《深入理解 JNI》：JNI 对象操作

对象的创建与销毁

在JNI中，创建和销毁Java对象是常见的操作。这涉及到使用JNIEnv指针提供的函数来实例化Java类并管理对象的生命周期。

创建Java对象

要创建一个Java对象，首先需要获取表示该对象类的 jclass ，然后使用 JNIEnv 的 NewObject 函数。 NewObject 函数需要三个参数：

1. jclass：表示要实例化的Java类的类引用。
2. jmethodID：表示要调用的构造方法的ID。
3. jvalue* args：一个指向参数数组的指针，如果构造方法没有参数，则为NULL。

示例代码：创建Java对象

假设我们有一个Java类 Person ，它有一个构造方法接受两个参数：一个String和一个int。

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // ... 其他方法 ...
}
```

在C/C++代码中创建 Person 对象的示例：

```
JNIEXPORT jobject JNICALL Java_ClassName_createPerson(JNIEnv *env, jobject obj, jstring nam
e, jint age) {
    // 获取Person类的引用
    jclass personClass = (*env)->FindClass(env, "com/example/Person");
    if (personClass == NULL) {
        return NULL; // 类未找到
```

```
    }

    // 获取构造方法的ID
    jmethodID constructor = (*env)->GetMethodID(env, personClass, "<init>", "(Ljava/lang/Strin
g;I)V");
    if (constructor == NULL) {
        return NULL; // 构造方法未找到
    }

    // 创建Person对象
    jobject person = (*env)->NewObject(env, personClass, constructor, name, age);
    return person;
}
```

销毁Java对象

在JNI中，不需要显式地销毁Java对象，因为Java的垃圾收集器会自动处理不再使用的对象。但是，当使用JNI时，需要注意管理本地引用。

每次调用JNI函数时，可能会创建本地引用。这些引用只在当前线程的当前本地方法调用期间有效。一旦本地方法返回，所有本地引用都会被自动释放。然而，如果你在本地代码中存储了对Java对象的引用，需要手动释放这些引用，以避免局部引用表溢出。

释放本地引用

使用 DeleteLocalRef 函数可以手动释放本地引用：

```
(*env)->DeleteLocalRef(env, localRef);
```

示例代码：释放Java对象引用

```
JNIEXPORT void JNICALL Java_ClassName_someMethod(JNIEnv *env, jobject obj) {
    // ... 创建和使用Java对象 ...

    // 假设localRef是之前创建的Java对象的本地引用
    jobject localRef = ...;

    // 使用完毕后释放引用
    (*env)->DeleteLocalRef(env, localRef);
}
```

注意事项

- 创建Java对象时，确保类路径和构造方法签名正确。
- 在创建对象后，如果不再需要，及时释放本地引用。
- 不要尝试手动销毁Java对象，这应该由垃圾收集器处理。

通过上述方法，JNI允许本地代码创建和销毁Java对象，从而实现与Java代码的无缝集成。正确管理对象的生命周期和引用是使用JNI时需要注意的重要方面。

访问Java对象字段

在JNI中，访问Java对象的字段是一项常见操作。这包括读取字段的值和修改字段的值。为了实现这一点，我们需要使用 JNIEnv 指针提供的函数来获取字段的ID，然后通过这个ID来访问字段。

获取字段ID

要访问Java对象的字段，首先需要获取该字段的 jfieldID 。这可以通过调用 JNIEnv 的 GetFieldID 函数来实现。 GetFieldID 需要三个参数：

1. jclass：表示包含字段的Java类的类引用。
2. 字段名称：字段的名称，以字符串形式提供。
3. 字段签名：字段的类型签名，以字符串形式提供。

字段签名遵循特定的格式，例如：

- int 类型的签名是 I
- boolean 类型的签名是 Z
- `java.lang.String`类型的签名是`Ljava/lang/String;`
- 数组的签名以 [开头，后跟元素类型的签名，例如`int[]`的签名是`[I`

访问字段

一旦我们有了字段的 jfieldID ，就可以使用 JNIEnv
指针的相应函数来读取或修改字段的值。根据字段的类型，有不同的函数可供选择，例如：

- GetIntField 和 SetIntField ：用于访问 int 类型的字段。
- GetBooleanField 和 SetBooleanField ：用于访问 boolean 类型的字段。
- GetObjectField 和 SetObjectField ：用于访问对象引用的字段。
- ...等等，对于每种类型都有对应的访问函数。

示例代码：访问Java对象字段

假设我们有一个Java类 Person ，它有两个字段：一个 String 类型的 name 和一个 int 类型的 age 。

```
public class Person {
    public String name;
    public int age;

    // ... 构造方法和其他方法 ...
}
```

在C/C++代码中访问 Person 对象字段的示例：

```
JNIEXPORT void JNICALL Java_ClassName_accessPersonFields(JNIEnv *env, jobject obj) {
    // 获取Person类的引用
    jclass personClass = (*env)->GetObjectClass(env, obj);

    // 获取字段ID
    jfieldID nameField = (*env)->GetFieldID(env, personClass, "name", "Ljava/lang/String;");
    jfieldID ageField = (*env)->GetFieldID(env, personClass, "age", "I");

    if (nameField == NULL || ageField == NULL) {
        return; // 字段未找到
    }

    // 读取字段值
    jstring name = (*env)->GetObjectField(env, obj, nameField);
    jint age = (*env)->GetIntField(env, obj, ageField);

    // 打印字段值
    // ... 使用JNI函数将jstring转换为C字符串并打印 ...

    // 修改字段值
    jstring newName = (*env)->NewStringUTF(env, "New Name");
    (*env)->SetObjectField(env, obj, nameField, newName);
    (*env)->SetIntField(env, obj, ageField, 30);
}
```

在这个示例中，我们首先获取了 Person 类的引用，然后获取了 name 和 age 字段的ID。接着，我们使用 GetObjectField 和 GetIntField 函数读取字段的值，并使用

SetObjectField 和 SetIntField 函数修改字段的值。

访问Java对象字段时需要注意以下几点：

- 确保字段名称和签名正确无误。
- 处理可能出现的异常，例如字段未找到或非法访问。
- 在访问字段后，如果创建了新的对象引用（如使用 NewStringUTF ），记得在不需要时释放这些引用。

通过上述方法，JNI允许本地代码访问和修改Java对象的字段，从而实现了与Java代码的紧密协作。这是JNI强大功能的体现之一，它允许Java和本地代码共同操作对象的属性。

调用Java对象方法

在JNI中，除了访问Java对象的字段外，还经常需要调用Java对象的方法。这一过程与调用普通Java方法类似，但需要通过JNI特定的函数和机制来实现。

获取MethodID

要调用Java对象的方法，首先需要获取该方法的 MethodID 。这可以通过调用 JNIEnv 指针的 GetMethodID 函数来完成。 GetMethodID 函数需要三个参数：

1. jclass：表示包含方法的Java类的类引用。
2. 方法名称：要调用的方法的名称。
3. 方法签名：方法的签名，描述了方法的参数类型和返回类型。

方法签名的格式与字段签名类似，但用于描述方法。例如，一个接受两个参数（一个 int 和一个 String ）且没有返回值的方法的签名是`(ILjava/lang/String;)V`。

调用方法

一旦获取了方法的 MethodID ，就可以使用 JNIEnv
指针的相应函数来调用该方法。根据方法的返回类型，有不同的函数可供选择，例如：

- CallVoidMethod ：调用无返回值的方法。
- CallObjectMethod ：调用返回对象引用的方法。
- CallIntMethod ：调用返回 int 类型的方法。
- ...等等，对于每种返回类型都有对应的调用函数。

示例代码：调用Java对象方法

假设我们有一个Java类 Calculator ，它有一个方法 add ，接受两个 int 参数并返回它们的和。

```
public class Calculator {
    public int add(int a, int b) {
```

```
        return a + b;
    }
}
```

在C/C++代码中调用 Calculator 对象的 add 方法的示例：

```
JNIEXPORT jint JNICALL Java_ClassName_callAddMethod(JNIEnv *env, jobject obj, jint a, jint b) {
    // 获取Calculator类的引用
    jclass calculatorClass = (*env)->GetObjectClass(env, obj);

    // 获取add方法的ID
    jmethodID addMethod = (*env)->GetMethodID(env, calculatorClass, "add", "(II)I");
    if (addMethod == NULL) {
        return -1; // 方法未找到
    }

    // 调用add方法
    jint result = (*env)->CallIntMethod(env, obj, addMethod, a, b);
    return result;
}
```

在这个示例中，我们首先获取了 Calculator 类的引用，然后获取了 add 方法的ID。接着，我们使用 CallIntMethod 函数调用了 add 方法，并返回了结果。

调用Java对象方法时需要注意以下几点：

- 确保方法名称和签名正确无误。
- 处理可能出现的异常，例如方法未找到或非法访问。
- 如果方法有返回值，确保使用正确的调用函数来接收返回值。
- 如果方法抛出异常，可以使用JNI的异常处理函数来检查和处理这些异常。

通过上述方法，JNI允许本地代码调用Java对象的方法，从而实现了与Java代码的紧密协作。这是JNI强大功能的体现之一，它使得本地代码能够利用Java对象提供的各种方法和功能。