第 5 章

NDK 开发常用的数据类型及使用方法

经过前 4 章的系统学习，读者应当已经能够熟练地创建并配置一个满足项目需求的 NDK 工程。在此基础上，本章将聚焦于 NDK 开发中的数据类型及其使用技巧。深入理解这些知识点，对于实现 Java 应用程序与原生代码之间的顺畅通信至关重要。

5.1 基础类型说明

基本数据类型在 JNI 中可以直接与 C/C++的基本数据类型相对应。为了实现这种映射关系，JNI 使用 typedef 来定义这些基本类型之间的对应关系。Java 与 Native 之间的数据类型映射关系见表 5-1。

表 5-1 基本数据类型

Java	JNI	C/C++	大小
boolean	jboolean	uint8_t	无符号 8 位
byte	jbyte	int8_t	有符号 8 位
char	jchar	uint16_t	无符号 16 位
short	jshort	int16_t	有符号 16 位
int	jint	int32_t	有符号 32 位
long	jlong	int64_t	有符号 64 位
float	jfloat	float	32 位
double	jdouble	double	64 位

基本类型在 jni.h 文件中的定义，代码如下：

```
//第 5 章/jni.h
/* Primitive types that match up with Java equivalents. */
typedef uint8_t  jboolean; /* unsigned 8 bits */
typedef int8_t   jbyte;    /* signed 8 bits */
```

```
typedef uint16_t jchar;    /* unsigned 16 bits */
typedef int16_t  jshort;   /* signed 16 bits */
typedef int32_t  jint;     /* signed 32 bits */
typedef int64_t  jlong;    /* signed 64 bits */
typedef float    jfloat;   /* 32-bit IEEE 754 */
typedef double   jdouble;  /* 64-bit IEEE 754 */

/* "cardinal indices and sizes" */
typedef jint    jsize;
```

5.2 引用类型说明

关于引用类型，在 Java 语言中所有类都继承自 java.lang.Object，然而，C 语言并未引入类和对象的概念，并且 Java 类的内部结构在原生代码层面并未直接暴露，因此，在 C 语言中，普遍使用 void*代替任意类型，它作为一个通用指针类型，能够指向任意类型的对象，而在 C++中，为了模拟 Java 中的类，定义了一个空的类 class Object {};，作为原生代码中的占位符，以此来与 Java 中的对象概念相对应。由于 C 和 C++的实现并不相同，所以 C 和 C++对应的实际类型是独立的，详细见表 5-2。

表 5-2 引用数据类型

Java	JNI	C	C++
java.lang.Class	jclass	jobject	_jclass *
java.lang.Throwable	jthrowable	jobject	_jthrowable*
java.lang.String	jstring	jobject	_jstring *
Other objects	jobject	void *	_jobject*
java.lang.Object[]	jobjectArray	jarray	_jobjectArray*
boolean[]	jbooleanArray	jarray	_jbooleanArray*
byte[]	jbyteArray	jarray	_jbyteArray*
char[]	jcharArray	jarray	_jcharArray*
short[]	jshortArray	jarray	_jshortArray*
int[]	jintArray	jarray	_jintArray*
long[]	jlongArray	jarray	_jlongArray*
float[]	jfloatArray	jarray	_jfloatArray*
double[]	jdoubleArray	jarray	_jdoubleArray*
Other arrays	Jarray	jarray	jarray*

5.2.1 C 语言下的引用类型

C 语言中对引用数据类型的定义，代码如下：

```
/*
 * Reference types, in C.
 */
typedef void*         jobject;
typedef jobject       jclass;
typedef jobject       jstring;
typedef jobject       jarray;
typedef jarray        jobjectArray;
typedef jarray        jbooleanArray;
typedef jarray        jbyteArray;
typedef jarray        jcharArray;
typedef jarray        jshortArray;
typedef jarray        jintArray;
typedef jarray        jlongArray;
typedef jarray        jfloatArray;
typedef jarray        jdoubleArray;
typedef jobject       jthrowable;
typedef jobject       jweak;
```

5.2.2　C++语言下的引用类型

C++语言中对引用数据类型的定义，代码如下：

```
/*
 * Reference types, in C++
 */
class _jobject {};
class _jclass : public _jobject {};
class _jstring : public _jobject {};
class _jarray : public _jobject {};
class _jobjectArray : public _jarray {};
class _jbooleanArray : public _jarray {};
class _jbyteArray : public _jarray {};
class _jcharArray : public _jarray {};
class _jshortArray : public _jarray {};
class _jintArray : public _jarray {};
class _jlongArray : public _jarray {};
class _jfloatArray : public _jarray {};
class _jdoubleArray : public _jarray {};
class _jthrowable : public _jobject {};

typedef _jobject*       jobject;
typedef _jclass*        jclass;
typedef _jstring*       jstring;
typedef _jarray*        jarray;
typedef _jobjectArray*  jobjectArray;
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 _jthrowable*    jthrowable;
typedef _jobject*       jweak;
```

typedef 是一种为数据类型定义别名的机制，它有助于增强代码的可读性和可维护性。从上述代码来看，除了基本数据类型外，在引用类型的处理上，C 语言最终将其视为 void*，而 C++则将其视为_jobject*。尽管全部统一使用 void*或_jobject*作为表示是可行的，但分别使用不同的类型别名是为了提高代码的可读性和可扩展性。这样做有助于开发者更好地理解代码中指针的用途，以及它们所指向的对象类型，同时也为未来可能的类型扩展提供了更高的灵活性，因此，这种类型别名的使用方式不仅符合编程规范，也体现了良好的编程实践。

5.3 UTF-8 和 UTF-16 字符串

在讲解数据类型的操作函数之前，首先需要对字符集进行一番探讨。之所以要专门讲解字符集，是因为不同的编程语言和平台可能采用不同的字符集，这直接影响了字符类型在内存中的占用空间及字符的编码方式。

特别地，我们注意到基本类型中的 char 在 Java 和 C/C++中的表现是有所不同的。对于熟悉 C 语言的读者来讲，知道 C 语言中的 char 类型通常占用一字节的空间，用于存储 ASCII 字符集中的字符，然而，在 Java 语言中，char 类型却占用了两字节的空间。同样地，JNI 中定义的 jchar 类型也占用了两字节。

这种差异的存在，主要是因为 Java 和 C/C++使用了不同的字符集。Java 采用的是 Unicode 字符集，这是一个能够涵盖世界上绝大多数语言和字符的宽字符集。由于 Unicode 字符集中包含了大量的字符，每个字符需要更多的空间来进行编码，因此 Java 中的 char 类型需要两字节来存储一个 Unicode 字符，而 C 语言则通常使用 ASCII 字符集，这是一个只包含基本拉丁字母和符号的较小字符集，因此其 char 类型只需一字节就能存储一个 ASCII 字符。

因此，在使用 jchar 时，需要格外注意字符编码的问题。如果 Java 中的字符串仅包含 ASCII 字符，则可以直接将 jchar 转换为 char 使用，因为 ASCII 字符在 Unicode 中的表示与 ASCII 字符集中的表示是一致的，然而，如果 Java 字符串包含非 ASCII 字符（如汉字），则需要谨慎处理字符编码的转换问题。通常，可以使用 Java 中的 String 类的 getBytes 方法，将字符串转换为指定字符集（如 UTF-8）的字节数组，然后在 C/C++中利用相应的字符集函数将这些字节数组转换为 char 数组。

综上所述，了解 Java 和 C/C++在字符集处理上的差异，对于两者之间进行数据类型转换和通信至关重要。接下来，我们将探讨常用数据类型的操作函数，以便更高效地实现 Java

与原生代码之间的交互。

5.4　常用数据类型操作函数的使用

5.4.1　String 字符串的使用

1. 字符串创建

在原生代码（C 或 C++代码）中，可以使用 NewString()函数来创建一个采用 Unicode 编码格式的字符串实例。使用 NewStringUTF()函数则用于创建采用 UTF-8 编码格式的字符串实例。这两个函数都接受一个 C 语言风格的字符串（C 字符串）作为参数，并返回一个 Java 字符串的引用类型，即 jstring 类型的值。

使用 C 语言创建字符串实例，代码如下：

```
//创建一个 Unicode 编码格式的字符串实例
jstring str = (*env)->NewString(env, "hello world");
//创建一个 UTF-8 编码格式的字符串实例
jstring str = (*env）->NewStringUTF(env, "hello world");
```

使用 C++语言创建字符串实例，代码如下：

```
//创建一个 Unicode 编码格式的字符串实例
jstring str = env->NewString("hello world");
//创建一个 UTF-8 编码格式的字符串实例
jstring str = env->NewStringUTF("hello world");
```

细心观察，使用 C 语言创建字符串和使用 C++创建字符串略有不同，除 C 语言版本在参数上多了一个 env 参数外，还有 env 的使用方式不同。这主要取决于在 C/C++中对于 JNIEnv 定义的不同。以一个普通的 JNI 函数代码举例，第 1 个参数为 JNIEnv *env，代码如下：

```
JNIEXPORT void JNICALL
Java_com_example_javap_TestJni_test(JNIEnv *env, jobject thiz) {

}
```

在 JNI 的头文件 jni.h 中，JNIEnv 分为 C 语言和 C++语言定义两种版本，代码如下：

```
//_JNIEnv 定义
struct _JNIEnv {
const struct JNINativeInterface* functions;
//…省略
};

#if defined(__cplusplus)    //C++中的实现
typedef _JNIEnv JNIEnv;
#else
```

```
typedef const struct JNINativeInterface* JNIEnv;   //C 语言中的实现
#endif
```

在 C 语言中，JNIEnv 的定义实际上是指向 struct JNINativeInterface 的指针，即 JNIEnv 等同于 struct JNINativeInterface*，因此，在 C 语言环境中，当我们看到一个 JNIEnv* env 作为函数参数时，env 实际上是一个指向指针的指针，也就是一个双重指针。这意味着在使用 env 所指向的结构体中的函数或成员时，需要进行解引用操作。

然而，在 C++中，情况有所不同。在 C++中，JNIEnv 通常被定义为_JNIEnv 结构的一个别名，而_JNIEnv 结构内部包含了一个指向 struct JNINativeInterface 的指针，因此，在 C++的上下文中，当 JNIEnv* env 作为参数传递时，env 是一个指向_JNIEnv 结构的指针，而该结构内部已经包含了指向 JNINativeInterface 的指针，所以，在 C++中，可以直接使用指针操作。

除了上述内容外，值得注意的是，C++对 JNIEnv 进行了封装处理。在其内部结构中，它保留了一个指向 JNINativeInterface 结构体的指针，因此，在 C++中调用 JNI 接口函数时，JNINativeInterface 会被直接作为函数的第 1 个参数传入，这就意味着在 C++环境中调用 JNI 接口时，开发者无须再次显式地传入 env 作为函数的第 1 个参数，而在后续的讲解中，将更多地侧重于函数本身的定义及在 C 语言环境下的使用方式。

2. 获取字符串 UTFChars

在原生代码（C 或 C++编写的代码）中，当需要与 Java 端的字符串进行交互时，可以使用 GetStringUTFChars 函数来获取 Java 字符串的 UTF-8 编码表示。该函数返回一个指向字节数组的指针，该字节数组表示采用修改后的 UTF-8 编码的字符串。该数组在使用 ReleaseStringUTFChars 函数释放之前一直有效。

接口定义，代码如下：

```
/**
 * 获取字符串 UTFChars
 * @param env      JNI 接口指针
 * @param string    Java 字符串对象
 * @param isCopy    指向布尔值的指针
 * @return 返回指向修改后的 UTF-8 字符串的指针，如果操作失败，则返回 NULL
 */
const char * GetStringUTFChars(JNIEnv *env, jstring string,jboolean *isCopy);
```

isCopy 是一个传出参数，在不为 NULL 的情况下，如果指针指向 Java 字符串的副本，则*isCopy 会被设置为 JNI_TRUE；如果*isCopy 被赋值为 JNI_FALSE，则说明直接指向 Java 字符串；如果不关心是否复制，则直接传入 NULL 即可。

接口函数的使用，示例代码如下：

```
jstring str;
jboolean isCopy;
const char *cString = (*env)->GetStringUTFChars(env, str, &isCopy);
  if (NULL == cString) {
```

```
        LOGE("获取 C 字符串失败\n");
    }
    if (isCopy == JNI_FALSE){
        LOGE("cString 指向 Java 字符串");
    }else{
        LOGE("cString 指向 Java 字符串的副本");
    }
```

3. 释放字符串 UTFChars

使用 GetStringUTFChars 获取的字符串必须主动释放，否则会造成内存泄漏。

接口定义，代码如下：

```
/**
 * 释放字符串 UTFChars
 * @param env    JNI 接口指针
 * @param string Java 字符串对象
 * @param utf    指向修改后的 UTF-8 字符串的指针
 */
void ReleaseStringUTFChars(JNIEnv *env, jstring string, const char *utf)
```

接口函数的使用，示例代码如下：

```
jstring javaString;
const char * cString;
(*env)->ReleaseStringUTFChars(env, javaString, cString);
```

4. 获取字符串长度

使用 GetStringLength()函数获取 Java 字符串长度，该函数有一个参数 jstring，指向 Java 的字符串。返回字符串长度 jsize（int）。

接口定义，代码如下：

```
/**
 * 获取字符串长度
 * @param env    JNI 接口指针
 * @param string Java 字符串对象
 * @return 返回 Java 字符串的长度
 */
jsize GetStringLength(JNIEnv *env, jstring string);
```

接口函数的使用，示例代码如下：

```
jstring javaString;
jsize length = (*env)->GetStringLength(env, javaString);
```

5.4.2　数组操作

JNI 把 Java 数组当作引用类型来处理，和基本类型一样，JNI 也提供了对 Java 数组进行

处理的函数，以下对数组的操作以 Int 数组举例。

1. 创建数组

使用 New<Type>Array()函数在原生代码中创建数组，其中<Type>可以是基本类型中的任意一种，例如 NewIntArray()。该函数接受一个描述数组大小的参数，与 NewString()函数一样，在失败时返回 NULL。

接口定义，代码如下：

```
/**
 * 创建一个新的 Java 数组
 * @param env    JNI 接口指针
 * @param length 数组长度
 * @return 返回一个 Java 数组，如果无法构造该数组，则返回 NULL
 */
ArrayType New<Type>Array(JNIEnv *env, jsize length);
```

接口函数的使用，示例代码如下：

```
//创建一个 Java Int 数组
jintArray javaArray;
javaArray = (*env)->NewIntArray(env, 10);
if (NULL != javaArray){
   LOGE("可以操作数组了");
}
```

2. 获取 Java 数组

JNI 提供了两种访问 Java 数组元素的方法，可以将 Java 数组复制成 C 数组进行操作或者直接返回指向 Java 数组的指针。

1）操作数组副本

使用 Get<Type>ArrayRegion()函数将给定的 Java 数组复制到给定的 C 数组中。

接口函数的定义，代码如下：

```
/**
 * 将给定的 Java 数组复制到给定的 C 数组中
 * @param env     JNI 接口指针
 * @param array   Java 数组
 * @param start   起始索引
 * @param len     要复制的元素数量
 * @param buf     目标缓冲区
 */
void GetIntArrayRegion(JNIEnv *env, jintArray array, jsize start, jsize len,
jint *buf)
```

接口函数的使用，示例代码如下：

```
//定义一个本地数组
jint nativeArray[10];
```

```
//从第 0 个元素开始，将 10 个 Java 数组中的元素复制到本地数组中
(*env)->GetIntArrayRegion(env, javaArray, 0, 10, nativeArray);
```

当使用 Get<Type>ArrayRegion()函数时，需要确保提供的本地数组有效且足够大，以便存储从 Java 数组中提取的元素。如果 start 和 len 参数指定的范围超出了数组的实际大小，则 JNI 将抛出异常。复制成功后，原生代码就可以像使用普通的 C 数组一样使用和修改。

当原生代码想将所作的修改提交给 Java 数组时，可以使用 Set<Type>ArrayRegion()函数将 C 数组复制回 Java 数组，示例代码如下：

```
jintArray javaArray;
jint nativeArray[10];

//从第 0 个元素开始，将 10 个 Java 数组中的元素复制到本地数组中
(*env)->GetIntArrayRegion(env, javaArray, 0, 10, nativeArray);
  //数组操作
  for (int i = 0; i < 10; ++i) {
        nativeArray[i] = nativeArray[i] + 10;
      }
  //将修改后的数组提交到 Java 数组
(*env)->SetIntArrayRegion(env, javaArray, 0, 10, nativeArray);
```

2）操作数组指针

使用 Get<Type>ArrayElements()函数获取指向数组的元素的直接指针。在调用 Release<Type>ArrayElements()函数之前，返回的指针一直有效。由于返回的数组可能是 Java 数组的副本，因此在调用 Release<Type>ArrayElements()函数之前，对返回数组所做的更改不一定会反映到原始 Java 数组中。

接口函数的定义，代码如下：

```
/**
 * 获取指向数组的元素的指针
 * @param env              JNI 接口指针
 * @param javaArray        Java 数组
 * @param isCopy           指向布尔值的指针
 * @return   返回指向数组元素的指针，如果操作失败，则返回 NULL
 */
jint* GetIntArrayElements(JNIEnv* env, jintArray javaArray, jboolean* isCopy);


/**
 * 释放指向 Java 数组元素的直接指针
 * @param env     JNI 接口指针
 * @param array   Java 数组对象
 * @param elems   指向数组元素的指针
 * @param mode    释放模式
```

```
    */void Release<Type>ArrayElements(JNIEnv *env, ArrayType array Type *elems,
jint mode);
```

接口函数的使用，示例代码如下：

```
jboolean isCopy;
jint * cArray = (*env)->GetIntArrayElements(env, javaArray, &isCopy);
if (cArray == NULL){
   LOGE("java 数组获取失败");
   return ;
}
if (isCopy == JNI_TRUE){
   LOGE("数组指针指向 Java 数组副本");
}else if (isCopy == JNI_FALSE){
   LOGE("Java 数组直接指向 Java 数组");
}
//释放指向 Java 数组元素的直接指针
(*env)->ReleaseIntArrayElements(env, javaArray, cArray, 0);
```

值得注意的是 Release<Type>ArrayElements()函数的最后一个参数 mode。该 mode 参数提供有关如何释放数组缓冲区的信息，如果 elems 不是数组中元素的副本，则 mode 无效，否则 mode 会有以下影响，详细见表 5-3。

表 5-3　数组释放模式

mode	影　响
0	将内容复制到 Java 数组并释放 elems 缓冲区
JNI_COMMIT	将内容复制到 Java 数组但不释放 elems 缓冲区
JNI_ABORT	释放缓冲区而不将可能的更改复制到 Java 缓冲区

在大多数情况下，开发者将 0 传递给参数 mode 以确保固定数组和复制数组的行为一致。虽然其他选项可以使程序员更好地控制内存管理，但在使用时应格外小心。

注意：从 JDK/JRE 1.1 开始，程序员可以使用 Get/Release<Type>ArrayElements()函数来获取指向原始数组元素的指针。如果 VM 支持 pinning，则返回指向原始数据的指针，否则将制作一份副本。

3. 获取 Java 端数组的直接指针

从 JDK/JRE 1.3 开始引入的新函数允许本机代码获取指向数组元素的直接指针，即使 VM 不支持 pinning 也是如此。

接口函数的定义，代码如下：

```
/**
 *  获取指向数组的元素的直接指针
 * @param env     JNI 接口指针
```