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。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);

}

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:

#if defined(HAVE\_PTHREADS)

pthread\_cond\_t mCond;

#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) {

#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。

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;

}