

本章涉及的源代码文件名称及位置

下面是本章分析的源码文件名和它的位置。

- ❑ `RefBase.h`(`framework/base/include/utils/RefBase.h`)
- ❑ `RefBase.cpp`(`framework/base/libs/utils/RefBase.cpp`)
- ❑ `Thread.cpp`(`framework/base/libs/utils/Thread.cpp`)
- ❑ `Thread.h`(`framework/base/include/utils/Thread.h`)
- ❑ `Atomic.h`(`system/core/include/cutils/Atomic.h`)
- ❑ `AndroidRuntime.cpp`(`framework/base/core/jni/AndroidRuntime.cpp`)
- ❑ `Looper.java`(`framework/base/core/Java/Android/os/Looper.java`)
- ❑ `Handler.java`(`framework/base/core/Java/Android/os/ Handler.java`)
- ❑ `HandlerThread.java`(`framework/base/core/Java/Android/os/ HandlerThread.java`)

5.1 概述

初次接触 Android 源码时，见到最多的一定是 sp 和 wp。即使你只是沉迷于 Java 世界的编码，那么 Looper 和 Handler 也是避不开的。本章的目的，就是把经常碰到的这些内容中的“拦路虎”一网打尽，将它们彻底搞懂。至于弄明白它们有什么好处，就仁者见仁，智者见智了。个人觉得 Looper 和 Handler 相对会更实用一些。

5.2 以“三板斧”揭秘 RefBase、sp 和 wp

RefBase 是 Android 中所有对象的始祖，类似于 MFC 中的 CObject 及 Java 中的 Object 对象。在 Android 中，RefBase 结合 sp 和 wp，实现了一套通过引用计数的方法来控制对象生命周期的机制。就如我们想像的那样，这三者的关系非常暧昧。初次接触 Android 源码的人往往会被那个随处可见的 sp 和 wp 搞晕了头。


什么是 sp 和 wp 呢？其实，sp 并不是我开始所想的 smart pointer（C++ 语言中有这个东西），它真实的意思应该是 strong pointer，而 wp 则是 weak pointer 的意思。我认为，Android 推出这一套机制可能是模仿 Java，因为 Java 世界中有所谓 weak reference 之类的东西。sp 和 wp 的目的，就是为了帮助健忘的程序员回收 new 出来的内存。

说明 我还是喜欢赤裸裸地管理内存的分配和释放。不过，目前 sp 和 wp 的使用已经深入到 Android 系统的各个角落，想把它去掉真是不太可能了。

这三者的关系比较复杂，都说程咬金的“三板斧”很厉害，那么我们就借用这三板斧，揭密其间的暧昧关系。

5.2.1 第一板斧——初识影子对象

我们的“三板斧”，其实就是三个例子。相信这三板斧劈下去，你会很容易理解它们。

 [--> 例子 1]

```
// 类 A 从 RefBase 派生，RefBase 是万物的始祖。
class A: public RefBase
{
    //A 没有任何自己的功能。
}
int main()
{
    A* pA = new A;
    {
        // 注意我们的 sp、wp 对象是在 {} 中创建的，下面的代码先创建 sp，然后创建 wp。
        sp<A> spA(pA);
        wp<A> wpA(spA);
    }
}
```

```

        // 大括号结束前，先析构 wp，再析构 sp。
    }
}

```

例子够简单吧？但也需一步一步分析这斧子是怎么劈下去的。

1. RefBase 和它的影子

类 A 从 RefBase 中派生。使用的是 RefBase 构造函数。代码如下所示：

👉 [-->RefBase.cpp]

```

RefBase::RefBase()
    : mRefs(new weakref_impl(this)) // 注意这句话
{
    // mRefs 是 RefBase 的成员变量，类型是 weakref_impl，我们暂且叫它影子对象。
    // 所以 A 有一个影子对象。
}

```

mRefs 是引用计数管理的关键类，需要进一步观察。它是从 RefBase 的内部类 weakref_type 中派生出来的。

先看看它的声明：

```

class RefBase::weakref_impl : public RefBase::weakref_type
// 从 RefBase 的内部类 weakref_type 派生。

```

由于 Android 频繁使用 C++ 内部类的方法，所以初次阅读 Android 代码时可能会有点不太习惯，C++ 的内部类和 Java 的内部类相似，但有一点不同，即它需要一个显式的成员指向外部类对象，而 Java 的内部类对象有一个隐式的成员指向外部类对象的。

说明 内部类在 C++ 中的学名叫 nested class（内嵌类）。

👉 [-->RefBase.cpp::weakref_impl 构造]

```

weakref_impl(RefBase* base)
    : mStrong(INITIAL_STRONG_VALUE) // 强引用计数，初始值为 0x1000000。
    , mWeak(0) // 弱引用计数，初始值为 0。
    , mBase(base) // 该影子对象所指向的实际对象。
    , mFlags(0)
    , mStrongRefs(NULL)
    , mWeakRefs(NULL)
    , mTrackEnabled(!DEBUG_REFS_ENABLED_BY_DEFAULT)
    , mRetain(false)
{
}

```

如你所见，new 了一个 A 对象后，其实还 new 了一个 weakref_impl 对象，这里称它为

影子对象，另外我们称 A 为实际对象。

这里有一个问题：影子对象有什么用？

可以仔细想一下，是不是发现影子对象成员中有两个引用计数？一个强引用，一个弱引用。如果知道引用计数和对象生死有些许关联的话，就容易想到影子对象的作用了。

说明 按上面的分析来看，在构造一个实际对象的同时，还会悄悄地构造一个影子对象，在嵌入式设备的内存不是很紧俏的今天，这个影子对象的内存占用已经不成问题了。

2.sp 上场

程序继续运行，现在到了：

```
sp<A> spA(pA);
```

请看 sp 的构造函数，它的代码如下所示（注意，sp 是一个模板类，对此不熟悉的读者可以去翻翻书，或者干脆把所有出现的 T 都换成 A）：

 [-->RefBase.h::sp(T* other)]

```
template<typename T>
sp<T>::sp(T* other) // 这里的 other 就是刚才创建的 pA。
    : m_ptr(other) // sp 保存了 pA 的指针。
{
    if (other) other->incStrong(this); // 调用 pA 的 incStrong。
}
```

OK，战场转到 RefBase 的 incStrong 中。它的代码如下所示：

 [-->RefBase.cpp]

```
void RefBase::incStrong(const void* id) const
{
    // mRefs 就是刚才在 RefBase 构造函数中 new 出来的影子对象。
    weakref_impl* const refs = mRefs;

    // 操作影子对象，先增加弱引用计数。
    refs->addWeakRef(id);
    refs->incWeak(id);
    .....
```

先来看看影子对象的这两个 weak 函数都干了些什么。

(1) 眼见而心不烦

下面看看第一个函数 addWeakRef，代码如下所示：

 [-->RefBase.cpp]

```
void addWeakRef(const void* /*id*/) { }
```

呵呵，addWeakRef啥都没做，因为这是 release 版走的分支。调试版的代码我们就不讨论了，它是给创造 RefBase、sp，以及 wp 的人调试用的。

说明 调试版分支的代码很多，看来创造它们的人也在为不理解它们之间的暧昧关系痛苦不已。

总之，一共有这么几个不用考虑的函数，下面都已列出来了。以后再碰见它们，干脆就直接跳过去：

```
void addStrongRef(const void* /*id*/) { }
void removeStrongRef(const void* /*id*/) { }
void addWeakRef(const void* /*id*/) { }
void removeWeakRef(const void* /*id*/) { }
void printRefs() const { }
void trackMe(bool, bool) { }
```

继续我们的征程。再看 incWeak 函数，代码如下所示：

👉 [-->RefBase.cpp]

```
void RefBase::weakref_type::incWeak(const void* id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->addWeakRef(id); // 上面说了，非调试版什么都不干。
    const int32_t c = android_atomic_inc(&impl->mWeak);
    // 原子操作，影子对象的弱引用计数加1。
    // 千万记住影子对象的强弱引用计数的值，这是彻底理解 sp 和 wp 的关键。
}
```

好，我们再回到 incStrong，继续看代码：

👉 [-->RefBase.cpp]

```
.....
// 刚才增加了弱引用计数。
// 再增加强引用计数。
refs->addStrongRef(id); // 非调试版这里什么都不干。
// 下面函数为原子加1操作，并返回旧值。所以 c=0x1000000，而 mStrong 变为 0x1000001。
const int32_t c = android_atomic_inc(&refs->mStrong);
if (c != INITIAL_STRONG_VALUE) {
    // 如果 c 不是初始值，则表明这个对象已经被强引用过一次了。
    return;
}
// 下面这个是原子加操作，相当于执行 refs->mStrong + (-0x1000000)，最终 mStrong=1。
android_atomic_add(-INITIAL_STRONG_VALUE, &refs->mStrong);
/*
    如果是第一次引用，则调用 onFirstRef，这个函数很重要，派生类可以重载这个函数，完成一些
    初始化工作。
*/
const_cast<RefBase*>(this)->onFirstRef();
}
```

说明 android_atomic_xxx 是 Android 平台提供的原子操作函数，原子操作函数是多线程编程中的常见函数，读者可以学习原子操作函数的相关知识，本章后面也会对其进行介绍。

(2) sp 构造的影响

sp 构造完后，它给这个世界带来了什么？

那就是在 RefBase 中影子对象的强引用计数变为 1，且弱引用计数也变为 1。


更准确的说法是，sp 的出生导致影子对象的强引用计数加 1，且弱引用计数也加 1。

(3) wp 构造的影响

继续看 wp，例子中的调用方式如下：

```
wp<A> wpA(spA)
```

wp 有好几个构造函数，原理都一样。来看这个最常见的：

 [-->RefBase.h::wp(const sp<T>& other)]

```
template<typename T>
wp<T>::wp(const sp<T>& other)
    : m_ptr(other.m_ptr) //wp 的成员变量 m_ptr 指向实际对象。
{
    if (m_ptr) {
        // 调用 pA 的 createWeak，并且保存返回值到成员变量 m_refs 中。
        m_refs = m_ptr->createWeak(this);
    }
}
```

 [-->RefBase.cpp]

```
RefBase::weakref_type* RefBase::createWeak(const void* id) const
{
    // 调用影子对象的 incWeak，这个我们刚才讲过了，它会导致影子对象的弱引用计数增加 1。
    mRefs->incWeak(id);
    return mRefs; // 返回影子对象本身。
}
```

我们可以看到，wp 化后，影子对象的弱引用计数将增加 1，所以现在弱引用计数为 2，而强引用计数仍为 1。另外，wp 中有两个成员变量，一个保存实际对象，另一个保存影子对象。sp 只有一个成员变量，用来保存实际对象，但这个实际对象内部已包含了对应的影子对象。

OK，wp 创建完了，现在开始进行 wp 的析构。

(4) wp 析构的影响

wp 进入析构函数，则表明它快要离世了，代码如下所示：

 [-->RefBase.h]

```
template<typename T>
```

```
wp<T>::~~wp()
{
    if (m_ptr) m_refs->decWeak(this); // 调用影子对象的 decWeak, 由影子对象的基类实现。
}
```

👉 [-->RefBase.cpp]

```
void RefBase::weakref_type::decWeak(const void* id)
{
    // 把基类指针转换成子类(影子对象)的类型, 这种做法有些违背面向对象编程的思想。
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->removeWeakRef(id); // 非调试版不做任何事情。

    // 原子减1, 返回旧值, c=2, 而弱引用计数从2变为1。
    const int32_t c = android_atomic_dec(&impl->mWeak);
    if (c != 1) return; // c=2, 直接返回。

    // 如果c为1, 则弱引用计数为0, 这说明没用弱引用指向实际对象, 需要考虑是否释放内存。
    // OBJECT_LIFETIME_XXX和生命周期有关系, 我们后面再说。
    if ((impl->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
        if (impl->mStrong == INITIAL_STRONG_VALUE)
            delete impl->mBase;
        else {
            delete impl;
        }
    } else {
        impl->mBase->onLastWeakRef(id);
        if ((impl->mFlags & OBJECT_LIFETIME_FOREVER) != OBJECT_LIFETIME_FOREVER) {
            delete impl->mBase;
        }
    }
}
```

在例1中, wp析构后, 弱引用计数减1。但由于此时强引用计数和弱引用计数仍为1, 所以没有对象被干掉, 即没有释放实际对象和影子对象占据的内存。

(5) sp析构的影响

下面进入sp的析构。

👉 [-->RefBase.h]

```
template<typename T>
sp<T>::~~sp()
{
    if (m_ptr) m_ptr->decStrong(this); // 调用实际对象的 decStrong, 由 RefBase 实现。
}
```

👉 [-->RefBase.cpp]

```
void RefBase::decStrong(const void* id) const
```

```

{
    weakref_impl* const refs = mRefs;
    refs->removeStrongRef(id); // 调用影子对象的 removeStrongRef, 啥都不干。
    // 注意, 此时强弱引用计数都是 1, 下面函数调用的结果是 c=1, 强引用计数为 0。
    const int32_t c = android_atomic_dec(&refs->mStrong);
    if (c == 1) { // 对于我们的例子, c 为 1
        // 调用 onLastStrongRef, 表明强引用计数减为 0, 对象有可能被 delete。
        const_cast<RefBase*>(this)->onLastStrongRef(id);
        // mFlags 为 0, 所以会通过 delete this 把自己干掉。
        // 注意, 此时弱引用计数仍为 1。
        if ((refs->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
            delete this;
        }
        .....
    }
}

```

先看 delete this 的处理, 它会导致 A 的析构函数被调用。再来看 A 的析构函数, 代码如下所示:

👉 [--> 例子 1::~~A()]

```

// A 的析构直接导致进入 RefBase 的析构。
RefBase::~~RefBase()
{
    if (mRefs->mWeak == 0) { // 弱引用计数不为 0, 而是 1。
        delete mRefs;
    }
}

```

RefBase 的 delete this 自杀行为没有把影子对象干掉, 但我们还在 decStrong 中, 可从 delete this 接着往下看:

👉 [--> RefBase.cpp]

```

.... // 接前面的 delete this
if ((refs->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
    delete this;
}
// 注意, 实际数据对象已经被干掉了, 所以 mRefs 也没有用了, 但是 decStrong 刚进来
// 的时候就把 mRefs 保存到 refs 了, 所以这里的 refs 指向影子对象。
refs->removeWeakRef(id);
refs->decWeak(id); // 调用影子对象 decWeak
}

```

👉 [--> RefBase.cpp]

```

void RefBase::weakref_type::decWeak(const void* id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);

```



```
impl->removeWeakRef(id); // 非调试版不做任何事情。

// 调用前影子对象的弱引用计数为 1，强引用计数为 0，调用结束后 c=1，弱引用计数为 0。
const int32_t c = android_atomic_dec(&impl->mWeak);
if (c != 1) return;

// 这次弱引用计数终于变为 0 了，并且 mFlags 为 0，mStrong 也为 0。
if ((impl->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
    if (impl->mStrong == INITIAL_STRONG_VALUE)
        delete impl->mBase;
    else {
        delete impl; // impl 就是 this，把影子对象也就是自己干掉。
    }
} else {
    impl->mBase->onLastWeakRef(id);
    if ((impl->mFlags & OBJECT_LIFETIME_FOREVER) != OBJECT_LIFETIME_FOREVER) {
        delete impl->mBase;
    }
}
}
```

好，第一板斧劈下去了！来看看它的结果是什么。

3. 第一板斧的结果

第一板斧过后，来总结一下刚才所学的知识：

- ❑ RefBase 中有一个隐含的影子对象，该影子对象内部有强弱引用计数。
- ❑ sp 化后，强弱引用计数各增加 1，sp 析构后，强弱引用计数各减 1。
- ❑ wp 化后，弱引用计数增加 1，wp 析构后，弱引用计数减 1。

完全彻底地消灭 RefBase 对象，包括让实际对象和影子对象灭亡，这些都是由强弱引用计数控制的，另外还要考虑 flag 的取值情况。当 flag 为 0 时，可得出如下结论：

- ❑ 强引用为 0 将导致实际对象被 delete。
- ❑ 弱引用为 0 将导致影子对象被 delete。

5.2.2 第二板斧——由弱生强

再看第二个例子，代码如下所示：

👉 [--> 例子 2]

```
int main()
{
    A *pA = new A();
    wp<A> wpA(pA);
    sp<A> spA = wpA.promote(); // 通过 promote 函数，得到一个 sp。
}
```

对 A 的 wp 化，不再做分析了。按照前面所讲的知识，wp 化后仅会使弱引用计数加 1，所以此处 wp 化的结果是：

影子对象的弱引用计数为 1，强引用计数仍然是初始值 0x1000000。

wpA 的 promote 函数是从一个弱对象产生一个强对象的重要函数，试看——

1. 由弱生强的方法

代码如下所示：

👉 [-->RefBase.h]

```
template<typename T>
sp<T> wp<T>::promote() const
{
    return sp<T>(m_ptr, m_refs); // 调用 sp 的构造函数。
}
```

👉 [-->RefBase.h]

```
template<typename T>
sp<T>::sp(T* p, weakref_type* refs)
    : m_ptr((p && refs->attemptIncStrong(this)) ? p : 0) // 有点看不清楚。
{
    // 上面那行代码够简洁，但是不方便阅读，我们写成下面这样：
    /*
        T* pTemp = NULL;
        // 关键函数 attemptIncStrong
        if(p != NULL && refs->attemptIncStrong(this) == true)
            pTemp = p;

        m_ptr = pTemp;
    */
}
```

2. 成败在此一举

由弱生强的关键函数是 attemptIncStrong，它的代码如下所示：

👉 [-->RefBase.cpp]

```
bool RefBase::weakref_type::attemptIncStrong(const void* id)
{
    incWeak(id); // 增加弱引用计数，此时弱引用计数变为 2。
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    int32_t curCount = impl->mStrong; // 这个仍是初始值。
    // 下面这个循环，在多线程操作同一个对象时可能会循环多次。这里可以不去管它，
    // 它的目的就是使强引用计数增加 1。
    while (curCount > 0 && curCount != INITIAL_STRONG_VALUE) {
        if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
            break;
        }
    }
}
```

```

    }
    curCount = impl->mStrong;
}

if (curCount <= 0 || curCount == INITIAL_STRONG_VALUE) {
    bool allow;
/*
    下面这个 allow 的判断极为精妙。impl 的 mBase 对象就是实际对象，有可能已经被 delete 了。
    curCount 为 0，表示强引用计数肯定经历了 INITIAL_STRONG_VALUE->1->...->0 的过程。
    mFlags 就是根据标志来决定是否继续进行 || 或 && 后的判断，因为这些判断都使用了 mBase，
    如不做这些判断，一旦 mBase 指向已经回收的地址，你就等着 segment fault 吧！
    其实，咱们大可不必理会这些东西，因为它不影响我们的分析和理解。
*/
    if (curCount == INITIAL_STRONG_VALUE) {
        allow = (impl->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK
            || impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
    } else {
        allow = (impl->mFlags & OBJECT_LIFETIME_WEAK) == OBJECT_LIFETIME_WEAK
            && impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
    }
    if (!allow) {
        // allow 为 false，表示不允许由弱生强，弱引用计数要减去 1，这是因为咱们进来时加过一次。
        decWeak(id);
        return false; // 由弱生强失败。
    }

    // 允许由弱生强，强引用计数要增加 1，而弱引用计数已经增加过了。
    curCount = android_atomic_inc(&impl->mStrong);
    if (curCount > 0 && curCount < INITIAL_STRONG_VALUE) {
        impl->mBase->onLastStrongRef(id);
    }
}
impl->addWeakRef(id);
impl->addStrongRef(id); // 两个函数调用没有作用。
if (curCount == INITIAL_STRONG_VALUE) {
    // 强引用计数变为 1。
    android_atomic_add(-INITIAL_STRONG_VALUE, &impl->mStrong);
    // 调用 onFirstRef，通知该对象第一次被强引用。
    impl->mBase->onFirstRef();
}
return true; // 由弱生强成功。
}

```

3. 第二板斧的结果

promote 完成后，相当于增加了一个强引用。根据上面所学的知识可知：

由弱生强成功后，强弱引用计数均增加 1。所以现在影子对象的强引用计数为 1，弱引用计数为 2。

5.2.3 第三板斧——破解生死魔咒

1. 延长生命的魔咒

RefBase 为我们提供了一个这样的函数：

```
extendObjectLifetime(int32_t mode)
```

另外还定义了一个枚举：

```
enum {
    OBJECT_LIFETIME_WEAK      = 0x0001,
    OBJECT_LIFETIME_FOREVER = 0x0003
};
```

注意：FOREVER 的值是 3，用二进制表示是 B11，而 WEAK 的二进制是 B01，也就是说 FOREVER 包括了 WEAK 的情况。

上面这两个枚举值，是破除强弱引用计数作用的魔咒。先观察 flags 为 OBJECT_LIFETIME_WEAK 的情况，见下面的例子。

👉 [--> 例子 3]

```
class A:public RefBase
{
    public A()
    {
        extendObjectLifetime(OBJECT_LIFETIME_WEAK); // 在构造函数中调用。
    }
}

int main()
{
    A *pA = new A();
    wp<A> wpA(pA); // 弱引用计数加 1。
    {
        sp<A> spA(pA) // sp 后，结果是强引用计数为 1，弱引用计数为 2。
    }
    ....
}
```

sp 的析构将直接调用 RefBase 的 decStrong，它的代码如下所示：

👉 [--> RefBase.cpp]

```
void RefBase::decStrong(const void* id) const
{
    weakref_impl* const refs = mRefs;
    refs->removeStrongRef(id);
    const int32_t c = android_atomic_dec(&refs->mStrong);
    if (c == 1) { // 上面进行原子操作后，强引用计数为 0
        const_cast<RefBase*>(this)->onLastStrongRef(id);
        // 注意这句话。如果 flags 不是 WEAK 或 FOREVER 的话，将 delete 数据对象。
        // 现在我们的 flags 是 WEAK，所以不会 delete 它。
    }
}
```

```

        if ((refs->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
            delete this;
        }
    }
    refs->removeWeakRef(id);
    refs->decWeak(id); // 调用前弱引用计数是 2。
}

```

然后调用影子对象的 `decWeak`。再来看它的处理，代码如下所示：

👉 [-->RefBase.cpp::weakref_type 的 `decWeak()` 函数]

```

void RefBase::weakref_type::decWeak(const void* id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->removeWeakRef(id);
    const int32_t c = android_atomic_dec(&impl->mWeak);
    if (c != 1) return; // c 为 2，弱引用计数为 1，直接返回。
    /*
     * 假设我们现在到了例子中的 wp 析构之处，这时也会调用 decWeak，在调用上面的原子减操作后
     * c=1，弱引用计数变为 0，此时会继续往下运行。由于 mFlags 为 WEAK，所以不满足 if 的条件。
     */
    if ((impl->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
        if (impl->mStrong == INITIAL_STRONG_VALUE)
            delete impl->mBase;
        else {
            delete impl;
        }
    } else { // flag 为 WEAK，满足 else 分支的条件。
        impl->mBase->onLastWeakRef(id);
        /*
         * 由于 flags 值满足下面这个条件，所以实际对象会被 delete，根据前面的分析可知，实际对象的
         * delete 会检查影子对象的弱引用计数，如果它为 0，则会把影子对象也 delete 掉。
         * 由于影子对象的弱引用计数此时已经为 0，所以影子对象也会被 delete。
         */
        if ((impl->mFlags&OBJECT_LIFETIME_FOREVER) != OBJECT_LIFETIME_FOREVER) {
            delete impl->mBase;
        }
    }
}

```

2. LIFETIME_WEAK 的魔力

看完上面的例子，我们发现什么了？

在 `LIFETIME_WEAK` 的魔法下，强引用计数为 0，而弱引用计数不为 0 的时候，实际对象没有被 `delete`！只有当强引用计数和弱引用计数同时为 0 时，实际对象和影子对象才会被 `delete`。

3. 魔咒大揭秘

至于 LIFETIME_FOREVER 的破解，就不用再来一斧子了，我直接给出答案：

- ❑ flags 为 0，强引用计数控制实际对象的生命周期，弱引用计数控制影子对象的生命周期。强引用计数为 0 后，实际对象被 delete。所以对于这种情况，应记住的是，使用 wp 时要由弱生强，以免收到 segment fault 信号。
- ❑ flags 为 LIFETIME_WEAK，强引用计数为 0，弱引用计数不为 0 时，实际对象不会被 delete。当弱引用计数减为 0 时，实际对象和影子对象会同时被 delete。这是功德圆满的情况。
- ❑ flags 为 LIFETIME_FOREVER，对象将长生不老，彻底摆脱强弱引用计数的控制。所以你要在适当的时候杀死这些“老妖精”，免得她祸害“人间”。

5.2.4 轻量级的引用计数控制类 LightRefBase

上面介绍的 RefBase，是一个重量级的引用计数控制类。那么，究竟有没有一个简单些的引用计数控制类呢？Android 为我们提供了一个轻量级的 LightRefBase。这个类非常简单，我们不妨一起来看看。

👉 [-->RefBase.h]

```
template <class T>
class LightRefBase
{
public:
    inline LightRefBase() : mCount(0) { }
    inline void incStrong(const void* id) const {
        //LightRefBase 只有一个引用计数控制量 mCount。incStrong 的时候使它增加 1。
        android_atomic_inc(&mCount);
    }
    inline void decStrong(const void* id) const {
        //decStrong 的时候减 1，当引用计数变为零的时候，delete 掉自己。
        if (android_atomic_dec(&mCount) == 1) {
            delete static_cast<const T*>(this);
        }
    }
    inline int32_t getStrongCount() const {
        return mCount;
    }

protected:
    inline ~LightRefBase() { }

private:
    mutable volatile int32_t mCount; // 引用计数控制变量。
};
```

LightRefBase 类够简单吧？不过它是一个模板类，我们该怎么用它呢？下面给出一个例子，其中类 A 是从 LightRefBase 派生的，写法如下：

```
class A:public LightRefBase<A> // 注意派生的时候要指明是 LightRefBase<A>。
{
public:
    A(){};
    ~A(){};
};
```

另外，我们从 LightRefBase 的定义中可以知道，它支持 sp 的控制，因为它只有 incStrong 和 decStrong 函数。

5.2.5 题外话——三板斧的来历

从代码量上看，RefBase、sp 和 wp 的代码量并不多，但里面的关系，尤其是 flags 的引入，曾一度让我眼花缭乱。当时，我确实很希望能自己调试一下这些例子，但在设备上调试 native 代码，需要花费很大的精力，即使是通过输出 log 的方式来调试也需要花很多时间。该怎么解决这一难题？

既然它的代码不多而且简单，那何不把它移植到台式机的开发环境下，整一个类似的 RefBase 呢？有了这样的构想，我便用上了 Visual Studio。至于那些原子操作，Windows 平台上有很直接的 InterlockedExchangeXXX 与之对应，真的是踏破铁鞋无觅处，得来全不费功夫！（在 Linux 平台上，不考虑多线程的话，将原子操作换成普通的非原子操作不是也可以吗？如果更细心更负责任的话，你可以自己用汇编来实现常用的原子操作，内核代码中有现成的函数，一看就会明白。）

如果把破解代码看成是攻城略地的话，我们必须学会灵活多变，而且应力求破解方法日臻极致！

5.3 Thread 类及常用同步类分析

Thread 类是 Android 为线程操作而做的一个封装。代码在 Thread.cpp 中，其中还封装了一些与线程同步相关的类（既然是封装，要掌握它，最重要的当然是掌握与 Pthread 相关的知识）。我们先分析 Threa 类，进而再介绍与常用同步类相关的知识。

5.3.1 一个变量引发的思考

Thread 类虽说挺简单，但其构造函数中的那个 canCallJava 却一度让我感到费解。因为我一直使用的是自己封装的 Pthread 类。当发现 Thread 构造函数中竟然存在这样一个东西时，很担心自己封装的 Pthread 类会不会有什么重大问题，因为当时我还从来没考虑过 Java 方面的问题。

// canCallJava 表示这个线程是否会使用 JNI 函数。为什么需要一个这样的参数呢？
Thread(bool canCallJava = true)。

我们必须得了解它实际创建的线程函数是什么。Thread 类真实的线程是创建在 run 函数中的。

1. 一个变量，两种处理

先来看一段代码：

👉 [-->Thread.cpp]

```
status_t Thread::run(const char* name, int32_t priority, size_t stack)
{
    Mutex::Autolock _l(mLock);
    ....
    // 如果 mCanCallJava 为真，则调用 createThreadEtc 函数，线程函数是 _threadLoop。
    // _threadLoop 是 Thread.cpp 中定义的一个函数。
    if (mCanCallJava) {
        res = createThreadEtc(_threadLoop, this, name, priority,
                               stack, &mThread);
    } else {
        res = androidCreateRawThreadEtc(_threadLoop, this, name, priority,
                                         stack, &mThread);
    }
}
```

上面的 mCanCallJava 将线程创建函数的逻辑分为两个分支，虽传入的参数都有 _threadLoop，但它们调用的函数却不同。先直接看 mCanCallJava 为 true 的这个分支，代码如下所示：

👉 [-->Thread.h:createThreadEtc() 函数]

```
inline bool createThreadEtc(thread_func_t entryFunction,
                           void *userData,
                           const char* threadName = "android:unnamed_thread",
                           int32_t threadPriority = PRIORITY_DEFAULT,
                           size_t threadStackSize = 0,
                           thread_id_t *threadId = 0)
{
    return androidCreateThreadEtc(entryFunction, userData, threadName,
                                   threadPriority, threadStackSize, threadId) ? true : false;
}
```

它调用的是 androidCreateThreadEtc 函数，相关代码如下所示：

```
// gCreateThreadFn 是函数指针，它在初始化时和 mCanCallJava 为 false 时使用是同一个
// 线程创建函数。那么有地方会修改它吗？
static android_create_thread_fn gCreateThreadFn = androidCreateRawThreadEtc;
int androidCreateThreadEtc(android_thread_func_t entryFunction,
                           void *userData, const char* threadName,
```



```

        int32_t threadPriority, size_t threadStackSize,
        android_thread_id_t *threadId)
    {
        return gCreateThreadFn(entryFunction, userData, threadName,
                               threadPriority, threadStackSize, threadId);
    }

```

如果没有人修改这个函数指针，那么 `mCanCallJava` 就是虚晃一枪，并无什么作用。不过，代码中有的地方是会修改这个函数指针的指向的，请看——

2. zygote 偷梁换柱

在本书 4.2.1 节的第 2 点所介绍的 `AndroidRuntime` 调用 `startReg` 的地方，就有可能修改这个函数指针，其代码如下所示：

👉 [--> `AndroidRuntime.cpp`]

```

/*static*/ int AndroidRuntime::startReg(JNIEnv* env)
{
    // 这里会修改函数指针为 javaCreateThreadEtc。
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);
    return 0;
}

```

如果 `mCanCallJava` 为 `true`，则将调用 `javaCreateThreadEtc`。那么，这个函数有什么特殊之处呢？来看其代码，如下所示：

👉 [--> `AndroidRuntime.cpp`]

```

int AndroidRuntime::javaCreateThreadEtc(
    android_thread_func_t entryFunction,
    void* userData,
    const char* threadName,
    int32_t threadPriority,
    size_t threadStackSize,
    android_thread_id_t* threadId)
{
    void** args = (void**) malloc(3 * sizeof(void*));
    int result;
    args[0] = (void*) entryFunction;
    args[1] = userData;
    args[2] = (void*) strdup(threadName);
    // 调用的还是 androidCreateRawThreadEtc，但线程函数却换成了 javaThreadShell。
    result = androidCreateRawThreadEtc(AndroidRuntime::javaThreadShell, args,
                                       threadName, threadPriority, threadStackSize, threadId);
    return result;
}

```

👉 [-->AndroidRuntime.cpp]

```
int AndroidRuntime::javaThreadShell(void* args) {
    .....
    int result;
    // 把这个线程 attach 到 JNI 环境中，这样这个线程就可以调用 JNI 的函数了。
    if (javaAttachThread(name, &env) != JNI_OK)
        return -1;
    // 调用实际的线程函数干活。
    result = (*(android_thread_func_t)start)(userData);
    // 从 JNI 环境中 detach 出来。
    javaDetachThread();
    free(name);
    return result;
}
```

3. 费力能讨好

你明白 mCanCallJava 为 true 的目的了吗？它创建的新线程将：

- ❑ 在调用你的线程函数之前会 attach 到 JNI 环境中，这样，你的线程函数就可以无忧无虑地使用 JNI 函数了。
- ❑ 线程函数退出后，它会从 JNI 环境中 detach，释放一些资源。

注意 第二点尤其重要，因为进程退出前，dalvik 虚拟机会检查是否有 attach 了，如果最后有未 detach 的线程，则会直接 abort（这不是一件好事）。如果你关闭 JNI check 选项，就不会做这个检查，但我觉得，这个检查和资源释放有关系，建议还是重视。如果直接使用 POSIX 的线程创建函数，那么凡是使用过 attach 的，最后就都需要 detach！

Android 为了 dalvik 的健康真是费尽心机呀。

4. 线程函数 _threadLoop 介绍

无论一分为二是如何处理的，最终都会调用线程函数 _threadLoop，为什么不直接调用用户传入的线程函数呢？莫非 _threadLoop 会有什么暗箱操作吗？下面我们来看：

👉 [-->Thread.cpp]

```
int Thread::_threadLoop(void* user)
{
    Thread* const self = static_cast<Thread*>(user);
    sp<Thread> strong(self->mHoldSelf);
    wp<Thread> weak(strong);
    self->mHoldSelf.clear();

    #if HAVE_ANDROID_OS
        self->mTid = gettid();
    #endif
}
```

```

bool first = true;

do {
    bool result;
    if (first) {
        first = false;
        //self 代表继承 Thread 类的对象，第一次进来时将调用 readyToRun，看看是否准备好。
        self->mStatus = self->readyToRun();
        result = (self->mStatus == NO_ERROR);

        if (result && !self->mExitPending) {
            result = self->threadLoop();
        }
    } else {
        /*
        调用子类实现的 threadLoop 函数，注意这段代码运行在一个 do-while 循环中。
        这表示即使我们的 threadLoop 返回了，线程也不一定会退出。
        */
        result = self->threadLoop();
    }
}

/*
线程退出的条件：
1) result 为 false。这表明，如果子类在 threadLoop 中返回 false，线程就可以退出。这属于主动退出的情况，是 threadLoop 自己不想继续干活了，所以返回 false。读者在自己的代码中千万别写错 threadLoop 的返回值。
2) mExitPending 为 true，这个变量可由 Thread 类的 requestExit 函数设置，这种情况属于被动退出，因为由外界强制设置了退出条件。
*/
if (result == false || self->mExitPending) {
    self->mExitPending = true;
    self->mLock.lock();
    self->mRunning = false;
    self->mThreadExitedCondition.broadcast();
    self->mLock.unlock();
    break;
}
strong.clear();
strong = weak.promote();
} while(strong != 0);

return 0;
}

```

关于 `_threadLoop`，我们就介绍到这里。请读者务必注意下面一点：
`threadLoop` 运行在一个循环中，它的返回值可以决定是否退出线程。

5.3.2 常用同步类

同步，是多线程编程中不可避免的话题，同时也是一个非常复杂的问题。这里只简单介绍一下 Android 提供的同步类。这些类，只对系统提供的多线程同步函数（这种函数我们称为 Raw API）进行了面向对象的封装，读者必须先理解 Raw API，然后才能真正掌握其具体用法。

提示 要了解 Windows 下的多线程编程，有很多参考资料，而有关 Linux 下完整系统阐述多线程编程的书籍目前较少，这里推荐一本含金量较高的著作《Programming with POSIX Thread》（本书只有英文版，由 Addison-Wesley 出版）。

Android 提供了两个封装好的同步类，它们是 Mutex 和 Condition。这是重量级的同步技术，一般内核都会有对应的支持。另外，OS 还提供了简单的原子操作，这些也算是同步技术中的一种。下面分别来介绍这三种东西。

1. 互斥类——Mutex

Mutex 是互斥类，用于多线程访问同一个资源的时候，保证一次只有一个线程能访问该资源。在《Windows 核心编程》^①一书中，对于这种互斥访问有一个很形象的比喻：想象你在飞机上如厕，这时卫生间的信息牌上显示“有人”，你必须等里面的人出来后才可进去。这就是互斥的含义。

下面来看 Mutex 的实现方式，它们都很简单。

(1) Mutex 介绍

其代码如下所示：

👉 [-->Thread.h::Mutex 的声明和实现]

```
inline Mutex::Mutex(int type, const char* name) {
    if (type == SHARED) {
        //type 如果是 SHARED，则表明这个 Mutex 支持跨进程的线程同步。
        // 以后我们在 Audio 系统和 Surface 系统中会经常见到这种用法。
        pthread_mutexattr_t attr;
        pthread_mutexattr_init(&attr);
        pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
        pthread_mutex_init(&mMutex, &attr);
        pthread_mutexattr_destroy(&attr);
    } else {
        pthread_mutex_init(&mMutex, NULL);
    }
}

inline Mutex::~Mutex() {
    pthread_mutex_destroy(&mMutex);
}
```

^① 本书中文版由机械工业出版社出版，原书作者 Jeffrey Richter。

```

}
inline status_t Mutex::lock() {
    return -pthread_mutex_lock(&mMutex);
}
inline void Mutex::unlock() {
    pthread_mutex_unlock(&mMutex);
}
inline status_t Mutex::tryLock() {
    return -pthread_mutex_trylock(&mMutex);
}

```

关于 Mutex 的使用，除了初始化外，最重要的是 lock 和 unlock 函数的使用，它们的使用方法如下：

- ❑ 要想独占卫生间，必须先调用 Mutex 的 lock 函数。这样，这个区域就被锁住了。如果这块区域之前已被别人锁住，lock 函数则会等待，直到可以进入这块区域为止。系统保证一次只有一个线程能 lock 成功。
- ❑ 当你“方便”完毕，记得调用 Mutex 的 unlock 以释放互斥区域。这样，其他人的 lock 才可以成功返回。
- ❑ 另外，Mutex 还提供了一个 trylock 函数，该函数只是尝试去锁住该区域，使用者需要根据 trylock 的返回值来判断是否成功锁住了该区域。

注意 以上这些内容都和 Raw API 有关，不了解它的读者可自行学习相关知识。在 Android 系统中，多线程也是常见和重要的编程手段，务必请大家重视。

Mutex 类确实比 Raw API 方便好用，不过还是稍显麻烦。

(2) AutoLock 介绍

AutoLock 类是定义在 Mutex 内部的一个类，它其实是一帮“懒人”搞出来的，为什么这么说呢？先来看看使用 Mutex 有多麻烦：

- ❑ 显示调用 Mutex 的 lock。
- ❑ 在某个时候记住要调用该 Mutex 的 unlock。

以上这些操作都必须一一对应，否则会出现“死锁”！在有些代码中，如果判断分支特别多，你会发现 unlock 这句代码被写得比比皆是，如果稍有不慎，在某处就会忘了写它。有什么好办法能解决这个问题吗？终于有人想出来一个好办法，就是充分利用了 C++ 的构造和析构函数，只需看一看 AutoLock 的定义就会明白。代码如下所示：

👉 [-->Thread.h Mutex::Autolock 声明和实现]

```

class Autolock {
public:
    // 构造的时候调用 lock。
    inline Autolock(Mutex& mutex) : mLock(mutex) { mLock.lock(); }
    inline Autolock(Mutex* mutex) : mLock(*mutex) { mLock.lock(); }

```

```

        // 析构的时候调用 unlock。
        inline ~Autolock() { mLock.unlock(); }
private:
    Mutex& mLock;
};

```

AutoLock 的用法很简单：

❑ 先定义一个 Mutex，如 Mutex xlock。

❑ 在使用 xlock 的地方，定义一个 AutoLock，如 AutoLock autoLock (xlock)。

由于 C++ 对象的构造和析构函数都是自动被调用的，所以在 AutoLock 的生命周期内，xlock 的 lock 和 unlock 也就自动被调用了，这样就省去了重复书写 unlock 的麻烦，而且 lock 和 unlock 的调用肯定是一一对应的，这样就绝对不会出错。

2. 条件类——Condition

多线程同步中的条件类对应的是下面这种使用场景：

线程 A 做初始化工作，而其他线程比如线程 B、C 必须等到初始化工作完后才能工作，即线程 B、C 在等待一个条件，我们称 B、C 为等待者。

当线程 A 完成初始化工作时，会触发这个条件，那么等待者 B、C 就会被唤醒。触发这个条件的 A 就是触发者。

上面的使用场景非常形象，而且条件类提供的函数也非常形象，它的代码如下所示：

👉 [-->Thread.h:: Condition 的声明和实现]

```

class Condition {
public:
    enum {
        PRIVATE = 0,
        SHARED = 1
    };

    Condition();
    Condition(int type); // 如果 type 是 SHARED，表示支持跨进程的条件同步
    ~Condition();
    // 线程 B 和 C 等待事件，wait 这个名字是不是很形象呢？
    status_t wait(Mutex& mutex);
    // 线程 B 和 C 的超时等待，B 和 C 可以指定等待时间，当超过这个时间，条件却还不满足，则退出等待。
    status_t waitRelative(Mutex& mutex, nsecs_t reltime);
    // 触发者 A 用来通知条件已经满足，但是 B 和 C 只有一个会被唤醒。
    void signal();
    // 触发者 A 用来通知条件已经满足，所有等待者都会被唤醒。
    void broadcast();

private:
#ifdef HAVE_PTHREADS
    pthread_cond_t mCond;
#endif
};

```

```

#else
    void*    mState;
#endif
}

```

声明很简单，定义也很简单，代码如下所示：

```

inline Condition::Condition() {
    pthread_cond_init(&mCond, NULL);
}
inline Condition::Condition(int type) {
    if (type == SHARED) { // 设置跨进程的同步支持。
        pthread_condattr_t attr;
        pthread_condattr_init(&attr);
        pthread_condattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
        pthread_cond_init(&mCond, &attr);
        pthread_condattr_destroy(&attr);
    } else {
        pthread_cond_init(&mCond, NULL);
    }
}
inline Condition::~Condition() {
    pthread_cond_destroy(&mCond);
}
inline status_t Condition::wait(Mutex& mutex) {
    return -pthread_cond_wait(&mCond, &mutex.mMutex);
}
inline status_t Condition::waitRelative(Mutex& mutex, nsecs_t reltime) {
#ifdef HAVE_PTHREAD_COND_TIMEDWAIT_RELATIVE
    struct timespec ts;
    ts.tv_sec = reltime/1000000000;
    ts.tv_nsec = reltime%1000000000;
    return -pthread_cond_timedwait_relative_np(&mCond, &mutex.mMutex, &ts);
    ..... // 有些系统没有实现 POSIX 的相关函数，所以不同的系统需要调用不同的函数。
#endif
}
inline void Condition::signal() {
    pthread_cond_signal(&mCond);
}
inline void Condition::broadcast() {
    pthread_cond_broadcast(&mCond);
}

```

可以看出，Condition 的实现全是凭借调用了 Raw API 的 pthread_cond_xxx 函数。这里要重点说明的是，Condition 类必须配合 Mutex 来使用。什么意思？

在上面的代码中，不论是 wait、waitRelative、signal 还是 broadcast 的调用，都放在一个 Mutex 的 lock 和 unlock 范围中，尤其是 wait 和 waitRelative 函数的调用，这是强制性的。

来看一个实际的例子，加深一下对 Condition 类和 Mutex 类的印象。这个例子是 Thread

类的 `requestExitAndWait`，目的是等待工作线程退出，代码如下所示：

👉 [-->Thread.cpp]

```
status_t Thread::requestExitAndWait()
{
    .....
    requestExit(); // 设置退出变量 mExitPending 为 true。
    Mutex::Autolock _l(mLock); // 使用 Autolock, mLock 被锁住。
    while (mRunning == true) {
        /*
         * 条件变量的等待，这里为什么要通过 while 循环来反复检测 mRunning？
         * 因为某些时候即使条件类没有被触发，wait 也会返回。关于这个问题，强烈建议读者阅读
         * 前面推荐的《Programming with POSIX Thread》一书。
         */
        mThreadExitedCondition.wait(mLock);
    }

    mExitPending = false;
    // 退出前，局部变量 Mutex::Autolock _l 的析构会被调用，unlock 也就会被自动调用。
    return mStatus;
}
```

那么，什么时候会触发这个条件呢？是在工作线程退出前。其代码如下所示：

👉 [-->Thread.cpp]

```
int Thread::_threadLoop(void* user)
{
    Thread* const self = static_cast<Thread*>(user);
    sp<Thread> strong(self->mHoldSelf);
    wp<Thread> weak(strong);
    self->mHoldSelf.clear();

    do {
        .....
        result = self->threadLoop(); // 调用子类的 threadLoop 函数。
        .....
        // 如果 mExitPending 为 true，则退出。
        if (result == false || self->mExitPending) {
            self->mExitPending = true;
            // 退出前触发条件变量，唤醒等待者。
            self->mLock.lock(); // lock 锁住。
            // mRunning 的修改位于锁的保护中。如果你阅读了前面推荐的书，这里也就不难理解了。
            self->mRunning = false;
            self->mThreadExitedCondition.broadcast();
            self->mLock.unlock(); // 释放锁。
            break; // 退出循环，此后该线程函数会退出。
        }
        .....
    }
```

```

    } while(strong != 0);

    return 0;
}

```

关于 Android 多线程的同步类，暂时介绍到此吧。当然，这些类背后所隐含的知识及技术是读者需要倍加重视的。

提示 希望我们能养成一种由点及面的学习方法。以我们的同步类为例，假设你是第一次接触多线程编程，也学会了如何使用 Mutex 和 Condition 这两个类，不妨以这两个类代码中所传递的知识作为切入点，把和多线程相关的所有知识（这个知识不仅仅是函数的使用，还包括多线程的原理，多线程的编程模型，甚至是现在很热门的并行多核编程）普遍了解一下。只有深刻理解并掌握了原理等基础和框架性的知识后，才能以不变应万变，才能做到游刃有余。

3. 原子操作函数介绍

什么是原子操作？所谓原子操作，就是该操作绝不会在执行完毕前被任何其他任务或事件打断，也就是说，原子操作是最小的执行单位。

上面这句话放到代码中是什么意思？请看一个例子：

👉 [--> 例子]

```

static int g_flag = 0; // 全局变量 g_flag
static Mutex lock ;// 全局的锁
// 线程 1 执行 thread1。
void thread1()
{
    //g_flag 递减，每次操作前锁住。
    lock.lock();
    g_flag--;
    lock.unlock();
}
// 线程 2 中执行 thread2 函数。
void thread2()
{
    lock.lock();
    g_flag++; // 线程 2 对 g_flag 进行递增操作，每次操作前要取得锁。
    lock.unlock();
}

```

为什么需要 Mutex 来帮忙呢？因为 g_flag++ 或 g_flag-- 操作都不是原子操作。从汇编指令的角度看，C/C++ 中的一条语句对应了数条汇编指令。以 g_flag++ 操作为例，它生成的汇编指令可能就是以下三条：

- ❑ 从内存中取数据到寄存器。
- ❑ 对寄存器中的数据进行递增操作，结果还在寄存器中。
- ❑ 寄存器的结果写回内存。

这三条汇编指令，如果按正常的顺序连续执行是没有问题的，但在多线程时就不能保证了。例如，线程 1 在执行第一条指令后，线程 2 由于调度的原因，抢在线程 1 之前连续执行完了三条指令。这样，线程 1 继续执行指令时，它所使用的值就不是线程 2 更新后的值，而是之前的旧值。再对这个值进行操作便没有意义了。

在一般情况下，处理这种问题可以使用 Mutex 来加锁保护，但 Mutex 的使用方法比它所保护的内容还要复杂，例如，锁的使用将导致从用户态转入内核态，有较大的浪费。那么，有没有简便些的办法让这些加、减等操作不被中断呢？

答案是肯定的，但这需要 CPU 的支持。在 X86 平台上，一个递增操作可以用下面的内嵌汇编语句来实现：

```
#define LOCK "lock;"
INT32 InterlockedIncrement(INT32* lpAddend)
{
    /*
     * 这是我们在 Linux 平台上实现 Windows API 时使用的方法。
     * 其中在 SMP 系统上，LOCK 定义成 "lock;" 表示锁总线，这样同一时刻就只能有一个 CPU 访问总线了。
     * 非 SMP 系统，LOCK 定义成空。由于 InterlockedIncrement 要返回递增前的旧值，所以我们
     * 使用了 xaddl 指令，它先交换源和目的的操作数，再进行递增操作。
     */
    INT32 i = 1;
    __asm__ __volatile__(
        LOCK "xaddl %0, %1"
        : "+r" (i), "+m" (*lpAddend)
        : : "memory");
    return *lpAddend;
}
```

Android 提供了相关的原子操作函数。这里有必要介绍一下各个函数的作用。

[-->Atomic.h]，注意该文件位于 system/core/include/cutils 目录中。

```
// 原子赋值操作，结果是 *addr=value。
void android_atomic_write(int32_t value, volatile int32_t* addr);
// 下面所有函数的返回值都是操作前的旧值。
// 原子加 1 和原子减 1。
int32_t android_atomic_inc(volatile int32_t* addr);
int32_t android_atomic_dec(volatile int32_t* addr);
// 原子加法操作，value 为被加数。
int32_t android_atomic_add(int32_t value, volatile int32_t* addr);
// 原子“与”和“或”操作。
int32_t android_atomic_and(int32_t value, volatile int32_t* addr);
int32_t android_atomic_or(int32_t value, volatile int32_t* addr);
/*
条件交换的原子操作。只有在 oldValue 等于 *addr 时，才会把 newValue 赋值给 *addr。
```

这个函数的返回值须特别注意。返回值非零，表示没有进行赋值操作。返回值为零，表示进行了原子操作。

```
*/
int android_atomic_cmpxchg(int32_t oldvalue, int32_t newvalue,
                           volatile int32_t* addr);
```

有兴趣的话，读者可以对上述函数的实现进行深入研究，其中：

❑ X86 平台的实现在 system/core/libcutils/Atomic.c 中，注意其代码在 #elif defined(__i386__) || defined(__x86_64__) 所包括的代码段内。

❑ ARM 平台的实现在 system/core/libcutils/atomic-android-arm.S 汇编文件中。

原子操作的最大好处在于避免了锁的使用，这对整个程序运行效率的提高有很大帮助。目前，在多核并行编程中，最高境界就是完全不使用锁。当然，它的难度可想而知是巨大的。

5.4 Looper 和 Handler 类分析

就应用程序而言，Android 系统中 Java 的应用程序和其他系统上相同，都是靠消息驱动来工作的，它们大致的工作原理如下：

- ❑ 有一个消息队列，可以往这个消息队列中投递消息。
- ❑ 有一个消息循环，不断从消息队列中取出消息，然后处理。

我们用图 5-1 来展示这个工作过程：

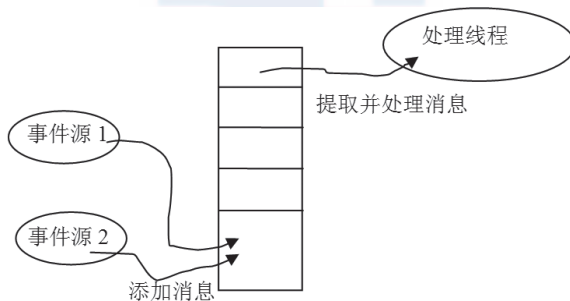


图 5-1 线程和消息处理的原理图

从图中可以看出：

- ❑ 事件源把待处理的消息加入到消息队列中，一般是加至队列尾，一些优先级高的消息也可以加至队列头。事件源提交的消息可以是按键、触摸屏等物理事件产生的消息，也可以是系统或应用程序本身发出的请求消息。
- ❑ 处理线程不断从消息队列头中取出消息并处理，事件源可以把优先级高的消息放到队列头，这样，优先级高的消息就会首先被处理。

在 Android 系统中，这些工作主要由 Looper 和 Handler 来实现：

- ❑ Looper 类，用于封装消息循环，并且有一个消息队列。

❑ Handler 类，有点像辅助类，它封装了消息投递、消息处理等接口。
 Looper 类是其中的关键。先来看看它是怎么做的。

5.4.1 Looper 类分析

我们以 Looper 使用的一个常见例子来分析这个 Looper 类。

👉 [--> 例子 1]

```
// 定义一个 LooperThread。
class LooperThread extends Thread {
    public Handler mHandler;
    public void run() {
        // ① 调用 prepare。
        Looper.prepare();
        .....
        // ② 进入消息循环。
        Looper.loop();
    }
}
// 应用程序使用 LooperThread。
{
    .....
    new LooperThread().start(); // 启动新线程，线程函数是 run
}
```

上面的代码一共有两个关键调用（即①和②），我们对其逐一进行分析。

1. 准备好了吗

第一个调用函数是 Looper 的 prepare 函数。它会做什么工作呢？其代码如下所示：

👉 [--> Looper.java]

```
public static final void prepare() {
    // 一个 Looper 只能调用一次 prepare。
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    // 构造一个 Looper 对象，设置到调用线程的局部变量中。
    sThreadLocal.set(new Looper());
}
// sThreadLocal 定义
private static final ThreadLocal sThreadLocal = new ThreadLocal();
```

ThreadLocal 是 Java 中的线程局部变量类，全名应该是 Thread Local Variable。我觉得它的实现和操作系统提供的线程本地存储（TLS）有关系。总之，该类有两个关键函数：

- ❑ set：设置调用线程的局部变量。
- ❑ get：获取调用线程的局部变量。

注意 set/get 的结果都和调用这个函数的线程有关。ThreadLocal 类可参考 JDK API 文档或 Android API 文档。

根据上面的分析可知，prepare 会在调用线程的局部变量中设置一个 Looper 对象。这个调用线程就是 LooperThread 的 run 线程。先看看 Looper 对象的构造，其代码如下所示：

👉 [-->Looper.java]

```
private Looper() {
    // 构造一个消息队列。
    mQueue = new MessageQueue();
    mRun = true;
    // 得到当前线程的 Thread 对象。
    mThread = Thread.currentThread();
}
```

prepare 函数很简单，它主要干了一件事：

在调用 prepare 的线程中，设置了一个 Looper 对象，这个 Looper 对象就保存在这个调用线程的 TLV 中。而 Looper 对象内部封装了一个消息队列。

也就是说，prepare 函数通过 ThreadLocal 机制，巧妙地把 Looper 和调用线程关联在一起了。要了解这样做的目的是什么，需要再看第二个重要函数。

2. Looper 循环

代码如下所示：

👉 [-->Looper.java]

```
public static final void loop() {
    Looper me = myLooper(); // myLooper 返回保存在调用线程 TLV 中的 Looper 对象。
    // 取出这个 Looper 的消息队列。
    MessageQueue queue = me.mQueue;
    while (true) {
        Message msg = queue.next();
        // 处理消息，Message 对象中有一个 target，它是 Handler 类型。
        // 如果 target 为空，则表示需要退出消息循环。
        if (msg != null) {
            if (msg.target == null) {
                return;
            }
            // 调用该消息的 Handler，交给它的 dispatchMessage 函数处理。
            msg.target.dispatchMessage(msg);
            msg.recycle();
        }
    }
}

// myLooper 函数返回调用线程的线程局部变量，也就是存储在其中的 Looper 对象。
public static final Looper myLooper() {
```

```
        return (Looper)sThreadLocal.get();
    }
}
```

通过上面的分析会发现，Looper 的作用是：

- ❑ 封装了一个消息队列。
- ❑ Looper 的 prepare 函数把这个 Looper 和调用 prepare 的线程（也就是最终的处理线程）绑定在一起了。
- ❑ 处理线程调用 loop 函数，处理来自该消息队列的消息。

当事件源向这个 Looper 发送消息的时候，其实是把消息加到这个 Looper 的消息队列里了。那么，该消息就将由和 Looper 绑定的处理线程来处理。可事件源又是怎么向 Looper 消息队列添加消息的呢？来看下一节。

3. Looper、Message 和 Handler 的关系

Looper、Message 和 Handler 之间也存在暧昧关系，不过要比 RefBase 那三个简单得多，用两句话就可以说清楚：

- ❑ Looper 中有一个 Message 队列，里面存储的是一个个待处理的 Message。
- ❑ Message 中有一个 Handler，这个 Handler 是用来处理 Message 的。

其中，Handler 类封装了很多琐碎的工作。先来认识一下这个 Handler。

5.4.2 Handler 分析

1. 初识 Handler

Handler 中所包括的成员：

👉 [-->Handler.java]

```
final MessageQueue mQueue; // Handler 中也有一个消息队列。
final Looper mLooper; // 也有一个 Looper。
final Callback mCallback; // 有一个回调用的类。
```

这几个成员变量是怎么使用的呢？这首先得分析 Handler 的构造函数。Handler 一共有四个构造函数，它们主要的区别是在对上面三个重要成员变量的初始化上。我们试对其进行逐一的分析。

👉 [-->Handler.java]

```
// 构造函数 1
public Handler() {
    // 获得调用线程的 Looper。
    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException(.....);
    }
}
```

```

        // 得到 Looper 的消息队列。
        mQueue = mLooper.mQueue;
        // 无 callback 设置。
        mCallback = null;
    }

    // 构造函数 2
    public Handler(Callback callback) {
        mLooper = Looper.myLooper();
        if (mLooper == null) {
            throw new RuntimeException(.....);
        }
        // 和构造函数 1 类似, 只不过多了一个设置 callback。
        mQueue = mLooper.mQueue;
        mCallback = callback;
    }

    // 构造函数 3
    public Handler(Looper looper) {
        mLooper = looper; //looper 由外部传入, 是哪个线程的 Looper 不确定。
        mQueue = looper.mQueue;
        mCallback = null;
    }

    // 构造函数 4, 和构造函数 3 类似, 只不过多了 callback 设置。
    public Handler(Looper looper, Callback callback) {
        mLooper = looper;
        mQueue = looper.mQueue;
        mCallback = callback;
    }
}

```

在上述构造函数中, Handler 中的消息队列变量最终都会指向 Looper 的消息队列, Handler 为何要如此做?

2. Handler 的真面目

根据前面的分析可知, Handler 中的消息队列实际就是某个 Looper 的消息队列, 那么, Handler 如此安排的目的何在?

在回答这个问题之前, 我先来问一个问题:

怎么往 Looper 的消息队列插入消息?

如果不知道 Handler, 这里有一个很原始的方法可解决上面这个问题:

- ☐ 调用 Looper 的 myQueue, 它将返回消息队列对象 MessageQueue。
- ☐ 构造一个 Message, 填充它的成员, 尤其是 target 变量。
- ☐ 调用 MessageQueue 的 enqueueMessage, 将消息插入消息队列。

这种原始方法的确很麻烦, 且极容易出错。但有了 Handler 后, 我们的工作就变得异常简单了。Handler 更像一个辅助类, 帮助我们简化编程的工作。

(1) Handler 和 Message

Handler 提供了一系列函数，帮助我们完成创建消息和插入消息队列的工作。这里只列举其中一二。要掌握详细的 API，则需要查看相关的文档。

```
// 查看消息队列中是否有消息码是 what 的消息。
final boolean      hasMessages(int what)
// 从 Handler 中创建一个消息码是 what 的消息。
final Message      obtainMessage(int what)
// 从消息队列中移除消息码是 what 的消息。
final void          removeMessages(int what)
// 发送一个只填充了消息码的消息。
final boolean       sendEmptyMessage(int what)
// 发送一个消息，该消息添加到队列尾。
final boolean       sendMessage(Message msg)
// 发送一个消息，该消息添加到队列头，所以优先级很高。
final boolean       sendMessageAtFrontOfQueue(Message msg)
```

只需对上面这些函数稍作分析，就能明白其他的函数。现以 sendMessage 为例，其代码如下所示：

 [-->Handler.java]

```
public final boolean sendMessage(Message msg)
{
    return sendMessageDelayed(msg, 0); // 调用 sendMessageDelayed
}
```

 [-->Handler.java]

```
// delayMillis 是以当前调用时间为基础的相对时间
public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    // 调用 sendMessageAtTime，把当前时间算上
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}
```

 [-->Handler.java]

```
//uptimeMillis 是绝对时间，即 sendMessageAtTime 函数处理的是绝对时间
public boolean sendMessageAtTime(Message msg, long uptimeMillis){
    boolean sent = false;
    MessageQueue queue = mQueue;
    if (queue != null) {
        // 把 Message 的 target 设置为自己，然后加入到消息队列中
        msg.target = this;
        sent = queue.enqueueMessage(msg, uptimeMillis);
    }
}
```

```

        return sent;
    }

```

看到上面这些函数我们可以预见，如果没有 Handler 的辅助，当我们自己操作 MessageQueue 的 enqueueMessage 时，得花费多大工夫！

Handler 把 Message 的 target 设为自己，是因为 Handler 除了封装消息添加等功能外还封装了消息处理的接口。

(2) Handler 的消息处理

刚才，我们往 Looper 的消息队列中加入了一个消息，按照 Looper 的处理规则，它在获取消息后会调用 target 的 dispatchMessage 函数，再把这个消息派发给 Handler 处理。Handler 在这块是如何处理消息的呢？

👉 [-->Handler.java]

```

public void dispatchMessage(Message msg) {
    // 如果 Message 本身有 callback，则直接交给 Message 的 callback 处理
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        // 如果本 Handler 设置了 mCallback，则交给 mCallback 处理
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        // 最后才是交给子类处理
        handleMessage(msg);
    }
}

```

dispatchMessage 定义了一套消息处理的优先级机制，它们分别是：

- ❑ Message 如果自带了 callback 处理，则交给 callback 处理。
- ❑ Handler 如果设置了全局的 mCallback，则交给 mCallback 处理。
- ❑ 如果上述都没有，该消息则会被交给 Handler 子类实现的 handleMessage 来处理。当然，这需从 Handler 派生并重载 handleMessage 函数。

在通常情况下，我们一般都是采用第三种方法，即在子类中通过重载 handleMessage 来完成处理工作的。

至此，Handler 知识基本上讲解完了，可是在实际编码过程中还有一个重要问题需要警惕，下一节内容就会谈及此问题。

5.4.3 Looper 和 Handler 的同步关系

Looper 和 Handler 会有什么同步关系呢？它们之间确实有同步关系，而且如果不注意此

关系，定会铸成大错！

同步关系肯定与多线程有关，我们来看下面的一个例子：

👉 [--> 例子 2]

```
// 先定义一个 LooperThread 类
class LooperThread extends Thread {
    public Looper myLooper = null; // 定义一个 public 的成员 myLooper，初值为空。
    public void run() { // 假设 run 在线程 2 中执行
        Looper.prepare();
        // myLooper 必须在这个线程中赋值
        myLooper = Looper.myLooper();
        Looper.loop();
    }
}
// 下面这段代码在线程 1 中执行，并且会创建线程 2
{
    LooperThread lpThread = new LooperThread();
    lpThread.start(); // start 后会创建线程 2
    Looper looper = lpThread.myLooper; // <===== 注意
    // thread2Handler 和线程 2 的 Looper 挂上钩
    Handler thread2Handler = new Handler(looper);
    // sendMessage 发送的消息将由线程 2 处理
    threadHandler.sendMessage(...)
}
```

上面这段代码的目的很简单：

- ❑ 线程 1 中创建线程 2，并且线程 2 通过 Looper 处理消息。
- ❑ 线程 1 中得到线程 2 的 Looper，并且根据这个 Looper 创建一个 Handler，这样发送给该 Handler 的消息将由线程 2 处理。

但很可惜，上面的代码是有问题的。如果我们熟悉多线程，就会发现标有“注意”的那行代码存在着严重问题。myLooper 的创建是在线程 2 中，而 looper 的赋值在线程 1 中，很有可能此时线程 2 的 run 函数还没来得及给 myLooper 赋值，这样线程 1 中的 looper 将取到 myLooper 的初值，也就是 looper 等于 null。另外，

```
Handler thread2Handler = new Handler(looper) 不能替换成
Handler thread2Handler = new Handler(Looper.myLooper())
```

这是因为，myLooper 返回的是调用线程的 Looper，即 Thread1 的 Looper，而不是我们想要的 Thread2 的 Looper。

对这个问题，可以采用同步的方式进行处理。你是不是有点迫不及待地想完善这个例子了？其实 Android 早就替我们想好了，它提供了一个 HandlerThread 来解决这个问题。

5.4.4 HandlerThread 介绍

HandlerThread 完美地解决了 myLooper 可能为空的问题。下面来看看它是怎么做的，代码如下所示：

👉 [-->HandlerThread]

```
public class HandlerThread extends Thread{
// 线程1 调用 getLooper 来获得新线程的 Looper
public Looper getLooper() {
    .....
    synchronized (this) {
        while (isAlive() && mLooper == null) {
            try {
                wait(); // 如果新线程还未创建 Looper, 则等待
            } catch (InterruptedException e) {
            }
        }
    }
    return mLooper;
}

// 线程2 运行它的 run 函数, looper 就是在 run 线程里创建的。
public void run() {
    mTid = Process.myTid();
    Looper.prepare(); // 创建这个线程上的 Looper
    synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll(); // 通知取 Looper 的线程1, 此时 Looper 已经创建好了。
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
}
```

HandlerThread 很简单，小小的 wait/ notifyAll 就解决了我们的难题。为了避免重复发明轮子，我们还是多用 HandlerThread 类吧！

5.5 本章小结

本章主要分析了 Android 代码中最常见的几个类：其中在 Native 层包括与对象生命周期相关的 RefBase、sp、wp、LightRefBase 类，以及 Android 为多线程编程提供的 Thread 类和相关的同步类；Java 层则包括使用最为广泛的 Handler 类和 Looper 类。另外，还分析了类 HandlerThread，它降低了创建和使用带有消息队列的线程的难度。