정의하는 인터페이스. 클래스 또는 인터페이스가 언로드된 후 메서드 또는 필드 ID 가 유효하지 않게 됩니다.

프로그래머는 해당 필드 또는 메서드를 확인할 수 있는 클래스 또는 인터페이스에서 필드 또는 메서드 ID를 파생시킬 수 있습니다. 필드 또는 메소드는 클래스 또는 인터페이스 자체에서 정의되거나 수퍼클래스 또는 수퍼 인터페이스에서 상속될 수 있습니다. JVM(Java™ Virtual Machine) 사양에는 필드 및 메서드를 해결하는 정확 한 규칙이 포함되어 있습니다. JNI 구현은 동일한 필드 또는 메서드 정의가 이러한 두 클래스 또는 인터페이스에서 확인되는 경우 두 클래스 또는 인터페이스에서 지정된 이름 및 설명자에 대해 동일한 필드 또는 메서드 ID를 파생해야 합니다.

예를 들어 B가 필드 `fld`를 정의하고 C가 B에서 `fld`를 상속하는 경우 프로그램 `mer`는 클래스 B 와 C 모두에서 필드 이름 "fld"에 대해 동일한 필드 ID를 얻도록 보장됩니다.

JNI는 필드 및 메서드 ID가 내부적으로 구현되는 방식에 대한 제한을 두지 않습니다.

지정된 클래스 또는 인터페이스에서 필드 ID를 얻으려면 필드 이름과 필드 설명자가 모두 필요합니다. 필드가

JNI 설계 개요

프로그래밍 오류 확인 없음 11.8.1

Java 프로그래밍 언어에서 오버로드되지 않습니다. 그러나 클래스 파일에 오버로드된 필드가 있고 이러한 클래스 파일을 JVM(Java Virtual Machine)에서 실행하는 것은 합법적입니다. 따라서 JNI는 Java 프로그래밍 언어용 컴파일러에서 생성되지 않은 올바른 클래스 파일을 처리할 수 있습니다.

프로그래머는 메소드 또는 필드의 이름과 유형을 이미 알고 있는 경우에만 JNI를 사용하여 메소드를 호출하거나 필드에 액세스할 수 있습니다. 이에 비해 Java Core Reflection API를 사용하면 프로그래머가 지정된 클래스 또는 인터페이스에서 필드 및 메서드 집합을 결정할 수 있습니다. 네이티브 코드에서도 클래스 또는 인터페이스 유형을 반영할 수 있는 것이 때때로 유용합니다. Java 2 SDK 펌리스 1.2는 기존 Java Core Reflection API와 함께 작동하도록 설계된 새로운 JNI 기능을 제공합니다. 새 함수에는 JNI 필드 ID와 java.lang.reflect.Field 클래스의 인스턴스 간에 변환하는 한 쌍과 JNI 메서드 ID와 java.lang.reflect.Method 클래스의 인스턴스 간에 변환하는 또 다른 쌍이 포함됩니다.

11.8 오류 및 예외

JNI 프로그래밍에서 발생하는 오류는 Java 가상 머신 구현에서 발생하는 예외와 다릅니다. 프로그래머 오류는 JNI 기능의 오용으로 인해 발생합니다. 예를 들어 프로그래머는 실수로 GetFieldID에 대한 클래스 참조 대신 개체 참조를 전달할 수 있습니다. 예를 들어 네이티브 코드가 JNI를 통해 개체를 할당하려고 할 때 발생하는 메모리 부족 상황으로 인해 Java 가상 머신 예외가 발생합니다.

11.8.1 프로그래밍 오류 확인 안 함

JNI 함수는 프로그래밍 오류를 확인하지 않습니다. 잘못된 인수를 JNI 함수에 전달하면 정의되지 않은 동작이 발생합니다. 이 디자인 결정의 이유는 다음과 같습니다.

- JNI 함수가 가능한 모든 오류 조건을 확인하도록 강제하면 성능이 저하됩니다.
모든(일반적으로 올바른) 네이티브 메서드의 성능.
- 많은 경우 이러한 작업을 수행하기 위한 런타임 유형 정보가 충분하지 않습니다.
확인 중.

대부분의 C 라이브러리 함수는 프로그래밍 오류를 방지하지 않습니다. 예를 들어 printf 함수는 일반적으로 잘못된 주소를 수신할 때 오류 코드를 반환하는 대신 런타임 오류를 트리거합니다. C 라이브러리 함수가 가능한 모든 오류 조건을 확인하도록 강제하면 사용자 코드에서 한 번, 라이브러리에서 다시 한 번 이러한 확인이 중복될 수 있습니다.

11.8.2 Java 가상 머신 예외

JNI 설계 개요

JNI 사양은 프로그래밍 오류를 확인하기 위해 가상 머신을 요구하지 않지만 가상 머신 구현은 일반적인 실수에 대한 확인을 제공하도록 권장됩니다. 예를 들어 가상 머신은 JNI 함수 테이블(§ 11.5.2)의 디버그 버전에서 더 많은 검사를 수행할 수 있습니다.

11.8.2 Java 가상 머신 예외

JNI는 기본 프로그래밍 언어의 예외 처리 메커니즘에 의존하지 않습니다. 원시 코드는 Java 가상 머신이 Throw 또는 ThrowNew를 호출하여 예외를 발생시키도록 할 수 있습니다. 보류 중인 예외가 현재 스레드에 기록됩니다. Java 프로그래밍 언어에서 발생한 예외와 달리 네이티브 코드에서 발생한 예외는 현재 실행을 즉시 방해하지 않습니다.

기본 언어에는 표준 예외 처리 메커니즘이 없습니다.

따라서 JNI 프로그래머는 잠재적으로 예외를 throw할 수 있는 각 작업 후에 예외를 확인하고 처리해야 합니다. JNI 프로그래머는 두 가지 방법으로 예외를 처리할 수 있습니다.

- 기본 메서드는 즉시 반환하도록 선택하여 예외를 발생시킬 수 있습니다.
네이티브 메서드 호출을 시작한 코드에서 발생합니다.
- 네이티브 코드는 `ExceptionClear`를 호출하여 예외를 지운 다음
자체 예외 처리 코드를 실행합니다.

후속 JNI 함수를 호출하기 전에 보류 중인 예외를 확인, 처리 및 지우는 것이 매우 중요합니다. 보류 중인 예외가 있는 대부분의 JNI 함수를 호출하면 정의되지 않은 결과가 발생합니다. 다음은 보류 중인 예외가 있을 때 안전하게 호출할 수 있는 JNI 함수의 전체 목록입니다.

예외 발생

예외 설명

예외지우기

예외 확인

`ReleaseStringChars`

`ReleaseStringUTFChars`

`ReleaseStringCritical`

`Release<Type>ArrayElements`

`ReleasePrimitiveArrayCritical`

로컬 참조 삭제

`GlobalRef 삭제`

`WeakGlobalRef 삭제`

모니터 종료

처음 네 개의 함수는 예외 처리와 직접 관련이 있습니다. 나머지는 다양한 가상 머신 리소스를 해제한다는 점에서 공통적입니다.

JNI를 통해 노출됩니다. 예외가 발생할 때 리소스를 해제할 수 있어야 하는 경우가 많습니다.

11.8.3 비동기 예외

한 스레드는 `Thread.stop`을 호출하여 다른 스레드에서 비동기 예외를 발생시킬 수 있습니다. 비동기 예외는 다음이 발생할 때까지 현재 스레드의 네이티브 코드 실행에 영향을 주지 않습니다.

- 네이티브 코드는 동기식을 발생시킬 수 있는 JNI 함수 중 하나를 호출합니다.
예외 또는
- 네이티브 코드는 `ExceptionOccurred`를 사용하여 동기 및 비동기를 확인합니다.
명시적으로 만성 예외.

잠재적으로 동기 예외를 발생시킬 수 있는 JNI 함수만
비동기 예외를 확인하십시오.

기본 메서드는 현재 스레드가 적절한 시간 내에 비동기 예외에 응답하도록 필요한 위치(예: 다른 예외 검사가 없는 긴밀한 루프)에 `ExceptionOccurred` 검사를 삽입할 수 있습니다.

비동기 예외를 생성하는 Java 스레드 API인 `Thread.stop`은 Java 2 SDK 릴리스 1.2에서 더 이상 사용되지 않습니다. 프로그래머는 일반적으로 신뢰할 수 없는 프로그램으로 이어지기 때문에 `Thread.stop` 사용을 강력히 권장하지 않습니다.
이것은 특히 JNI 코드의 문제입니다. 예를 들어 오늘날 10개로 작성된 많은 JNI 라이브러리는 이 섹션에서 설명하는 비동기 예외 확인 규칙을 신중하게 따르지 않습니다.

JNI 유형

이것 장은 JNI에서 정의한 표준 데이터 유형을 지정합니다. C 및 C++ 코드는 이러한 유형을 참조하기 전에 헤더 파일 jni.h를 포함해야 합니다.

12.1 기본 및 참조 유형

JNI는 Java 프로그래밍 언어의 기본 유형 및 참조 유형에 해당하는 C/C++ 유형 세트를 정의합니다.

12.1.1 기본 유형

다음 표에서는 Java 프로그래밍 언어의 기본 유형과 JNI의 해당 유형에 대해 설명합니다. Java 프로그래밍 언어의 해당 항목과 마찬가지로 JNI의 모든 기본 유형에는 잘 정의된 크기가 있습니다.

자바 언어 유형	네이티브 유형	설명
부울	jboolean	부호 없는 8비트
바이트	제이바이트	부호 있는 8비트
숏	jchar	부호 없는 16비트
짧은	jshort	부호 있는 16비트
정수	진트	서명된 32비트
긴	제롱	서명된 64비트
뜨다	jfloat	32비트
더블	제이더블	64비트

12.1.2 참조 유형

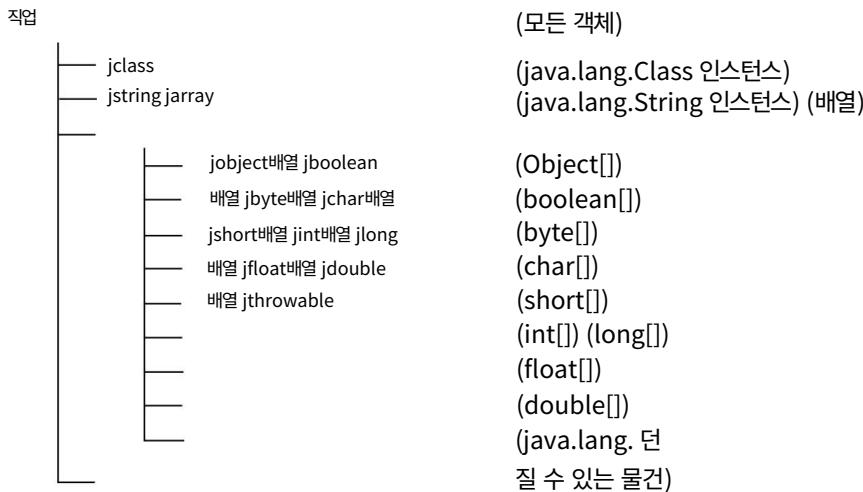
JNI 유형

`jsize` 정수 유형은 기본 인덱스 및 크기를 설명하는 데 사용됩니다.

```
typedef 진트 jsize;
```

12.1.2 참조 유형

JNI에는 Java 프로그래밍 언어의 다양한 참조 유형에 해당하는 많은 참조 유형이 포함되어 있습니다.
JNI 참조 유형은 아래 표시된 계층 구조로 구성됩니다.



C 프로그래밍 언어에서 사용될 때 다른 모든 JNI 참조 유형은 `jobject`와 동일하게 정의됩니다.
예를 들어:

```
typedef jobject jclass;
```

C++ 프로그래밍 언어에서 사용되는 경우 JNI는 더미 클래스 세트를 도입하여 다양한 참조 유형 간의 하위 유형 지정 관계를 표현합니다.

```
클래스 _jobject {};
클래스 _jclass : 공개 _jobject {};
클래스 _jthrowable : 공개 _jobject {};
```

JNI 유형

jvalue 유형 12.1.3 _

```
클래스 _jstring : 공개 _jobject {};
클래스 _jarray : 공개
_jobject {};
클래스 _jbooleanArray: 공개 _jarray {};
클래스
_jbyteArray : 공개 _jarray {};
클래스 _jcharArray: 공개 _jarray {};
클래스 _jshortArray: 공개 _jarray {};
클래스 _jintArray : 공개
_jarray {};
클래스 _ jlongArray : 공개 _jarray {};
클래스 _jfloatArray:
공개 _jarray {};
클래스 _jdoubleArray: 공개 _jarray {};
클래스
_jobjectArray: 공개 _jarray {};
```

```
typedef _jobject *jobject; typedef
_jclass *jclass; typedef _jthrowable
*jthrowable; typedef _jstring *jstring; typedef
_jarray *자레이;
typedef _jbooleanArray
*jbooleanArray; typedef _jbyteArray *jbyteArray;
typedef _jcharArray *jcharArray; typedef _jshortArray
*jshortArray; typedef _jintArray *jintArray; typedef
_jlongArray *jlongArray; typedef _jfloatArray *jfloatArray;
typedef _jdoubleArray *jdoubleArray; typedef
_jobjectArray *jobjectArray;
```

12.1.3 jvalue 유형

jvalue 유형은 참조 유형과 기본 유형의 합집합입니다. 다음과 같이 정의됩니다.

```
typedef union jvalue { jboolean
z; 제이바이트 b; jchar jshort
s; 진트 나; jlong jfloat;
작업 객체 l; } 제이값;
```

12.2 필드 및 메소드 ID

메서드 및 필드 ID는 일반 C 포인터 유형입니다.

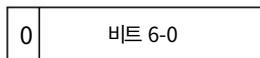
```
구조체 _jfieldID; /* 불투명한 구조 */ typedef struct _jfieldID *jfieldID; /* 필드 ID */
struct _jmethodID; /* 불투명한 구조 */ typedef struct _jmethodID *jmethodID; /
* 메서드 ID */
```

12.3 문자열 형식

JNI는 C 문자열을 사용하여 클래스 이름, 필드 및 메서드 이름, 필드 및 메서드 설명자를 나타냅니다. 이러한 문자열은 UTF-8 형식입니다.

12.3.1 UTF-8 문자열

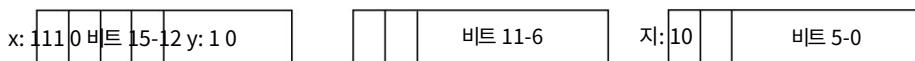
UTF-8 문자열은 null이 아닌 ASCII 문자만 포함하는 문자 시퀀스가 문자당 1바이트만 사용하여 표시될 수 있도록 인코딩되지만 최대 16비트의 문자도 표시될 수 있습니다. '\u0001' ~ '\u007F' 범위의 모든 문자는 다음과 같이 단일 바이트로 표시됩니다.



바이트의 7비트 데이터는 표현되는 문자의 값을 제공합니다. null 문자 ('\u0000') 와 '\u0080' ~ '\u07FF' 범위의 문자는 다음과 같이 바이트 쌍 x 및 y로 표시됩니다.



바이트는 값이 $((x \& 0x1f) << 6) + (y \& 0x3f)$ 인 문자를 나타냅니다.
 '\u0080' ~ '\uFFFF' 범위의 문자는 3바이트 x, y 및 z로 표시됩니다.



값이 $((x \& 0xf) << 12) + (y \& 0x3f) << 6) + (z \& 0x3f)$ 인 문자는 3바이트로 표시됩니다.

이 형식과 표준 UTF-8 형식 간에는 두 가지 차이점이 있습니다.

먼저 널바이트 (byte)0은 1바이트 형식이 아닌 2바이트 형식으로 인코딩된다. 즉, JNI UTF-8 문자열에는 null 이 포함되지 않습니다.

둘째, 1바이트, 2바이트 및 3바이트 형식만 사용됩니다. JNI는 더 긴 UTF-8 형식을 인식하지 않습니다.

12.3.2 클래스 설명자

클래스 설명자는 클래스 또는 인터페이스의 이름을 나타냅니다. "." 를 대체하여 Java™ 언어 사양에 정의된 대로 완전한 클래스 또는 인터페이스 이름에서 파생될 수 있습니다 . "/" 문자 가 있는 문자 . 예를 들어 java.lang.String 의 클래스 설명자는 다음과 같습니다.

"자바/언어/문자열"

배열 클래스는 "[" 문자 뒤에 필드를 사용하여 구성됩니다.

요소 유형의 설명자(§ 12.3.3). "int[]" 에 대한 클래스 설명자는 다음과 같습니다.

"[나"

"double[][][]" 에 대한 클래스 설명자는 다음과 같습니다.

"[[[디"

12.3.3 필드 설명자

8개의 기본 유형에 대한 필드 설명자는 다음과 같습니다.

필드 설명자	자바 언어 유형
지	부울
비	바이트
씨	숏
엑스	짧은
나	정수
제이	긴
제트	뜨다
디	더블

12.3.4 메서드 설명자

JNI 유형

참조 유형의 필드 설명자는 "L" 문자로 시작하고 그 뒤에 클래스 설명자가 오고 ";"로 끝납니다. 성격. 배열 유형의 필드 설명자는 배열 클래스의 클래스 설명자와 동일한 규칙에 따라 구성됩니다. 다음은 참조 유형 및 Java 프로그래밍 언어 대응 항목에 대한 필드 설명자의 몇 가지 예입니다.

필드 설명자	자바 언어 유형
"Ljava/lang/문자열;"	끈
"[나"	정수[]
"[Ljava/lang/객체;"	물체[]

12.3.4 메서드 설명자

메서드 설명자는 모든 인수 유형의 필드 설명자를 한 쌍의 괄호 안에 배치하고 그 뒤에 반환 유형의 필드 설명자를 배치하여 구성됩니다. 인수 유형 사이에는 공백이나 기타 구분 문자가 없습니다. "V"는 void 메서드 반환 유형을 나타내는 데 사용됩니다. 생성자는 반환 유형으로 "V"를 사용하고 이름으로 "<init>"를 사용합니다.

다음은 JNI 메서드 설명자와 해당 메서드 및 생성자 유형의 몇 가지 예입니다.

메서드 설명자	자바 언어 유형
"()Ljava/lang/문자열;"	문자열 에프();
"(ILjava/lang/Class;)J"	긴 f(int i, 클래스 c);
"([비)V"	String(바이트[]바이트);

12.4 상수

JNIEXPORT 및 JNICALL은 JNI 함수와 네이티브 메서드 구현 모두의 호출 및 연결 규칙을 정하는 데 사용되는 매크로입니다. 프로그램 mer는 함수 반환 유형 앞에 JNIEXPORT 매크로를 배치하고 함수 이름과 반환 유형 사이에 JNICALL 매크로를 배치해야 합니다. 예를 들어:

JNI 유형	상수	12.4
--------	----	------

JNIEXPORT 진트 JNICALL
Java_pkg_Cls_f(JNIEnv *env, jobject this);

pkg.Cls.f를 구현하는 C 함수의 프로토타입입니다.

```
jint(JNICALL *f_ptr)(JNIEnv *env, jobject this);
```

Java_pkg_Cls_f 함수를 할당할 수 있는 함수 포인터 변수입니다.
JNI_FALSE 및 JNI_TRUE는 jboolean 유형에 대해 정의된 상수입니다.

```
#define JNI_FALSE 0 #define  
JNI_TRUE 1
```

JNI_OK는 JNI 함수의 성공적인 반환 값을 나타내며 JNI_ERR은 때때로 오류 조건을 나타내는 데 사용됩니다.

```
#define JNI_OK  
#define JNI_ERR 0 (-1)
```

JNI 사양에는 현재 표준 오류 코드 세트가 포함되어 있지 않기 때문에 일부 오류 조건은 JNI_ERR로 표시되지 않습니다. JNI 함수는 성공 시 JNI_OK를 반환하고 실패 시 음수를 반환합니다.

다음 두 상수는 기본 배열의 기본 복사본을 해제하는 함수에서 사용됩니다. 이러한 함수의 예는 ReleaseIntArrayElements입니다.

JNI_COMMIT은 원시 배열이 Java 가상 머신의 원래 배열로 다시 복사되도록 합니다. JNI_ABORT는 새 콘텐츠를 다시 복사하지 않고 기본 배열에 할당된 메모리를 해제합니다.

```
#define JNI_COMMIT 1  
#define JNI_ABORT 2
```

Java 2 SDK 릴리스 1.2에는 JNI 버전 번호를 나타내는 두 개의 상수가 도입되었습니다.

```
#define JNI_VERSION_1_1 0x00010001 /* JNI 버전 1.1 */ #define JNI_VERSION_1_2  
0x00010002 /* JNI 버전 1.2 */
```

기본 애플리케이션은 다음 조건부 컴파일을 수행하여 jni.h 파일의 1.1 또는 1.2 버전에 대해 컴파일되는지 여부를 결정할 수 있습니다.

```
#ifdef JNI_VERSION_1_2 /* Java
2 SDK 1.2의 jni.h에 대해 컴파일 */ #else /* JDK 1.1의 jni.h에 대해 컴파일
*/ #endif
```

다음 상수는

JavaVM 인터페이스의 일부인 GetEnv 함수 :

```
#define JNI_EDETACHED (-2) /* VM에서 분리된 스레드 */ #define JNI_EVERSION (-3) /* JNI 버전 오류 */
```

13 장

JNI 함수

이것 장은 JNI 함수를 지정합니다. JNI 프로그래머에게 적용되는 제한 사항을 설명하기 위해 "반드시"라는 용어를 사용합니다. 예를 들어 특정 JNI 함수가 NULL이 아닌 개체를 받아야 하는 경우 해당 함수에 NULL을 전달하지 않는 것은 프로그래머의 책임입니다. 결과적으로 JNI 구현은 해당 함수에서 수신 가능한 NULL 포인터를 확인할 필요가 없습니다. 프로그래머가 해당 함수에 NULL을 전달하면 결과 동작이 정의되지 않습니다.

JNI 기능에 대한 요약부터 시작하겠습니다. 이 장의 본문에는 모든 JNI 함수의 세부 사양이 포함되어 있습니다.

13.1 JNI 기능 요약

JNI 함수는 정의된 위치와 용도에 따라 네 가지 범주 중 하나로 분류됩니다. 우선, 가상 머신 구현은 기본 기능 세트를 내보냅니다. 이러한 함수는 호출 인터페이스의 일부입니다. 네이티브 애플리케이션에서 가상 머신 인스턴스 생성과 같은 작업을 수행하는 데 사용할 수 있습니다. 둘째, JavaVM 인터페이스는 가상 머신 인스턴스를 나타냅니다. JavaVM 인터페이스는 예를 들어 네이티브 스레드를 가상 머신 인스턴스에 연결할 수 있는 기능을 제공합니다. 셋째, 네이티브 메서드를 구현하는 네이티브 라이브러리는 가상 머신 구현이 네이티브 라이브러리를 로드 및 언로드할 때 호출되는 특수 핸들러 함수를 내보낼 수 있습니다. 마지막으로 JNIEnv 인터페이스는 개체 생성, 필드 액세스 및 메서드 호출과 같은 JNI 기능을 지원합니다.

13.1.1 직접 내보낸 호출 인터페이스 함수

가상 머신 구현은 호출 인터페이스의 일부로 다음 세 가지 기능을 직접 내보냅니다.

JNI_GetDefaultJavaVMInitArgs
JNI_CreateJavaVM
JNI_GetCreatedJavaVM

JavaVM 인터페이스

JNI 함수

JNI_GetDefaultJavaVMInitArgs 함수는 가상 머신 인스턴스를 생성하는 데 사용 되는 기본 초기화 인수 값을 제공합니다. 이 정보는 JDK 릴리스 1.1의 가상 머신 구현에만 적용됩니다. 이 함수는 Java 2 SDK 릴리스 1.2에서 더 이상 유용하지 않지만 이전 버전과의 호환성을 위해 여전히 지원됩니다.

JNI_CreateJavaVM 함수는 주어진 초기화 인수 세트에 따라 가상 머신 인스턴스를 생성합니다. C 구조의 필드를 설정하여 JDK 릴리스 1.1에서 초기화 인수를 지정합니다. Java 2 SDK 릴리스 1.2는 초기화 인수를 지정하는 보다 유연한 방법을 지원하지만 이전 버전과의 호환성을 위해 동일한 JDK 1.1 스타일 초기화 구조를 계속 지원합니다.

JNI_GetCreatedJavaVMs 함수는 현재 프로세스에서 생성된 모든 가상 머신 인스턴스를 반환합니다. 특정 JNI 구현은 동일한 프로세스에서 둘 이상의 가상 머신 인스턴스를 생성할 수 있을 필요가 없습니다. JDK 릴리스 1.1과 Java 2 SDK 릴리스 1.2 모두 동일한 프로세스에서 둘 이상의 가상 머신 인스턴스 생성을 지원하지 않습니다.

가상 머신 인스턴스를 포함하는 기본 애플리케이션은 이러한 기능을 호출할 수 있습니다. 기본 애플리케이션은 이러한 기능을 내보내는 가상 머신 라이브러리에 연결하거나 기본 동적 연결 메커니즘을 사용하여 가상 머신 라이브러리를 로드하고 런타임에 내보낸 기능을 찾을 수 있습니다.

13.1.2 JavaVM 인터페이스

JavaVM 인터페이스는 함수 테이블에 대한 포인터에 대한 포인터입니다. 함수 테이블의 처음 세 항목은 Microsoft COM 인터페이스와의 향후 호환성을 위해 예약되어 있으며 NULL로 설정됩니다. 나머지 4개 항목은 호출 인터페이스의 일부입니다.

DestroyJavaVM

현재 스레드 연결

현재 스레드 분리

GetEnv

단일 스레드에 고유한 JNIEnv 인터페이스 포인터 와 달리 JavaVM 인터페이스 포인터는 전체 가상 머신 인스턴스를 나타내며 가상 머신 인스턴스의 모든 스레드에 대해 유효합니다.

DestroyJavaVM 함수는 JavaVM 인터페이스 포인터로 표시된 가상 머신 인스턴스를 언로드 합니다. AttachCurrentThread 함수는 가상 머신 인스턴스의 일부로 실행되도록 현재 네이티브 스레드를 설정합니다. 스레드가 가상 머신 인스턴스에 연결되면 JNI 함수 호출을 만들어 개체 액세스 및 메서드 호출과 같은 작업을 수행 할 수 있습니다. Detach CurrentThread 기능은 가상 머신 인스턴스에 현재

JNI 함수

JNIEnv 인터페이스 _

스레드는 더 이상 JNI 함수 호출을 실행할 필요가 없으므로 가상 머신 구현이 정리를 수행하고 리소스를 확보할 수 있습니다.

GetEnv 함수는 Java 2 SDK 릴리스 1.2의 새로운 기능입니다. 그것은 두 가지 목적을 제공할 수 있습니다. 첫째, 현재 스레드가 가상 머신 인스턴스에 연결되어 있는지 확인하는 데 사용할 수 있습니다. 둘째, JavaVM 인터페이스 포인터에서 JNIEnv 인터페이스 와 같은 다른 인터페이스를 얻는 데 사용할 수 있습니다 .

13.1.3 네이티브 라이브러리에 정의된 함수

Java 2 SDK 릴리스 1.2에서는 프로그래머가 가상 머신 구현이 기본 라이브러리를 로드 및 언로드할 때 호출할 추가 핸들러 기능을 내보낼 수 있습니다. 가상 머신 구현이 네이티브 라이브러리를 로드할 때 내보낸 함수 항목 JNI_OnLoad를 검색하고 호출합니다 . 가상 머신 구현이 네이티브 라이브러리를 언로드할 때 내보낸 함수 항목 JNI_OnUnload를 검색하고 호출합니다.

13.1.4 JNIEnv 인터페이스

JNIEnv 인터페이스는 JNI의 핵심 기능을 지원합니다 . 가상 머신 구현은 JNIEnv 인터페이스 포인터를 각 네이티브 메서드에 대한 첫 번째 인수로 전달합니다. 원시 코드는 JavaVM 인터페이스 포인터에서 GetEnv 함수를 호출하여 JNIEnv 인터페이스 포인터를 얻을 수도 있습니다 . JNIEnv 인터페이스 포인터는 특정 스레드에서만 유효 하지만 JavaVM 인터페이스 포인터는 가상 머신 인스턴스의 모든 스레드에서 유효합니다.

JNIEnv 인터페이스 포인터는 함수 테이블에 대한 포인터에 대한 포인터입니다. 함수 테이블의 처음 세 항목은 Microsoft COM 인터페이스와의 향후 호환성을 위해 예약되어 있으며 NULL로 설정됩니다. 함수 테이블의 네 번째 항목은 나중에 사용하기 위해 예약되어 있으며 역시 NULL로 설정됩니다.

이 장의 나머지 부분에서는 JNIEnv 인터페이스 의 모든 항목을 자세히 다룹니다. 지금은 높은 수준의 개요를 제공합니다.

버전 정보

- GetVersion은 JNIEnv 인터페이스 의 버전을 반환합니다 .

클래스 및 인터페이스 작업

- DefineClass는 기본 바이트 배열 표현에서 클래스 또는 인터페이스 유형을 정의합니다.
원시 클래스 파일 데이터를 보냅니다.
- FindClass는 주어진 이름의 클래스 또는 인터페이스 유형에 대한 참조를 반환합니다 . •
GetSuperclass는 주어진 클래스나 인터페이스의 슈퍼클래스를 반환합니다.

JNIEnv 인터페이스 _

JNI 함수

- `IsAssignableFrom`은 한 클래스 또는 인터페이스의 인스턴스를 다른 클래스 또는 인터페이스의 인스턴스에 할당할 수 있는지 확인하고 런타임 유형 확인에 유용합니다.

예외

- `Throw` 및 `ThrowNew`는 현재 스레드에서 예외를 발생시킵니다. • `ExceptionOccurred` 및 `ExceptionCheck`는 현재 스레드에서 보류 중인 예외를 확인합니다. `ExceptionCheck`는 Java 2 SDK 릴리스 1.2의 새로운 기능입니다. • `ExceptionDescribe`는 보류 중인 예외에 대한 진단 메시지를 인쇄합니다.
합니다.
- `ExceptionClear`는 보류 중인 예외를 자웁니다. • `FatalError`는 메시지를 인쇄하고 현재 가상 머신을 종료합니다.
사례.

글로벌 및 로컬 참조

- `NewGlobalRef`는 전역 참조를 생성하고 `DeleteGlobalRef`는 하나를 삭제합니다.
- `NewWeakGlobalRef` 및 `DeleteWeakGlobalRef`는 약한 글로벌 참조를 관리합니다. `ences`. 둘 다 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.
- `DeleteLocalRef`는 로컬 참조에 필요한 가상 머신 리소스를 회수합니다.
에렌스.
- `NewLocalRef`는 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.
- `ConfirmLocalCapacity`는 생성할 고정된 수의 로컬 참조를 위해 현재 스레드에 공간을 예약합니다.
`ConfirmLocalCapacity`는 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.
- `PushLocalFrame` 및 `PopLocalFrame`은 로컬 참조에 대한 중첩 범위를 생성합니다. 두 기능 모두 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.

개체 작업

- `AllocObject`는 초기화되지 않은 개체를 할당합니다. • `NewObject`는 객체를 할당하고 생성자 중 하나를 실행합니다. • `GetObjectClass`는 주어진 인스턴스의 클래스를 반환합니다. • `IsInstanceOf`는 주어진 객체가 주어진 클래스 또는 인터의 인스턴스인지 확인합니다.
얼굴.
- `IsSameObject`는 두 참조가 동일한 개체를 참조하는지 확인합니다.

JNI 함수

JNIEnv 인터페이스 _

인스턴스 필드 액세스

- GetFieldID는 주어진 클래스에서 기호 조회를 수행하고 필드를 반환합니다.
명명된 인스턴스 필드의 ID입니다.
- Get<Type>Field 및 Set<Type>Field 패밀리 액세스 기능
인스턴스 필드.

정적 필드 액세스

- GetStaticFieldID는 주어진 클래스 또는 인터페이스에서 기호 조회를 수행합니다.
명명된 정적 필드의 필드 ID를 반환합니다.
- GetStatic<Type>Field 및 SetStatic<Type>Field 제품군의 기능은 정적 필드에 액세스합니다.

인스턴스 메서드 호출

- GetMethodID는 주어진 클래스 또는 인터페이스에서 기호 조회를 수행하고
인스턴스 메서드 또는 생성자의 메서드 ID를 반환합니다.
- Call<Type>Method 패밀리의 함수는 인스턴스 메소드를 호출합니다. •
CallNonvirtual<Type>Method 계열의 함수는 수퍼클래스의 인스턴스 메소드 또는 생성자를 호출합니다.

정적 메서드 호출

- GetStaticMethodID는 주어진 클래스에서 기호 조회를 수행하고 정적 메서드의 메서드 ID를 반환합니다.
- CallStatic<Type>Method 계열의 함수는 정적 메서드를 호출합니다.

문자열 작업

- NewString은 네이티브 유니코드를 나타내는 java.lang.String 객체를 생성합니다.
끈.
- NewStringUTF는 네이티브를 나타내는 java.lang.String 객체를 생성합니다.
UTF-8 문자열.
- GetStringLength는 java.lang.String 객체가 나타내는 문자열의 유니코드 문자 수를 반환합니다 .
- GetStringLengthUTF는 지정된 java.lang.String 객체가 나타내는 문자열의 모든 문자를 인코딩하는
데 필요한 UTF-8 바이트 수를 반환합니다 . • GetStringChars 및 ReleaseStringChars는 유니코드 문
자열에 대한 포인터로 java.lang.String 객체의 내용에 액세스합니다 .

JNIEnv 인터페이스 _	JNI 함수
• GetStringUTFChars 및 ReleaseStringUTFChars는 UTF-8 문자열에 대한 포인터로 java.lang.String 객체의 내용에 액세스합니다.	
• GetStringCritical 및 ReleaseStringCritical은 최소한의 오버헤드로 java.lang.String 객체의 내용에 액세스합니다. 두 기능 모두 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.	
• GetStringRegion 및 GetStringUTFRegion은 java.lang.String 객체의 내용을 원시 버퍼로 복사합니다. 두 기능 모두 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.	

어레이 작업

- GetArrayLength는 배열의 요소 수를 반환합니다. • NewObjectArray는 객체의 배열을 생성하는 반면 New<Type>Array 패밀리의 함수는 기본 유형의 배열을 생성합니다.
- GetObjectArrayElement 및 SetObjectArrayElement는 기본 코드에서 참조 유형의 배열에 액세스합니다.
- Get<Type>ArrayElements 및 Release<Type>Array Elements 제품군의 기능은 기본 유형의 배열에 있는 모든 요소에 액세스합니다. • Get<Type>ArrayRegion 및 Set<Type>ArrayRegion 제품군의 기능은 기본 유형의 배열 안팎으로 여러 요소를 복사합니다.
- GetPrimitiveArrayCritical 및 ReleasePrimitiveArrayCritical은 최소한의 오버헤드로 프리미티브 유형 배열의 요소에 액세스합니다. 두 기능 모두 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.

네이티브 메서드 등록

- RegisterNatives 및 UnregisterNatives를 사용하면 네이티브 코드가 적극적으로 링크할 수 있습니다. 기본 메서드 연결을 해제합니다.

모니터링 작업

- MonitorEnter 및 MonitorExit는 연결된 모니터에서 동기화됩니다. 사물.

JavaVM 인터페이스

- GetJavaVM은 현재 가상 머신 인스턴스에 대한 JavaVM 인터페이스 포인터를 반환합니다.

JNI 함수

JNIEnv 인터페이스 _

반사 지원

- `FromReflectedField`는 Java Core Reflection API의 `java.lang.reflect.Field` 인스턴스를 필드 ID로 변환합니다. `FromReflectedField`는 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.
- `FromReflectedMethod`는 `java.lang.reflect.Method`의 인스턴스 또는 `java.lang.reflect.Constructor`의 인스턴스를 메소드 ID로 변환합니다. `FromReflectedMethod`는 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.
- `ToReflectedField` 및 `ToReflectedMethod`는 반대 방향으로 변환을 수행합니다. 두 기능 모두 Java 2 SDK 릴리스 1.2의 새로운 기능입니다.

인터페이스 포인터의 유연성으로 인해 JNIEnv 인터페이스를 쉽게 발전시킬 수 있습니다(§ 11.5.2).

JNI 사양의 향후 버전에서는 JNIEnv의 현재 버전과 다른 JNIEnv2 인터페이스와 같은 새로운 인터페이스를 도입할 수 있습니다. 향후 가상 머신 구현은 JNIEnv 및 JNIEnv2 인터페이스를 동시에 지원하여 이전 버전과의 호환성을 유지할 수 있습니다. 네이티브 라이브러리의 `JNI_OnLoad` 처리기 반환 값은 네이티브 라이브러리에서 예상하는 JNI 인터페이스의 버전에 대해 가상 머신 구현에 알립니다. 예를 들어 네이티브 라이브러리는 현재 다음과 같이 네이티브 함수 `Java_Foo_f`를 사용하여 네이티브 메서드 `Foo.f`를 구현할 수 있습니다.

JNIEXPORT 무효 JNICALL

```
Java_Foo_f(JNIEnv *env, jobject this, jint arg) {
    ... (*env)->... /* JNIEnv 인터페이스에 대한 일부 호출 */
}
```

향후에는 동일한 기본 메소드가 다음과 같이 구현될 수도 있습니다.

```
/* JNI 인터페이스의 가상 미래 버전(JNI_VERSION_2_0) */ JNIEnv2 *g_env;
```

JNIEXPORT 진트 JNICALL

```
JNINonLoad(JavaVM *vm, 무효 *예약됨) {
```

```
진트레스; /* 전
역 변수의 캐시 JNIEnv2 인터페이스 포인터 */ res = (*vm)->GetEnv(vm, (void **)&g_env,
JNI_VERSION_2_0); if (res < 0) { 반환 해상도;

}
JNI_VERSION_2_0을 반환합니다. /* 필요한 JNI 버전 */
}
```

JNI 함수 사양

JNI 함수

```
JNIEXPORT 무효 JNICALL
Java_Foo_f(jobject this, jint arg) {
    ... (*g_env)->... /* JNIEnv2 인터페이스에 대한 일부 호출 */
}
```

인터페이스 진화를 강조하기 위해 여러 가지 면에서 가상의 미래 JNIEnv2 인터페이스를 JNIEnv 인터페이스 와 다르게 만들었습니다 . JNIEnv2 인터페이스는 스레드 로컬 필요가 없으므로 전역 변수에 캐시될 수 있습니다 . JNIEnv2 인터페이스는 Java_Foo_f 기본 함수 에 대한 첫 번째 인수로 전달될 필요가 없습니다 . Java_Foo_f 와 같은 네이티브 메서드 구현 함수의 이름 인코딩 규칙(§ 11.3)은 변경할 필요가 없습니다. 가상 머신 구현은 JNI_OnLoad 핸들러의 반환 값에 의존하여 지정된 네이티브 라이브러리에서 네이티브 메서드 구현 함수의 인수 전달 규칙을 결정 합니다.

13.2 JNI 함수의 사양

이 섹션에는 JNI 함수의 전체 사양이 포함되어 있습니다. 각 기능에 대해 다음과 같은 정보를 제공합니다.

- 함수 프로토타입 • 매개 변수, 반환 값 및 가능한 항목을 포함한 자세한 설명 예외
- JNIEnv 및 JavaVM 인터페이스 함수 테이블

JNI 함수

AllocObject

AllocObject

원기

`jobject AllocObject(JNIEnv *env, jclass clazz);`

설명

개체의 생성자를 호출하지 않고 새 개체를 할당합니다. 개체에 대한 로컬 참조를 반환합니다.

`clazz` 인수는 배열 클래스를 참조하면 안 됩니다. 배열 개체를 할당하려면 `New<Type>Array` 함수 계열을 사용하십시오 .

`NewObject`, `NewObjectV` 또는 `NewObjectA`를 사용하여 개체를 할당하고 생성자 중 하나를 실행합니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 27 .

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`clazz`: 할당할 개체의 클래스에 대한 참조입니다.

반환 값 새로 할당된 개체에 대한 로컬 참조를 반환하거나 개체를 할당할 수 없는 경우 `NULL`을 반환합니다. 이 함수의 호출이 예외를 `throw`한 경우에만 `NULL`을 반환합니다 .

예외

`InstantiationException`: 클래스가 인터페이스 또는 추상 클래스인 경우.

`OutOfMemoryError`: 시스템의 메모리가 부족한 경우.

현재 스레드 연결

JNI 함수

현재 스레드 연결

원기

`jint AttachCurrentThread(JavaVM *vm, void **penv, void *args);`

설명

현재 스레드를 지정된 가상 머신 인스턴스에 연결합니다.
 가상 머신 인스턴스에 연결되면 기본 스레드에는 연관된 `java.lang.Thread` 인스턴스가 있습니다. 연결된 네이티브 스레드는 JNI 함수 호출을 실행할 수 있습니다. 네이티브 스레드는 `DetachCurrentThread`를 호출하여 분리될 때까지 가상 머신 인스턴스에 연결된 상태로 유지됩니다.

이미 연결된 스레드를 연결하려고 하면 `penv` 가 가리키는 값이 현재 스레드의 `JNIEnv`로 설정됩니다.

네이티브 스레드는 두 개의 JVM(Java Virtual Machine) 인스턴스에 동시에 연결할 수 없습니다.

JDK 릴리스 1.1에서 두 번째 인수는 `JNIEnv` 인터페이스 포인터를 받습니다. 세 번째 인수는 예약되어 있으며 NULL로 설정해야 합니다. 기본 `java.lang.Thread` 생성자는 연결된 `java.lang.Thread` 인스턴스에 대한 스레드 이름(예: "Thread-123")을 자동으로 생성합니다. `java.lang.Thread` 인스턴스는 가상 머신 구현에 의해 생성된 기본 스레드 그룹 "main"에 속합니다.

Java 2 SDK 릴리스 1.2에서는 릴리스 1.1 동작을 유지하기 위해 세 번째 인수를 NULL로 설정할 수 있습니다. 또는 세 번째 인수는 다음 구조를 가리킬 수 있습니다.

```
typedef struct {진트 버
    전; 문자 * 이름; 작업
    그룹;
} JavaVMAttachArgs;
```

버전 필드는 두 번째 인수를 통해 다시 전달된 `JNIEnv` 인터페이스의 버전을 지정합니다. Java 2 SDK 릴리스 1.2에서 허용하는 유효한 버전은 `JNI_VERSION_1_1` 및 `JNI_VERSION_1_2`입니다.

이름 필드가 NULL이 아닌 경우 연관된 `java.lang.Thread` 인스턴스의 이름을 지정하는 UTF-8 문자열을 가리킵니다. 이름 필드가 NULL이면 기본 `java.lang.Thread` 구성

JNI 함수

현재 스레드 연결

tor는 연관된 java.lang.Thread 인스턴스에 대한 스레드 이름(예: "Thread-123")을 생성합니다.

group 필드가 NULL 이 아닌 경우 새로 생성된 java.lang.Thread 인스턴스가 추가된 스레드 그룹의 전역 참조를 지정합니다. 그룹 필드가 NULL 이면 java.lang.Thread 인스턴스가 가상 머신 구현에 의해 생성된 기본 스레드 그룹 "main"에 추가됩니다.

결합

JavaVM 인터페이스 기능 테이블의 인덱스 4.

매개변수

vm: 현재 스레드가 연결될 가상 머신 인스턴스입니다.

penv: 현재 스레드에 대한 JNIEnv 인터페이스 포인터가 배치될 위치에 대한 포인터입니다.

args: 예약됨(JDK 릴리스 1.1에서) 또는 JavaVM AttachArgs 구조에 대한 포인터 (Java 2 SDK 릴리스 1.2에서).

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수를 반환합니다.

예외

없음.

<Type>메서드 호출

JNI 함수

<Type>메서드 호출

원기

```
<NativeType> Call<Type>Method(JNIEnv *env, jobject obj,
jmethodID methodID, ...);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

<Type>메서드 호출	<네이티브 유형>
CallVoid 메서드	무효의
CallObject 메서드	작업
CallBoolean 메서드 jboolean	제이바이트
CallByte 메서드	jchar
CallChar 메서드	jshort
CallShort 메서드	진트
CallInt 메서드	jlong
CallLong 메서드	jfloat
CallFloat 메서드	제이더블
CallDouble 메서드	

설명

캐치에서 메서드 ID를 사용하여 지정된 인스턴스 메서드를 호출합니다.

프로그래머는 methodID 인수 바로 뒤에 메서드에 전달될 모든 인수를 배치합니다. Call<Type>Method 함수는 이러한 인수를 받아 프로그래머가 호출하려는 메소드에 전달합니다.

JNI 함수

<Type>메서드 호출

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

<Type>메서드 호출	색인
CallVoid 메서드	61
CallObject 메서드	34
CallBoolean방법 37	
CallByte 메서드	40
CallChar 메서드	43
CallShort 메서드	46
CallInt 메서드	49
CallLong 메서드	52
CallFloat 메서드	55
CallDouble 메서드	58

매개변수

env : JNIEnv 인터페이스 포인터.

obj: 메서드가 호출되는 개체에 대한 참조입니다.

methodID: 호출할 메서드를 나타내는 메서드 ID.

추가 인수: 메서드에 전달할 인수입니다.

반환 값 메서드를 호출한 결과입니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

<Type>MethodA 호출

JNI 함수

<Type>MethodA 호출

원기

```
<NativeType> Call<Type>MethodA(JNIEnv *env,
jobject obj, jmethodID methodID, jvalue *args);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

Call<Type>MethodA <NativeType>	
CallVoidMethodA	무효의
CallObjectMethodA jobject	
CallBooleanMethodA jboolean	
CallByteMethodA	제이바이트
CallCharMethodA	jchar
CallShortMethodA	jshort
CallIntMethodA	진트
CallLongMethodA	jlong
CallFloatMethodA	jfloat
CallDoubleMethodA jdouble	

설명

캐시에서 메서드 ID를 사용하여 지정된 인스턴스 메서드를 호출합니다.

프로그래머는 메소드에 대한 모든 인수를 methodID 인수 바로 다음에 오는 jvalue 배열에 배치합니다. Call <Type>MethodA 루틴은 이 배열의 인수를 수락한 다음 프로그램 mer가 호출하려는 메서드에 인수를 전달합니다.

JNI 함수

<Type>MethodA 호출

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

Call<Type>MethodA 인덱스	
CallVoidMethodA	63
CallObjectMethodA	36
CallBooleanMethodA	39
CallByteMethodA	42
CallCharMethodA	45
CallShortMethodA	48
CallIntMethodA	51
CallLongMethodA	54
CallFloatMethodA	57
CallDoubleMethodA	60

매개변수

env : JNIEnv 인터페이스 포인터.

obj: 메서드가 호출되는 개체에 대한 참조입니다.

methodID: 호출할 메서드를 나타내는 메서드 ID.

args: 메서드에 전달할 인수의 배열입니다.

반환 값 메서드를 호출한 결과를 반환합니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

<Type>MethodV 호출

JNI 함수

<Type>MethodV 호출

원기

```
<NativeType> Call<Type>MethodV(JNIEnv *env,
jobject obj, jmethodID methodID, va_list args);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

Call<Type>MethodV <NativeType>	
CallVoidMethodV	무효의
CallObjectMethodV jobject	
CallBooleanMethodV jboolean	
CallByteMethodV	제이바이트
CallCharMethodV	jchar
CallShortMethodV	jshort
CallIntMethodV	진트
CallLongMethodV	jlong
CallFloatMethodV	jfloat
CallDoubleMethodV jdouble	

설명

캐시에서 메서드 ID를 사용하여 지정된 인스턴스 메서드를 호출합니다.

프로그래머는 메서드에 대한 모든 인수를 methodID 인수 바로 뒤에 오는 va_list 유형의 args 인수에 배치합니다. Call <Type>MethodV 루틴은 인수를 받아들인 다음 프로그래머가 호출하려는 메소드에 인수를 전달합니다.

JNI 함수

<Type>MethodV 호출

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

Call<Type>MethodV 인덱스	
CallVoidMethodV	62
CallObjectMethodV	35
CallBooleanMethodV	38
CallByteMethodV	41
CallCharMethodV	44
CallShortMethodV	47
CallIntMethodV	50
CallLongMethodV	53
CallFloatMethodV	56
CallDoubleMethodV	59

매개변수

env : JNIEnv 인터페이스 포인터.

obj: 메서드가 호출되는 개체에 대한 참조입니다.

methodID: 호출할 메소드의 메소드 ID.

args: 호출된 메서드에 전달된 인수의 va_list 입니다.

반환 값 메서드를 호출한 결과를 반환합니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

CallNonvirtual<Type>방법

JNI 함수

CallNonvirtual<Type>방법

원기

```
<NativeType> CallNonvirtual<Type>Method(
    JNIEnv *env, jobject obj, jclass clazz, jmethodID
    methodID, ...);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

CallNonvirtual<Type>방법	<네이티브 유형>
CallNonvirtualVoidMethod	무효의
CallNonvirtualObjectMethod	직업
CallNonvirtualBooleanMethod	jboolean
CallNonvirtualByteMethod	제이바이트
CallNonvirtualCharMethod	jchar
CallNonvirtualShortMethod	jshort
CallNonvirtualIntMethod	진트
CallNonvirtualLongMethod	jlong
CallNonvirtualFloatMethod	jfloat
CallNonvirtualDoubleMethod	제이더블

설명

캐시에서 클래스 및 메서드 ID를 사용하여 지정된 인스턴스 메서드를 호출합니다.

CallNonvirtual <Type>Method 함수군과 Call<Type>Method 함수군은 다릅니다.

Call<Type>Method 함수는 캐시의 실제 클래스를 기반으로 메서드를 호출하는 반면, CallNonvirtual<Type>Method 루틴은 메서드 ID를 가져온 clazz 매개 변수로 지정된 클래스를 기반으로 메서드를 호출합니다.

clazz 매개변수는 객체의 실제 클래스 또는 상위 클래스 중 하나를 참조해야 합니다.

프로그래머는 methodID 인수 바로 뒤에 메서드에 전달될 모든 인수를 배치합니다. CallNonvirtual <Type>Method 루틴은 이러한 인수를 받아 프로그래머가 호출하려는 메소드에 전달합니다.

JNI 함수

CallNonvirtual<Type>방법

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

CallNonvirtual<Type>방법	색인
CallNonvirtualVoidMethod	91
CallNonvirtualObjectMethod	64
CallNonvirtualBooleanMethod	67
CallNonvirtualByteMethod	70
CallNonvirtualCharMethod	73
CallNonvirtualShortMethod	76
CallNonvirtualIntMethod	79
CallNonvirtualLongMethod	82
CallNonvirtualFloatMethod	85
CallNonvirtualDoubleMethod	88

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 메서드 ID가 파생된 클래스에 대한 참조입니다.

obj: 메서드가 호출되는 개체에 대한 참조입니다.

methodID: 클래스 참조에 유효한 메서드 ID
클라즈.

추가 인수: 메서드에 전달할 인수입니다.

반환 값 메서드를 호출한 결과를 반환합니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

CallNonvirtual<Type>MethodA

JNI 함수

CallNonvirtual<Type>MethodA

원기

```
<NativeType> CallNonvirtual<Type>MethodA(
    JNIEnv *env, jobject obj, jclass clazz, jmethodID
    methodID, jvalue *args);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

CallNonvirtual<Type>MethodA	<네이티브 유형>
CallNonvirtualVoidMethodA	무효의
CallNonvirtualObjectMethodA	직업
CallNonvirtualBooleanMethodA	jboolean
CallNonvirtualByteMethodA	제이바이트
CallNonvirtualCharMethodA	jchar
CallNonvirtualShortMethodA	jshort
CallNonvirtualIntMethodA	진트
CallNonvirtualLongMethodA	jlong
CallNonvirtualFloatMethodA	jfloat
CallNonvirtualDoubleMethodA	제이더블

설명

개체에서 클래스 및 메서드 ID를 사용하여 지정된 인스턴스 메서드를 호출합니다.

CallNonvirtual <Type>MethodA 함수군과 Call<Type>MethodA 함수군은 다릅니다.

Call<Type>MethodA 함수는 개체의 실제 클래스를 기반으로 메서드를 호출하는 반면, CallNonvirtual<Type>MethodA 루틴은 메서드 ID를 가져온 clazz 매개 변수로 지정된 클래스를 기반으로 메서드를 호출합니다.

clazz 매개변수는 객체의 실제 클래스 또는 상위 클래스 중 하나를 참조해야 합니다.

프로그래머는 메소드에 대한 모든 인수를 methodID 인수 바로 뒤에 오는 jvalue의 args 배열에 배치합니다. CallNonvirtual <Type>MethodA 루틴은 이 배열의 인수를 수락한 다음 프로그래머가 호출하려는 메서드에 인수를 전달합니다.

JNI 함수

CallNonvirtual<Type>MethodA

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

CallNonvirtual<Type>MethodA	색인
CallNonvirtualVoidMethodA	93
CallNonvirtualObjectMethodA	66
CallNonvirtualBooleanMethodA	69
CallNonvirtualByteMethodA	72
CallNonvirtualCharMethodA	75
CallNonvirtualShortMethodA	78
CallNonvirtualIntMethodA	81
CallNonvirtualLongMethodA	84
CallNonvirtualFloatMethodA	87
CallNonvirtualDoubleMethodA	90

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 메서드 ID가 파생된 클래스에 대한 참조입니다.

obj: 메서드가 호출되는 개체에 대한 참조입니다.

methodID: 클래스 참조에 유효한 메서드 ID
클래스.

args: 메서드에 전달할 인수의 배열입니다.

반환 값 메서드를 호출한 결과를 반환합니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

CallNonvirtual<Type>MethodV

JNI 함수

CallNonvirtual<Type>MethodV

원기

```
<NativeType> CallNonvirtual<Type>MethodV(
    JNIEnv *env, jobject obj, jclass clazz, jmethodID
    methodID, va_list args);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

CallNonvirtual<Type>MethodV	<네이티브 유형>
CallNonvirtualVoidMethodV	무효의
CallNonvirtualObjectMethodV	직업
CallNonvirtualBooleanMethodV	jboolean
CallNonvirtualByteMethodV	제이바이트
CallNonvirtualCharMethodV	jchar
CallNonvirtualShortMethodV	jshort
CallNonvirtualIntMethodV	진트
CallNonvirtualLongMethodV	jlong
CallNonvirtualFloatMethodV	jfloat
CallNonvirtualDoubleMethodV	제이더블

설명

개체에서 클래스 및 메서드 ID를 사용하여 지정된 인스턴스 메서드를 호출합니다. methodID 인수는 클래스 clazz에서 GetMethodID를 호출하여 얻어야 합니다.

CallNonvirtual <Type>MethodV 함수 계열과 Call<Type>MethodV 함수 계열은 다릅니다.

Call<Type>MethodV 함수는 개체의 실제 클래스를 기반으로 메서드를 호출하는 반면 CallNonvirtual<Type>MethodV 루틴은 메서드 ID를 가져오는 clazz 매개 변수로 지정된 클래스를 기반으로 메서드를 호출합니다.

clazz 매개변수는 객체의 실제 클래스 또는 상위 클래스 중 하나를 참조해야 합니다.

프로그래머는 메서드에 대한 모든 인수를 methodID 인수 바로 뒤에 오는 va_list 유형의 args 인수에 배치합니다. CallNonvirtual <Type>MethodV 루틴은 인수를 수락한 다음 프로그래머가 호출하려는 메서드에 인수를 전달합니다.

JNI 함수

CallNonvirtual<Type>MethodV

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다 .

CallNonvirtual<Type>MethodV	색인
CallNonvirtualVoidMethodV	92
CallNonvirtualObjectMethodV	65
CallNonvirtualBooleanMethodV	68
CallNonvirtualByteMethodV	71
CallNonvirtualCharMethodV	74
CallNonvirtualShortMethodV	77
CallNonvirtualIntMethodV	80
CallNonvirtualLongMethodV	83
CallNonvirtualFloatMethodV	86
CallNonvirtualDoubleMethodV	89

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 메서드 ID가 파생된 클래스에 대한 참조입니다.

obj: 메서드가 호출되는 개체에 대한 참조입니다.

methodID: 클래스 참조에 유효한 메서드 ID
클라즈.

args: 메서드에 전달할 인수의 va_list 입니다 .

반환 값 메서드를 호출한 결과를 반환합니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

CallStatic<Type>메서드	JNI 함수
---------------------	--------

CallStatic<Type>메서드

원기

```
<NativeType> CallStatic<Type>메서드(
    JNIEnv *env, jclass clazz, jmethodID
    methodID, ...);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

CallStatic<Type>메서드	<네이티브 유형>
CallStaticVoid 메서드	무효의
CallStaticObject 메서드	작업
CallStaticBoolean 메서드	jboolean
CallStaticByteMethod	제이바이트
CallStaticChar 메서드	jchar
CallStaticShortMethod	jshort
CallStaticInt 메서드	진트
CallStaticLongMethod	jlong
CallStaticFloat 메서드	jfloat
CallStaticDoubleMethod	제이더블

설명

클래스에서 메서드 ID를 사용하여 지정된 정적 메서드를 호출합니다. 메서드는 clazz의 상위 클래스 중 하나에서 정의될 수 있지만 clazz에서 액세스 할 수 있어야 합니다.

프로그래머는 methodID 인수 바로 뒤에 메서드에 전달될 모든 인수를 배치해야 합니다. CallStatic <Type>Method 루틴은 이러한 인수를 받아 프로그래머가 호출하려는 정적 메소드에 전달합니다.

JNI 함수

CallStatic<Type>메서드

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

CallStatic<Type>메서드	색인
CallStaticVoid 메서드	141
CallStaticObject 메서드	114
CallStaticBoolean 메서드	117
CallStaticByteMethod	120
CallStaticChar 메서드	123
CallStaticShortMethod	126
CallStaticInt 메서드	129
CallStaticLongMethod	132
CallStaticFloat 메서드	135
CallStaticDoubleMethod	138

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 정적 메서드가 호출되는 클래스 개체에 대한 참조입니다.

methodID: 호출할 메서드의 정적 메서드 ID입니다.

추가 인수: 정적 메서드에 전달할 인수입니다.

반환 값: 정적 메서드 호출 결과를 반환합니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

CallStatic<Type>MethodA

JNI 함수

CallStatic<Type>MethodA

원기

```
<NativeType> CallStatic<Type>MethodA(
    JNIEnv *env, jclass clazz, jmethodID
    methodID, jvalue *args);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

CallStatic<Type>MethodA	<네이티브 유형>
CallStaticVoidMethodA	무효의
CallStaticObjectMethodA	직업
CallStaticBooleanMethodA	jboolean
CallStaticByteMethodA	제이바이트
CallStaticCharMethodA	jchar
CallStaticShortMethodA	jshort
CallStaticIntMethodA	진트
CallStaticLongMethodA	jlong
CallStaticFloatMethodA	jfloat
CallStaticDoubleMethodA	제이더블

설명

클래스에서 메서드 ID를 사용하여 지정된 정적 메서드를 호출합니다. 메서드는 clazz의 상위 클래스 중 하나에서 정의될 수 있지만 clazz에서 액세스 할 수 있어야 합니다.

프로그래머는 methodID 인수 바로 뒤에 오는 jvalue 의 args 배열에 메서드에 대한 모든 인수를 배치해야 합니다. CallStatic <Type>MethodA 루틴은 이 배열의 인수를 수락한 다음 프로그래머가 호출하려는 정적 메서드에 인수를 전달합니다.

JNI 함수

CallStatic<Type>MethodA

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

CallStatic<Type>MethodA	색인
CallStaticVoidMethodA	143
CallStaticObjectMethodA	116
CallStaticBooleanMethodA	119
CallStaticByteMethodA	122
CallStaticCharMethodA	125
CallStaticShortMethodA	128
CallStaticIntMethodA	131
CallStaticLongMethodA	134
CallStaticFloatMethodA	137
CallStaticDoubleMethodA	140

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 정적 메서드가 호출되는 클래스 개체에 대한 참조입니다.

methodID: 호출할 메서드의 정적 메서드 ID입니다.

args: 정적 메서드에 전달할 인수의 배열입니다.

반환 값 정적 메서드 호출 결과를 반환합니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

CallStatic<Type>MethodV	JNI 함수
-------------------------	--------

CallStatic<Type>MethodV

원기

```
<NativeType> CallStatic<Type>MethodV(
    JNIEnv *env, jclass clazz, jmethodID
    methodID, va_list args);
```

양식

이 함수 계열은 10개의 멤버로 구성됩니다.

CallStatic<Type>MethodV	<네이티브 유형>
CallStaticVoidMethodV	무효의
CallStaticObjectMethodV	직업
CallStaticBooleanMethodV	jboolean
CallStaticByteMethodV	제이바이트
CallStaticCharMethodV	jchar
CallStaticShortMethodV	jshort
CallStaticIntMethodV	진트
CallStaticLongMethodV	jlong
CallStaticFloatMethodV	jfloat
CallStaticDoubleMethodV	제이더블

설명

클래스에서 메서드 ID를 사용하여 지정된 정적 메서드를 호출합니다. 메서드는 clazz의 상위 클래스 중 하나에서 정의될 수 있지만 clazz에서 액세스 할 수 있어야 합니다.

프로그래머는 methodID 인수 바로 뒤에 오는 va_list 유형의 args 인수에 메서드에 대한 모든 인수를 배치해야 합니다. CallStatic <Type>MethodV 루틴은 인수를 승인한 다음 프로그래머가 호출하려는 정적 메소드에 인수를 전달합니다.

JNI 함수

CallStatic<Type>MethodV

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다 .

CallStatic<Type>MethodV	색인
CallStaticVoidMethodV	142
CallStaticObjectMethodV	115
CallStaticBooleanMethodV	118
CallStaticByteMethodV	121
CallStaticCharMethodV	124
CallStaticShortMethodV	127
CallStaticIntMethodV	130
CallStaticLongMethodV	133
CallStaticFloatMethodV	136
CallStaticDoubleMethodV	139

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 정적 메서드가 호출되는 클래스 개체에 대한 참조입니다.

methodID: 호출할 메서드의 정적 메서드 ID입니다.

args: 정적 메서드에 전달할 인수의 va_list 입니다 .

반환 값 정적 메서드 호출 결과를 반환합니다.

예외

메서드 실행 중에 발생하는 모든 예외입니다.

클래스 정의	JNI 함수
클래스 정의	
원기	jclass DefineClass(JNIEnv *env, const char *name, jobject loader, const jbyte *buf, jsize bufLen);
설명	클래스 또는 인터페이스를 나타내는 원시 클래스 데이터의 버퍼에서 java.lang.Class 인스턴스를 만듭니다 . 원시 클래스 데이터의 형식은 Java™ Virtual Machine 사양에 의해 지정됩니다.
	이 함수는 java.lang.ClassLoader.defineClass 메서드 보다 약간 더 일반적 입니다 . 이 함수는 null 클래스 로더를 사용하여 클래스 또는 인터페이스를 정의 할 수 있습니다 . java.lang.ClassLoader.defineClass 메서드는 인스턴스 메 소드이므로 java.lang.ClassLoader 인스턴스가 필요 합니다 .
결합	JNIEnv 인터페이스 함수 테이블 의 인덱스 5 .
매개변수	env : JNIEnv 인터페이스 포인터. name: 정의할 클래스 또는 인터페이스의 이름입니다. 로더: 정의된 클래스 또는 인터페이스에 할당된 클래스 로더. buf: 원시 클래스 파일 데이터를 포함하는 버퍼. bufLen: 버퍼 길이.
	반환 값 새로 정의된 클래스 또는 인터페이스 개체에 대한 로컬 참조를 반환하거나 예외 가 발생하면 NULL을 반환합니다. 이 함수의 호출이 예외를 throw 한 경우에만 NULL을 반환합니다 .
예외	ClassFormatError: 클래스 데이터가 유효한 클래스 또는 인터페이스를 지 정하지 않은 경우. NoClassDefFoundError: 클래스 데이터가 정의할 명명된 클래스 또 는 인터페이스를 지정하지 않은 경우. ClassCircularityError: 클래스 또는 인터페이스가 자체 슈퍼클래스 또는 슈 퍼인터페이스인 경우. OutOfMemoryError: 시스템의 메모리가 부족한 경우.

JNI 함수

GlobalRef 삭제

GlobalRef 삭제

원기

무효 DeleteGlobalRef(JNIEnv *env, jobject gref);

설명

gref 가 가리키는 전역 참조를 삭제합니다 . gref 인수는 전역 참조 또는 NULL 이어야 합니다. NULL이 아닌 동일한 전역 참조는 두 번 이상 삭제하면 안 됩니다. NULL 전역 참조를 삭제하는 것은 작동하지 않습니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 22 .

매개변수

env : JNIEnv 인터페이스 포인터.

gref: 삭제할 전역 참조입니다.

예외

없음.

로컬 참조 삭제

JNI 함수

로컬 참조 삭제

원기

```
무효 DeleteLocalRef(JNIEnv *env, jobject lref);
```

설명

`lref` 가 가리키는 로컬 참조를 삭제합니다. `lref` 인수는 로컬 참조 이거나 NULL이어야 합니다. NULL이 아닌 동일한 로컬 참조는 두 번 이상 삭제하면 안 됩니다. NULL 로컬 참조를 삭제하는 것은 작동하지 않습니다.

최상위 로컬 참조 프레임에 속하지 않는 로컬 참조를 삭제하면 작동하지 않습니다. 각 네이티브 메서드 호출은 새로운 로컬 참조 프레임을 생성합니다. `PushLocalFrame` 함수(Java 2 SDK 릴리스 1.2에 추가됨)도 새 로컬 참조 프레임을 만듭니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 23.

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`lref`: 삭제할 로컬 참조.

예외

없음.

JNI 함수

WeakGlobalRef 삭제

WeakGlobalRef 삭제

원기

```
무효 DeleteWeakGlobalRef(JNIEnv *env, jobject  
wref);
```

설명

약한 전역 참조를 삭제합니다. wref 인수는 약한 전역 참조여야 합니다. 동일한 약한 전역 참조를 두 번 이상 삭제하면 안 됩니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 227 .

매개변수

env : JNIEnv 인터페이스 포인터.

wref: 삭제할 약한 전역 참조.

예외

없음.

DestroyJavaVM

JNI 함수

DestroyJavaVM

원기

```
jint DestroyJavaVM(JavaVM *vm);
```

설명

가상 머신 인스턴스를 언로드하고 리소스를 회수합니다.

가상 머신 인스턴스를 언로드하려고 시도하기 전에 현재 스레드가 유일한 사용자 스레드가 될 때까지 시스템이 차단됩니다. 연결된 스레드가 잠금, 창 등과 같은 시스템 리소스를 보유할 수 있기 때문에 이러한 제한이 존재합니다.

가상 머신 구현은 이러한 리소스를 자동으로 해제할 수 없습니다. 가상 머신 인스턴스가 언로드될 때 메인 스레드를 유일한 실행 스레드로 제한함으로써 임의의 스레드가 보유한 시스템 리소스를 해제하는 부담은 프로그래머에게 있습니다.

DestroyJavaVM에 대한 지원은 JDK 릴리스 1.1에서 완전하지 않았습니다. 주 스레드만 DestroyJavaVM을 호출할 수 있습니다. 가상 머신 구현은 기본 스레드가 유일한 사용자 수준 스레드가 될 때까지 차단되고 음수 오류 코드를 반환합니다.

Java 2 SDK 릴리스 1.2는 여전히 가상 머신 인스턴스 언로드를 지원하지 않습니다. 그러나 DestroyJavaVM 사용에는 약간의 완화가 있습니다. 모든 스레드는 DestroyJavaVM을 호출할 수 있습니다.

오류 코드를 반환하기 전에 현재 스레드가 유일한 사용자 스레드가 될 때까지 가상 머신 구현이 차단됩니다.

결합

JavaVM 인터페이스 기능 테이블의 인덱스 3.

매개변수

vm: 제거될 가상 머신 인스턴스.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수를 반환합니다.

예외

없음.

JNI 함수

현재 스레드 분리

현재 스레드 분리

원기

`jint DetachCurrentThread(JavaVM *vm);`

설명

가상 머신 인스턴스에서 현재 스레드를 분리합니다. 이 스레드가 보유한 모든 모니터가 해제됩니다. 이 스레드가 죽기를 기다리는(즉, 이 스레드에서 `Thread.join` 수행) 모든 스레드에 알림이 전송됩니다.

JDK 릴리스 1.1에서는 기본 스레드를 가상 머신 인스턴스에서 분리할 수 없습니다. 대신 `DestroyJavaVM`을 호출하여 전체 가상 머신 인스턴스를 언로드해야 합니다.

Java 2 SDK 릴리스 1.2에서는 기본 스레드가 가상 머신 인스턴스에서 분리될 수 있습니다.

결합

JavaVM 인터페이스 기능 테이블의 인덱스 5.

매개변수

`vm`: 현재 스레드가 분리될 가상 머신 인스턴스입니다.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수를 반환합니다.

예외

없음.

로컬 용량 보장

JNI 함수

로컬 용량 보장

원기

`jint ConfirmLocalCapacity(JNIEnv *env, jint 용량);`

설명

현재 스레드에서 적어도 지정된 수의 로컬 참조를 만들 수 있는지 확인합니다.

네이티브 메서드에 들어가기 전에 가상 머신 구현은 현재 스레드에서 최소 16개의 로컬 참조를 생성할 수 있는지 확인합니다.

보장된 용량보다 더 많은 로컬 참조를 할당하면 가상 머신 구현에 사용 가능한 메모리가 충분한지 여부에 따라 즉각적인 실패로 이어질 수도 있고 그렇지 않을 수도 있습니다. 가상 머신 구현은 보장된 용량을 초과하는 추가 로컬 참조를 위한 메모리를 제공할 수 없는 경우 `FatalError`를 호출합니다.

디버깅 지원을 위해 가상 머신 구현은 사용자가 보장된 용량보다 더 많은 로컬 참조를 생성할 때 사용자에게 경고를 줄 수 있습니다. Java 2 SDK 릴리스 1.2에서 프로그래머는 `-verbose:jni` 명령줄 옵션을 제공하여 이러한 경고 메시지를 볼 수 있습니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 26 .

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`용량`: 생성될 로컬 참조 수.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수를 반환합니다.
예외를 throw합니다.

예외

`OutOfMemoryError`: 시스템의 메모리가 부족한 경우.

JNI 함수

예외 확인

예외 확인

원기

`jboolean ExceptionCheck(JNIEnv *env);`

설명

예외가 발생했는지 확인합니다. 네이티브 코드가 `ExceptionClear`를 호출하거나 네이티브 메서드 호출자가 예외를 처리할 때까지 예외는 계속 `throw`됩니다.

이 함수와 `ExceptionOccurred`의 차이점은 이 함수는 보류 중인 예외가 있는지 여부를 나타내는 `jboolean`을 반환하는 반면 `ExceptionOccurred`는 보류 중인 예외에 대한 로컬 참조를 반환하거나 보류 중인 예외가 없는 경우 `NULL`을 반환한다는 것입니다 .

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 228 .

매개변수

`env : JNIEnv` 인터페이스 포인터.

반환 값 보류 중인 예외가 있으면 `JNI_TRUE`를 반환하고 보류 중인 예외가 없으면 `JNI_FALSE`를 반환합니다.

예외

없음.

예외지우기	JNI 함수
원기	무효 ExceptionClear(JNIEnv *env);
설명	현재 스레드에서 현재 throw되고 있는 보류 중인 예외를 지웁니다. 현재 예외가 발생하지 않으면 이 함수는 아무 효과가 없습니다. 이 함수는 다른 스레드에서 보류 중인 예외에는 영향을 미치지 않습니다.
	ExceptionDescribe 에는 보류 중인 예외를 지우는 부작용도 있습니다.
결합	JNIEnv 인터페이스 함수 테이블 의 인덱스 17 .
매개변수	env : JNIEnv 인터페이스 포인터.
예외	없음.

JNI 함수

예외 설명

예외 설명

원기

무효 예외 설명(JNIEnv *env);

설명

보류 중인 예외와 스택의 역추적을 시스템 오류 보고 채널 System.out.err에 인쇄합니다. 디버깅을 위해 제공되는 편의 루틴입니다.

이 함수는 보류 중인 예외를 지우는 부작용이 있습니다.

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스 16 .

매개변수

env : JNIEnv 인터페이스 포인터.

예외

없음.

예외 발생	JNI 함수
예외 발생	
원기	jthrowable ExceptionOccurred(JNIEnv *env);
설명	현재 스레드에서 예외가 보류 중인지 확인합니다. 네이티브 코드가 ExceptionClear를 호출하거나 네이티브 메서드 호출자가 예외를 처리할 때 까지 예외는 보류 상태로 유지됩니다.
	이 함수와 ExceptionCheck (Java 2 SDK 릴리스 1.2에 추가됨)의 차이점은 ExceptionCheck는 보류 중인 예외가 있는지 여부를 나타내는 jboolean을 반환하는 반면 이 함수는 보류 중인 예외에 대한 로컬 참조를 반환하거나 없는 경우 NULL을 반환한다는 것입니다. 대기 중인 예외.
결합	JNIEnv 인터페이스 함수 테이블의 인덱스 15 .
매개변수	env : JNIEnv 인터페이스 포인터.
	반환 값 현재 스레드에서 보류 중인 예외 개체를 반환하거나 보류 중인 예외가 없으면 NULL을 반환합니다.
예외	없음.

JNI 함수

치명적 오류

치명적 오류

원기

```
무효 FatalError(JNIEnv *env, const char *msg);
```

설명

치명적인 오류를 발생시키고 가상 머신 구현이 복구될 것으로 기대하지 않습니다. stderr 와 같은 시스템 디버깅 채널에 메시지를 인쇄하고 가상 머신 인스턴스를 종료합니다.

이 함수는 반환하지 않습니다.

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스 18 .

매개변수

env : JNIEnv 인터페이스 포인터.

msg: 오류 메시지.

예외

없음.

클래스 찾기

JNI 함수

클래스 찾기

원기

`jclass FindClass(JNIEnv *env, const char *name);`

설명

명명된 클래스 또는 인터페이스에 대한 참조를 반환합니다. 이 기능은 JDK 릴리스 1.1에서 도입되었으며 Java 2 SDK 릴리스 1.2에서 확장되었습니다. JDK 릴리스 1.1에서 이 함수는 로컬로 정의된 클래스 또는 인터페이스를 로드합니다. 지정된 이름을 가진 클래스 또는 인터페이스에 대해 CLASSPATH 환경 변수로 지정된 디렉토리 및 zip 파일을 검색합니다.

Java 2 SDK 릴리스 1.2에서 `FindClass`는 현재 원시 메소드와 연관된 클래스 로더를 찾습니다. 네이티브 코드가 null 로더에 속하는 경우 부트스트랩 클래스 로더를 사용하여 명명된 클래스 또는 인터페이스를 로드합니다. 그렇지 않으면 해당 클래스 로더에서 `ClassLoader.loadClass` 메서드를 호출하여 명명된 클래스 또는 인터페이스를 로드합니다.

`FindClass`는 반환하는 클래스 또는 인터페이스를 초기화합니다.

`name` 인수는 클래스 설명자(§ 12.3.2)입니다. 예를 들어 `java.lang.String` 클래스의 설명자는 다음과 같습니다.

"자바/언어/문자열"

배열 클래스 `java.lang.Object[]`의 설명자는 다음과 같습니다.

"[Ljava/lang/객체;"

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 6.

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

이름: 반환할 클래스 또는 인터페이스의 설명자.

반환 값 명명 된 클래스 또는 인터페이스에 대한 로컬 참조를 반환하거나 클래스 또는 인터페이스를 로드할 수 없는 경우 NULL을 반환 합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

[JNI 함수](#)[클래스 찾기](#)

예외

ClassFormatError: 클래스 데이터가 유효한 클래스 또는 인터페이스를 지정하지 않은 경우.

ClassCircularityError: 클래스 또는 인터페이스가 자체 슈퍼클래스 또는 슈퍼인터페이스인 경우.

NoClassDefFoundError: 요청된 클래스 또는 인터페이스에 대한 정의를 찾을 수 없는 경우.

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

ExceptionInInitializerError: 클래스 또는 인터페이스 초기화에 실패한 경우.

FromReflected필드	JNI 함수
FromReflected필드	
원기	jfieldID FromReflectedField(JNIEnv *env, jobject 필드);
설명	java.lang.reflect.Field 인스턴스를 필드 ID로 변환합니다 . 이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.
결합	JNIEnv 인터페이스 함수 테이블 의 인덱스 8 .
매개변수	env : JNIEnv 인터페이스 포인터. 필드: java.lang.reflect.Field 인스턴스 에 대한 참조입니다 .
반환 값	java.lang.reflect.Field 의 주어진 인스턴스에 해당하는 필드 ID를 반환하거나 예외 가 발생 하면 NULL을 반환 합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다 .
예외	OutOfMemoryError: 시스템의 메모리가 부족한 경우.

JNI 함수

FromReflected 메서드

FromReflected 메서드

원기

```
jmethodID FromReflectedMethod(JNIEnv *env,
    작업객체 방법);
```

설명

`java.lang.reflect.Method` 인스턴스 또는 `java.lang.reflect.Constructor` 인스턴스를 메소드 ID로 변환합니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 7.

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

메소드: `java.lang.reflect.Method` 객체에 대한 참조 또는 `java.lang.reflect.Constructor` 객체에 대한 참조.

반환 값 `java.lang.reflect.Method` 클래스의 지정된 인스턴스 또는 `java.lang.reflect.Constructor` 클래스의 지정된 인스턴스에 해당하는 메서드 ID를 반환하거나 예외가 발생하면 `NULL`을 반환합니다. 이 함수의 호출이 예외를 `throw`한 경우에만 `NULL`을 반환합니다.

예외

`OutOfMemoryError`: 시스템의 메모리가 부족한 경우.

GetArrayLength	JNI 함수
----------------	--------

GetArrayLength

원기	jsize GetArrayLength(JNIEnv *env, jarray 배열);
----	---

설명	지정된 배열의 요소 수를 반환합니다. 배열 인수는 int 또는 double 과 같은 기본 유형 또는 java.lang.Object 의 하위 클래스 또는 기타 배열 유형 과 같은 참조 유형을 포함하여 모든 요소 유형의 배열을 나타낼 수 있습니다.
----	--

결합	JNIEnv 인터페이스 함수 테이블 의 인덱스 171 .
----	---------------------------------

매개변수	env : JNIEnv 인터페이스 포인터.
------	-------------------------

	배열: 길이를 결정할 배열 개체에 대한 참조입니다.
--	------------------------------

반환 값	배열의 길이를 반환합니다.
------	----------------

예외	없음.
----	-----

JNI 함수

Get<Type>ArrayElements

Get<Type>ArrayElements

원기

```
<NativeType> *Get<Type>ArrayElements(JNIEnv *env,
<ArrayType> 배열, jboolean *isCopy);
```

양식

이 함수 계열은 8개의 멤버로 구성됩니다.

Get<Type>ArrayElements	<배열 유형>	<네이티브 유형>
GetBooleanArrayElements	jbooleanArray	jboolean
GetByteArrayElements	jbyteArray	제이바이트
GetCharArray요소	jcharArray	jchar
GetShortArrayElements	jshortArray	jshort
GetIntArrayElements	진트어레이	진트
GetLongArrayElements	jlongArray	제롱
GetFloatArrayElements	jfloatArray	jfloat
GetDoubleArrayElements	jdoubleArray	제이더블

설명

기본 배열의 본문을 반환합니다. 결과는 해당 Release<Type>ArrayElements 함수가 호출될 때까지 유효합니다. 반환된 배열은 원래 배열의 복사본일 수 있으므로 반환된 배열에 대한 변경 사항은 해당 Release<Type>ArrayElements 가 호출될 때까지 원래 배열에 반드시 반영되지는 않습니다.

isCopy가 NULL 이 아닌 경우 복사본이 만들어지면 * isCopy가 JNI_TRUE 로 설정됩니다 . 복사본이 없으면 JNI_FALSE로 설정됩니다.

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스입니다 .

Get<Type>ArrayElements	색인
GetBooleanArrayElements	183
GetByteArrayElements	184
GetCharArray요소	185
GetShortArrayElements	186
GetIntArrayElements	187
GetLongArrayElements	188
GetFloatArrayElements	189
GetDoubleArrayElements	190

Get<Type>ArrayElements

JNI 함수

매개변수

env : JNIEnv 인터페이스 포인터.

배열: 액세스할 요소가 있는 기본 배열에 대한 참조입니다.

isCopy: 함수가 배열 요소의 복사본에 대한 포인터 또는 원래 배열 요소에 대한 직접 포인터를 반환했는지 여부를 나타내는 jboolean에 대한 포인터입니다.

반환 값 배열 요소에 대한 포인터를 반환하거나 예외가 발생하면 NULL을 반환합니다.
이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

JNI 함수

Get<Type>ArrayRegion

Get<Type>ArrayRegion

원기

```
무효 Get<Type>ArrayRegion(JNIEnv *env,
<ArrayType> 배열, jsize 시작, jsize len,
<NativeType> *buf);
```

양식

이 함수 계열은 8개의 멤버로 구성됩니다.

Get<Type>ArrayRegion	<배열 유형>	<네이티브 유형>
GetBooleanArrayRegion	jbooleanArray	jboolean
GetByteArrayRegion	jbyteArray	제이바이트
GetCharArray영역	jcharArray	jchar
GetShortArrayRegion	jshortArray	조트
GetIntArray영역	진트어레이	진트
GetLongArrayRegion	jlongArray	제롱
GetFloatArrayRegion	jfloatArray	jfloat
GetDoubleArrayRegion	jdoubleArray	제이더블

설명

기본 배열의 영역을 버퍼에 복사합니다. 배열 참조 및 buf 버퍼는 NULL 이 아니어야 합니다 .

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스입니다 .

Get<Type>ArrayRegion	색인
GetBooleanArrayRegion	199
GetByteArrayRegion	200
GetCharArray영역	201
GetShortArrayRegion	202
GetIntArray영역	203
GetLongArrayRegion	204
GetFloatArrayRegion	205
GetDoubleArrayRegion	206

Get<Type>ArrayRegion

JNI 함수

매개변수

env: JNIEnv 인터페이스 포인터.

배열: 복사할 요소가 있는 배열에 대한 참조입니다.

시작: 복사할 배열 요소의 시작 인덱스입니다.

len: 복사할 요소의 수.

buf: 대상 버퍼.

예외

ArrayIndexOutOfBoundsException: 영역의 인덱스 중 하나가 유효하지 않은 경우.

JNI 함수

Get<Type>필드

Get<Type>필드

원기

```
<NativeType> Get<Type>Field(JNIEnv *env, jobject obj,
jfieldID fieldID);
```

양식

이 함수군은 9개의 멤버로 구성됩니다.

Get<Type>필드	<네이티브 유형>
GetObjectField	직업
GetBoolean필드	jboolean
GetByteField	제이바이트
GetCharField	jchar
GetShortField	jshort
GetIntField	진트
GetLongField	jlong
GetFloat필드	jfloat
GetDoubleField	제이더블

설명

인스턴스의 필드 값을 반환합니다. 액세스할 필드는 필드 ID로 지정됩니다. 필드 ID는 obj 참조의 클래스에서 유효해야 합니다. obj 참조는 NULL 이 아니어야 합니다 .

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다 .

Get<Type>필드	색인
GetObjectField	95
GetBoolean필드	96
GetByteField	97
GetCharField	98
GetShortField	99
GetIntField	100
GetLongField	101
GetFloat필드	102
GetDoubleField	103

Get<Type>필드 JNI 함수

매개변수 env : JNIEnv 인터페이스 포인터.

obj: 액세스할 필드가 있는 인스턴스에 대한 참조입니다.

fieldID: 주어진 인스턴스의 필드 ID.

반환 값 필드의 값을 반환합니다.

예외 없음.

JNI 함수

GetEnv

GetEnv

원기

```
jint GetEnv(JavaVM *vm, void **penv, jint
interface_id);
```

설명

현재 스레드가 지정된 가상 머신 인스턴스에 연결되어 있지 않으면 *penv를 NULL로 설정하고 JNI_EDETACHED를 반환합니다. 지정된 인터페이스가 지원되지 않으면 *penv를 NULL로 설정하고 JNI_EVERSION을 반환합니다. 그렇지 않으면 *env를 적절한 인터페이스로 설정하고 JNI_OK를 반환합니다.

Java 2 SDK 릴리스 1.2는 JNI_VERSION_1_1 및 JNI_VERSION_1_2의 두 가지 유효한 인터페이스 버전을 지원합니다. 두 경우 모두 GetEnv는 *penv를 JNIEnv 인터페이스 포인터의 1.2 버전으로 설정합니다.

이 기능은 Java 2 SDK 릴리스 1.2에 추가되었습니다.

결합

JavaVM 인터페이스 기능 테이블의 인덱스 6 .

매개변수

vm: 가상 머신 인스턴스.

penv: 인터페이스 포인터를 저장하기 위한 위치.

interface_id: 인터페이스 버전 번호.

반환 값 성공 시 JNI_OK, 현재 스레드가 연결되지 않은 경우 JNI_EDETACHED, 지정된 인터페이스가 지원되지 않는 경우 JNI_EVERSION을 반환합니다.

예외

없음.

	GetFieldID	JNI 함수
	GetFieldID	
원기	jfieldID GetFieldID(JNIEnv *env, jclass clazz, const char *name, const char *sig);	
설명		클래스의 인스턴스 필드에 대한 필드 ID를 반환합니다. 필드는 이름과 설명자로 지정됩니다. 접근자 함수의 Get<Type>Field 및 Set<Type>Field 계열은 필드 ID를 사용하여 인스턴스 필드를 검색 합니다 . 이 필드는 clazz가 참조하는 클래스에서 액세스할 수 있어야 합니다 . 그러나 실제 필드는 clazz의 상위 클래스 중 하나에서 정의될 수 있습니다. clazz 참조는 NULL 이 아니어야 합니다 .
		GetFieldID로 인해 초기화되지 않은 클래스가 초기화됩니다.
		GetFieldID는 배열의 길이를 가져오는 데 사용할 수 없습니다. 대신 GetArrayLength를 사용하십시오 .
결합		JNIEnv 인터페이스 함수 테이블의 인덱스 94 .
매개변수	env : JNIEnv 인터페이스 포인터. clazz: 필드 ID가 파생될 클래스 개체에 대한 참조입니다.	
	이름: 0으로 끝나는 UTF-8 문자열의 필드 이름입니다. sig: 0으로 끝나는 UTF-8 문자열의 필드 설명자.	
		반환 값 작업이 실패하면 필드 ID 또는 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다 .
예외	NoSuchFieldError: 지정된 필드를 찾을 수 없는 경우. ExceptionInInitializerError: 클래스 이니셜라이저가 예외로 인해 실패하는 경우. OutOfMemoryError: 시스템의 메모리가 부족한 경우.	

JNI 함수

GetJavaVM

GetJavaVM

원기

```
jint GetJavaVM(JNIEnv *env, JavaVM **vm);
```

설명

현재 스레드가 연결된 JavaVM 인터페이스 포인터를 반환합니다 .
결과는 두 번째 인수가 가리키는 위치에 배치됩니다.

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스 219 .

매개변수

env : JNIEnv 인터페이스 포인터.

vm: 결과가 배치되어야 하는 위치에 대한 포인터입니다.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수 값을 반환합니다.

이 함수의 호출이 예외를 throw한 경우에만 음수를 반환합니다.

예외

없음.

GetMethodID	JNI 함수
-------------	--------

GetMethodID

원기	<pre>jmethodID GetMethodID(JNIEnv *env, jclass clazz, const char *name, const char *sig);</pre>
설명	클래스 또는 인터페이스의 인스턴스 메서드에 대한 메서드 ID를 반환합니다. 메서드는 clazz의 슈퍼클래스 또는 슈퍼인터페이스 중 하나에서 정의되고 clazz에 의해 상속될 수 있습니다. 메서드는 이름과 설명자로 결정됩니다. clazz 참조는 NULL이 아니어야 합니다.
<p>GetMethodID는 초기화되지 않은 클래스 또는 인터페이스가 초기화되도록 합니다.</p>	
<p>생성자의 메소드 ID를 얻으려면 메소드 이름으로 "<init>"를 제공하고 리턴 유형으로 "V"를 제공하십시오. 예를 들어 다음 코드 세그먼트는 String(char []) 생성자에 대한 메서드 ID를 가져옵니다.</p>	
<pre>jmethodID cid = (*env)->GetMethodID(env, Class_java_lang_String, "<init>", "([C)V");</pre>	
결합	JNIEnv 인터페이스 함수 테이블의 인덱스 33.
매개변수	<p>env : JNIEnv 인터페이스 포인터.</p> <p>clazz: 메서드 ID가 파생될 클래스 또는 인터페이스 개체에 대한 참조입니다.</p>
<p>이름: 0으로 끝나는 UTF-8 문자열의 메서드 이름입니다.</p>	
<p>sig: 0으로 끝나는 UTF-8 문자열의 메소드 설명자.</p>	
<p>반환 값 메서드 ID를 반환하거나 작업이 실패하면 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.</p>	
예외	<p>NoSuchMethodError: 지정된 메소드를 찾을 수 없는 경우.</p> <p>ExceptionInInitializerError: 클래스 또는 인터페이스 정적 초기화 프로그램이 예외로 인해 실패하는 경우.</p> <p>OutOfMemoryError: 시스템의 메모리가 부족한 경우.</p>

JNI 함수

GetObjectArrayElement

GetObjectArrayElement

원기

```
jobject GetObjectArrayElement(JNIEnv *env, jobjectArray 배열, jsize 인덱스);
```

설명

java.lang.Object 배열의 요소를 반환합니다. 배열 참조는 NULL이 아니어야 합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 173.

매개변수

env : JNIEnv 인터페이스 포인터.

array: 요소에 액세스할 java.lang.Object 배열에 대한 참조입니다.

인덱스: 배열 인덱스.

반환 값 요소에 대한 로컬 참조를 반환합니다.

예외

ArrayIndexOutOfBoundsException: 인덱스가 배열에서 유효한 인덱스를 지정하지 않는 경우.

GetObjectClass	JNI 함수
----------------	--------

GetObjectClass

원기 `jclass GetObjectClass(JNIEnv *env, jobject obj);`

설명 객체의 클래스를 반환합니다. `obj` 참조는 NULL이 아니어야 합니다.

결합 `JNIEnv` 인터페이스 함수 테이블의 인덱스 31.

매개변수 `env`: `JNIEnv` 인터페이스 포인터.

`obj`: 클래스를 가져올 개체에 대한 참조입니다.

반환 값 주어진 개체의 클래스에 대한 로컬 참조를 반환합니다.

예외 없음.

JNI 함수

GetPrimitiveArrayCritical

GetPrimitiveArrayCritical

원기

```
void * GetPrimitiveArrayCritical(JNIEnv *env, jarray 배열, jboolean
                                *isCopy);
```

설명

기본 배열의 본문에 대한 포인터를 반환합니다. 결과는 해당 ReleasePrimitiveArray Critical 함수가 호출될 때까지 유효합니다.

이 함수 쌍 내부의 코드를 "중요 영역"에서 실행되는 것으로 취급하는 것이 중요합니다. 임계 영역 내에서 네이티브 코드는 다른 JNI 함수를 호출해서는 안 되며, 네이티브 코드는 현재 스레드가 차단되고 가상 머신 인스턴스에서 다른 스레드를 기다리게 할 수 있는 시스템 호출을 수행해서는 안 됩니다.

이러한 제한으로 인해 가상 머신 구현이 고정을 지원하지 않는 경우에도 네이티브 코드가 복사되지 않은 버전의 어레이를 얻을 가능성이 높아집니다. 예를 들어 가상 머신 구현은 기본 코드가 GetPrimitiveArrayCritical을 통해 얻은 배열에 대한 포인터를 보유하고 있을 때 일시적으로 가비지 수집을 비활성화할 수 있습니다.

여러 쌍의 Get/ReleasePrimitiveArrayCritical 호출이 겹칠 수 있습니다.

```
jint len = (*env)->GetArrayLength(env, arr1); jbyte *a1 = (*env)->
GetPrimitiveArrayCritical(env, arr1, 0); if (a1 == NULL) { ... /* 메모리
부족 오류 */ }

}

jbyte *a2 = (*env)->
GetPrimitiveArrayCritical(env, arr2, 0); if (a2 == NULL) { ... /
* 메모리 부족 오류 */

}

memcpy(a1, a2, len); (*env)->ReleasePrimitiveArrayCritical(env, arr2, a2, 0); (*env)->ReleasePrimitiveArrayCritical(env, arr1, a1, 0);
```

GetPrimitiveArrayCritical은 가상 머신 구현이 내부적으로 다른 형식의 어레이를 나타내는 경우(비연속적으로, 예를 들어

GetPrimitiveArrayCritical

JNI 함수

플레). 따라서 가능한 메모리 부족 상황에 대해 반환 값을 NULL에 대해 확인하는 것이 중요합니다.

isCopy가 NULL이 아닌 경우 복사본이 만들어지면 * isCopy가 JNI_TRUE로 설정됩니다. 또는 복사본이 만들어지지 않으면 JNI_FALSE로 설정됩니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 222.

매개변수

env : JNIEnv 인터페이스 포인터.

배열: 액세스할 요소가 있는 배열에 대한 참조입니다.

isCopy: jboolean에 대한 포인터.

반환 값 배열 요소에 대한 포인터를 반환하거나 작업이 실패하면 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

JNI 함수

GetStaticFieldID

GetStaticFieldID

원기

```
jfieldID GetStaticFieldID(JNIEnv *env, jclass clazz,
                           const char *name, const char *sig);
```

설명

클래스 또는 인터페이스의 정적 필드에 대한 필드 ID를 반환합니다. 이 필드는 clazz 가 참조하는 클래스나 인터페이스 또는 해당 슈퍼클래스나 슈퍼인터페이스 중 하나에서 정의될 수 있습니다. clazz 참조는 NULL 이 아니어야 합니다. 필드는 이름과 설명자로 지정됩니다. 접근자 함수의 GetStatic<Type>Field 및 Set Static<Type>Field 패밀리는 정적 필드 ID를 사용하여 정적 필드를 검색합니다.

GetStaticFieldID는 초기화되지 않은 클래스 또는 인터페이스가 초기화되도록 합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 144.

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 액세스할 정적 필드가 있는 클래스 또는 인터페이스 개체에 대한 참조입니다.

이름: 0으로 끝나는 UTF-8 문자열의 정적 필드 이름입니다.

sig: 0으로 끝나는 UTF-8 문자열의 필드 설명자.

반환 값 작업이 실패하면 필드 ID 또는 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

NoSuchFieldError: 지정된 정적 필드를 찾을 수 없는 경우.

ExceptionInInitializerError: 클래스 또는 인터페이스 정적 초기화 프로그램이 예외로 인해 실패하는 경우.

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

GetStatic<Type>필드

JNI 함수

GetStatic<Type>필드

원기

```
<NativeType> GetStatic<Type>Field(JNIEnv *env, jclass clazz,
jfieldID fieldID);
```

양식

이 함수군은 9개의 멤버로 구성됩니다.

GetStatic<Type>필드	<네이티브 유형>
GetStaticObjectField	직업
GetStaticBoolean필드	jboolean
GetStaticByteField	제이바이트
GetStaticCharField	jchar
GetStaticShortField	jshort
GetStaticIntField	진트
GetStaticLongField	jlong
GetStaticFloat필드	jfloat
GetStaticDoubleField	제이더블

설명

이 접근자 루틴 계열은 클래스 또는 인터페이스의 정적 필드 값을 반환합니다. clazz 참조는 NULL 이 아니어야 합니다 . 액세스할 필드는 GetStaticFieldID를 호출하여 얻은 필드 ID로 지정됩니다 .

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스입니다 .

GetStatic<Type>필드	색인
GetStaticObjectField	145
GetStaticBoolean필드	146
GetStaticByteField	147
GetStaticCharField	148
GetStaticShortField	149
GetStaticIntField	150
GetStaticLongField	151
GetStaticFloat필드	152
GetStaticDoubleField	153

JNI 함수

GetStatic<Type>필드

매개변수

env: JNIEnv 인터페이스 포인터.

clazz: 액세스할 정적 필드가 있는 클래스 또는 인터페이스 개체에 대한 참조입니다.

fieldID: 액세스할 정적 필드의 ID입니다.

반환 값 정적 필드의 값을 반환합니다.

예외

없음.

GetStaticMethodID	JNI 함수
-------------------	--------

GetStaticMethodID

원기

```
jmethodID GetStaticMethodID(JNIEnv *env, jclass clazz,
                           const char *name, const char *sig);
```

설명

클래스의 정적 메서드에 대한 메서드 ID를 반환합니다. 메서드는 이름과 설명자로 지정됩니다. 메서드는 clazz 가 참조하는 클래스 또는 해당 슈퍼 클래스 중 하나에서 정의할 수 있습니다. clazz 참조는 NULL 이 아니어야 합니다 .

GetStaticMethodID는 초기화되지 않은 클래스를 초기화합니다.

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스 113 .

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 정적 메서드를 호출할 클래스 개체에 대한 참조입니다.

이름: 0으로 끝나는 UTF-8 문자열의 정적 메서드 이름입니다.

sig: 0으로 끝나는 UTF-8 문자열의 메소드 설명자.

반환 값

메서드 ID를 반환하거나 작업이 실패하면 NULL을 반환 합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다 .

예외

NoSuchMethodError: 지정된 정적 메서드를 찾을 수 없는 경우.

ExceptionInInitializerError: 클래스 아나셀라이저가 예외로 인해 실패하는 경우.

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

JNI 함수

GetStringChars

GetStringChars

원기

```
const jchar * GetStringChars(JNIEnv *env, jstring 문자열,
jboolean *isCopy);
```

설명

문자열의 유니코드 문자 배열에 대한 포인터를 반환합니다. 이 포인터는 `ReleaseStringChars`가 호출될 때까지 유효합니다.

`isCopy`가 NULL 이 아닌 경우 복사본이 만들어지면 * `isCopy`가 JNI_TRUE 로 설정됩니다. 또는 복사본이 만들어지지 않으면 JNI_FALSE 로 설정됩니다.

결합

`JNIEnv` 인터페이스 함수 테이블 의 인덱스 165 .

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

문자열: 액세스할 요소가 있는 문자열 개체에 대한 참조입니다.

`isCopy`: 문자열의 복사본이 반환되는지 여부를 나타내는 `jboolean` 에 대한 포인터입니다.

반환 값 유니코드 문자열에 대한 포인터를 반환하거나 작업이 실패하면 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

`OutOfMemoryError`: 시스템의 메모리가 부족한 경우.

	GetStringCritical	JNI 함수
	GetStringCritical	
원기	const jchar * GetStringCritical(JNIEnv *env, jstring 문자열, jboolean *isCopy);	
설명	jstring 참조의 내용에 대한 포인터를 반환합니다. 이 함수의 의미 체계는 GetStringChars 함수와 유사합니다. 가능한 경우 가상 머신 구현은 주어진 문자열의 요소에 대한 포인터를 반환합니다. 그렇지 않으면 복사본이 만들어집니다. 포인터는 ReleaseStringCritical이 호출될 때까지 유효합니다.	
	isCopy가 NULL이 아닌 경우 복사본이 만들어지면 * isCopy가 JNI_TRUE로 설정됩니다. 복사본이 없으면 JNI_FALSE로 설정됩니다.	
	이 함수와 해당 ReleaseStringCritical 함수를 사용하는 방법에는 상당한 제한이 있습니다. GetStringCritical 및 ReleaseStringCritical 호출로 둘러싸인 코드 세그먼트에서 네이티브 코드는 임의의 JNI 호출을 실행하거나 현재 스레드가 가상 머신 인스턴스의 다른 스레드를 차단하고 대기하도록 해서는 안 됩니다.	
	GetStringCritical의 제한 사항은 GetPrimitiveArrayCritical의 제한 사항과 동일합니다.	
	이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.	
결합	JNIEnv 인터페이스 함수 테이블의 인덱스 224.	
매개변수	env : JNIEnv 인터페이스 포인터. 문자열: 액세스할 요소가 있는 문자열 개체에 대한 참조입니다.	
	isCopy: 복사본이 만들어졌는지 여부를 나타내는 jboolean에 대한 포인터입니다.	
반환 값	유니코드 문자열에 대한 포인터를 반환하거나 작업이 실패하면 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.	
예외	OutOfMemoryError: 시스템의 메모리가 부족한 경우.	

JNI 함수

GetStringLength

GetStringLength

원기

```
jsize GetStringLength(JNIEnv *env, jstring string);
```

설명

문자열을 구성하는 유니코드 문자의 수를 반환합니다. 주어진 문자열 참조는 NULL이 아니어야 합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 164 .

매개변수

env : JNIEnv 인터페이스 포인터.

문자열: 길이를 결정할 문자열 개체에 대한 참조입니다.

반환 값 문자열의 길이를 반환합니다.

예외

없음.

GetStringRegion	JNI 함수
GetStringRegion	
원기	<pre>void GetStringRegion(JNIEnv *env, jstring str, jsize start, jsize len, jchar *buf);</pre>
설명	<p>오프셋 시작에서 시작하여 유니코드 문자의 len 번호를 복사합니다 . 지정된 버퍼 buf에 문자를 복사합니다.</p> <p>이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.</p>
결합	JNIEnv 인터페이스 함수 테이블 의 인덱스 220 .
매개변수	<p>env : JNIEnv 인터페이스 포인터.</p> <p>str: 복사할 문자열 개체에 대한 참조입니다.</p> <p>시작: 복사를 시작할 문자열 내의 오프셋.</p> <p>len: 복사할 유니코드 문자 수입니다.</p> <p>buf: 유니코드 문자를 보관할 버퍼에 대한 포인터.</p>
예외	<p>StringIndexOutOfBoundsException: 인덱스 오버플로인 경우 오류가 발생합니다.</p>

JNI 함수

GetStringUTFChars

GetStringUTFChars

원기

```
const jbyte * GetStringUTFChars(JNIEnv *env,
                                jstring 문자열, jboolean *isCopy);
```

설명

문자열의 UTF-8 문자 배열에 대한 포인터를 반환합니다.
 이 배열은 ReleaseStringUTFChars에 의해 해제될 때까지 유효합니다.

isCopy가 NULL이 아닌 경우 복사본이 만들어지면 * isCopy가 JNI_TRUE로 설정됩니다. 복사본이 없으면 JNI_FALSE로 설정됩니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 169.

매개변수

env : JNIEnv 인터페이스 포인터.

문자열: 액세스할 요소가 있는 문자열 개체에 대한 참조입니다.

isCopy: 복사본이 만들어졌는지 여부를 나타내는 jboolean에 대한 포인터입니다.

반환 값 UTF-8 문자열에 대한 포인터를 반환하거나 작업이 실패하면 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

GetStringUTF길이	JNI 함수
GetStringUTF길이	
원기	jsize GetStringUTFLength(JNIEnv *env, jstring 문자열);
설명	UTF-8 형식의 문자열을 나타내는 데 필요한 바이트 수를 반환합니다. 길이에는 후행 0 문자가 포함되지 않습니다. 주어진 문자열 참조 는 NULL이 아니어야 합니다.
결합	JNIEnv 인터페이스 함수 테이블 의 인덱스 168 .
매개변수	env : JNIEnv 인터페이스 포인터. 문자열: UTF-8 길이를 결정할 문자열 개체에 대한 참조입니다.
반환 값	문자열의 UTF-8 길이를 반환합니다.
예외	없음.

JNI 함수

GetStringUTFRegion

GetStringUTFRegion

원기

```
void GetStringUTFRegion(JNIEnv *env, jstring str, jsize 시작, jsize len,
                      char *buf);
```

설명

매트에 대해 유니코드 문자의 len 번호를 UTF-8로 변환합니다 .
 이 함수는 오프셋 시작에서 변환을 시작하고 결과를 지정된 버퍼
 buf에 배치합니다. str 참조 및 buf 버퍼는 NULL 이 아니어야 합니
 다 .

len 인수는 복사할 UTF-8 문자 수가 아니라 변환할 유니코드 문자 수를 나타냅니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 221 .

매개변수

env : JNIEnv 인터페이스 포인터.

str: 복사할 문자열 개체에 대한 참조입니다.

시작: 복사를 시작할 문자열 내의 오프셋.

len: 복사할 유니코드 문자 수입니다.

buf: UTF-8 문자를 보관할 버퍼에 대한 포인터.

예외

StringIndexOutOfBoundsException: 인덱스 오버플로인 경우
 오류가 발생합니다.

GetSuperclass	JNI 함수
---------------	--------

GetSuperclass

원기	jclass GetSuperclass(JNIEnv *env, jclass clazz);
----	--

설명	주어진 클래스의 슈퍼클래스를 반환합니다. clazz가 java.lang.Object 클래스 이외의 클래스를 나타내는 경우 이 함수는 clazz로 지정된 클래스의 수퍼 클래스에 대한 참조를 반환합니다.
----	---

clazz가 java.lang.Object 클래스를 나타내거나 clazz가 인터페이스를 나타내는 경우 이 함수는 NULL을 반환합니다.

결합	JNIEnv 인터페이스 함수 테이블의 인덱스 10.
----	------------------------------

매개변수	env : JNIEnv 인터페이스 포인터.
------	-------------------------

	clazz: 슈퍼클래스를 결정할 클래스 객체에 대한 참조.
--	----------------------------------

반환 값 clazz로 표현되는 클래스의 슈퍼 클래스 또는 NULL을 반환합니다.

예외	없음.
----	-----

JNI 함수

GetVersion

GetVersion

원기

`jint GetVersion(JNIEnv *env);`

설명

JNIEnv 인터페이스의 버전을 반환합니다. JDK 릴리스 1.1에서 GetVersion은 0x00010001을 반환합니다. Java 2 SDK 릴리스 1.2에서 GetVersion은 0x00010002를 반환합니다. JNI의 1.1 및 1.2 버전을 모두 지원하는 가상 머신 구현은 버전이 0x00010002인 하나의 JNIEnv 인터페이스만 제공합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 4.

매개변수

env : JNIEnv 인터페이스 포인터.

반환 값 JNIEnv 인터페이스의 버전을 반환합니다.

예외

없음.

IsAssignableFrom	JNI 함수
------------------	--------

IsAssignableFrom	
------------------	--

원기	jboolean IsAssignableFrom(JNIEnv *env, jclass clazz1, jclass clazz2);
----	--

설명	클래스 또는 인터페이스 clazz1 의 개체를 클래스 또는 인터페이스 clazz2 로 안전하게 캐스팅할 수 있는지 여부를 결정합니다. clazz1 과 clazz2는 모두 NULL 이 아니어야 합니다 .
----	--

결합	JNIEnv 인터페이스 함수 테이블 의 인덱스 11 .
----	--------------------------------

매개변수	env : JNIEnv 인터페이스 포인터.
------	-------------------------

	clazz1: 첫 번째 클래스 또는 인터페이스 인수.
--	-------------------------------

	clazz2: 두 번째 클래스 또는 인터페이스 인수.
--	-------------------------------

반환 값 다음 중 하나라도 참이면 JNI_TRUE를 반환합니다 .

- 첫 번째 및 두 번째 인수는 동일한 클래스를 참조하거나
상호 작용.
- 첫 번째 인수는 두 번째 인수의 하위 클래스를 참조합니다.
멘션.
- 첫 번째 인수는 인터페이스 중 하나로 두 번째 인수가 있는 클래스를 참조합니다.
- 첫 번째 인수와 두 번째 인수는 모두 요소 유형이 X 와 Y 인 배열 클래스를 참
조하고 IsAssignableFrom(env, X, Y)은 JNI_TRUE 입니다 .

그렇지 않으면 이 함수는 JNI_FALSE를 반환합니다.

예외	없음.
----	-----

JNI 함수

IsInstanceOf**IsInstanceOf**

원기

`jboolean IsInstanceOf(JNIEnv *env, jobject obj, jclass clazz);`

설명

개체가 클래스 또는 인터페이스의 인스턴스인지 여부를 테스트합니다.
`clazz` 참조는 NULL 이 아니어야 합니다.

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스 32 .

매개변수

`env`: JNIEnv 인터페이스 포인터.`obj`: 개체에 대한 참조입니다.`clazz`: 클래스 또는 인터페이스에 대한 참조.

반환 값 `obj`가 `clazz` 로 캐스팅될 수 있는 경우 , `obj`가 null 개체를 나타내는 경우 또는 `obj`가 이미 수
 집된 개체에 대한 약한 전역 참조인 경우 JNI_TRUE를 반환합니다 . 그렇지 않
 으면 JNI_FALSE를 반환합니다.

예외

없음.

IsSameObject	JNI 함수
원기	jboolean IsSameObject(JNIEnv *env, jobject ref1, jobject ref2);
설명	<p>두 참조가 동일한 개체를 참조하는지 여부를 테스트합니다. NULL 참조는 null 개체를 참조합니다.</p> <p>Java 2 SDK 릴리스 1.2에서는 이 함수를 사용하여 약한 전역 참조가 참조하는 개체가 활성 상태인지 여부를 확인할 수도 있습니다.</p>
결합	JNIEnv 인터페이스 함수 테이블의 인덱스 24.
매개변수	<p>env : JNIEnv 인터페이스 포인터.</p> <p>ref1: 개체에 대한 참조입니다.</p> <p>ref2: 개체에 대한 참조입니다.</p>
	<p>반환 값 ref1과 ref2가 동일한 개체를 참조하는 경우 JNI_TRUE를 반환합니다. 그렇지 않으면 JNI_FALSE를 반환합니다.</p> <p>Java 2 SDK 릴리스 1.2에서 약한 전역 참조 wref가 라이브 개체를 참조하는 한 IsSameObject(env, wref, NULL)는 JNI_FALSE를 반환합니다. wref가 참조하는 개체가 가비지 수집된 후 IsSameObject(env, wref, NULL) 호출은 JNI_TRUE를 반환합니다.</p>
예외	없음.

JNI 함수

JNI_CreateJavaVM

JNI_CreateJavaVM

원기

```
jint JNI_CreateJavaVM(JavaVM **pvm, void **penv, void *vm_args);
```

설명

가상 머신 인스턴스를 로드하고 초기화합니다. 가상 머신 인스턴스가 초기화되면 현재 스레드를 기본 스레드라고 합니다. 또한 이 함수는 env 인수를 기본 스레드의 JNIEnv 인터페이스 포인터로 설정합니다.

JDK 릴리스 1.1 및 Java 2 SDK 릴리스 1.2는 단일 프로세서에서 둘 이상의 가상 머신 인스턴스 생성을 지원하지 않습니다.
운.

JDK 릴리스 1.1에서 JNI_CreateJavaVM에 대한 두 번째 인수는 항상 JNIEnv 포인터의 주소입니다. 세 번째 인수는 JDK 릴리스 1.1에 특정한 초기화 구조 JDK1_1InitArgs에 대한 포인터입니다. vm_args의 버전 필드는 0x00010001로 설정되어야 합니다.

JDK1_1InitArgs 구조는 모든 가상 머신 구현에서 이식 가능하도록 설계되지 않았습니다. JDK 릴리스 1.1 구현의 요구 사항을 반영합니다.

```
typedef 구조체 JDK1_1InitArgs {
    /* 자바 VM 버전 */ jint 버전;

    /* 시스템 속성. */ 문자 **속성;

    /* 소스 파일이 컴파일된 클래스 파일보다 최신인지 확인 여부. */ 진
    트 체크소스;

    /* java.lang.Thread 스레드의 최대 기본 스택 크기.
    */ jint nativeStackSize;

    /* 최대 java.lang.Thread 스택 크기. */ 진트 javaStackSize;

    /* 초기 힙 크기. */
}
```

JNI_CreateJavaVM

JNI 함수

```

    진트 minHeapSize;

    /* 최대 힙 크기. */ 진트 maxHeapSize;

    /* 확인해야 하는 바이트 코드: * 0 -- 없음, * 1 -- 원격으로 로드된
       코드, * 2 -- 모든 코드. */ jint verifyMode;

```

```
/* 클래스 로딩을 위한 로컬 디렉토리 경로. */ const char *classpath;
```

```
/* 모든 VM 메시지를 리디렉션하는 기능에 대한 후크. */ jint
(*vfprintf)(FILE *fp, const char *format, va_list args);
```

```
/* VM 종료 후크. */ void (*exit)
(jint 코드);
```

```
/* VM 중단 후크. */ 무효 (*중단)();
```

```
/* 클래스 GC 활성화 여부. */ 진트 enableClassGC;
```

```
/* GC 메시지가 나타날지 여부. */ 진트 enableVerboseGC;
```

```
/* 비동기 GC 허용 여부. */ jint disableAsyncGC;
```

```
/* 세 개의 예약된 필드. */ 진트 예약0; 진트 예약
1; 진트 예약2; } JDK1_1InitArgs;
```

Java 2 SDK 릴리스 1.2는 JDK 릴리스 1.1과의 역호환성을 유지합니다. 초기화 인수가 JDK1_1InitArgs 구조를 가리키는 경우 JNI_CreateJavaVM은 JDK 릴리스 1.1에서와 마찬가지로 여전히 작동합니다.

JNI 함수

JNI_CreateJavaVM

또한 Java 2 SDK 릴리스 1.2에서는 Java 2 SDK 릴리스 1.2 및 향후 가상 머신 구현과 함께 작동할 일반 가상 머신 초기화 구조가 도입되었습니다.

JNI_CreateJavaVM 은 Java VMInitArgs 구조를 세 번째 인수로 받아들입니다. 고정된 옵션 집합을 포함하는 JDK1_1InitArgs 와 달리 JavaVMInitArgs 는 기호 이름/값 쌍을 사용하여 임의의 가상 머신 시작 옵션을 인코딩합니다. JavaVMInitArgs 는 다음과 같이 정의됩니다.

```
typedef 구조체 JavaVMInitArgs { 진트 버전; 진트
nOptions;

JavaVMOption *옵션; jboolean
ignoreUnrecognized;
} JavaVMInitArgs;
```

버전 필드는 JNI_VERSION_1_2로 설정되어야 합니다. (반대로 JDK1_1InitArgs 의 버전 필드는 JNI_VERSION_1_1 로 설정해야 합니다.) options 필드는 다음 유형의 배열입니다.

```
typedef 구조체 JavaVMOption { char
*optionString; 무효 *extraInfo;

} 자바VM옵션;
```

배열의 크기는 JavaVMInitArgs 의 nOptions 필드로 표시됩니다. ignoreUnrecognized 가 JNI_TRUE 인 경우 JNI_CreateJavaVM은 "-X" 또는 "_" 로 시작하는 인식되지 않은 모든 옵션 문자열을 무시합니다. ignoreUnrecognized 가 JNI_FALSE 이면 JNI_CreateJavaVM은 인식할 수 없는 옵션 문자열을 발견하는 즉시 JNI_ERR을 반환합니다. 모든 JVM(Java Virtual Machine) 구현은 다음 표준 옵션 세트를 인식해야 합니다.

JNI_CreateJavaVM

JNI 함수

optionString	의미
-D<이름>=<값>	시스템 속성 설정
-말 수가 많은	상세 출력을 활성화합니다. 이 옵션 뒤에는 콜론과 VM에서 인쇄할 메시지 종류를 나타내는 쉼표로 구분된 이름 목록이 올 수 있습니다. 예를 들어,
	-자세한 정보:gc, 클래스 GC 및 클래스 로딩 관련 메시지를 인쇄하도록 VM에 지시합니다. 표준 이름에는 gc, class 및 jni가 포함됩니다. 구현별 이름은 "X"로 시작해야 합니다.
vfprintf	extraInfo는 vfprintf 후크에 대한 포인터입니다.
출구	extraInfo는 종료 후크에 대한 포인터입니다.
종단하다	extraInfo는 중단 후크에 대한 포인터입니다.

또한 가상 머신 구현은 자체 구현 종속 문자열 세트를 지원할 수 있습니다. 구현 종속 옵션 문자열은 "-X" 또는 밑줄 ("_")로 시작해야 합니다. 예를 들어 Java 2 SDK 릴리스 1.2는 프로그래머가 초기 및 최대 힙 크기를 지정할 수 있도록 -Xms 및 -Xmx 옵션을 지원합니다. "-X"로 시작하는 옵션은 "java" 명령줄에서 지정할 수 있습니다.

다음은 Java 2 SDK 릴리스 1.2에서 가상 머신 인스턴스를 생성하는 예제 코드입니다.

JNI 함수

JNI_CreateJavaVM

```

JavaVMInitArgs vm_args;
JavaVMOption 옵션[4]; /* JIT 비활성화 */
options[0].optionString = "-Djava.compiler=없음"; /* 사용자 클래스 */
options[1].optionString = "-Djava.class.path=c:\\myclasses"; /* 네이티브 lib 경로 */
options[2].optionString = "-Djava.library.path=c:\\mylibs"; /* JNI 메시지 출력 */
options[3].optionString = "-verbose:jni"; vm_args.version = JNI_VERSION_1_2; vm_args.options = 옵션;
vm_args.nOptions = 4; vm_args.ignoreUnrecognized = TRUE;

```

```

res = JNI_CreateJavaVM(&vm, (void **)&env, &vm_args);

if (res < 0) { ... /* 오류 발생 */ }

```

결합

Java 가상 머신을 구현하는 네이티브 라이브러리에서 내보냅니다.

매개변수

pvm: 결과 JavaVM 인터페이스 포인터가 배치될 위치에 대한 포인터입니다.

penv: 기본 스레드에 대한 JNIEnv 인터페이스 포인터가 배치될 위치에 대한 포인터입니다.

args: JVM(Java Virtual Machine) 초기화 인수.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수를 반환합니다.

예외

없음.

JNI_GetCreatedJavaVM

JNI 함수

JNI_GetCreatedJavaVM

원기 `jint JNI_GetCreatedJavaVMs(JavaVM **vmBuf, jsize bufLen,
jsize *nVMs);`

설명 생성된 모든 가상 머신 인스턴스에 대한 포인터를 반환합니다. 이 함수는 가상 머신 인스턴스에 대한 포인터를 생성된 순서대로 버퍼 vmBuf 에 기록합니다. 이것해야 bufLen 항목 수를 씁니다.

마지막으로 *nVM 에서 생성된 총 가상 머신 인스턴스 수를 반환합니다.

JDK 릴리스 1.1 및 Java 2 SDK 릴리스 1.2는 단일 프로에서 둘 이상의 가상 머신 인스턴스 생성을 지원하지 않습니다.
문.

결합 Java 가상 머신을 구현하는 네이티브 라이브러리에서 내보냅니다.

매개변수 `vmBuf: 가상 머신 인스턴스에 대한 포인터가 배치될 버퍼에 대한 포인터.`

`bufLen: 버퍼의 길이.`

`nVMs: 정수에 대한 포인터.`

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수를 반환합니다.

예외 없음.

JNI 함수

JNI_GetDefaultJavaVMInitArgs

JNI_GetDefaultJavaVMInitArgs

원기

```
jint JNI_GetDefaultJavaVMInitArgs(void *vm_args);
```

설명

JVM(Java Virtual Machine) 구현을 위한 기본 구성을 반환합니다. 이 함수를 호출하기 전에 네이티브 코드는 vm_args 의 버전 필드를 0x00010001 로 설정해야 합니다 .

JDK 릴리스 1.1에서 이 함수는 JDK1_1InitArgs 구조에 대한 포인터를 인수로 취하고 성공적인 반환 시 해당 구조를 초기화합니다. 이 함수를 호출하기 전에 JDK1_1InitArgs 의 버전 필드를 0x00010001 로 설정해야 합니다 .

JNI_CreateJavaVM 함수의 사양은 JDK1_1InitArgs 구조의 내부를 설명합니다 .

Java 2 SDK 릴리스 1.2의 새로운 가상 머신 초기화 구조에서는 더 이상 프로그래머가 JNI_GetDefaultJavaVMInitArgs를 호출할 필요가 없습니다. 이 함수는 여전히 지원되지만 Java 2 SDK 릴리스 1.2에서는 더 이상 유용하지 않습니다.

결합

Java 가상 머신을 구현하는 네이티브 라이브러리에서 내보냅니다.

매개변수

vm_args: 기본 인수가 채워지는 VM 관련 초기화 구조에 대한 포인터입니다.

반환 값 요청된 버전이 지원되면 0을 반환합니다. 그렇지 않으면 요청된 버전이 지원되지 않으면 음수를 반환합니다.

예외

없음.

JNI_OnLoad	JNI 함수
------------	--------

JNI_OnLoad

원기 `jint JNI_OnLoad(JavaVM *vm, void *reserved);`

설명 지정된 네이티브 라이브러리에 대한 초기화 작업을 수행하고 네이티브 라이브러리에 필요한 JNI 버전을 반환합니다. 가상 머신 구현은 예를 들어 System.load Library에 대한 호출을 통해 네이티브 라이브러리가 로드될 때 JNI_OnLoad를 호출합니다. JNI_OnLoad는 네이티브 라이브러리에 필요한 JNIEnv 인터페이스 버전을 반환해야 합니다.

`System.loadLibrary`는 이 함수의 실행을 트리거합니다.
 JNI_OnLoad 함수는 네이티브 라이브러리에 필요한 JNI 버전 번호를 반환합니다. 네이티브 라이브러리가 JNI_OnLoad 함수를 내보내지 않는 경우 가상 머신 구현은 라이브러리에 JNI 버전 JNI_VERSION_1_1만 필요하다고 가정합니다. 가상 머신 구현이 JNI_OnLoad에서 반환된 버전 번호를 인식하지 못하는 경우 네이티브 라이브러리를 로드할 수 없습니다.

JNI_OnLoad 후크에 대한 지원이 Java 2 SDK 1.2에 추가되었습니다.

결합 네이티브 메서드 구현이 포함된 네이티브 라이브러리에서 내보냅니다.

매개변수 `vm`: 네이티브 라이브러리를 로드한 JVM(Java Virtual Machine) 인스턴스에 대한 포인터입니다.

예약됨: 현재 사용되지 않음. 이 매개변수는 NULL로 설정되며 향후 사용을 위해 예약됩니다.

반환 값 네이티브 JNIEnv 인터페이스 버전 번호를 반환합니다.
도서관 필요.

예외 없음.

JNI 함수

JNI_OnUnload

JNI_OnUnload

원기

```
void JNI_OnUnload(JavaVM *vm, void *reserved);
```

설명

기본 라이브러리에 대한 정리 작업을 수행합니다. 기본 라이브러리를 포함하는 클래스 로더가 가비지 수집될 때 가상 머신 구현은 JNI_OnUnload를 호출합니다. 이 기능을 사용하여 정리 작업을 수행할 수 있습니다. 이 함수는 알 수 없는 컨텍스트(예: 종료자)에서 호출되기 때문에 프로그래머는 Java 가상 머신 서비스를 사용할 때 신중해야 하며 임의의 JNI 함수 호출을 삼가야 합니다.

JNI_OnUnload 후크에 대한 지원이 Java 2 SDK 1.2에 추가되었습니다.

결합

네이티브 메서드 구현을 포함하는 네이티브 라이브러리에서 내보냅니다.

매개변수

vm: JVM (Java Virtual Machine) 인스턴스에 대한 포인터입니다.

예약됨: 현재 사용되지 않습니다. 이 매개변수는 NULL로 설정되며 나중에 사용할 수 있도록 예약되어 있습니다.

예외

없음.

모니터엔터

JNI 함수

모니터 입력

원기

`jint MonitorEnter(JNIEnv *env, jobject obj);`

설명

`obj` 가 참조하는 개체와 연결된 모니터를 입력합니다 .
`obj` 참조는 NULL 이 아니어야 합니다 .

각 개체에는 연결된 모니터가 있습니다. 현재 스레드가 이미 `obj` 와 연결된 모니터를 소유하고 있는 경우 이 스레드가 모니터에 들어간 횟수를 나타내는 모니터의 카운터를 증가시킵니다. `obj` 와 관련된 모니터를 어떤 스레드도 소유하고 있지 않으면 현재 스레드가 모니터의 소유자가 되어 이 모니터의 항목 수를 1로 설정합니다. 다른 스레드가 이미 `obj` 와 관련된 모니터를 소유하고 있는 경우 현재 스레드는 모니터가 해제된 다음 다시 소유권을 얻으려고 시도합니다.

MonitorEnter JNI 함수 호출을 통해 입력된 모니터는 monitorexit JVM (Java Virtual Machine) 명령어 또는 동기화된 메서드 반환을 사용하여 종료할 수 없습니다 . MonitorEnter JNI 함수 호출 및 monitoreenter JVM (Java Virtual Machine) 명령은 동일한 개체와 연결된 모니터에 들어가기 위해 경합할 수 있습니다 .

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스 217 .

매개변수

`env` : JNIEnv 인터페이스 포인터.`obj`: 관련 모니터가 입력될 개체에 대한 참조입니다.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수 값을 반환합니다.

이 함수의 호출이 예외를 throw한 경우에만 음수를 반환합니다.

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

JNI 함수

모니터 종료

모니터 종료

원기

`jint MonitorExit(JNIEnv *env, jobject obj);`

설명

지정된 개체와 연결된 모니터를 종료합니다. 현재 스레드는 obj가 참조하는 개체와 연결된 모니터의 소유자여야 합니다. obj 참조는 NULL 이 아니어야 합니다.

스레드는 이 모니터에 진입한 횟수를 나타내는 카운터를 감소시킵니다. 카운터 값이 0이 되면 현재 스레드가 모니터를 해제합니다.

기본 코드는 동기화된 메서드 또는 모니터 입력 JVM (Java Virtual Machine) 명령을 통해 입력된 모니터를 종료하기 위해 MonitorExit를 사용해서는 안 됩니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 218 .

매개변수

env : JNIEnv 인터페이스 포인터.

obj: 연결된 모니터가 종료될 개체에 대한 참조입니다.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수 값을 반환합니다.

이 함수의 호출이 예외를 throw한 경우에만 음수를 반환합니다.

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

IllegalMonitorStateException: 현재 스레드가 모니터를 소유하지 않는 경우.

NewGlobalRef

JNI 함수

NewGlobalRef

원기

`jobject NewGlobalRef(JNIEnv *env, jobject obj);`

설명

`obj` 인수 가 참조하는 개체에 대한 새 전역 참조를 만듭니다 . `obj` 인수 는 글로벌, 약한 글로벌 또는 로컬 참조일 수 있습니다. 전역 참조는 `DeleteGlobalRef` 를 호출하여 명시적으로 삭제해야 합니다 .

결합

`JNIEnv` 인터페이스 함수 테이블 의 인덱스 21 .

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`obj`: 글로벌 또는 로컬 참조.

반환 값 전역 참조를 반환합니다. 시스템 메모리가 부족하거나 주어진 인수가 NULL 이거나 주어진 참조가 이미 가비지 수집된 개체를 참조하는 약한 전역 참조인 경우 결과는 NULL 입니다.

예외

없음.

JNI 함수

NewLocalRef

NewLocalRef

원기

`jobject NewLocalRef(JNIEnv *env, jobject ref);`

설명

`ref` 와 동일한 개체를 참조하는 새 로컬 참조를 만듭니다 . 주어진 참조는 글로벌, 약한 글로벌 또는 로컬 참조일 수 있습니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 25 .

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`ref`: 함수가 새 로컬 참조를 생성하는 개체에 대한 참조입니다.

반환 값 로컬 참조를 반환합니다. 시스템 메모리가 부족하거나 주어진 인수가 NULL 이거나 주어진 참조가 이미 가비지 수집된 개체를 참조하는 약한 전역 참조인 경우 결과는 NULL 입니다.

예외

없음.

새 객체	JNI 함수
------	--------

새 객체

원기 `jobject NewObject(JNIEnv *env, jclass clazz, jmethodID methodID, ...);`

설명 새 개체를 생성합니다. 메서드 ID는 호출할 생성자 메서드를 나타냅니다. 이 ID는 메서드 이름이 "<init>"이고 반환 유형이 "V"인 GetMethodID를 호출하여 얻을 수 있습니다. 생성자는 상위 클래스 중 하나가 아니라 clazz 가 참조하는 클래스에서 정의되어야 합니다.

clazz 인수는 배열 클래스를 참조하면 안 됩니다.

프로그래머는 생성자에 전달할 모든 인수를 methodID 인수 바로 뒤에 배치합니다.

NewObject는 이러한 인수를 받아들이고 프로그래머가 호출하려는 생성자에게 전달합니다.

결합 JNIEnv 인터페이스 함수 테이블의 인덱스 28.

매개변수 env : JNIEnv 인터페이스 포인터.

clazz: 인스턴스가 생성될 클래스 객체에 대한 참조.

methodID: 새로 생성된 인스턴스에서 실행할 생성자의 메서드 ID.

추가 인수: 생성자에게 전달될 인수.

반환 값 개체에 대한 로컬 참조를 반환하거나 개체를 구성할 수 없는 경우 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외 InstantiationException: 클래스가 인터페이스 또는 추상 클래스인 경우.

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

생성자에 의해 발생한 모든 예외.

JNI 함수

NewObjectA

NewObjectA

원기

```
jobject NewObjectA(JNIEnv *env, jclass clazz,
jmethodID methodID, jvalue *args);
```

설명

새 개체를 생성합니다. 메서드 ID는 호출할 생성자 메서드를 나타냅니다. 이 ID는 메서드 이름이 "<init>"이고 반환 유형이 "V"인 GetMethodID를 호출하여 얻을 수 있습니다. 생성자는 상위 클래스 중 하나가 아니라 clazz 가 참조하는 클래스에서 정의되어야 합니다.

clazz 인수는 배열 클래스를 참조하면 안 됩니다.

프로그래머는 생성자에게 전달될 모든 인수를 methodID 인수 바로 뒤에 오는 jvalue의 args 배열에 배치합니다. NewObjectA는 이 배열의 인수를 받아들이고 프로그래머가 호출하려는 생성자에게 전달합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 30 .

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 인스턴스가 생성될 클래스 객체에 대한 참조.

methodID: 새로 생성된 인스턴스에서 실행할 생성자의 메서드 ID.

args: 생성자에 전달할 인수의 배열입니다.

반환 값 개체에 대한 로컬 참조를 반환하거나 개체를 구성할 수 없는 경우 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다 .

예외

InstantiationException: 클래스가 인터페이스 또는 추상 클래스인 경우.

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

생성자에 의해 발생한 모든 예외.

NewObjectV

JNI 함수

NewObjectV

원기

```
jobject NewObjectV(JNIEnv *env, jclass clazz,
                     jmethodID methodID, va_list 인수);
```

설명

새 개체를 생성합니다. 메서드 ID는 호출할 생성자 메서드를 나타냅니다. 이 ID는 메서드 이름이 "<init>"이고 반환 유형이 "V"인 GetMethodID를 호출하여 얻을 수 있습니다. 생성자는 상위 클래스 중 하나가 아니라 clazz 가 참조하는 클래스에서 정의되어야 합니다.

clazz 인수는 배열 클래스를 참조하면 안 됩니다.

프로그래머는 생성자에게 전달될 모든 인수를 methodID 인수 바로 뒤에 오는 va_list 유형의 args 인수에 배치합니다. NewObjectV는 이러한 인수를 받아들이고 프로그래머가 호출하려는 생성자에게 전달합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 29 .

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 인스턴스가 생성될 클래스 객체에 대한 참조.

methodID: 새로 생성된 인스턴스에서 실행할 생성자의 메서드 ID.

args: 생성자에 전달할 인수의 va_list입니다.

반환 값 개체에 대한 로컬 참조를 반환하거나 개체를 구성할 수 없는 경우 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

InstantiationException: 클래스가 인터페이스 또는 추상 클래스인 경우.

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

생성자에 의해 발생한 모든 예외.

JNI 함수

NewObjectArray

NewObjectArray

원기

```
jarray NewObjectArray(JNIEnv *env, jsize 길이, jclass elementType,
 jobject initialElement);
```

설명

클래스 또는 인터페이스 elementType에 객체를 포함하는 새 배열을 생성합니다 . 모든 요소는 초기에 initialElement 로 설정됩니다 . 길이 인수는 0일 수 있습니다. elementType 참조는 NULL이 아니어야 합니다 .

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 172 .

매개변수

env : JNIEnv 인터페이스 포인터.

길이 : 생성할 배열의 요소 수.

elementType : 배열 요소의 클래스 또는 인터페이스.

initialElement : 초기화 값 개체에 대한 참조입니다. 이 값은 NULL일 수 있습니다.

반환 값 배열 개체에 대한 로컬 참조를 반환하거나 배열을 구성할 수 없는 경우 NULL 을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL 을 반환합니다 .

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

새로운<Type>배열

JNI 함수

새로운<Type>배열

원기 `<ArrayType> New<Type>Array(JNIEnv *env,
jsize 길이);`

양식

이 함수 계열은 8개의 멤버로 구성됩니다.

새로운<Type>배열	<배열 유형>
NewBooleanArray	jbooleanArray
NewByteArray	jbyteArray
NewCharArray	jcharArray
NewShortArray	jshortArray
NewIntArray	진트어레이
NewLongArray	jlong배열 jfloat배
NewFloatArray	열
NewDoubleArray	jdoubleArray

설명

기본 요소 유형의 새 배열을 구성합니다. 새로 구성된 배열의 모든 요소는 0으로 초기화됩니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다 .

새로운<Type>배열	색인
NewBooleanArray	175
NewByteArray	176
NewCharArray	177
NewShortArray	178
NewIntArray	179
NewLongArray	180
NewFloatArray	181
NewDoubleArray	182

매개변수

env : JNIEnv 인터페이스 포인터.

길이: 생성할 배열의 요소 수.

JNI 함수

새로운<Type>배열

반환 값 기본 배열에 대한 로컬 참조를 반환하거나 배열을 구성할 수 없는 경우 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

새 문자열

JNI 함수

새 문자열

원기

```
jstring NewString(JNIEnv *env,
                   const jchar *uchars, jsize len);
```

설명

지정된 유니코드 문자에서 java.lang.String 객체를 생성합니다 .

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스 163 .

매개변수

env : JNIEnv 인터페이스 포인터.

uchars: 문자열을 구성하는 유니코드 시퀀스에 대한 포인터.

len: 유니코드 문자열의 길이.

반환 값 문자열 개체에 대한 로컬 참조를 반환하거나 문자열을 구성할 수 없는 경우 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다 .

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

JNI 함수

NewStringUTF

NewStringUTF

원기

```
jstring NewStringUTF(JNIEnv *env, const char  
*bytes);
```

설명

UTF-8 문자 배열에서 새 java.lang.String 객체를 생성합니다. 바이트로 가리키는
UTF-8 문자는 0으로 끝납니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 167 .

매개변수

env: JNIEnv 인터페이스 포인터 또는 문자열을 구성할 수 없는 경우
NULL입니다 .

바이트: 문자열을 구성하는 UTF-8 문자 시퀀스에 대한 포인터입니다.

반환 값 문자열 개체에 대한 로컬 참조를 반환하거나 문자열을 구성할 수 없는 경우
NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만
NULL을 반환합니다 .

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

NewWeakGlobalRef

JNI 함수

NewWeakGlobalRef

원기

```
jweak NewWeakGlobalRef(JNIEnv *env, jobject obj);
```

설명

`obj` 가 참조하는 개체에 대한 새로운 약한 전역 참조를 만듭니다. 약한 전역 참조는 특별한 종류의 전역 참조입니다. 일반 전역 참조와 달리 약한 전역 참조는 기본 개체가 가비지 수집될 수 있도록 합니다. 전역 또는 로컬 참조를 사용하는 모든 상황에서 약한 전역 참조를 사용할 수 있습니다. 가비지 수집기가 실행될 때 개체가 약한 참조에서만 참조되는 경우 기본 개체를 해제합니다. 해제된 개체를 가리키는 약한 전역 참조는 기능적으로 NULL 참조와 동일합니다. 프로그래머는 약한 참조를 NULL 과 비교하기 위해 `IsSame Object` 함수를 사용하여 약한 전역 참조가 해제된 개체를 가리키는지 여부를 감지할 수 있습니다.

JNI의 약한 전역 참조는 Java 2 SDK 릴리스 1.2의 일부로 사용할 수 있는 Java 약한 참조 (`java.lang.ref`) API의 단순화된 버전입니다. JNI 약한 전역 참조는 Java 약한 참조 API에 있는 네 가지 유형의 약한 참조보다 약합니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 226.

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`obj`: 약한 전역 참조가 생성될 개체입니다.

반환 값 `obj`가 `null`을 참조하거나 `obj` 가 가비지 수집 개체에 대한 약한 전역 참조이거나 가상 머신 구현의 메모리가 부족한 경우 `NULL` 을 반환합니다.

예외

`OutOfMemoryError`: 시스템의 메모리가 부족한 경우.

JNI 함수

팝로컬프레임

팝로컬프레임

원기

`jobject PopLocalFrame(JNIEnv *env, jobject 결과);`

설명

스택에서 현재(최상위) 로컬 참조 프레임을 팝합니다. 또한 이 함수는 프레임에 포함된 모든 로컬 참조를 해제하고 주어진 결과 개체에 대한 이전 로컬 참조 프레임의 로컬 참조를 반환합니다.

이전 프레임에 대한 참조를 반환할 필요가 없으면 결과 매개변수에 `NULL`을 전달합니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 20.

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

결과: 이전 로컬 참조 프레임으로 전달될 객체.

반환 값 이전 로컬 참조 프레임의 로컬 참조를 반환합니다.

두 번째 인수와 동일한 개체를 참조합니다.

예외

없음.

푸시로컬프레임

JNI 함수

푸시로컬 프레임

원기

`jint PushLocalFrame(JNIEnv *env, jint 용량);`

설명

최소한 지정된 수의 로컬 참조를 생성할 수 있는 새 로컬 참조 프레임을 생성합니다. 새 프레임에서 생성된 모든 로컬 참조는 PopLocalFrame이 호출될 때 해제됩니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 19 .

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`용량` : 로컬 참조 프레임에서 생성될 로컬 참조의 최대 수입니다.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수를 반환하고 `OutOfMemoryError`를 발생시킵니다. 이 함수의 호출이 예외를 `throw`한 경우에만 음수를 반환합니다.

예외

`OutOfMemoryError`: 시스템의 메모리가 부족한 경우.

JNI 함수

원주민 등록

원주민 등록

원기

```
jint RegisterNatives(JNIEnv *env, jclass clazz,
                      const JNINativeMethod *메서드, jint nMethods);
```

설명

clazz 인수로 지정된 클래스에 네이티브 메서드를 등록합니다. methods 매개 변수는 네이티브 메서드의 이름, 설명자 및 함수 포인터를 포함하는 JNI NativeMethod 구조의 배열을 지정합니다. nMethods 매개 변수는 배열의 기본 메서드 수를 지정합니다.

JNINativeMethod 구조는 다음과 같이 정의됩니다.

```
typedef struct { char *
    이름; 문자 *서명; 무
    효 *fnPtr;
} JNINativeMethod;
```

JNINativeMethod 구조의 fnPtr 필드는 기본 메소드를 구현하는 유효한 함수 포인터여야 합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 215.

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 네이티브 메서드가 등록될 클래스 개체에 대한 참조입니다.

methods: 등록할 네이티브 메서드입니다.

nMethods: 등록할 네이티브 메서드의 수.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수 값을 반환합니다.

이 함수의 호출이 예외를 throw한 경우에만 음수를 반환합니다.

예외

NoSuchMethodError: 지정된 메서드를 찾을 수 없거나 메서드가 네이티브가 아닌 경우.

Release<Type>ArrayElements JNI 함수

Release<Type>ArrayElements

원기 무효 Release<Type>ArrayElements(JNIEnv *env,
<ArrayType> 배열, <NativeType> *elems, 진트 모드);

양식 이 함수 계열은 8개의 멤버로 구성됩니다.

Release<Type>ArrayElements <ArrayType>	<네이티브 유형>
ReleaseBooleanArrayElements jbooleanArray jboolean	
ReleaseByteArrayElements jbyteArray	제이바이트
ReleaseCharArrayElements jcharArray	jchar
ReleaseShortArrayElements jshortArray	jshort
ReleaseIntArrayElements 진트어레이 이	진트
ReleaseLongArrayElements jlongArray	제롬
ReleaseFloatArrayElements jfloatArray	jfloat
ReleaseDoubleArrayElements jdoubleArray jdouble	

설명 네이티브 코드가 해당 Get<Type>ArrayElements 함수를 사용하여 파생된 기본 배열 요소에 더 이상 액세스할 필요가 없음을 가상 머신 구현에 알립니다. 필요한 경우 이 기능은 요소에 대한 모든 변경 사항을 원래 배열에 다시 복사합니다.

mode 인수는 배열 버퍼를 해제하는 방법에 대한 정보를 제공합니다. mode 인수는 elem이 array에 있는 요소의 복사본이 아닌 경우 효과가 없습니다. 그렇지 않은 경우 모드는 다음 표에 표시된 대로 다음과 같은 영향을 미칩니다.

방법	행위
0	요소 버퍼를 다시 복사하고 해제하십시오.
JNI_COMMIT	다시 복사하지만 요소 버퍼를 해제하지 않습니다.
JNI_ABORT	elems 버퍼에서 가능한 변경 사항을 다시 복사하지 않고 버퍼를 해제합니다.

대부분의 경우 프로그래머는 고정된 배열과 복사된 배열 모두에 대해 일관된 동작을 보장하기 위해 mode 인수에 0을 전달합니다. 다른 옵션은 프로그래머에게 메모리 관리에 대한 더 많은 제어권을 제공하므로 매우 주의해서 사용해야 합니다.

JNI 함수

Release<Type>ArrayElements

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

Release<Type>ArrayElements	색인
ReleaseBooleanArrayElements 191	
ReleaseByteArrayElements 192	
ReleaseCharArrayElements 193	
ReleaseShortArrayElements 194	
ReleaseIntArrayElements 195	
ReleaseLongArrayElements 196	
ReleaseFloatArrayElements 197	
ReleaseDoubleArrayElements 198	

매개변수

env : JNIEnv 인터페이스 포인터.

배열: 배열 객체에 대한 참조입니다.

elems: 배열 요소에 대한 포인터.

모드: 릴리스 모드.

예외

없음.

ReleasePrimitiveArrayCritical

JNI 함수

ReleasePrimitiveArrayCritical

원기

```
void ReleasePrimitiveArrayCritical(JNIEnv *env, jarray 배열, void
*caray, jint 모드);
```

설명

네이티브 코드가 더 이상 이전 GetPrimitiveArrayCritical 호출의 결과인 caray에 액세스할 필요가 없음을 가상 머신 구현에 알립니다. 필요한 경우 이 기능은 caray에 대한 모든 변경 사항을 원래 어레이에 다시 복사합니다.

mode 인수는 배열 버퍼를 해제하는 방법에 대한 정보를 제공합니다. caray가 array에 있는 요소의 복사본이 아닌 경우 mode 인수는 효과가 없습니다. 그렇지 않은 경우 모드는 다음 표에 표시된 대로 다음과 같은 영향을 미칩니다.

방법	행위
0	다시 복사하고 caray 버퍼를 해제합니다.
JNI_COMMIT	다시 복사하지만 caray 버퍼를 해제하지 않습니다.
JNI_ABORT	caray 버퍼에서 가능한 변경 사항을 다시 복사하지 않고 버퍼를 해제합니다.

대부분의 경우 프로그래머는 복사된 배열의 일관된 동작을 보장하기 위해 mode 인수에 0을 전달합니다. 다른 옵션은 프로그래머에게 메모리 관리에 대한 더 많은 제어권을 제공하므로 매우 주의해서 사용해야 합니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 223.

매개변수

env : JNIEnv 인터페이스 포인터.

배열: 배열 개체에 대한 참조입니다.

caray: 배열 요소에 대한 포인터.

모드: 릴리스 모드.

예외

없음.

JNI 함수

ReleaseStringChars

ReleaseStringChars

원기

무효 ReleaseStringChars(JNIEnv *env,
jstring 문자열, const jchar *chars);

설명

네이티브 코드가 더 이상 문자에 액세스할 필요가 없음을 가상 머신 구현에 알립니다. chars 인수는 GetStringChars 를 사용하여 문자열에서 얻은 포인터입니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 166 .

매개변수

env : JNIEnv 인터페이스 포인터.
문자열: 문자열 개체에 대한 참조입니다.
chars: 유니코드 문자열에 대한 포인터.

예외

없음.

ReleaseStringCritical

JNI 함수

ReleaseStringCritical

원기

무효 ReleaseStringCritical(JNIEnv *env,
jstring 문자열, const jchar *carray);

설명

기본 코드가 더 이상 carray에 액세스할 필요가 없음을 가상 머신 구현에 알립니다. carray 인수는 GetStringCritical을 사용하여 문자열에서 얻은 포인터입니다.

GetStringCritical 및 ReleaseStringCritical 호출로 둘러싸인 코드 세그먼트에서 네이티브 코드는 임의의 JNI 호출을 실행하거나 현재 스레드가 가상 머신 인스턴스의 다른 스레드를 차단하고 대기하도록 해서는 안 됩니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 225.

매개변수

env : JNIEnv 인터페이스 포인터.

문자열: 문자열 개체에 대한 참조입니다.

chars: 유니코드 문자열에 대한 포인터.

예외

없음.

JNI 함수

ReleaseStringUTF문자

ReleaseStringUTF문자

원기

무효 ReleaseStringUTFChars(JNIEnv *env, jstring 문자열,
const char *utf);

설명

네이티브 코드가 더 이상 네이티브 문자열 utf에 액세스할 필요가 없음을 가상 머신 구현에 알립니다. utf 인수는 GetStringUTFChars를 사용하여 문자열에서 파생 된 포인터입니다 .

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스 169 .

매개변수

env : JNIEnv 인터페이스 포인터.

문자열: 문자열 개체에 대한 참조입니다.

utf: UTF-8 문자열에 대한 포인터.

예외

없음.

Set<Type>ArrayRegion

JNI 함수

Set<Type>ArrayRegion

원기

```
무효 Set<Type>ArrayRegion(JNIEnv *env,
<ArrayType> 배열, jsize 시작, jsize len,
<NativeType> *buf);
```

양식

이 함수 계열은 8개의 멤버로 구성됩니다.

Set<Type>ArrayRegion	<배열 유형>	<네이티브 유형>
SetBooleanArrayRegion	jbooleanArray	jboolean
SetByteArrayRegion	jbyteArray	제이바이트
SetCharArrayRegion	jcharArray	jchar
SetShortArrayRegion	jshortArray	jshort
SetIntArrayRegion	진트어레이	진트
SetLongArrayRegion	jlongArray	제롱
SetFloatArrayRegion	jfloatArray	jfloat
SetDoubleArrayRegion	jdoubleArray	제이더블

설명

버퍼에서 기본 배열의 영역을 다시 복사합니다. 배열 참조 및 buf 버퍼는 NULL 이 아니어야 합니다 .

결합

JNIEnv 인터페이스 함수 테이블 의 인덱스입니다 .

Set<Type>ArrayRegion	색인
SetBooleanArrayRegion	207
SetByteArrayRegion	208
SetCharArrayRegion	209
SetShortArrayRegion	210
SetIntArrayRegion	211
SetLongArrayRegion	212
SetFloatArrayRegion	213
SetDoubleArrayRegion	214

JNI 함수

Set<Type>ArrayRegion

매개변수

`env : JNIEnv 인터페이스 포인터.``배열: 요소가 복사되는 기본 배열에 대한 참조입니다.``시작: 기본 배열의 시작 인덱스입니다.``len: 복사할 요소의 수.``buf: 소스 버퍼.`

예외

`ArrayIndexOutOfBoundsException: 영역의 인덱스 중 하나가 유효하지 않은 경우.`

<유형>필드 설정

JNI 함수

<유형>필드 설정

원기 `void Set<Type>Field(JNIEnv *env, jobject obj, jfieldID fieldID,
<NativeType> 값);`

양식

이 함수군은 9개의 멤버로 구성됩니다.

<유형>필드 설정	<네이티브 유형>
SetObjectField	직업
SetBooleanField	jboolean
SetByteField	제이바이트
SetCharField	jchar
ShortField 설정	jshort
SetIntField	진트
LongField 설정	jlong
SetFloatField	jfloat
DoubleField 설정	제이더블

설명

개체의 인스턴스 필드 값을 설정합니다. 개체 참조는 NULL이 아니어야 합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다.

<유형>필드 설정	색인
SetObjectField	104
SetBooleanField	105
SetByteField	106
SetCharField	107
ShortField 설정	108
SetIntField	109
LongField 설정	110
SetFloatField	111
DoubleField 설정	112

JNI 함수

<유형>필드 설정

매개변수

env : JNIEnv 인터페이스 포인터.

obj: 개체에 대한 참조입니다.

fieldID: 필드 ID.

값: 필드의 새 값입니다.

예외

없음.

SetObjectArrayElement	JNI 함수
-----------------------	--------

SetObjectArrayElement

원기 `void SetObjectArrayElement(JNIEnv *env, jobjectArray 배열, jsize 인덱스, jobject 값);`

설명 개체 배열의 요소를 설정합니다. 배열 참조는 NULL이 아니어야 합니다.

결합 JNIEnv 인터페이스 함수 테이블의 인덱스 174.

매개변수 env : JNIEnv 인터페이스 포인터.

array: 해당 요소에 액세스할 배열에 대한 참조입니다.

index: 액세스할 배열 요소의 인덱스.

값: 배열 요소의 새 값입니다.

예외 ArrayIndexOutOfBoundsException: 인덱스가 배열에서 유효한 인덱스를 지정하지 않는 경우.

ArrayStoreException: 값의 클래스가 배열 요소 클래스의 하위 클래스가 아닌 경우.

JNI 함수

SetStatic<유형>필드

SetStatic<유형>필드

원기

```
무효 SetStatic<Type>Field(JNIEnv *env, jclass clazz,
jfieldID fieldID,
<NativeType> 값);
```

양식

이 함수군은 9개의 멤버로 구성됩니다.

SetStatic<유형>필드	<네이티브 유형>
SetStaticObjectField	직업
SetStaticBooleanField	jboolean
SetStaticByteField	제이바이트
SetStaticCharField	jchar
SetStaticShortField	jshort
SetStaticIntField	진트
SetStaticLongField	jlong
SetStaticFloatField	jfloat
SetStaticDoubleField	제이더블

설명

클래스 또는 인터페이스의 정적 필드 값을 설정합니다. 액세스할 필드는 필드 ID로 지정됩니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스입니다 .

SetStatic<유형>필드	색인
SetStaticObjectField	154
SetStaticBooleanField	155
SetStaticByteField	156
SetStaticCharField	157
SetStaticShortField	158
SetStaticIntField	159
SetStaticLongField	160
SetStaticFloatField	161
SetStaticDoubleField	162

SetStatic<유형>필드

JNI 함수

매개변수

env : JNIEnv 인터페이스 포인터.

clazz: 정적 필드에 액세스할 클래스 또는 인터페이스에 대한 참조입니다.

fieldID: 액세스할 정적 필드를 나타내는 ID.

값: 필드의 새 값입니다.

예외

없음.

JNI 함수

던지다

던지다

원기

`jint Throw(JNIEnv *env, jthrowable obj);`

설명

`java.lang.Throwable` 객체가 발생하도록 합니다. `throw`된 예외는 현재 스레드에서 보류되지만 네이티브 코드 실행을 즉시 중단하지는 않습니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 13.

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`obj`: `java.lang.Throwable` 객체입니다.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수 값을 반환합니다.

예외

주어진 `java.lang.Throwable` 객체.

[새로 만들기](#)[JNI 함수](#)

새로 만들기

원기

```
jint ThrowNew(JNIEnv *env, jclass clazz, const char
               *message);
```

설명

message 로 지정된 메시지를 사용하여 지정된 클래스에서 예외 객체를 생성하고 해당 예외를 throw합니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 14 .

매개변수

env : JNIEnv 인터페이스 포인터.
clazz: java.lang.Throwable 의 하위 클래스입니다 .
메시지: java.lang.Throw 가능 객체를 구성하는 데 사용되는 메시지입니다 .

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 다음과 같은 경우 음수 값을 반환합니다.
 지정된 예외를 throw할 수 없습니다.

예외

새로 생성된 java.lang.Throwable 객체 또는 이 객체를 생성할 때 발생하는 예외입니다.

JNI 함수

ToReflectedField

ToReflectedField

원기

```
jobject ToReflectedField(JNIEnv *env, jclass cls, jfieldID fieldID,
jboolean isStatic);
```

설명

cls에서 파생된 필드 ID를 java.lang.reflect.Field 클래스의 인스턴스로 변환합니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 12.

매개변수

env : JNIEnv 인터페이스 포인터.

cls : 클래스 또는 인터페이스에 대한 참조.

fieldID : JNI 필드 ID입니다.

isStatic : 필드 ID가 정적 필드를 나타내는지 여부.

반환 값 java.lang.reflect.Field 클래스의 인스턴스를 반환합니다. 그렇지 않으면 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

ToReflected 메서드

JNI 함수

ToReflected 메서드

원기

```
 jobject ToReflectedMethod(JNIEnv *env, jclass cls, jmethodID
                           methodID, jboolean isStatic);
```

설명

cls 에서 파생된 메소드 ID를 java.lang.reflect.Method 클래스의 인스턴스 또는 java.lang.reflect.Constructor 클래스의 인스턴스로 변환합니다.

이 함수는 Java 2 SDK 릴리스 1.2에서 도입되었습니다.

결합

JNIEnv 인터페이스 함수 테이블의 인덱스 9.

매개변수

env : JNIEnv 인터페이스 포인터.

cls: 클래스 또는 인터페이스에 대한 참조.

methodID: 메서드 ID.

isStatic: 메서드 ID가 정적 메서드를 참조하는지 여부.

반환 값 java.lang.reflect.Method 클래스의 인스턴스 또는 java.lang.reflect.Constructor 클래스의 인스턴스를 반환합니다. 그렇지 않으면 NULL을 반환합니다. 이 함수의 호출이 예외를 throw한 경우에만 NULL을 반환합니다.

예외

OutOfMemoryError: 시스템의 메모리가 부족한 경우.

JNI 함수

원주민 등록 취소

원주민 등록 취소

원기`jint UnregisterNatives(JNIEnv *env, jclass clazz);`설명

클래스의 네이티브 메서드를 등록 취소합니다. 클래스는 네이티브 메서드 함수에 연결되거나 등록되기 전의 상태로 돌아갑니다.

이 함수는 일반 네이티브 코드에서 사용하면 안 됩니다. 대신 네이티브 라이브러리를 다시 로드하고 다시 연결하는 방법을 특수 프로그램에 제공합니다.

결합

`JNIEnv` 인터페이스 함수 테이블의 인덱스 216 .

매개변수

`env` : `JNIEnv` 인터페이스 포인터.

`clazz`: 네이티브 메서드를 등록 해제할 클래스 개체에 대한 참조입니다.

반환 값 성공 시 0을 반환합니다. 그렇지 않으면 음수 값을 반환합니다.

예외

없음.

색인

| 히브리어

중단 후크, 250, 252
AllocObject, 52, 181
Arnold, Ken, 3
ArrayIndexOutOfBoundsException, 37, 229 배열, 23
 액세스, 33 함수 중에서 선택, 36 효율적인 액세스, 158 함수
 요약, 35 다차원, 39 객체 대 프리미티브, 33 동시 액세스, 160 ASCII 문자열, 24 `asm_dispatch` 어셈블리 루틴, 122 어셈블리 구현 공유 스럽 디스패처, 122 시
 간 결정적 코드, 7

비동기 예외, 163 `atol` 예제, 109 기본 스레드 연결, 89 `attach.c` 예제, 89
AttachCurrentThread, 96, 182

| 베트남어

이진 호환성, 145 부울, 165 네이티브 코드의 경계, 133
Bracha, Gilad, 146 바이트, 165

| 씨

씨
 호출 규칙, 153 프로그래밍 언어, 3
C 셸, 17
C++
 프로그래밍 언어, 3에서 JNI 프로그래밍, 106 가상 함수 테이블, 153 캐싱 필드 및 메서드 ID, 53 사용 시점, 53 클래스 정의, 56 정확성에 필요, 135

`Call<Type>메소드`, 47, 184
`<Type>MethodA` 호출, 186
`Call<Type>MethodV`, 188 콜백, 46
`CallBoolean` 메서드, 184
`CallBooleanMethodA`, 186
`CallBooleanMethodV`, 188
`CallByteMethod`, 184
`CallByteMethodA`, 186
`CallByteMethodV`, 188
`CallChar` 메서드, 184
`CallCharMethodA`, 186
`CallCharMethodV`, 188
`CallDoubleMethod`, 184
`CallDoubleMethodA`, 186
`CallDoubleMethodV`, 188
`CallFloat` 메서드, 184
`CallFloatMethodA`, 186
`CallFloatMethodV`, 188 호출 규칙, 153

색인

CallIntMethod, 47, 184
 CallIntMethodA, 186
 CallIntMethodV, 188
 CallLong 메서드, 184
 CallLongMethodA, 186
 CallLongMethodV, 188
 CallNonvirtual<Type>방법, 51, 190
 CallNonvirtual<Type>MethodA, 192
 CallNonvirtual<Type>MethodV, 194
 CallNonvirtualBooleanMethod, 51, 190
 CallNonvirtualBooleanMethodA, 192
 CallNonvirtualBooleanMethodV, 194
 CallNonvirtualByteMethod, 190
 CallNonvirtualByteMethodA, 192
 CallNonvirtualByteMethodV, 194
 CallNonvirtualCharMethod, 190
 CallNonvirtualCharMethodA, 192
 CallNonvirtualCharMethodV, 194
 CallNonvirtualDoubleMethod, 190
 CallNonvirtualDoubleMethodA, 192
 CallNonvirtualDoubleMethodV, 194
 CallNonvirtualFloatMethod, 190
 CallNonvirtualFloatMethodA, 192
 CallNonvirtualFloatMethodV, 194
 CallNonvirtualIntMethod, 190
 CallNonvirtualIntMethodA, 192
 CallNonvirtualIntMethodV, 194
 CallNonvirtualLongMethod, 190
 CallNonvirtualLongMethodA, 192
 CallNonvirtualLongMethodV, 194
 CallNonvirtualObjectMethod, 190
 CallNonvirtualObjectMethodA, 192
 CallNonvirtualObjectMethodV, 194
 CallNonvirtualShortMethod, 190
 CallNonvirtualShortMethodA, 192
 CallNonvirtualShortMethodV, 194
 CallNonvirtualVoidMethod, 51, 190
 CallNonvirtualVoidMethodA, 192
 CallNonvirtualVoidMethodV, 194
 CallObjectMethod, 47, 184
 CallObjectMethodA, 186
 CallObjectMethodV, 188
 CallShort 방법, 184
 CallShortMethodA, 186
 CallShortMethodV, 188
 CallStatic<Type>방법, 196
 CallStatic<Type>MethodA, 198
 CallStatic<Type>MethodV, 200
 CallStaticBooleanMethod, 49, 196
 CallStaticBooleanMethodA, 198
 CallStaticBooleanMethodV, 200
 CallStaticByteMethod, 196
 CallStaticByteMethodA, 198
 CallStaticByteMethodV, 200
 CallStaticCharMethod, 196
 CallStaticCharMethodA, 198
 CallStaticCharMethodV, 200
 CallStaticDoubleMethod, 196
 CallStaticDoubleMethodA, 198
 CallStaticDoubleMethodV, 200
 CallStaticFloatMethod, 196
 CallStaticFloatMethodA, 198
 CallStaticFloatMethodV, 200
 CallStaticIntMethod, 196
 CallStaticIntMethodA, 198
 CallStaticIntMethodV, 200
 CallStaticLongMethod, 196
 CallStaticLongMethodA, 198
 CallStaticLongMethodV, 200
 CallStaticObjectMethod, 196
 CallStaticObjectMethodA, 198
 CallStaticObjectMethodV, 200
 CallStaticShortMethod, 196
 CallStaticShortMethodA, 198
 CallStaticShortMethodV, 200
 CallStaticVoidMethod, 49, 196
 CallStaticVoidMethodA, 198
 CallStaticVoidMethodV, 200
 CallVoidMethod, 47, 184
 CallVoidMethodA, 186
 CallVoidMethodV, 188
 CatchThrow 예, 73

참조

- Solaris C 컴파일러 참조
- CFunction 클래스
- callInt 메서드 구현,
- 119 사용, 114 구현, 118

- 사용, 113 자,
23, 165
씨열
- Microsoft Visual C++ 컴파일러 클래스 설명자, 39, 169를 참조하십시오.
- 클래스 로더 부트
- 스트랩 로더, 214 정의 로더,
146 위임, 147 가비지 수집,
151 JNI_OnUnload 핸들러, 104 유형 안전성, 150
 - ClassCircularityError, 202, 215 클래스, 4, 23 설명자, 169 JNI를 통한 로드, 39 ClassFormatError, 202, 215
 - ClassLoader .defineClass, 202
ClassLoader.findLibrary, 149
ClassLoader.loadClass, 214
- CLASSPATH, 214
- CMalloc 클래스 무
- 로 메서드, 115 구현,
117 사용, 114 코드 예제
atol, 109 attach.c, 89
- CacheThrow, 73 CFunction,
- 118 CMalloc , 117
CPointer, 117 HelloWorld,
11 InstanceFieldAccess,
42, 54 InstanceMethodCall,
46, 57 IntArray, 33, 34,
35 invoke.c, 83 KeyInput,
127 MyNewString, 51,
55, 62, 64, 71 ObjectArrayTest, 38
Prompt, 21, 25, 29 StaticFieldAccess,
45 StaticMethodCall, 50 웹 다운로드 주소, 9 Win32.CreateFile, 111 , 115
- CMalloc 클래스 무
- 로 메서드, 115 구현,
117 사용, 114 코드 예제
atol, 109 attach.c, 89
- CacheThrow, 73 CFunction,
- 118 CMalloc , 117
CPointer, 117 HelloWorld,
11 InstanceFieldAccess,
42, 54 InstanceMethodCall,
46, 57 IntArray, 33, 34,
35 invoke.c, 83 KeyInput,
127 MyNewString, 51,
55, 62, 64, 71 ObjectArrayTest, 38
Prompt, 21, 25, 29 StaticFieldAccess,
45 StaticMethodCall, 50 웹 다운로드 주소, 9 Win32.CreateFile, 111 , 115
- GetPrimitiveArrayCritical 의 교착 상태 ,
GetStringCritical 의 35 , 32
- DefineClass, 202 정의
- 로더, 146 위임, 147
- DeleteGlobalRef, 203
- DeleteGlobalWeakRef, 65
- DeleteLocalRef, 63, 204
- DeleteWeakGlobalRef, 70, 130, 205 설명자 클래스
- 클래스 설명자 필드 참조
- 필드 설명자 메서드 참조
- 메서드 설명자 참조
- DestroyJavaVM, 86, 206
- DetachCurrentThread, 92, 207
- DLL
- 동적 링크 라이브러리 dlopen,
88 dlsym, 88 double, 165 동적 링크 라이브러리, 16 참조

명령줄 옵션, 251 상수, 170 생성자, 46

NewObject, 51 호출, 51 도 참조하십시오.

이자형

EnsureLocalCapacity, 68, 208

색인

이름 인코딩의 이스케이
 프 시퀀스 , UTF-8 문자열의
 152개, 예제 프로그램 168개
 코드 예제 참조 ExceptionCheck,
 77, 209 ExceptionClear,
 75, 210 ExceptionDescribe, 75,
 211 ExceptionInInitializerError,
 215, 226, 228, 233, 236
 ExceptionOccurred, 75, 212 예외 비동기, 163 확인, 76,
 131 처리, 78, 161 유ти리티 함수, 79 보류, 25, 75, 162
 대 프로그래머 오류, 161 종료 후크, 250, 252

findLibrary 참조
 ClassLoader.findLibrary float, 23, 165
 외부 함수 인터페이스, 145 FORTRAN, 145 무료
 글로벌 참조, 69 로컬 참조, 67 네이티브 피어 리소
 스, 125 가상 머신 리소스, 139
 FromReflectedField, 105, 216
 FromReflectedMethod, 105, 217

에프

FatalError, 213 필드
 설명자, 44, 169 javap -s, 44 필드
 ID, 43, 160, 168 캐싱, 53,
 135 해결, 160 필드 액세스, 41 액세
 스 정적 대 인스턴스, 46 설명
 자 필드 설명자 인스턴스 참조
 인스턴스 필드 이름 참조 , 43 static
 정적 필드 파일 설명자 참조 ,
 123 FileDescriptor 클래스, 124 최종 액
 세스 제어, 138 종료, 125 종료자 참조 기본
 피어를 해제하는 종료자 , 126
 JNI_OnUnload, 104 FindClass, 39,
 214

G

cc 에 대한 -G 옵션 , 16개
 의 가비지 수집 클래스 로더,
 151개의 객체 복사, 156
 및 네이티브 메서드 인터
 페이스, 8개의 객체 재배치, 27개의 약한
 전역 참조, 65

Get<Type>ArrayElements, 35, 36, 219
 Get<Type>ArrayRegion, 36, 221
 Get<Type>필드, 223
 GetArrayLength, 35, 36, 218
 GetBooleanArrayElements, 219
 GetBooleanArrayRegion, 221
 GetBooleanField, 223
 GetByteArrayElements, 219
 GetByteArrayRegion, 221
 GetByteField, 223
 GetCharArrayElements, 219
 GetCharArray영역, 221
 GetCharField, 223
 GetDoubleArrayElements, 219
 GetDoubleArrayRegion, 221
 GetDoubleField, 223
 GetEnv, 97, 172, 225
 GetFieldID, 43, 160, 226
 GetFloatArrayElements, 219
 GetFloatArrayRegion, 221
 GetFloatField, 223
 GetIntArrayElements, 159, 219
 GetIntArrayRegion, 34, 159, 221

GetIntField, 43, 160, 223	시간
GetJavaVM, 97, 227	핸들 Win32 유형, 111
GetLongArrayElements, 219	HelloWorld 예제, 11 호스트 환
GetLongArrayRegion, 221	경, 4
GetLongField, 223	자바 플랫폼 참조
GetMethodID, 47, 228	HTTP 서버, 89
GetObjectArrayElement, 38, 229	
GetObjectClass, 43, 105, 230	
GetObjectField, 43, 223	
GetPrimitiveArrayCritical, 35, 36, 159, 231	나
GetProcAddress Win32 API, 88, 119	ID 대 참조, 134
GetShortArrayElements, 219	IllegalAccessException, 138
GetShortArrayRegion, 221	IllegalArgumentException, 73
GetShortField, 223	IllegalMonitorStateException, 94, 259 initIDs, 56 인
GetStatic<Type>필드, 234	라인 최종 필드 액세스, 138 기본 메소드, 58, 133 인스턴스
GetStaticBooleanField, 234	필드, 41 액세스 프로시저, 43 인스턴스 메소드, 46 호출, 수퍼
GetStaticByteField, 234	클래스의, 51 기본, 23 호출 단계, 47
GetStaticCharField, 234	InstanceFieldAccess 예, 42, 54
GetStaticDoubleField, 234	InstanceMethodCall 예, 46, 57 인스턴스, 23 가상 머신,
GetStaticFieldID, 46, 233	173 InstantiationException, 181, 262, 263, 264
GetStaticFloatField, 234	int, 23, 165 IntArray 예, 33, 34, 35 메서드 호출 인터페이
GetStaticIntField, 45, 234	스, 47 획득, 48 국제화 코드, 99, 138 호출 인터페이
GetStaticLongField, 234	스, 5, 83 invoke.c 예제, 83 IsAssignableFrom,
GetStaticMethodID, 49, 236	105, 246 isCopy 인수, 26 릴리스 리소스, 140
GetStaticObjectField, 234	
GetStaticShortField, 234	
GetStringChars, 26	
GetStringChars, 30, 139, 237	
GetStringCritical, 27, 30, 238	
GetStringLength, 26, 30, 239	
GetStringRegion, 29, 30, 240	
GetStringUTFChars, 24, 25, 30, 241	
GetStringUTFLength, 30, 242	
GetStringUTFRegion, 29, 30, 243	
GetSuperclass, 105, 244	
GetVersion, 245 전역	
참조, 64, 156 freeing, 69	
고슬링, 제임스, 3	IsInstanceOf, 105, 247
그린 스래드, 97, 141	IsSameObject, 66, 158, 248

색인

제이

jarray, 166, 167 Java

2 SDK, 8 Java API

Java 애플리케이션 프로

 그래밍 인터페이스 참조 Java 애플리케이션, 4 네이티브
 애플리케이션 참조 Java 애플리케이션 프로그래밍

인터페이스, 4 Java Core Reflection API, 161 Java

Development Kit, 7 Java 네이티브 인터페이스, 3 대

체 솔루션, 6가지 이점, 8가지 설계 목표, 145가지 기능 표, 22가지 기능, 22가지 사용의 의미, 6가지 성능 특성, 58가지 역할, 4가

지 버전 진화, 155, 179 사용 시기, 6 Java 플랫폼, 4 호스트 환

경 Java 프로그래밍 참조 언어, 3 자바 런타임 환경, 4 자바 런타입 인터프리터, 11, 83 자바 가상 머신, 4 인스턴스, 173 자

바 약한 참조 API, 270 java.class.path 속성, 85

java.io.FileDescriptor, 124 java.lang.Class, 202

java.lang.ClassLoader, 202 java.lang.Float, 121

java.lang.Integer, 121 java.lang.reflect 패키지, 105

java.lang.reflect.Constructor, 105, 217

java.lang.reflect.Field, 105, 216

java.lang.reflect.Method, 105, 217 java.lang.String

문자열 java.lang.Thread, 48 java.libr 참조 ary.path

속성, 17, 150 Java/Java 호출, 58 Java/네이티브 호출,

58

Java_ 접두사, 22

Java_VMIInitArgs, 251 javac

컴파일러, 11, 14 javah 도구,

9, 11, 14 javai.dll, 87

javai.lib, 87 javap 도구, 44

JavaVM 인터페이스 생성, 83 파

괴, 86 포인터 얻기, 97

JavaVMAttachArgs, 182

JavaVMIInitArgs, 85

JavaVMOption, 85

jboolean, 165 실수로

잘림, 132

jbooleanArray, 166, 167 jbyte,

165 jbyteArray, 166, 167 jchar,

165 jcharArray, 166, 167 jclass,

166, 167 jclass 대 jobject, 132

JDK

Java 개발 키트 참조

JDK1_1InitArgs, 85, 249

jdouble, 165 jdoubleArray,

166, 167 jfieldID, 168 jfloat, 23,

165 jfloatArray, 166, 167 jint,

23, 165 jintArray, 166, 167

JIT 컴파일러

just-in-time 컴파일러 참조 jlong,

165 jlongArray, 166, 167 jmethodID,

168

JNI

자바 네이티브 인터페이스 참조

jni.h 헤더 파일, 15, 165

JNI_ABORT 상수, 171, 276

JNI_AttachCurrentThread, 92

JNI_COMMIT 상수, 171, 276

JNI_CreateJavaVM, 85, 249

JNI_EDETACHED 상수, 172, 225
 JNI_ERR 상수, 171
 JNI_EVERSION 상수, 172, 225
 JNI_FALSE 상수, 171
 JNI_GetCreatedJavaVM, 97, 254
 JNI_GetDefaultJavaVMInitArgs, 85, 255
 JNI_NativeMethod, 273
 JNI_OK 상수, 171
 JNI_OnLoad, 97, 102, 256
 JNI_OnUnload, 104, 257
 JNI_ThreadAttachArgs, 92
 JNI_TRUE 상수, 171
 JNI_VERSION_1_1 상수, 171
 JNI_VERSION_1_2 상수, 85, 103, 171
 JNICALL 매크로, 22, 170
 JNIEnv, 22 이
 점, 155 포인터 얻
 기, 96 조작, 153 스레드 로컬,
 93, 141

 JNIEnv2, 179
 JNIEXPORT 매크로, 22, 170
 JNU_CallMethodByName, 79, 137
 JNU_FindCreateJavaVM, 88
 JNU_GetEnv, 103
 JNU_GetStringNativeChars, 100, 112, 138
 JNU_MonitorNotify, 96
 JNU_MonitorNotifyAll, 96
 JNU_MonitorWait, 96
 JNU_NewStringNative, 99, 138
 JNU_ThrowByName, 75
 jobject, 24, 166, 167
 jobjectArray, 24, 166, 167
 JRE
 Java Runtime Environment 참조
 jshort, 165 jshortArray, 166, 167 jsize, 166
 jstring, 24, 166, 167 jthrowable, 166, 167
 just-in-time compiler, 138 jvalue union
 type, 167 jvm.dll, 87 jvm.lib, 87

케이
 kernel32.dll, 115
 Kernighan, Brian, 3
 KeyInput 예, 127 KornShell,
 17 ksh, 17

얼
 cl 에 대한 -LD 옵션, 16
 LD_LIBRARY_PATH, 17, 87, 149 Lea,
 Doug, 93 Liang, Sheng, 146 libjava.so,
 86 libjvm.so, 87 cl 에 대한 -link 옵션,
 87 네이티브 애플리케이션 연결, 86 네이티
 브 메서드, 151 LISP, 145 cc 에 대한
 -ljava 옵션, 86 loadLibrary 참조
 System.loadLibrary LoadLibrary
 Win32 API, 88, 119 로컬 참조, 62,
 156 1.2 특정 기능, 68 과도한 생성,
 140 참조 -verbose:jni 옵션 해제, 67 방
 법 구현, 157 무효화 방법, 63 레지스트리,
 157 스레드 로컬, 94 유효성, 141 로케일,
 99 long, 165 longjmp, 97 참조
 에 대한 -lthread 옵션, 86

중
 mapLibraryName
 cl, 16, 87에 대한
 System.mapLibraryName -MD 옵션 참조

색인

메모리 누수, 67 메서드 설
명자, 48, 170 javap -s, 49 메서드 ID,
160, 168 캐싱, 53, 135 해결, 160

메서드 호출,
46 설명자

수퍼클래스, 51 인스턴스의 메서드 설명자 참조

인스턴스 메서드 정적 참조

정적 메서드 참조

Microsoft COM 인터페이스, 154, 174, 175 Microsoft Visual C++ 컴파일러, 16 MonitorEnter, 94, 258 monitoreenter 명령, 258, 259 MonitorExit, 94, 259 monitorexit 명령, 258 모니터, 94 msrvct.dll, 113 mutex_lock, 97 MyNewString 예제, 51, 55, 62, 64, 71

일반 메서드와의 차이점, 13 구현 참조 javah 도구
함수 프로토타입, 14 헤더 파일, 15 이름 인코딩,
152 링크, 151 등록, 101 쓰기 및 실행 단계, 11 네이티브 메서드 인터페이스, 7 릴리스 간 Java 네이티브 인터페이스 호환성 참조, JDK 1.0의 8가지 문제, 7개의 기본 수정자, 13개의 기본 프로그래밍 언어, 4개의 기본 스레드 모델, 97 개의 기본/Java 콜백, 58 New<Type>Array, 36, 266 NewBooleanArray, 266 NewByteArray, 266 NewCharArray, 52, 266 NewDoubleArray, 266 NewFloatArray, 266 NewGlobalRef, 64, 260 NewGlobalWeakRef, 65 NewIntArray, 266 NewLocalRef, 71, 261 NewLongArray, 266 NewObject, 52, 262 NewObjectA, 263 NewObjectArray, 39, 265 NewObjectV, 264 NewShortArray, 266 NewString, 85 NewString, 85, 30, 269 NewWeakGlobalRef, 130, 270 NoClassDefFoundError, 76, 202, 215 NoSuchFieldError, 226, 233 NoSuchMethodError, 47, 228, 236, 273

N

이름 인코딩 long,
117, 152 short,
152 네임스페이스,
146 네이티브 애플리케이션, 4 Java 애플리케이션 네이티브 코드 참조, 3 C, C++ 네이티브 라이브러리, 5 생성, 15 로드, 146 공유 라이브러리, 16 언로드, 151 네이티브 라이브러리 경로 참조
-Djava.library.path 설정, 16개의 기본 메서드, 5개의 인수, 22개의 선언, 13 도 참조하십시오.

영형

객체 배열, 33
Object.wait, notify 및 notifyAll, 95 ObjectArrayTest 예제, 38

- 개체 인스
 턴스 레이아웃, 8 일대일
 맵핑, 109 장단점, 116
- 대 공유 스텝, 116 OOPSLA, 146
 불투명 참조, 61 OutOfMemoryError, 25 참
 조
- 피
PATH, 17, 87
- 피어 클래스, 123 백포인
 터, 127 동기화 자유 메
 서드, 125
- JNI 작업의 성능
 기본 배열 액세스, 158 효율성 목표, 146 필
 드 액세스, 59 메서드 호출, 58 고정, 27,
 158
- PopLocalFrame, 69, 271 포팅
레이어, 133 기본 배열, 33 기본 유
형, 165 printf, 15 개인 액세스 제
어, 138 프로세스 분리, 6 프로그래
머 오류, 161
- 프롬프트 예, 21, 25, 29
푸시로컬프레임, 69, 272
- 아른 자형
- 캐싱 필드 및 메서드
 ID의 경쟁 조건, 네이티브 피어 해제 시 54개, 참
 조 유형 125개, 참조 비교 166 개, 해제 66개, 전
 역 66개
- 글로벌 레퍼런스 로컬 보기
- 로컬 참조 불투명, 23, 156
개의 관리 규칙, 70개의 악한 글로
벌 참조
 악한 글로벌 참조 리플렉션, 105, 161
참조
RegisterNatives, 101, 153, 273 사용, 102
 로컬 참조 레지스트리, 157
- Release<Type>ArrayElements, 35, 36, 274
ReleaseBooleanArrayElements, 274
ReleaseByteArrayElements, 274
ReleaseCharArrayElements, 274
ReleaseDoubleArrayElements, 274
ReleaseFloatArrayElements, 274
ReleaseIntArrayElements, 159, 171, 274
ReleaseLongArrayElements, 274
ReleasePrimitiveArrayCritical, 35, 36, 159, 276
- ReleaseShortArrayElements, 274
ReleaseStringChars, 26, 30, 139, 277
ReleaseStringCritical, 27, 30, 278
ReleaseStringUTFChars, 25, 30, 279 해결 필드
 및 메서드 ID, 160 제한
- GetStringCritical, 27,
238 의 GetPrimitiveArrayCritical, 35, 159, 231
 에 대한 제약 조건도 참조하십시오.
- 리치, 데니스, 3세
- 에스
- 보안_특성, 111
Set<Type>ArrayRegion, 36, 280
Set<Type>Field, 282
SetBooleanArrayRegion, 280
세트부울 필드, 282
SetByteArrayRegion, 280
세트바이트필드, 282
SetCharArrayRegion, 280
세트차필드, 282
SetDoubleArrayRegion, 280
세트더블필드, 282
SetFloatArrayRegion, 280

색인

SetFloatfield, 43, 282
 SetIntArrayregion, 34, 280
 SetIntfield, 282 SetJMP, 97
 SetLongArrayregion, 280
 Setlongfield, 282
 SetObjectArrayElement , 38, 284
 SetObjectfield, 43, 282 SetShortArrayregion,
 280 SetShortfield, 282 Field , 285 Field,
 285 Field , 282 285 SetStaticCharfield,
 285 SetStaticDoublefield, 285
 SetStaticFloatfield, 285 SetStaticIntfield,
 45, 285 SetStaticLongfield, 285
 SetStaticObjectfield , 285
 SetStatichortfield, 285 SetStringregion,
 30 SetStringutfregion, 30 Sh, 17 기본 도서
 관 See Stub, 116 Advantates , 116
 Advants, 116 Advantates, 116 Advantates,
 116 Advantates, 116 대 일대일 매핑, 116개
 short, 165개 Smalltalk, 145개 소켓 설명자,
 123개 Solaris 네이티브 스레드, 141개 공유 라
 이브러리, 16개 Solaris C 컴파일러, 16개 표준
 셀, 17개 정적 필드, 41개 액세스, 44개 정적 초기
 화 프로그램, 13개 호출됨 FindClass 제공 ,
 214 GetFieldID, 226 GetMethodID, 228
 GetStaticFieldID, 233
 GetStaticMethodID, 236

정적 메소드, 46 기본, 23
 호출 단계, 49
 StaticFieldAccess 예,
 45 StaticMethodCall 예, 50 stdcall, 116,
 153 stderr, 213 stdio.h 헤더 파일, 15 문자
 열, 23 참조 문자열 문자열 생성자, 99
 String.getBytes 메소드, 100
 StringIndexOutOfBoundsException, 29,
 240, 243 문자열, 23 액세스, 24 ASCII, 24 함
 수 중에서 선택, 31 구성, 26 리소스 해제,
 25 NULL 종료 C, 24 함수 요약, 29 유니코드,
 24 UTF-8, 24, 168 Stroustrup, Bjarne, 4
 스텝 함수, 112 하위 유형 지정, 106, 166 피어 클래스의 등
 기화 , 125 동기화된 블록, 94 System.loadLibrary,
 101, 148 사용 방법, 13 System.mapLibraryName,
 150 System.out,err, 211 System.
 runFinalization, 104

E|

tcsh, 17
 C++의 "this" 포인터, 14
 thr_create, 97 스레드 모델, 97,
 141 Green, 네이티브 및 사용자 스레
 드 모델 참조 Thread.start, 97 Thread.stop, 163

스레드, 93 프로

그래밍에 대한 제약, 93 스레드 로컬 JNIEnv,
153 로컬 참조의 유효성, 64

던지기, 162, 287

ThrowNew, 75, 162, 288

ToReflectedField, 105, 289

ToReflectedMethod, 105, 290 C++에

서 유형 변환, 107

Win32

CreateFile API, 110 동적 링
크 라이브러리, 16

Win32.CreateFile 예제, 111, 115 래퍼 클래스, 109

엑스

-Xcheck:jni 옵션, 131, 155

-Xmx 옵션, 85

유

유니코드 문자열, 24 NULL

로 끝나지 않음, 137

유닉스, 153

원주민 등록 취소, 291

UnsatisfiedLinkError, 17, 149, 153 사용자 스레

드 모델, 97

UTF-8 문자열, 24, 168 유틸리

티 함수, 70

JNU_CallMethodByName, 79

JNU_FindCreateJavaVM, 88

JNU_GetEnv, 103

JNUGetStringNativeChars, 100

JNU_MonitorNotify, 96

JNU_MonitorNotifyAll, 96

JNU_MonitorWait, 96

JNU_NewStringNative, 99

JNU_ThrowByName, 75

V

-verbose

gc, class, jni, 252

-verbose:jni 옵션, 69, 140 vfprintf

hook, 250, 252 가상 머신 인스턴스, 173

여

약한 전역 참조, 65, 네이티브 피어에서 158,

IsSameObject 사용 130, 66