

Android指针管理：RefBase,SP,WP

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);
```

个人信息

angeldeviljy@gmail.com

我的个人博客

我的CSDN

昵称：AngelDevil

园龄：5年2个月

粉丝：301

关注：3

[+加关注](#)

常用链接

[我的随笔](#)

[我的评论](#)

[我的参与](#)

[最新评论](#)

[我的标签](#)

[更多链接](#)

我的标签

[android \(30\)](#) [快速a](#)

[binder \(3\)](#) [Java \(2\)](#)

[JavaScript \(2\)](#) [Java](#)

[JVM \(1\)](#) [layout_we](#)

[looper \(1\)](#) [messag](#)

随笔分类(50)

[android\(30\)](#)

[Android Framework](#)

[C++\(1\)](#)

[Java\(1\)](#)

[JavaScript\(2\)](#)

[程序设计\(1\)](#)

[读书笔记\(2\)](#)

[技术相关\(1\)](#)

[快速Android开发系](#)

积分与排名

积分 - 99351

排名 - 1887

最新评论

```
~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)
```



1. Re:快速Android开
Volley

不错

2. Re:快速Android开
EventBus

@sclgxt不需要注
定义一个不会用到的c
果onEvent(Object e
用register如果在当前
一个onEvent开头的

3. Re:快速Android开
EventBus

Fragment之中可
个父类的Fragment之
还需注册吗

4. Re:快速Android开
EventBus

介绍的很详细，非

5. Re:快速Android开
Android-Async-Htt

好！！

阅读排行榜

1. Android动画学习:
Animation(123949)
2. Android自定义对
置,大小(67899)
3. 快速Android开发:
Android-Async-Htt
4. 自定义SimpleAda
5. 快速Android开发:
EventBus(40016)

评论排行榜

1. Git与Repo入门(5
2. 快速Android开发:
EventBus(33)
3. Android动画学习:
Animation(17)
4. Android中自定义
造函数中的第三个参
(15)
5. android中layout

推荐排行榜

1. Git与Repo入门(1
2. Android动画学习:
Animation(62)
3. 快速Android开发:
EventBus(28)
4. Android中自定义

```

: 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来决定是否释放对象。

mFlags可以为0或以下两

4

(请您对文章做出评价)

```
enum {
    OBJECT_LIFETIME_WEAK =
    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mFlags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);

#ifdef 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->mStrong`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。

作者：AngelDevil
出处：www.cnblogs.com/angeldevil
欢迎访问我的个人站点：angeldevil.me
转载请注明出处！

分类：android , Android Frameworks

标签：android

好文要顶

关注我

收藏该文







AngelDevil
关注 - 3
粉丝 - 301
[+ 加关注](#)

« 上一篇：Android图片异步加载

» 下一篇：Android编译系统

posted @ 2013-03-10 12:48 AngelDevil 阅读(7571) 评论(0) 编辑 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

[博客园首页](#) [博问](#) [新闻](#) [闪存](#) [程序员招聘](#) [知识库](#)

- 最新IT新闻：**
- 男子弄丢Apple ID iPhone 6、MacBook全部“阵亡”
 - 传戴尔出售旗下软件开发商减轻收购EMC压力
 - 微软发布多款Windows 10配件产品
 - 苹果收购前美信半导体芯片工厂 同三星做邻居
 - 互联网公司为何那么看重媒体
- » 更多新闻...

- 最新知识库文章：**
- Linux概念架构的理解
 - 从涂鸦到发布——理解API的设计过程
 - 好的架构是进化来的，不是设计来的
 - 被误解的MVC和被神化的MVVM
 - 再谈设计和编码
- » 更多知识库文章...