android底层通过引用计数实现智能指针，并且实现类强指针 sp类以及弱指针 wp类

这两个类都是模板类，模板类型必须继承自RefBase类

定义在

/system/core/include/utils/RefBase.h

/system/core/include/utils/StrongPointer.h

实现在

/system/core/libutils/RefBase.cpp

当一个指针的强引用计数为0的时候，释放指针

# [Android指针管理：RefBase,SP,WP](http://www.cnblogs.com/angeldevil/archive/2013/03/10/2952586.html)

Android中通过引用计数来实现智能指针，并且实现有强指针与弱指针。由对象本身来提供引用计数器，但是对象不会去维护引用计数器的值，而是由智能指针来管理。

要达到所有对象都可用引用计数器实现智能指针管理的目标，可以定义一个公共类，提供引用计数的方法，所有对象都去继承这个公共类，这样就可以实现所有对象都可以用引用计数来管理的目标，在Android中，这个公共类就是RefBase，同时还有一个简单版本LightRefBase。

RefBase作为公共基类提供了引用计数的方法，但是并不去维护引用计数的值，而是由两个智能指针来进行管理：sp(Strong Pointer)和wp(Weak Pointer)，代表强引用计数和弱引用计数。

# **一、轻量级引用计数的实现：LightRefBase**

**LightRefBase的实现很简单，只是内部保存了一个变量用于保存对象被引用的次数**，并提供了两个函数用于增加或减少引用计数。

template <class T>

class LightRefBase

{

public:

inline LightRefBase() : mCount(0) { }

inline void incStrong(const void\* id) const {

android\_atomic\_inc(&mCount);

}

inline void decStrong(const void\* id) const {

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

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:

mutable volatile int32\_t mCount;

};



# **二、sp(Strong Pointer)**

LightRefBase仅仅提供了引用计数的方法，具体引用数应该怎么管理，就要通过智能指针类来管理了，每当有一个智能指针指向对象时，对象的引用计数要加1，当一个智能指针取消指向对象时，对象的引用计数要减1，在C＋＋中，当一个对象生成和销毁时会自动调用（拷贝）构造函数和析构函数，所以，对对象引用数的管理就可以放到智能指针的（拷贝）构造函数和析构函数中。Android提供了一个**智能指针可以配合LightRefBase使用：sp**，sp的定义如下：



template <typename T>

class sp

{

public:

inline sp() : m\_ptr(0) { }

sp(T\* other);

sp(const sp<T>& other);

template<typename U> sp(U\* other);

template<typename U> sp(const sp<U>& other);

~sp();

// Assignment

sp& operator = (T\* other);

sp& operator = (const sp<T>& other);

template<typename U> sp& operator = (const sp<U>& other);

template<typename U> sp& operator = (U\* other);

//! Special optimization for use by ProcessState (and nobody else).

void force\_set(T\* other);

// Reset

void clear();

// Accessors

inline T& operator\* () const { return \*m\_ptr; }

inline T\* operator-> () const { return m\_ptr; }

inline T\* get() const { return m\_ptr; }

// Operators

COMPARE(==)

COMPARE(!=)

COMPARE(>)

COMPARE(<)

COMPARE(<=)

COMPARE(>=)

private:

template<typename Y> friend class sp;

template<typename Y> friend class wp;

void set\_pointer(T\* ptr);

T\* m\_ptr;

};



代码比较多，其中Accessors部分代码重载了\*、->操作符使我们使用sp的时候就像使用真实的对象指针一样，可以直接操作对象的属性或方法，COMPARE是宏定义，用于重载关系操作符，由于对引用计数的控制主要是由（拷贝）构造函数和析构函数控制，所以忽略其他相关代码后，sp可以精简为如下形式（赋值操作符也省略掉了，构造函数省略相似的两个）：



template <typename T>

class sp

{

public:

inline sp() : m\_ptr(0) { }

sp(T\* other);

sp(const sp<T>& other);

~sp();

private:

template<typename Y> friend class sp;

template<typename Y> friend class wp;

void set\_pointer(T\* ptr);

T\* m\_ptr;

};



默认构造函数使智能指针不指向任何对象，sp(T\* other)与sp(const sp<T>& other)的实现如下：



template<typename T>

sp<T>::sp(T\* other)

: m\_ptr(other)

{

if (other) other->incStrong(this);

}

template<typename T>

sp<T>::sp(const sp<T>& other)

: m\_ptr(other.m\_ptr)

{

if (m\_ptr) m\_ptr->incStrong(this);

}



内部变量m\_ptr指向实际对象，并调用实际对象的incStrong函数，T继承自LightRefBase，所以此处调用的是LightRefBase的incStrong函数，之后实际对象的引用计数加1。

当智能指针销毁的时候调用智能指针的析构函数：

template<typename T>

sp<T>::~sp()

{

if (m\_ptr) m\_ptr->decStrong(this);

}

调用实际对象即LightRefBase的decStrong函数，其实现如下：

inline void decStrong(const void\* id) const {

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

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

}

}

android\_atomic\_dec返回mCount减1之前的值，如果返回1表示这次减过之后引用计数就是0了，就把对象delete掉。

# **三、RefBase**

RefBase提供了更强大的引用计数的管理。



class RefBase

{

public:

void incStrong(const void\* id) const;

void decStrong(const void\* id) const;

void forceIncStrong(const void\* id) const;

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

int32\_t getStrongCount() const;

class weakref\_type

{

public:

RefBase refBase() const;

void incWeak(const void\* id);

void decWeak(const void\* id);

// acquires a strong reference if there is already one.

bool attemptIncStrong(const void\* id);

// acquires a weak reference if there is already one.

// This is not always safe. see ProcessState.cpp and BpBinder.cpp

// for proper use.

bool attemptIncWeak(const void\* id);

//! DEBUGGING ONLY: Get current weak ref count.

int32\_t getWeakCount() const;

//! DEBUGGING ONLY: Print references held on object.

void printRefs() const;

//! DEBUGGING ONLY: Enable tracking for this object.

// enable -- enable/disable tracking

// retain -- when tracking is enable, if true, then we save a stack trace

// for each reference and dereference; when retain == false, we

// match up references and dereferences and keep only the

// outstanding ones.

void trackMe(bool enable, bool retain);

};

weakref\_type\* createWeak(const void\* id) const;

weakref\_type\* getWeakRefs() const;

// DEBUGGING ONLY: Print references held on object.

inline void printRefs() const { getWeakRefs()->printRefs(); }

// DEBUGGING ONLY: Enable tracking of object.

inline void trackMe(bool enable, bool retain)

{

getWeakRefs()->trackMe(enable, retain);

}

typedef RefBase basetype;

protected:

RefBase();

virtual ~RefBase();

//! Flags for extendObjectLifetime()

enum {

OBJECT\_LIFETIME\_STRONG = 0x0000,

OBJECT\_LIFETIME\_WEAK = 0x0001,

OBJECT\_LIFETIME\_MASK = 0x0003

};

void extendObjectLifetime(int32\_t mode);

//! Flags for onIncStrongAttempted()

enum {

FIRST\_INC\_STRONG = 0x0001

};

virtual void onFirstRef();

virtual void onLastStrongRef(const void\* id);

virtual bool onIncStrongAttempted(uint32\_t flags, const void\* id);

virtual void onLastWeakRef(const void\* id);

private:

friend class weakref\_type;

class weakref\_impl;

RefBase(const RefBase& o);

RefBase& operator=(const RefBase& o);

weakref\_impl\* const mRefs;

};



不同于LightRefBase的是，RefBase内部并没有使用一个变量来维护引用计数，而是通过一个weakref\_impl \*类型的成员来维护引用计数，并且同时提供了强引用计数和弱引用计数。weakref\_impl继承于RefBase::weakref\_type，代码比较多，不过大都是调试代码，由宏定义分开，Release是不包含调试代码的，去除这些代码后其定义为：



#define INITIAL\_STRONG\_VALUE (1<<28)

class RefBase::weakref\_impl : public RefBase::weakref\_type

{

public:

volatile int32\_t mStrong;

volatile int32\_t mWeak;

RefBase\* const mBase;

volatile int32\_t mFlags;

weakref\_impl(RefBase\* base)

: mStrong(INITIAL\_STRONG\_VALUE)

, mWeak(0)

, mBase(base)

, mFlags(0)

{

}

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) { }

};



weakref\_impl中的函数都是作为调试用，Release版的实现都是空的，成员变量分别表示强引用数、弱引用数、指向实际对象的指针与flag，flag可控制实际对象的生命周期，取值为0或RefBase中定义的枚举值。

RefBase提供了incStrong与decStrong函数用于控制强引用计数值，其弱引用计数值是由weakref\_impl控制，强引用计数与弱引用数都保存在weakref\_impl \*类型的成员变量mRefs中。

RefBase同LightRefBase一样为对象提供了引用计数的方法，对引用计数的管理同样要由智能指针控制，由于RefBase同时实现了强引用计数与弱引用计数，所以就有两种类型的智能指针，sp(Strong Pointer)与wp(Weak Pointer)。

sp前面已经说过，其（拷贝）构造函数调用对象即RefBase的incStrong函数。



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

{

weakref\_impl\* const refs = mRefs;

refs->incWeak(id);

refs->addStrongRef(id);

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

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

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

return;

}

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

refs->mBase->onFirstRef();

}



addStrong的函数体为空，incStrong函数内部首先调用成员变量mRefs的incWeak函数将弱引用数加1，然后再将强引用数加1，由于android\_atomic\_inc返回变量的旧值，所以如果其不等于INITIAL\_STRONG\_VALUE就直接返回，则则是第一次由强智能指针（sp）引用，将其减去INITIAL\_STRONG\_VALUE后变成1，然后调用对象的onFirstRef。

成员变量mRefs是在对象的构造函数中初始化的：

RefBase::RefBase()

: mRefs(new weakref\_impl(this))

{

}

weakrel\_impl的incWeak继承自父类weakrel\_type的incWeak：



void RefBase::weakref\_type::incWeak(const void\* id)

{

weakref\_impl\* const impl = static\_cast<weakref\_impl\*>

impl->addWeakRef(id);

const int32\_t c = android\_atomic\_inc(&impl->mWeak);

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

}



addWeakRef实现同样为空，所以只是将弱引用计数加1。所以当对象被sp引用后，强引用计数与弱引用计数会同时加1。

当sp销毁时其析构函数调用对象即RefBase的decStrong函数：



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) {

const\_cast<RefBase\*>(this)->onLastStrongRef(id);

if ((refs->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK) {

delete this;

}

}

refs->removeWeakRef(id);

refs->decWeak(id);

}



decStrong中将强引用数与弱引用数同时减1，如果这是最后一个强引用的话，会调用对象的onLastStrongRef，并且判断成员变量mRefs的成员变量mFlags来决定是否在对象的强引用数为0时释放对象。

mFlags可以为0或以下两个枚举值：

enum {

OBJECT\_LIFETIME\_WEAK = 0x0001,

OBJECT\_LIFETIME\_FOREVER = 0x0003

};

mFlags的值可以通过extendObjectLifetime函数改变：

void RefBase::extendObjectLifetime(int32\_t mode)

{

android\_atomic\_or(mode, &mRefs->mFlags);

}

OBJECT\_LIFETIME\_FOREVER包含OBJECT\_LIFETIME\_WEAK（位运算中其二进制11包含01），所以当

refs->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK

为true时表示mFlags为0，实际对象的生命周期受强引用数控制，所以在强引用数为0时delete this，否则实际对象的生命周期就由弱引用数控制。

再来看decWeak：



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;

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;

}

}

}



将弱引用数减1，若减1后不为0直接返回，否则判断

(impl->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK

若判断结果为true：

    实际对象生命周期被强引用数控制，接下来判断：

mpl->mStrong == INITIAL\_STRONG\_VALUE

1. 如果判断为true表示对象只被弱引用引用过，现在弱引用数为0，直接删除实际对象。
2. 如果判断为false，表示对象曾经被强引用引用过，但现在强引用为变为0了（因为增加或减小强引用数时一定同时增加或减小弱引用数，所以弱引用数为0时，强引用数一定为0），弱引用数为0了，直接释放mRefs，而实际对象由于受强引用数控制，已经在RefBase::decStrong中被delete了。

若判断结果为false:

    判断mFlgs是否是OBJECT\_LIFETIME\_FOREVER，如果是，什么都不作由用户自己控制对象的生命周期，否则，实际对象的生命周期受弱引用数控制，现在弱引用数为0，delete实际对象。

# **四、wp(Weak Pointer)**

定义如下：



template <typename T>

class wp

{

public:

typedef typename RefBase::weakref\_type weakref\_type;

inline wp() : m\_ptr(0) { }

wp(T\* other);

wp(const wp<T>& other);

wp(const sp<T>& other);

template<typename U> wp(U\* other);

template<typename U> wp(const sp<U>& other);

template<typename U> wp(const wp<U>& other);

~wp();

// Assignment

wp& operator = (T\* other);

wp& operator = (const wp<T>& other);

wp& operator = (const sp<T>& other);

template<typename U> wp& operator = (U\* other);

template<typename U> wp& operator = (const wp<U>& other);

template<typename U> wp& operator = (const sp<U>& other);

void set\_object\_and\_refs(T\* other, weakref\_type\* refs);

// promotion to sp

sp<T> promote() const;

// Reset

void clear();

// Accessors

inline weakref\_type\* get\_refs() const { return m\_refs; }

inline T\* unsafe\_get() const { return m\_ptr; }

// Operators

COMPARE(==)

COMPARE(!=)

COMPARE(>)

COMPARE(<)

COMPARE(<=)

COMPARE(>=)

private:

template<typename Y> friend class sp;

template<typename Y> friend class wp;

T\* m\_ptr;

weakref\_type\* m\_refs;

};



同sp一样，m\_ptr指向实际对象，但wp还有一个成员变量m\_refs。



template<typename T>

wp<T>::wp(T\* other)

: m\_ptr(other)

{

if (other) m\_refs = other->createWeak(this);

}

template<typename T>

wp<T>::wp(const wp<T>& other)

: m\_ptr(other.m\_ptr), m\_refs(other.m\_refs)

{

if (m\_ptr) m\_refs->incWeak(this);

}

RefBase::weakref\_type\* RefBase::createWeak(const void\* id) const

{

mRefs->incWeak(id);

return mRefs;

}



可以看到，wp的m\_refs就是RefBase即实际对象的mRefs。

wp析构的时候减少弱引用计数：

template<typename T>

wp<T>::~wp()

{

if (m\_ptr) m\_refs->decWeak(this);

}

由于弱指针没有重载\*与->操作符，所以不能直接操作指向的对象，虽然有unsafe\_get函数，但像名字所示的，不建议使用，直接使用实际对象指针的话就没必要用智能指针了。

因为弱指针不能直接操作对象，所以要想操作对象的话就要将其转换为强指针，即wp::promote方法：



template<typename T>

sp<T> wp<T>::promote() const

{

return sp<T>(m\_ptr, m\_refs);

}

template<typename T>

sp<T>::sp(T\* p, weakref\_type\* refs)

: m\_ptr((p && refs->attemptIncStrong(this)) ? p : 0)

{

}



是否能从弱指针生成一个强指针关键是看refs->attemptIncStrong，看其定义：



bool RefBase::weakref\_type::attemptIncStrong(const void\* id)

{

incWeak(id);

weakref\_impl\* const impl = static\_cast<weakref\_impl\*>(this);

int32\_t curCount = impl->mStrong;

LOG\_ASSERT(curCount >= 0, "attemptIncStrong called on %p after underflow",

this);

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;

if (curCount == INITIAL\_STRONG\_VALUE) {

// Attempting to acquire first strong reference... this is allowed

// if the object does NOT have a longer lifetime (meaning the

// implementation doesn't need to see this), or if the implementation

// allows it to happen.

allow = (impl->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK

|| impl->mBase->onIncStrongAttempted(FIRST\_INC\_STRONG, id);

} else {

// Attempting to revive the object... this is allowed

// if the object DOES have a longer lifetime (so we can safely

// call the object with only a weak ref) and the implementation

// allows it to happen.

allow = (impl->mFlags&OBJECT\_LIFETIME\_WEAK) == OBJECT\_LIFETIME\_WEAK

&& impl->mBase->onIncStrongAttempted(FIRST\_INC\_STRONG, id);

}

if (!allow) {

decWeak(id);

return false;

}

curCount = android\_atomic\_inc(&impl->mStrong);

// If the strong reference count has already been incremented by

// someone else, the implementor of onIncStrongAttempted() is holding

// an unneeded reference. So call onLastStrongRef() here to remove it.

// (No, this is not pretty.) Note that we MUST NOT do this if we

// are in fact acquiring the first reference.

if (curCount > 0 && curCount < INITIAL\_STRONG\_VALUE) {

impl->mBase->onLastStrongRef(id);

}

}

impl->addWeakRef(id);

impl->addStrongRef(id);

#if PRINT\_REFS

LOGD("attemptIncStrong of %p from %p: cnt=%d\n", this, id, curCount);

#endif

if (curCount == INITIAL\_STRONG\_VALUE) {

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

impl->mBase->onFirstRef();

}

return true;

}



首先通过incWeak将弱引用数加1（被强指针sp引用会导致强引用数和弱引用数同时加1），然后：



int32\_t curCount = impl->mStrong;

while (curCount > 0 && curCount != INITIAL\_STRONG\_VALUE) {

if (android\_atomic\_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {

break;

}

curCount = impl->mStrong;

}



如果之前已经有强引用，直接将强引用数加1，android\_atomic\_cmpxchg表示如果impl->mStrong的值为curCount，则把impl->mString的值改为curCount+1，此处用while循环是防止其他线程已经增加了强引用数。

接下来：

if (curCount <= 0 || curCount == INITIAL\_STRONG\_VALUE)

表示对象目前没有强引用，这就要判断对象是否存在了。

如果curCount == INITIAL\_STRONG\_VALUE，表示对象没有被sp引用过。接下来判断：

allow = (impl->mFlags&OBJECT\_LIFETIME\_WEAK) != OBJECT\_LIFETIME\_WEAK

|| impl->mBase->onIncStrongAttempted(FIRST\_INC\_STRONG, id);

表示：如果对象的生命周期只受强引用控制，对象一定存在，要有强引用才可以管理对象的释放，所以一定会允许生成强引用；如果对象的生命周期受弱引用控制，调用对象的onIncStrongAttempted试图增加强引用，由于此时在弱引用中，弱引用一定不为0，对象也一定存在，调用onIncStrongAttempted的意图是因为类的实现者可能不希望用强引用引用对象。在RefBase中onIncStrongAttempted默认返回true：

bool RefBase::onIncStrongAttempted(uint32\_t flags, const void\* id)

{

return (flags&FIRST\_INC\_STRONG) ? true : false;

}

如果curCount <= 0（只会等于0），表示对象强引用数经历了INITIAL\_STRONG\_VALUE -->大于0 --> 0，接下来就要判断：

allow = (impl->mFlags&OBJECT\_LIFETIME\_WEAK) == OBJECT\_LIFETIME\_WEAK

&& impl->mBase->onIncStrongAttempted(FIRST\_INC\_STRONG, id);

如果对象的生命周期受强引用数控制，那么由于曾被sp引用过，现在强引用数又为0，对象就已经被delete了，所以就不能生成强引用，否则如果对象的生命周期受弱引用数控制，就通过onIncStrongAttempted看类的实现者是否希望当对象的强引用数变为0时可以再次被强引用引用。

if (!allow) {

decWeak(id);

return false;

}

如果allow为false表示不能从弱引用生成强引用，就要调用decWeak将弱引用减1（因为在promote入口先将弱引用加了1），然后返回false表示生成强引用失败。

if (curCount == INITIAL\_STRONG\_VALUE) {

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

impl->mBase->onFirstRef();

}

最后，如果curCount == INITIAL\_STRONG\_VALUE表示第一次被sp引用，调用对象的onFirstRef函数。

# 五、总结

RefBase内部有一个指针指向实际对象，有一个weakref\_impl类型的指针保存对象的强／弱引用计数、对象生命周期控制。

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weakref\_impl \*对象用于保存实际对象的引用计数。sp 管理一个对象指针时，对象的强、弱引用数同时加1，sp销毁时，对象的强、弱引用数同时减1。

wp中有两个成员变量，一个保存实际对象，另一个是weakref\_impl \*对象。wp管理一个对象指针时，对象的弱引用计数加1，wp销毁时，对象的弱引用计数减1。

weakref\_impl中包含一个flag用于决定对象的生命周期是由强引用数控制还是由弱引用数控制：

* 当flag为0时，实际对象的生命周期由强引用数控制，weakref\_impl \*对象由弱引用数控制。
* 当flag为OBJECT\_LIFETIME\_WEAK时，实际对象的生命周期受弱引用数控制。
* 当flag为OBJECT\_LIFETIME\_FOREVER时，实际对象的生命周期由用户控制。

        可以用extendObjectLifetime改变flag的值。