深入理解常见类

5.1 综述

初次接触Android源码，最多见到的一定是sp和wp。如果你只是沉迷于Java世界，那么Looper和Handler也是避不开的。本章的目的，就是把经常碰见的这些内容中的“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，就是仁者见仁，智者见智了。我个人觉得，可能Looper和Handler会相对更实用一些。

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。

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，揭密其相互间的暧昧关系。

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(A);

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，需要进去观察。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内部类相似，但不同的是，它需要一个显示的成员指向外部类对象，而Java内部类对象就有一个隐式的成员指向外部类对象。

说明：内部类在C++中的学名叫nested class（内嵌类）。

[-->RefBase.cpp::weakref\_imple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。

按上面的分析，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不成问题了。

2. sp上场

程序继续运行，现在到了

sp<A> spA(A);

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。）

[-->system/core/include/utils/StrongPointer.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：

**void RefBase::incStrong(const void\* id) const**

**{**

**//mRefs就是刚才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->incWeak(id);**

**refs->addStrongRef(id);**

**const int32\_t c = android\_atomic\_inc(&refs->mStrong);**

**ALOG\_ASSERT(c > 0, "incStrong() called on %p after last strong ref", refs);**

**#if PRINT\_REFS**

**ALOGD("incStrong of %p from %p: cnt=%d\n", this, id, c);**

**#endif**

**if (c != INITIAL\_STRONG\_VALUE) {**

**return;**

**}**

**android\_atomic\_add(-INITIAL\_STRONG\_VALUE, &refs->mStrong);**

**refs->mBase->onFirstRef();**

**}**

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

先来看第一个函数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。

调试版分支的代码很多，看来创造它们的人，也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，我们都已列出来了。以后再碰见它们，干脆就直接跳过的是：

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：

**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 \_\_unused = 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**

**ALOG\_ASSERT(c >= 0, "incWeak called on %p after last weak ref", this);**

**}**

**void RefBase::incStrong(const void\* id) const**

**{**

**......**

**//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);**

**ALOG\_ASSERT(c > 0, "incStrong() called on %p after last strong ref", refs);**

**#if PRINT\_REFS**

**ALOGD("incStrong of %p from %p: cnt=%d\n", this, id, c);**

**#endif**

**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。**

**refs->mBase->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知识，本章后面将对其做介绍。

（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RefBase中影子对象的强引用计数变为1，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，弱引用计数加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：

**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，将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，现在开始进入wp的析构。

（4）wp析构的影响

wp进入析构函数，这表明它快要离世了。

**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**

**}**

**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);**

**ALOG\_ASSERT(c >= 1, "decWeak called on %p too many times", this);**

**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\_STRONG) {**

**// This is the regular lifetime case. The object is destroyed**

**// when the last strong reference goes away. Since weakref\_impl**

**// outlive the object, it is not destroyed in the dtor, and**

**// we'll have to do it here.**

**if (impl->mStrong == INITIAL\_STRONG\_VALUE) {**

**// Special case: we never had a strong reference, so we need to**

**// destroy the object now.**

**delete impl->mBase;**

**} else {**

**// ALOGV("Freeing refs %p of old RefBase %p\n", this, impl->mBase);**

**delete impl;**

**}**

**} else {**

**// less common case: lifetime is OBJECT\_LIFETIME\_{WEAK|FOREVER}**

**impl->mBase->onLastWeakRef(id);**

**if ((impl->mFlags&OBJECT\_LIFETIME\_MASK) == OBJECT\_LIFETIME\_WEAK) {**

**// this is the OBJECT\_LIFETIME\_WEAK case. The last weak-reference**

**// is gone, we can destroy the object.**

**delete impl->mBase;**

**}**

**}**

**}**

OK，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**

**constint32\_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，可接着从delete this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**

**//的时候就保存mRefs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**

**constint32\_t c = android\_atomic\_dec(&impl->mWeak);**

**if (c!= 1) return;**

**//这次弱引用计数终于变为0，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(A);**

**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。按照前面所学的知识，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) {**

**//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，则强引用计数要增加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3，二进制表示是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**

**{**

**publicA()**

**{**

**extendObjectLifetime(OBJECT\_LIFETIME\_WEAK);//在构造函数中调用**

**}**

**}**

**int main()**

**{**

**A \*pA =new A();**

**wp<A> wpA(A);//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);**

**constint32\_t c = android\_atomic\_dec(&refs->mStrong);**

**if (c == 1) { //上面原子操作后，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);**

**constint32\_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，调用上边的原子减操作后**

**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， 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，我直接的答案是：

· 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(\_\_attribute\_\_((unused)) 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(\_\_attribute\_\_((unused)) const void\* id) const {**

**if (android\_atomic\_dec(&mCount) == 1) {**

**//decStrong的时候减1，当引用计数变为零的时候，delete掉自己**

**delete static\_cast<const T\*>(this);**

**}**

**}**

**//! DEBUGGING ONLY: Get current strong ref count.**

**inline int32\_t getStrongCount() const {**

**return mCount;**

**}**

**typedef LightRefBase<T> basetype;**

**protected:**

**inline ~LightRefBase() { }**

**private:**

**friend class ReferenceMover;**

**inline static void renameRefs(size\_t n, const ReferenceRenamer& renamer) { }**

**inline static void renameRefId(T\* ref,**

**const void\* old\_id, const void\* new\_id) { }**

**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，但里边的关系，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的方式也需要很多时间。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？由于有了这样的构想，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类以及常用同步类的分析

Thread类是Android为线程操作而做的一个封装。代码在Threads.cpp中，其中还封装了一些与线程同步相关（既然是封装，要掌握它，最重要的当然是与Pthread相关的知识）的类。我们拟先行分析Thread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，但它构造函数中的那个canCallJava却一度使我感到费解。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：

**[-->Threads.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Threads.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：

**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，初始化时和mCanCallJava为false时使用的是同一个

//线程创建函数。那么有地方会修改它吗？

**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，很可惜，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

在第四章4.2.1的第2小节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);**

**ALOGV("--- registering native functions ---\n");**

**env->PushLocalFrame(200);**

**if (register\_jni\_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {**

**env->PopLocalFrame(NULL);**

**return -1;**

**}**

**env->PopLocalFrame(NULL);**

**//createJavaThread("fubar", quickTest, (void\*) "hello");**

**return 0;**

**}**

所以，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]**

**/\*static\*/ int AndroidRuntime::javaCreateThreadEtc(**

**android\_thread\_func\_t entryFunction,**

**void\* userData,**

**const char\* threadName,**

**int32\_t threadPriority,**

**size\_t threadStackSize,**

**android\_thread\_id\_t\* threadId)**

**{**

**// javaThreadShell must free**

**void\*\* args = (void\*\*) malloc(3 \* sizeof(void\*));**

**int result;**

**if (!threadName)**

**threadName = "unnamed thread";**

**args[0] = (void\*) entryFunction;**

**args[1] = userData;**

**args[2] = (void\*) strdup(threadName); // javaThreadShell must free**

**//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. 费力而讨好

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。

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，但是最后未detach的线程如果有，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JNIcheck。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

不论一分为二是如何处理的，最终的线程函数\_threadLoop都会被调用，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();**

**#ifdef HAVE\_ANDROID\_OS**

**// this is very useful for debugging with gdb**

**self->mTid = gettid();**

**#endif**

**bool first = true;**

**do {**

**bool result;**

**if (first) {**

**first = false;**

**//self代表继承Thread类的对象，第一次进来将调用readyToRun，看看是否准备好**

**self->mStatus = self->readyToRun();**

**result = (self->mStatus == NO\_ERROR);**

**if (result && !self->exitPending()) {**

**// Binder threads (and maybe others) rely on threadLoop**

**// running at least once after a successful ::readyToRun()**

**// (unless, of course, the thread has already been asked to exit**

**// at that point).**

**// This is because threads are essentially used like this:**

**// (new ThreadSubclass())->run();**

**// The caller therefore does not retain a strong reference to**

**// the thread and the thread would simply disappear after the**

**// successful ::readyToRun() call instead of entering the**

**// threadLoop at least once.**

**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();**

**}**

**// establish a scope for mLock**

**{**

**Mutex::Autolock \_l(self->mLock);**

**/\***

**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->mRunning = false;**

**// clear thread ID so that requestExitAndWait() does not exit if**

**// called by a new thread using the same thread ID as this one.**

**self->mThread = thread\_id\_t(-1);**

**// note that interested observers blocked in requestExitAndWait are**

**// awoken by broadcast, but blocked on mLock until break exits scope**

**self->mThreadExitedCondition.broadcast();**

**break;**

**}**

**}**

**// Release our strong reference, to let a chance to the thread**

**// to die a peaceful death.**

**strong.clear();**

**// And immediately, re-acquire a strong reference for the next loop**

**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回避的话题，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（这种函数我们也称之为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

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，保证一次只能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”，你必须等里边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：

**inline Mutex::Mutex(int type, \_\_attribute\_\_((unused)) 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);**

**}**

**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，它们的用法如下：

· 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与它相关的知识。在Android系统中，多线程也是常见和重要的编程手段，务请大家重视。

Mutex类确实比Raw API方便好用，不过还是稍显麻烦。来看下一节。

**[-->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一种使用场景：

· 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**

**};**

**enum WakeUpType {**

**WAKE\_UP\_ONE = 0,**

**WAKE\_UP\_ALL = 1**

**};**

**Condition();**

**Condition(int type); //如果type是SHARED，表示支持跨进程的条件同步**

**~Condition();**

**// Wait on the condition variable. Lock the mutex before calling.**

**//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);**

**// same with relative timeout**

**//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);**

**// Signal the condition variable, allowing exactly one thread to continue.**

**//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();**

**// Signal the condition variable, allowing one or all threads to continue.**

**void signal(WakeUpType type) {**

**if (type == WAKE\_UP\_ONE) {**

**signal();**

**} else {**

**broadcast();**

**}**

**}**

**// Signal the condition variable, allowing all threads to continue.**

**//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:**

**#if defined(HAVE\_PTHREADS)**

**pthread\_cond\_t mCond;**

**#else**

**void\* mState;**

**#endif**

**};**

**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) {**

**#if defined(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，所以不同系统需要调用不同的函数**

**#else // HAVE\_PTHREAD\_COND\_TIMEDWAIT\_RELATIVE**

**struct timespec ts;**

**#if defined(HAVE\_POSIX\_CLOCKS)**

**clock\_gettime(CLOCK\_REALTIME, &ts);**

**#else // HAVE\_POSIX\_CLOCKS**

**// we don't support the clocks here.**

**struct timeval t;**

**gettimeofday(&t, NULL);**

**ts.tv\_sec = t.tv\_sec;**

**ts.tv\_nsec= t.tv\_usec\*1000;**

**#endif // HAVE\_POSIX\_CLOCKS**

**ts.tv\_sec += reltime/1000000000;**

**ts.tv\_nsec+= reltime%1000000000;**

**if (ts.tv\_nsec >= 1000000000) {**

**ts.tv\_nsec -= 1000000000;**

**ts.tv\_sec += 1;**

**}**

**return -pthread\_cond\_timedwait(&mCond, &mutex.mMutex, &ts);**

**#endif // HAVE\_PTHREAD\_COND\_TIMEDWAIT\_RELATIVE**

**}**

**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？

· 上面代码中，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类使用的印象。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：

**status\_t Thread::requestExitAndWait()**

**{**

**Mutex::Autolock \_l(mLock); //使用Autolock，mLock被锁住**

**if (mThread == getThreadId()) {**

**ALOGW(**

**"Thread (this=%p): don't call waitForExit() from this "**

**"Thread object's thread. It's a guaranteed deadlock!",**

**this);**

**return WOULD\_BLOCK;**

**}**

**mExitPending = true;**

**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**

**前边推荐的《Programming with POSIX Thread》一书。**

**\*/**

**mThreadExitedCondition.wait(mLock);**

**}**

**// This next line is probably not needed any more, but is being left for**

**// historical reference. Note that each interested party will clear flag.**

**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，什么地方会触发这个条件呢？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。

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做为切入点，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，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\_flags++或者g\_flags—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\_flags++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，抢先在线程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的使用比它所要保护的内容还复杂，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，这样同一时刻只能有一个CPU访问总线。

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。

\*/

INT32i = 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]，注意该文件位置在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的和其他系统上的相同，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：

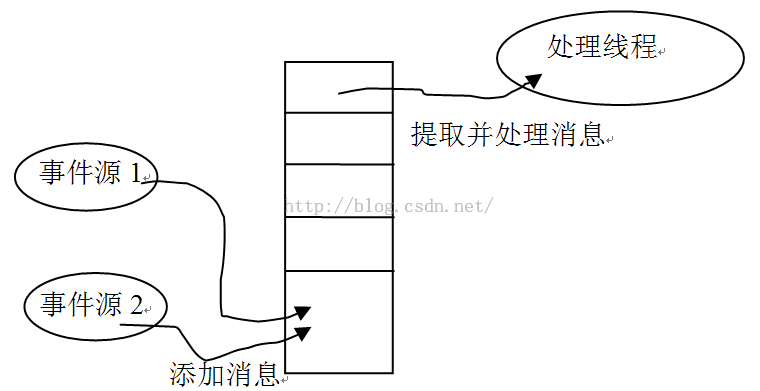


图5-1 线程和消息处理原理图

从图中可以看出：

· 事件源把待处理的消息加入到消息队列，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，也可以是来自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类的分析

我们以Looper使用的一个常见例子来分析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，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：

**private static void prepare(boolean quitAllowed) {**

**//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(quitAllowed));**

**}**

//sThreadLocal定义

static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();

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：

**private Looper(boolean quitAllowed) {**

**//构造一个消息队列**

**mQueue = new MessageQueue(quitAllowed);**

**//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Thread Local Variable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：

· Looper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。那么，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，里边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了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做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：

· 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。

2.1 Handler和Message

Handle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;**

**}**

看到上面这些函数可以想见，如果没有Handler的辅助，当我们自己操作MessageQueue的enqueueMessage时，得花费多大功夫！

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。

2.2 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，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，这需要从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。下一节内容就将谈及此问题。

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，定要铸成大错！

同步关系肯定和多线程有关，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的赋值则在线程1，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。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，还分析了方类HandlerThread，它降低了创建和使用带有消息队列的线程的难度。