# 《深入理解 JNI》JNI 高级特性

## 全局引用与弱全局引用

在JNI编程中，管理对象引用的生命周期是非常重要的。JNI提供了几种不同类型的引用，以适应不同的使用场景。其中，全局引用（Global Reference）和弱全局引用（Weak Global Reference）是两种常用的引用类型。

### 全局引用（Global Reference）

全局引用是JNI中最强的引用类型。一旦一个Java对象被全局引用指向，它就不会被垃圾收集器回收，即使所有其他的引用都已经超出作用域。全局引用通常用于以下场景：

- 当本地代码需要在多个方法调用之间保持对Java对象的引用时。
- 当本地代码需要将Java对象传递给其他线程时。

创建全局引用的函数是`JNIEnv`的`NewGlobalRef`。使用全局引用时需要注意，因为它们会阻止对象被垃圾回收，所以必须谨慎管理，避免内存泄漏。

示例代码：创建和使用全局引用

```
JNIEXPORT void JNICALL Java_ClassName_createGlobalRef(JNIEnv *env, jobject obj) {
    // 创建全局引用
    jobject globalRef = (*env)->NewGlobalRef(env, obj);
    if (globalRef == NULL) {
        return; // 创建失败
    }

    // ... 使用全局引用 ...

    // 使用完毕后删除全局引用
    (*env)->DeleteGlobalRef(env, globalRef);
}
```

### 弱全局引用（Weak Global Reference）

弱全局引用是一种较弱的全局引用，它允许垃圾收集器回收被引用的Java对象。弱全局引用通常用于以下场景：

- 当本地代码需要长期保持对Java对象的引用，但又不想阻止对象被垃圾回收时。

- 当需要缓存对象引用，但又希望缓存不会阻止对象被回收时。

创建弱全局引用的函数是`JNIEnv`的`NewWeakGlobalRef`。使用弱全局引用时，需要注意检查对象是否已经被回收。

示例代码：创建和使用弱全局引用

```
JNIEXPORT void JNICALL Java_ClassName_createWeakGlobalRef(JNIEnv *env, jobject obj) {
    // 创建弱全局引用
    jobject weakGlobalRef = (*env)->NewWeakGlobalRef(env, obj);
    if (weakGlobalRef == NULL) {
        return; // 创建失败
    }

    // ... 使用弱全局引用 ...

    // 检查对象是否已被回收
    jobject ref = (*env)->NewLocalRef(env, weakGlobalRef);
    if (ref != NULL) {
        // 对象仍然存在
        (*env)->DeleteLocalRef(env, ref);
    } else {
        // 对象已被回收
    }

    // 删除弱全局引用
    (*env)->DeleteWeakGlobalRef(env, weakGlobalRef);
}
```

### 注意事项

- 全局引用和弱全局引用都需要手动删除，以避免内存泄漏。
- 使用全局引用时要特别小心，确保不会因为忘记删除而导致内存泄漏。
- 弱全局引用虽然允许对象被垃圾回收，但在对象被回收后，相关的弱全局引用会自动失效，需要重新获取对象引用。

通过使用全局引用和弱全局引用，JNI提供了一种灵活的方式来管理Java对象的生命周期，使得本地代码可以在需要时保持对对象的引用，同时又不阻碍垃圾收集器的正常工作。

## 异常处理

在JNI编程中，处理异常是一个重要的环节。Java代码可能会抛出异常，而这些异常需要在本地代码中进行适当的处理。JNI提供了一组函数来检查和处理Java异常。

## 检查异常

当Java方法调用发生异常时，相关的JNI函数会返回错误代码，并且可以通过`JNIEnv`指针的`ExceptionOccurred`函数来检查是否有异常发生。

```
jthrowable exception = (*env)->ExceptionOccurred(env);
if (exception != NULL) {
    // 异常发生
}
```

## 清除异常

如果检测到异常，可以使用`ExceptionClear`函数来清除异常状态。这通常在处理完异常后进行，以便JNI调用链可以继续正常执行。

```
(*env)->ExceptionClear(env);
```

## 抛出异常

JNI允许本地代码抛出Java异常。可以使用`ThrowNew`函数来抛出一个新的异常。

```
jclass exceptionClass = (*env)->FindClass(env, "java/lang/RuntimeException");
if (exceptionClass != NULL) {
    (*env)->ThrowNew(env, exceptionClass, "An error occurred in native code");
}
```

## 示例代码：JNI中的异常处理

以下是一个简单的JNI函数示例，展示了如何检查和处理Java异常：

```
JNIEXPORT void JNICALL Java_ClassName_nativeMethod(JNIEnv *env, jobject obj) {
    // ... 执行一些操作 ...

    // 检查是否有异常发生
    if ((*env)->ExceptionOccurred(env)) {
        // 处理异常
        (*env)->ExceptionDescribe(env); // 打印异常信息到标准错误输出
```

```
        (*env)->ExceptionClear(env); // 清除异常状态

        // 可以选择抛出自定义异常或执行其他错误处理逻辑
        jclass exceptionClass = (*env)->FindClass(env, "java/lang/RuntimeException");
        if (exceptionClass != NULL) {
            (*env)->ThrowNew(env, exceptionClass, "An error occurred in native code");
        }
    }

    // ... 继续执行其他操作 ...
}
```

在本地代码中处理异常时需要注意以下几点：

- 不要忽略异常。如果Java代码抛出异常，本地代码应该检查并适当地处理它。
- 清除异常后，应该谨慎地决定程序的下一步行动，因为异常可能会导致对象处于不一致的状态。
- 抛出异常时，确保提供有意义的错误信息，以便Java层的代码可以理解和处理这些异常。

通过JNI提供的异常处理机制，本地代码可以与Java层的异常处理逻辑协同工作，确保程序的健壮性和稳定性。
以下是对第五部分第三小节“多线程与JNI”内容的扩充：

# 多线程与JNI

## 多线程支持概述

- JNI中的多线程模型：
    - 在JNI环境下，Java虚拟机（JVM）本身是支持多线程运行的。这意味着可以有多个Java线程同时存在并执行。然而，对于本地代码（C/C++等）与JVM之间的交互，在多线程场景下有特殊的规则和要求。
    - JVM的多线程模型基于操作系统的线程实现，例如在类Unix系统中通常基于pthread库。这种基于底层操作系统线程构建的方式使得JVM能够充分利用多核处理器的性能优势。
- 线程创建与管理：
    - 在JNI环境中创建线程，可以使用多种方式。如果是在Android平台上，可以利用Android提供的线程创建机制，如`pthread_create`函数（遵循POSIX线程标准）。在其他平台或者更通用的场景下，也可以使用C++11中的`std::thread`库来创建线程。
    - 当创建一个新线程时，需要考虑线程的生命周期管理。这包括线程启动时的初始化工作，例如设置线程特定的数据（如果需要），以及在线程结束时进行资源的清理和释放。对于JNI相关的线程，特别要注意与JVM的正确交互，确保不会出现

资源泄漏或者未定义的行为。

## AttachCurrentThread与DetachCurrentThread

- AttachCurrentThread：
  - `AttachCurrentThread`函数是JNI中用于将当前本地线程附着到JVM的关键函数。它的作用是让JVM识别这个本地线程，从而使得该线程能够调用JNI函数。
  - 在调用`AttachCurrentThread`时，需要传递一个指向`JNIEnv`指针的指针（通常为`JNIEnv **`类型）以及一些可选的线程属性参数。JVM会为这个线程分配一个`JNIEnv`实例，并将其地址通过传入的指针返回。
  - 例如，在C代码中可能的调用方式如下：

```
JavaVM *jvm;
JNIEnv *env;
// 假设jvm已经被正确初始化
(*jvm)->AttachCurrentThread(jvm, &env, NULL);
```

- 这个函数必须在本地线程首次调用任何JNI函数之前被调用，否则会导致未定义的行为。
  - DetachCurrentThread：
  - 当本地线程完成与JVM相关的操作后，必须调用`DetachCurrentThread`函数将线程从JVM中分离。这是因为JVM会为附着线程维护一些内部状态信息，如果不进行分离，可能会导致内存泄漏等问题。
  - 分离线程的操作相对简单，只需要传入对应的`JavaVM`指针即可。例如：

```
(*jvm)->DetachCurrentThread(jvm);
```

一旦线程被分离，就不能再在该线程中调用JNI函数，除非再次调用`AttachCurrentThread`重新附着。

## 线程局部存储与JNIEnv

- 线程局部存储（TLS）：
  - JNI使用线程局部存储来管理每个线程的`JNIEnv`指针。这意味着每个线程都有自己独立的`JNIEnv`实例，它们之间互不干扰。
  - TLS是一种机制，允许每个线程拥有其自己的数据副本。在JNI中，这种机制确保了当多个线程同时运行时，不会出现`JNIEnv`指针的混淆或者冲突。
  - 从实现角度来看，在底层操作系统或者JVM内部，通过特定的数据结构和算法来维护每个线程与它的`JNIEnv`指针之间的映射关系。

JNIEnv的使用限制：

- 强调`JNIEnv`指针具有很强的线程局限性，它只能在创建它的线程中使用。这是因为`JNIEnv`指针指向的是与特定线程相关的JVM内部数据结构和函数表。
- 如果试图将一个线程的`JNIEnv`指针传递给另一个线程并使用，会导致不可预测的结果，如程序崩溃或者数据损坏。

## 同步与线程安全

- 同步机制：
  - 在多线程环境中，为了确保数据的一致性和正确性，需要使用同步机制。在JNI编程中，可以利用Java层的`synchronized`关键字来实现对象级别的同步。例如，在Java代码中对共享对象的方法进行同步声明，在本地代码中调用这些同步方法时，就会遵循Java的同步规则。
  - 同时，在C/C++ 层面，可以使用互斥锁（mutex）来实现更细粒度的同步控制。例如，在访问共享资源（如全局变量或者文件句柄）之前，先获取互斥锁，访问结束后释放锁。这样可以防止多个线程同时访问共享资源导致的竞态条件。
  - 除了互斥锁，还可以使用条件变量（condition variable）来实现线程间的通信和协调。例如，一个线程可以等待某个条件满足（通过条件变量），而另一个线程在满足条件后通知等待的线程继续执行。
- 线程安全：
  - 在JNI编程中确保线程安全是一个综合性的任务。除了正确使用同步机制外，还需要注意数据的初始化和清理顺序。例如，在多个线程中共享的Java对象或者本地数据结构，要确保在所有线程开始访问之前已经正确初始化，并且在所有线程都不再使用之后进行彻底的清理。
  - 另外，对于JNI函数本身的调用也要遵循线程安全的规则。有些JNI函数可能在内部有特定的线程假设或者限制，需要仔细阅读JNI文档并正确使用。

## 示例代码

- 多线程示例：
  - 以下是一个简单的JNI多线程示例，展示了如何创建线程、附着到JVM、调用JNI函数以及分离线程。

```
#include <jni.h>
#include <pthread.h>

JavaVM *jvm;

void *nativeThreadFunc(void *arg) {
    JNIEnv *env;
    // 附着线程到JVM
    (*jvm)->AttachCurrentThread(jvm, &env, NULL);

    // 这里可以进行JNI相关的操作，例如调用Java方法或者访问Java对象
    // 假设已经有一个Java类的引用jclass cls和一个方法ID mid
    jobject obj = // 获取Java对象的方式
```

```
    (*env)->CallVoidMethod(env, obj, mid);

    // 分离线程
    (*jvm)->DetachCurrentThread(jvm);
    return NULL;
}

JNIEXPORT void JNICALL Java_ClassName_nativeMethod(JNIEnv *env, jobject thiz) {
    // 获取JavaVM指针
    (*env)->GetJavaVM(env, &jvm);

    pthread_t thread;
    // 创建线程
    pthread_create(&thread, NULL, nativeThreadFunc, NULL);
    // 等待线程结束（这里可以根据实际需求决定是否等待）
    pthread_join(thread, NULL);
}
```