TODO

<https://blog.csdn.net/csdn_of_coder/article/details/72792866>

Android系统时间更新主要涉及到两种时间：NITZ时间和NTP时间。NITZ时间与手机服务运营商关系密切；而NTP网络时间则较为独立，访问某个服务器，获取时间即可。NTP时间获取的过程较为单一，且由于工作中主要是涉及到Android NTP时间的获取；所以这里先只介绍Android系统获取NTP时间的流程。

NTP其实是一种协议，Android系统中通过一个Socket向一个提供NTP服务的服务器发送请求来获取NTP时间。既然借助于Socket通信，那么功能实现的过程中就会涉及到服务器地址、请求超时时间等信息的配置。这部分我们在后续的分析过程中都会看到。

# SystemClock

*/\*\*  
 \* Returns milliseconds since boot, including time spent in sleep.  
 \*  
 \** ***@return*** *elapsed milliseconds since boot.  
 \*/***native public static long** elapsedRealtime();

*/\*\*  
 \* Returns milliseconds since boot, not counting time spent in deep sleep.  
 \*  
 \** ***@return*** *milliseconds of non-sleep uptime since boot.  
 \*/***native public static long** uptimeMillis();

# TODO

<http://www.voidcn.com/article/p-xehsnzzs-ev.html>

<https://www.jianshu.com/p/ca10702401ba>

<https://blog.csdn.net/yin1031468524/article/details/65447849>

SntpClient原理

Dump机制

getActiveNetworkInfo这个判断是有问题的

**final** NetworkInfo ni = mCM == **null** ? **null** : mCM.getActiveNetworkInfo();  
**if** (ni == **null** || !ni.isConnected()) {  
 **if** (LOGD) Log.d(TAG, **"forceRefresh: no connectivity"**);  
 **return false**;  
}

# 启动流程

## systemserver进程启动时

Android NTP时间更新主要由NetworkUpdateService服务处理，它决定何时更新NTP时间。在systemserver进程启动时，它会被初始化；同时，它是Binder的子类：

networkTimeUpdater = new NetworkTimeUpdateService(context);

ServiceManager.addService("network\_time\_update\_service", networkTimeUpdater);

首先创建服务实例，接着将该服务发布到系统中。创建完服务对象，并当系统准备完成后，会直接启动该服务：

try {

if (networkTimeUpdaterF != null) networkTimeUpdaterF.systemRunning();

} catch (Throwable e) {

reportWtf("Notifying NetworkTimeService running", e);

}

手段就是调用NetworkTimeUpdateService::systemRunning()方法。通过这两步，时间更新服务就处于工作状态了。

那么我们通过分析这两个步骤，来理解NetworkTimeUpdateService的工作流程

## NetworkTimeUpdateService的构造函数

public NetworkTimeUpdateService(Context context) {

mContext = context;

mTime = NtpTrustedTime.getInstance(context);//初始化NtpTrustedTime实例,它负责NTP时间的获取

mAlarmManager = (AlarmManager) mContext.getSystemService(Context.ALARM\_SERVICE);//获取闹钟服务

//该PendingIntent对象用于当某次Ntp时间获取失败后,开启一个定时服务,去再次请求时间

Intent pollIntent = new Intent(ACTION\_POLL, null);

mPendingPollIntent = PendingIntent.getBroadcast(mContext, POLL\_REQUEST, pollIntent, 0);

//初始化了几个成员变量,它们决定了Ntp时间更新的细节

mPollingIntervalMs = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_ntpPollingInterval);

mPollingIntervalShorterMs = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_ntpPollingIntervalShorter);

mTryAgainTimesMax = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_ntpRetry);

mTimeErrorThresholdMs = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_ntpThreshold);

mWakeLock = ((PowerManager) context.getSystemService(Context.POWER\_SERVICE)).newWakeLock(

PowerManager.PARTIAL\_WAKE\_LOCK, TAG);

首先初始化NtpTrustedTime类实例，它负责NTP时间获取的具体操作，而NetworkTimeUpdateService服务决定何时触发NTP时间请求。

NtpTrustedTime实例的构造过程如下，它采用了单例设计模式：

### NtpTrustedTime

Sdf

private NtpTrustedTime(String server, long timeout) {

if (LOGD) Log.d(TAG, "creating NtpTrustedTime using " + server);

mServer = server;

mTimeout = timeout;

}

public static synchronized NtpTrustedTime getInstance(Context context) {

if (sSingleton == null) {

final Resources res = context.getResources();

final ContentResolver resolver = context.getContentResolver();

//读取用户配置的NTP服务器地址、超时信息等

final String defaultServer = res.getString(

com.android.internal.R.string.config\_ntpServer);

final long defaultTimeout = res.getInteger(

com.android.internal.R.integer.config\_ntpTimeout);

final String secureServer = Settings.Global.getString(

resolver, Settings.Global.NTP\_SERVER);

final long timeout = Settings.Global.getLong(

resolver, Settings.Global.NTP\_TIMEOUT, defaultTimeout);

final String server = secureServer != null ? secureServer : defaultServer;

sSingleton = new NtpTrustedTime(server, timeout);

sContext = context;

}

return sSingleton;

}

读取配置文件和数据库配置的Ntp服务器地址和请求超时时间等信息，然后创建NtpTrustedTime对象并返回。其中，涉及到的服务器地址和超时信息会在config.xml和数据库中配置。如这里的默认NTP请求地址配置为：

<!-- Remote server that can provide NTP responses. -->

<string translatable="false" name="config\_ntpServer">2.android.pool.ntp.org</string>

NtpTrustedTime封装了具体请求NTP时间的接口，NetworkTimeUpdateService如果需要更新NTP时间，通过创建的mTime实例即可实现。

### 参数初始化

创建完NtpTrustedTime实例后，还会获取Alarm服务对象；因为，当某刻NTP请求失败，系统会定时地去尝试再次获取NTP时间。其中创建的mPendingPollIntent就代表我们的这种意图。随后，这里初始化了几个long型变量，它会在NTP时间请求过程中决定一些细节分支处理，所以我们要明确它们的具体含义：

//初始化了三个成员变量,它们决定了Ntp时间更新的细节

mPollingIntervalMs = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_ntpPollingInterval);//86400000ms

mPollingIntervalShorterMs = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_ntpPollingIntervalShorter);//60000ms

mTryAgainTimesMax = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_ntpRetry);//3

**mTimeErrorThresholdMs** = mContext.getResources().getInteger(

com.android.internal.R.integer.config\_**ntpThreshold**);//5000

mPollingIntervalMs：Normal polling frequency in milliseconds -> 每两次刷新请求之间的时间间隔；决定是否需要刷新NTP时间**==24H==1D**

mPollingIntervalShorterMs：Try-again polling interval in milliseconds, in case the network request failed -> 某次请求失败后，再次去请求的时间间隔==60s==1min

mTryAgainTimesMax：Number of times to try again with the shorter interval, before backing off until the normal polling interval. A value < 0 indicates infinite -> 最大的尝试请求次数

**mTimeErrorThresholdMs**：If the time difference is greater than this threshold in milliseconds, then update the time -> 当新旧时间差别大于该值时，更新时间

初始化了这些成员变量，NetworkTimeUpdateService的创建工作就完成了。接着就要去启动该服务了。

*<!-- Remote server that can provide NTP responses. -->*<**string translatable="false" name="config\_ntpServer"**>2.android.pool.ntp.org</**string**>  
*<!-- Normal polling frequency in milliseconds -->*<**integer name="config\_ntpPollingInterval"**>86400000</**integer**>  
*<!-- Try-again polling interval in milliseconds, in case the network request failed -->*<**integer name="config\_ntpPollingIntervalShorter"**>60000</**integer**>  
*<!-- Number of times to try again with the shorter interval, before backing  
 off until the normal polling interval. A value < 0 indicates infinite. -->*<**integer name="config\_ntpRetry"**>3</**integer**>  
*<!-- If the time difference is greater than this threshold in milliseconds,  
 then update the time. -->*<**integer name="config\_ntpThreshold"**>5000</**integer**>  
*<!-- Timeout to wait for NTP server response in milliseconds. -->*<**integer name="config\_ntpTimeout"**>5000</**integer**>

## systemRunning()

sdf

NetworkTimeUpdateService实例创建完成后，会调用systemRunning()启动它：

/\*\* Initialize the receivers and initiate the first NTP request \*/

public void systemRunning() {

registerForTelephonyIntents();

registerForAlarms();

registerForConnectivityIntents();

HandlerThread thread = new HandlerThread(TAG);

thread.start();

mHandler = new MyHandler(thread.getLooper());

// Check the network time on the new thread

mHandler.obtainMessage(EVENT\_POLL\_NETWORK\_TIME).sendToTarget();

mSettingsObserver = new SettingsObserver(mHandler, EVENT\_AUTO\_TIME\_CHANGED);

mSettingsObserver.observe(mContext);

}

说是启动，其实主要的工作就是初始化了一些触发时间更新的条件。

### 注册广播

Dgf

private void registerForTelephonyIntents() {

IntentFilter intentFilter = new IntentFilter();

intentFilter.addAction(TelephonyIntents.ACTION\_NETWORK\_SET\_TIME);

intentFilter.addAction(TelephonyIntents.ACTION\_NETWORK\_SET\_TIMEZONE);

mContext.registerReceiver(mNitzReceiver, intentFilter);

}

private void registerForAlarms() {

mContext.registerReceiver(

new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

mHandler.obtainMessage(EVENT\_POLL\_NETWORK\_TIME).sendToTarget();

}

}, new IntentFilter(ACTION\_POLL));

}

private void registerForConnectivityIntents() {

IntentFilter intentFilter = new IntentFilter();

intentFilter.addAction(ConnectivityManager.CONNECTIVITY\_ACTION);

mContext.registerReceiver(mConnectivityReceiver, intentFilter);

}

/\*\* Receiver for Nitz time events \*/

private BroadcastReceiver mNitzReceiver = new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

String action = intent.getAction();

if (DBG) Log.d(TAG, "Received " + action);

if (TelephonyIntents.ACTION\_NETWORK\_SET\_TIME.equals(action)) {

mNitzTimeSetTime = SystemClock.elapsedRealtime();

} else if (TelephonyIntents.ACTION\_NETWORK\_SET\_TIMEZONE.equals(action)) {

mNitzZoneSetTime = SystemClock.elapsedRealtime();

}

}

};

/\*\* Receiver for ConnectivityManager events \*/

private BroadcastReceiver mConnectivityReceiver = new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

String action = intent.getAction();

if (ConnectivityManager.CONNECTIVITY\_ACTION.equals(action)) {

if (DBG) Log.d(TAG, "Received CONNECTIVITY\_ACTION ");

// Don't bother checking if we have connectivity, NtpTrustedTime does that for us.

Message message = mHandler.obtainMessage(EVENT\_NETWORK\_CHANGED);

// Send with a short delay to make sure the network is ready for use

mHandler.sendMessageDelayed(message, NETWORK\_CHANGE\_EVENT\_DELAY\_MS);

}

}

};

三个register方法分别注册了该服务关心的一些广播，其中一些广播的处理会通过发送消息事件去触发时间更新流程。其中，网络连接更新变化是会触发时间更新的；这在MyHandler的处理中可以看到。

### MyHandler

MyHandler是NetworkTimeUpdateService内处理这些引发时间更新的异步消息事件的类，它工作在单独的线程中：

/\*\* Handler to do the network accesses on \*/

private class MyHandler extends Handler {

public MyHandler(Looper l) {

super(l);

}

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

case EVENT\_AUTO\_TIME\_CHANGED:

case EVENT\_POLL\_NETWORK\_TIME:

case EVENT\_NETWORK\_CHANGED:

onPollNetworkTime(msg.what);

break;

}

}

}

MyHandler中处理了三个事件，这样表明只在三种情况下，会主动去触发时间更新流程：

监听到Settings.Global.AUTO\_TIME字段有变化，表明用户是否使用自动时间

服务内部开启的时间更新定时服务，如某次时间更新失败，需要在下一次再次去触发

网络连接状态发生变化，如网络连接状态从未连接到连接，此时就需要触发时间更新

### SettingsObserver

SettingsObserver会监听Settings.Global.AUTO\_TIME字段是否变化，它对应的就是上面介绍的第一种触发情况：

/\*\* Observer to watch for changes to the AUTO\_TIME setting \*/

private static class SettingsObserver extends ContentObserver {

private int mMsg;

private Handler mHandler;

SettingsObserver(Handler handler, int msg) {

super(handler);

mHandler = handler;

mMsg = msg;

}

void observe(Context context) {

ContentResolver resolver = context.getContentResolver();

resolver.registerContentObserver(Settings.Global.getUriFor(Settings.Global.AUTO\_TIME),

false, this);

}

@Override

public void onChange(boolean selfChange) {

mHandler.obtainMessage(mMsg).sendToTarget();

}

}

当监听到Settings.Global.AUTO\_TIME字段有变化时，会发送消息事件EVENT\_AUTO\_TIME\_CHANGED给MyHandler；如果字段值设为了1，表明用户希望自动设置时间，那么此时就会去触发时间更新。

systemRunning()中还主动触发了一次时间更新操作：

// Check the network time on the new thread

mHandler.obtainMessage(EVENT\_POLL\_NETWORK\_TIME).sendToTarget();

通过发送EVENT\_POLL\_NETWORK\_TIME事件；三种情况下，时间更新的流程都由onPollNetworkTime()处理，分析一种情况就等于分析了三种情况。直接看它的处理过程：

\* Checks if the user prefers to automatically set the time.

\*/

private boolean isAutomaticTimeRequested() {

return Settings.Global.getInt(

mContext.getContentResolver(), Settings.Global.AUTO\_TIME, 0) != 0;

}

## onPollNetworkTimeUnderWakeLock

private void onPollNetworkTime(int event) {

// If Automatic time is not set, don't bother.

if (!isAutomaticTimeRequested()) return;

mWakeLock.acquire();

try {

onPollNetworkTimeUnderWakeLock(event);

} finally {

mWakeLock.release();

}  
 }  
}

首先判断Settings.Global.AUTO\_TIME字段的值，如果为1，则去自动更新、设置时间。

**private void** onPollNetworkTimeUnderWakeLock(**int** event) {  
 **final long** refTime = SystemClock.elapsedRealtime();////获取当前设备从开机算起的时间  
 *// If NITZ time was received less than mPollingIntervalMs time ago,  
 // no need to sync to NTP.*

//NITZ时间部分,如果两次时间的差异小于mPollingIntervalMs的值,则退出时间更新流程

//这是由基带通信决定的，手机才会触发这里**if** (mNitzTimeSetTime != NOT\_SET && refTime - mNitzTimeSetTime < mPollingIntervalMs) {  
 resetAlarm(mPollingIntervalMs);  
 **return**;  
 }  
 **final long** currentTime = System.currentTimeMillis();  
 **if** (DBG) Log.d(TAG, **"System time = "** + currentTime);  
 *// Get the NTP time* **if** (mLastNtpFetchTime == NOT\_SET || refTime >= mLastNtpFetchTime + mPollingIntervalMs  
 || event == EVENT\_AUTO\_TIME\_CHANGED) {  
 **if** (DBG) Log.d(TAG, **"Before Ntp fetch"**);  
  
 *// force refresh NTP cache when outdated* **if** (mTime.getCacheAge() >= mPollingIntervalMs) {  
 mTime.forceRefresh();  
 }  
  
 *// only update when NTP time is fresh* **if** (mTime.getCacheAge() < mPollingIntervalMs) {  
 **final long** ntp = mTime.currentTimeMillis();  
 mTryAgainCounter = 0;  
 *// If the clock is more than N seconds off or this is the first time it's been  
 // fetched since boot, set the current time.* **if** (Math.abs(ntp - currentTime) > mTimeErrorThresholdMs  
 || mLastNtpFetchTime == NOT\_SET) {  
 *// Set the system time* **if** (DBG && mLastNtpFetchTime == NOT\_SET  
 && Math.abs(ntp - currentTime) <= mTimeErrorThresholdMs) {  
 Log.d(TAG, **"For initial setup, rtc = "** + currentTime);  
 }  
 **if** (DBG) Log.d(TAG, **"Ntp time to be set = "** + ntp);  
 *// Make sure we don't overflow, since it's going to be converted to an int* **if** (ntp / 1000 < Integer.MAX\_VALUE) {  
 SystemClock.setCurrentTimeMillis(ntp);  
 }  
 } **else** {  
 **if** (DBG) Log.d(TAG, **"Ntp time is close enough = "** + ntp);  
 }  
 mLastNtpFetchTime = SystemClock.elapsedRealtime();  
 } **else** {  
 *// Try again shortly* mTryAgainCounter++;  
 **if** (mTryAgainTimesMax < 0 || mTryAgainCounter <= mTryAgainTimesMax) {  
 resetAlarm(mPollingIntervalShorterMs);  
 } **else** {  
 *// Try much later* mTryAgainCounter = 0;  
 resetAlarm(mPollingIntervalMs);  
 }  
 **return**;  
 }  
 }  
 resetAlarm(mPollingIntervalMs);  
}

onPollNetworkTimeUnderWakeLock()函数处理如下：

private void onPollNetworkTimeUnderWakeLock(int event) {

final long refTime = SystemClock.elapsedRealtime();

// If NITZ time was received less than mPollingIntervalMs time ago,

// no need to sync to NTP.

if (mNitzTimeSetTime != NOT\_SET && refTime - mNitzTimeSetTime < mPollingIntervalMs) {//

resetAlarm(mPollingIntervalMs);

return;

}

final long currentTime = System.currentTimeMillis();//获取当前时间

if (DBG) Log.d(TAG, "System time = " + currentTime);

// Get the NTP time

if (mLastNtpFetchTime == NOT\_SET || refTime >= mLastNtpFetchTime + mPollingIntervalMs

|| event == EVENT\_AUTO\_TIME\_CHANGED) {//如果我们是第一次设置时间,或者两次更新时间的间隔大于了mPollingIntervalMs值,或者当前触发更新的事件是EVENT\_AUTO\_TIME\_CHANGED;则继续更新流程

if (DBG) Log.d(TAG, "Before Ntp fetch");

// force refresh NTP cache when outdated

if (mTime.getCacheAge() >= mPollingIntervalMs) {//如果这次时间更新的时间节点和上一次时间更新时获取到结果的时间节点之间的差异大于mPollingIntervalMs时,认为时间过期,需要强制刷新时间;

mTime.forceRefresh();

}

// only update when NTP time is fresh

if (mTime.getCacheAge() < mPollingIntervalMs) {//这种情况下认为NTP时间是新鲜的、有效的

final long ntp = mTime.currentTimeMillis();//获取当前的NTP时间

mTryAgainCounter = 0;

// If the clock is more than N seconds off or this is the first time it's been

// fetched since boot, set the current time.

if (Math.abs(ntp - currentTime) > mTimeErrorThresholdMs

|| mLastNtpFetchTime == NOT\_SET) {

// Set the system time

if (DBG && mLastNtpFetchTime == NOT\_SET

&& Math.abs(ntp - currentTime) <= mTimeErrorThresholdMs) {

Log.d(TAG, "For initial setup, rtc = " + currentTime);

}

if (DBG) Log.d(TAG, "Ntp time to be set = " + ntp);

// Make sure we don't overflow, since it's going to be converted to an int

if (ntp / 1000 < Integer.MAX\_VALUE) {

SystemClock.setCurrentTimeMillis(ntp);//将NTP时间设置到内核中,此时时间才会生效

}

} else {

if (DBG) Log.d(TAG, "Ntp time is close enough = " + ntp);

}

mLastNtpFetchTime = SystemClock.elapsedRealtime();

} else {//否则,一段时间之后再去触发时间更新;根据当前的重试次数,时间间隔的设置也会变化

// Try again shortly

mTryAgainCounter++;

if (mTryAgainTimesMax < 0 || mTryAgainCounter <= mTryAgainTimesMax) {

//设置定时任务,隔一段时间后再去尝试刷新时间

resetAlarm(mPollingIntervalShorterMs);

} else {

// Try much later

mTryAgainCounter = 0;

//设置定时任务,隔一段时间后再去尝试刷新时间

resetAlarm(mPollingIntervalMs);

}

return;

}

}

resetAlarm(mPollingIntervalMs);//设置定时任务,隔一段时间后再去尝试刷新时间

}

NTP时间的刷新动作是由NtpTrustedTime::forceRefresh()函数实现：

@Override

public boolean forceRefresh() {

if (TextUtils.isEmpty(mServer)) {

// missing server, so no trusted time available

return false;

}

// We can't do this at initialization time: ConnectivityService might not be running yet.

synchronized (this) {

if (mCM == null) {

mCM = (ConnectivityManager) sContext.getSystemService(Context.CONNECTIVITY\_SERVICE);

}

}

final NetworkInfo ni = mCM == null ? null : mCM.getActiveNetworkInfo();

if (ni == null || !ni.isConnected()) {

if (LOGD) Log.d(TAG, "forceRefresh: no connectivity");

return false;

}

if (LOGD) Log.d(TAG, "forceRefresh() from cache miss");

final SntpClient client = new SntpClient();

if (client.requestTime(mServer, (int) mTimeout)) {

mHasCache = true;

mCachedNtpTime = client.getNtpTime();

mCachedNtpElapsedRealtime = client.getNtpTimeReference();

mCachedNtpCertainty = client.getRoundTripTime() / 2;

return true;

} else {

return false;

}

}

它内部借助SntpClient类来实现此时的NTP socket请求：

final SntpClient client = new SntpClient();

if (client.requestTime(mServer, (int) mTimeout)) {

mHasCache = true;

mCachedNtpTime = client.getNtpTime();

mCachedNtpElapsedRealtime = client.getNtpTimeReference();

mCachedNtpCertainty = client.getRoundTripTime() / 2;

return true;

} else {

return false;

}

SntpClient封装了获取NTP时间的socket接口：

/\*\*

\* Sends an SNTP request to the given host and processes the response.

\*

\* @param host host name of the server.

\* @param timeout network timeout in milliseconds.

\* @return true if the transaction was successful.

\*/

public boolean requestTime(String host, int timeout) {

InetAddress address = null;

try {

address = InetAddress.getByName(host);//

} catch (Exception e) {

if (DBG) Log.d(TAG, "request time failed: " + e);

return false;

}

return requestTime(address, NTP\_PORT, timeout);

}

public boolean requestTime(InetAddress address, int port, int timeout) {

DatagramSocket socket = null;

try {

socket = new DatagramSocket();

socket.setSoTimeout(timeout);

byte[] buffer = new byte[NTP\_PACKET\_SIZE];

DatagramPacket request = new DatagramPacket(buffer, buffer.length, address, port);

// set mode = 3 (client) and version = 3

// mode is in low 3 bits of first byte

// version is in bits 3-5 of first byte

buffer[0] = NTP\_MODE\_CLIENT | (NTP\_VERSION << 3);

// get current time and write it to the request packet

final long requestTime = System.currentTimeMillis();//开始请求的时间节点记录,当前时间

final long requestTicks = SystemClock.elapsedRealtime();//开始请求的时间时钟记录,从开机算起

writeTimeStamp(buffer, TRANSMIT\_TIME\_OFFSET, requestTime);

socket.send(request);//开始发送请求

// read the response

DatagramPacket response = new DatagramPacket(buffer, buffer.length);

socket.receive(response);//开始读取响应

final long responseTicks = SystemClock.elapsedRealtime();//读取响应的时间时钟记录,从开机算起

final long responseTime = requestTime + (responseTicks - requestTicks);//响应时间 = 开始发送请求的时间节点 + (响应时钟节点 - 请求时钟节点)

// extract the results

final byte leap = (byte) ((buffer[0] >> 6) & 0x3);

final byte mode = (byte) (buffer[0] & 0x7);

final int stratum = (int) (buffer[1] & 0xff);

final long originateTime = readTimeStamp(buffer, ORIGINATE\_TIME\_OFFSET);

final long receiveTime = readTimeStamp(buffer, RECEIVE\_TIME\_OFFSET);

final long transmitTime = readTimeStamp(buffer, TRANSMIT\_TIME\_OFFSET);

/\* do sanity check according to RFC \*/

// TODO: validate originateTime == requestTime.

checkValidServerReply(leap, mode, stratum, transmitTime);

long roundTripTime = responseTicks - requestTicks - (transmitTime - receiveTime);

// receiveTime = originateTime + transit + skew

// responseTime = transmitTime + transit - skew

// clockOffset = ((receiveTime - originateTime) + (transmitTime - responseTime))/2

// = ((originateTime + transit + skew - originateTime) +

// (transmitTime - (transmitTime + transit - skew)))/2

// = ((transit + skew) + (transmitTime - transmitTime - transit + skew))/2

// = (transit + skew - transit + skew)/2

// = (2 \* skew)/2 = skew

long clockOffset = ((receiveTime - originateTime) + (transmitTime - responseTime))/2;

if (DBG) {

Log.d(TAG, "round trip: " + roundTripTime + "ms, " +

"clock offset: " + clockOffset + "ms");

}

// save our results - use the times on this side of the network latency

// (response rather than request time)

mNtpTime = responseTime + clockOffset;//计算得到的ntp时间

mNtpTimeReference = responseTicks;//读取响应的时间节点

mRoundTripTime = roundTripTime;

} catch (Exception e) {

if (DBG) Log.d(TAG, "request time failed: " + e);

return false;

} finally {

if (socket != null) {

socket.close();

}

}

return true;

}

一个标准的socket发送请求-读取响应的模式。实现中，记录了一些特殊的时间节点，我们需要注意。

## 系统是如何判定当前NTP时间是否需要刷新呢？

系统是如何判定当前NTP时间是否需要刷新呢？

// force refresh NTP cache when outdated

if (mTime.getCacheAge() >= mPollingIntervalMs) {//如果这次时间更新的时间节点和上一次时间更新时获取到结果的时间节点之间的差异大于mPollingIntervalMs时,认为时间过期,需要强制刷新时间;

mTime.forceRefresh();

}

@Override

public long getCacheAge() {

if (mHasCache) {

return SystemClock.elapsedRealtime() - mCachedNtpElapsedRealtime;//mCachedNtpElapsedRealtime = client.getNtpTimeReference();

} else {

return Long.MAX\_VALUE;

}

}

/\*\*

\* Returns the reference clock value (value of SystemClock.elapsedRealtime())

\* corresponding to the NTP time.

\*

\* @return reference clock corresponding to the NTP time.

\*/

public long getNtpTimeReference() {

return mNtpTimeReference;

}

具体的信息可以参考里面的注释。NTP时间刷新完成后，我们还需要将它设置到系统中，其实也就是设置到kernel中：

// only update when NTP time is fresh

if (mTime.getCacheAge() < mPollingIntervalMs) {//这种情况下认为NTP时间是新鲜的、有效的

final long ntp = mTime.currentTimeMillis();//获取当前的NTP时间

mTryAgainCounter = 0;

// If the clock is more than N seconds off or this is the first time it's been

// fetched since boot, set the current time.

if (Math.abs(ntp - currentTime) > mTimeErrorThresholdMs

|| mLastNtpFetchTime == NOT\_SET) {

// Set the system time

if (DBG && mLastNtpFetchTime == NOT\_SET

&& Math.abs(ntp - currentTime) <= mTimeErrorThresholdMs) {

Log.d(TAG, "For initial setup, rtc = " + currentTime);

}

if (DBG) Log.d(TAG, "Ntp time to be set = " + ntp);

// Make sure we don't overflow, since it's going to be converted to an int

if (ntp / 1000 < Integer.MAX\_VALUE) {

SystemClock.setCurrentTimeMillis(ntp);//将NTP时间设置到内核中,此时时间才会生效

}

} else {

if (DBG) Log.d(TAG, "Ntp time is close enough = " + ntp);

}

mLastNtpFetchTime = SystemClock.elapsedRealtime();

}

判断获取到的NTP时间的有效性，随后SystemClock.setCurrentTimeMillis()函数会将时间设置到内核中，让其生效；这样我们的时间就得到更新了。

---------------------

## poll 轮询 重试轮询resetAlarm

如果某次的时间刷新操作失败了，系统会设置定时任务，在某个时间段之后再去触发时间刷新：

\* Cancel old alarm and starts a new one for the specified interval.

\*

\* @param interval when to trigger the alarm, starting from now.

\*/

private void resetAlarm(long interval) {

mAlarmManager.cancel(mPendingPollIntent);

long now = SystemClock.elapsedRealtime();

long next = now + interval;

mAlarmManager.set(AlarmManager.ELAPSED\_REALTIME, next, mPendingPollIntent);

}

定时器任务触发时，会引起如下处理：

//用于定时器

private void registerForAlarms() {

mContext.registerReceiver(

new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

mHandler.obtainMessage(EVENT\_POLL\_NETWORK\_TIME).sendToTarget();

}

}, new IntentFilter(ACTION\_POLL));

}

向MyHandler发送EVENT\_POLL\_NETWORK\_TIME事件，再次去调用onPollNetworkTime()函数处理当前的时间刷新请求，重复之前的分析流程。

NetworkTimeUpdateService服务启动及NTP时间刷新流程就分析完成了。

## 参考

[系统网络时间更新NetworkTimeUpdateService服务解析](https://blog.csdn.net/csdn_of_coder/article/details/72792866)

# 时区

Settings.Global.AUTO\_TIME\_ZONE

系统基本是没有修改

android-7.1.1\_r1/frameworks/opt/telephony/src/java/com/android/internal/telephony/ServiceStateTracker.java