

Android – Eine Einführung

Tasks & Threads

Andreas Wilhelm

Institut für Informatik
Georg-August-Universität Göttingen

www.avedo.net

Überblick

- ▶ Applikation läuft in einem einzigen Thread (Main- oder UI-Thread)
- ▶ Main-Thread verarbeitet Events und verwaltet Benutzeroberfläche
- ▶ Größere Operationen im Main-Thread blockieren diesen
- ▶ Falls Main-Thread länger als fünf Sekunden blockiert zeigt Android ANR-Dialog (Application Not Responding)
- ▶ Android Grafik-Toolkit eintrittsinvariant
 - Komponente kann nicht aus verschiedenen Threads verwendet werden
 - Änderungen an der grafischen Oberfläche nur im Main-Thread

Zwei Regeln sind im Zusammenhang mit Threads wichtig:

- 1 Länger andauernde Operationen sollten in Threads ausgelagert werden.
- 2 Änderungen an der grafischen Oberfläche sollte nur der Main-Thread vornehmen.

Klassen

- ▶ Android verbindet Java-Concurrent-API (*java.util.concurrent*) mit Eigenimplementationen (*android.os*)
- ▶ Einfache Konstrukte für Nebenläufigkeit *Threads* und *AsyncTasks*
- ▶ Kommunikation mit Main-Thread über *Handler*
- ▶ Zeitlich festgelegte oder periodische Ausführung mit *Timer* und *ScheduledThreadPoolExecutor*

Überblick

- ▶ Verwendung von Java Threads (*java.lang.Thread*)
- ▶ Zwei Möglichkeiten Threads zu nutzen:
 - 1 Ableiten von *Thread* und Überschreiben der *run()* Methode
 - 2 Übergabe eines *Runnable*s beim Erstellen
- ▶ Problem: Aktualisieren der grafischen Oberfläche:
 - 1 *Activity.runOnUiThread(Runnable)* führt *Runnable* im Main-Thread aus
 - 2 Methoden *post(Runnable)* und *postDelayed(Runnable, long)* delegieren Operation direkt an betreffendes *View*
 - 3 Verwendung eines *Handlers*

Implementierung

```
public void onSyncClicked(View v) {  
    new Thread(  
        new Runnable() {  
            public void run() {  
5                final Drawable pic = fetchOneImage(resourcePath, localPath);  
                iview.post(  
                    new Runnable() {  
                        public void run() {  
10                            iview.setDrawable(pic);  
                        }  
                    }  
                );  
            }  
        }  
    ).start();  
15 }
```

Listing : Download im Thread

Probleme mit Threads

- ▶ Applikationen laufen in einem Thread
- ▶ Blockierter Main-Thread führt zu Anzeige von ANR-Dialog
- ▶ Android Grafik-Toolkit eintrittsinvariant
- ▶ Kommunikation zwischen dem betreffenden Thread und dem Main-Thread nötig

Überblick

- ▶ Handler ermöglicht Kommunikation mit anderen Threads
- ▶ Handler wird mit genau einem Thread und dessen Nachrichten-Stapel assoziiert
- ▶ Handler wird an erstellenden Thread gebunden
- ▶ Liefert Nachrichten und Operationen (*Runnables*) an dessen Stapel
- ▶ Beeinflussung der Zeitablaufsteuerung mit *post*()* und *send*()* Methoden
- ▶ *post*()* reihen *Runnables* in den Stapel ein
- ▶ *send*()* Methoden fügen Nachrichten dem Stapel hinzu, die beim verlassen des Stapels in *handleMessage()* verarbeitet werden
- ▶ Handler erlauben zeitlich gebundene oder periodische Ausführung

Implementierung

```

public class Downloader extends Activity {
    private Button startDownload;
    private ProgressBar progressBar;
    private final Handler handler = new Handler();

5
    public void onCreate(Bundle savedInstanceState) {
        // Setup the activity, ...
        super.onCreate(savedInstanceState);
        setContentView(R.layout.downloader_activity);

10
        // ... fetch the progressbar and setup the download button.
        progressBar = (ProgressBar) findViewById(R.id.progressBar);
        startDownload = (Button) findViewById(R.id.startDownload);
        startDownload.setOnClickListener(
15
            new View.OnClickListener() {
                public void onClick(View v) {
                    progressBar.setProgress(0);
                    progressBar.setVisibility(View.VISIBLE);

20
                    Thread downloadThread = new Thread(
                        new Runnable() {
                            int size = getPackageSize();
                            int currentSize = 0;
                            while(true) {
25
                                handler.post(new Runnable() {
                                    int currentSize += getChunkSize();
                                    progressBar.setProgress(size / currentSize);
                                });
                            }
                        });
30
                }
            });
    }
}

```

Listing : ProgressBar aus Thread aktualisieren

Überblick

- ▶ Einsatz von Handlern nicht komfortabel
- ▶ AsyncTask kapselt die Erstellung von Threads und Handlern
- ▶ Verwendung durch Ableitung

onPreExecute() Wird vor der eigentlichen Operation im Main-Thread aufgerufen. Initialisierung der grafischen Komponenten ohne Handler.

doInBackground(Params...) Wird direkt nach Beendigung von *onPreExecute()* im Hintergrund-Thread aufgerufen und führt die tatsächliche Operation aus. Parameter, die bei der Erstellung des AsyncTasks mit übergeben wurden, werden an die Methode gereicht. Änderungen, die im Main-Thread angezeigt werden sollen, können mit einem Aufruf von *publishProgress()* an die Methode *onProgressUpdate()* weitergegeben werden. Das Ergebnis dieses Schritts wird beim Aufruf der Methode *onPostExecute()* weitergegeben.

onProgressUpdate(Progress...) Wird im Main-Thread ausgeführt und kann dazu verwendet werden, den aktuellen Status der Operation in der grafischen Oberfläche dazustellen.

onPostExecute(Result) Wird im Main-Thread ausgeführt und bekommt als Eingabewert das Ergebnis des Hintergrund-Threads übergeben. Sie kann dazu genutzt werden die Ergebnisse der Operation grafisch dazustellen.

Die Klasse

AsyncTask ist eine generische Klasse, die drei Typen bei der Instanzierung erwartet:

- Params** Die Parameter sind die Werte, die beim Aufruf der Methode *execute()* angegeben und weiter an die Methode *doInBackground()* gereicht werden.
- Progress** Das Array vom Typ "*Progress*" ist der Eingabeparameter der Methode *onProgressUpdate()*, die sich auf Basis dieser Daten um die Aktualisierung der grafischen Oberfläche kümmert.
- Result** Dieser Typ ist der Typ des Rückgabewerts, den die Methode *doInBackground()* liefert und an die Methode *onPostExecute()* weitergibt, die sich um die abschließende Aktualisierung der grafischen Oberfläche kümmert.

Implementierung

```
public class OneDownloader extends Activity {  
    ...  
  
    private class Downloader extends AsyncTask<URL, Integer, Long> {  
5        public void onPreExecute() {  
            progressBar.setProgress(100, 0, false);  
            progressBar.setVisibility(View.VISIBLE);  
        }  
  
10        protected Long doInBackground(URL... urls) {  
            int count = urls.length;  
            long totalSize = 0;  
  
            for (int i = 0; i < count; i++) {  
15                totalSize += Downloader.downloadFile(urls[i]);  
                publishProgress((int) ((i / (float) count) * 100));  
  
                if (isCancelled())  
                    break;  
20            }  
  
            return totalSize;  
        }  
  
25        public void onProgressUpdate(String... args) {  
            progressBar.setProgress(100, args[0], false);  
        }  
  
        protected void onPostExecute(Long result) {  
30            progressBar.setText("Download complete").setProgress(0, 0, false);  
        }  
    };  
}
```

Listing : AsyncTask einsetzen

Implementierung II

```
public class OneDownloader extends Activity {  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
5        // Setup the activity, ...  
        ...  
  
        // ... and setup the download button.  
        startDownload.setOnClickListener(  
10            new View.OnClickListener() {  
                @Override  
                public void onClick(View v) {  
                    new Downloader().execute(new Url("www.example.com/a.zip"));  
15                }  
            });  
        }  
        ...  
    }
```

Listing : AsyncTask aufrufen

Anmerkungen

- ▶ Deutlich übersichtlicher als Verwendung von Handlern
- ▶ Gleiche Operation kann iterativ auf mehrere Eingaben angewendet werden (*execute(url1, url2, ..., urlN)*)
- ▶ Abbrechen eines Tasks mit *cancel(boolean)*
- ▶ Abbrechen führt anstatt *onPostExecute()*, *onCancelled()* aus

Zustand des Threads

Da ein AsyncTask jeder Zeit abgebrochen werden kann, sollte regelmäßig überprüft werden, ob dies geschehen ist. Dazu kann die Methode *isCancelled()* verwendet werden.

Synchronisation

- ▶ AsyncTask synchronisiert Callback-Aufrufe
→ Daten stehen im nächsten Calllback bereit
- ▶ Beispiel: Im Konstruktor oder *onPreExecute()* gesetzte Daten stehen in *doInBackground()* zur Verfügung

Es ist dafür wichtig folgendes zu beachten:

- 1 Instanziierung der Klasse AsyncTask muss im Main-Thread vorgenommen werden
- 2 Das Starten des Tasks mit *execute()* muss im Main-Thread geschehen
- 3 Keine der oben beschriebenen Methoden (*onPreExecute()*, *doInBackground()*, ...) sollte direkt aufgerufen werden – darum kümmert sich das System
- 4 Jeder Task kann nur ein einziges mal ausgeführt werden

Nebenläufigkeit

- ▶ AsyncTask führt mehrere Operationen seriell in einem einzelnen Thread aus
- ▶ Mehrere Threads können über *executeOnExecutor()* verwendet werden
- ▶ Bis Android 1.6 Standard-Verfahren
- ▶ Seit Android 3.0 serielle Variante Standard

Überblick

- ▶ Verwendung durch Ableitung
- ▶ Ausführung einmaliger oder wiederkehrender Aufgaben
- ▶ Ausführung der Aufgaben sequentiell in nur einem Thread
→ Ausführung kann zeitlicher Verschiebung unterliegen
- ▶ Aufgaben werden im *One-Shot*-Modus zu absolutem oder relativem Zeitpunkt ausgeführt
- ▶ Regelmäßig wiederkehrende Aufgaben werden in festem Intervall oder mit festgelegtem zeitlichen Abstand ausgeführt
- ▶ Implementierung der Aufgaben als `TimerTask`

Timer beenden

Man sollte bei der Verwendung von Timern darauf achten, dass diese explizit mit *cancel()* beendet werden, wenn sie nicht mehr benötigt werden, um Ressourcen, wie den Thread, freizugeben. Andernfalls würden diese Ressourcen nie freigegeben und wären blockiert.

Implementierung

```
public class TimerActivity extends Activity {  
    // Place the attributs here!  
  
    @Override  
5    protected void onCreate(Bundle savedInstanceState) {  
        // Do the setup here!  
    }  
  
10    public void onStartStopClicked(View v) {  
        if(!this.isRunning) {  
            // Setup the timer and task and start the scheduling.  
            this.counterTimer = new Timer();  
            this.counterTask = new CounterTask();  
            this.counterTimer.schedule(this.counterTask, 1000, 1000);  
15  
            // Change the button label and the running state.  
            this.startStopBtn.setText(R.string.lblStop);  
            this.isRunning = true;  
        } else {  
20            // Cancel the current task.  
            this.counterTask.cancel();  
  
            // Change the button label and the running state.  
            this.startStopBtn.setText(R.string.lblStart);  
            this.isRunning = false;  
25  
        }  
    }  
30 }  
...
```

Listing : Timer einsetzen

Implementierung II

```

public class TimerActivity extends Activity {
    ...

    public void onResetClicked(View v) {
        // Make a toast ...
        Toast.makeText(this, "Canceled timer!", Toast.LENGTH_SHORT).show();

        // ... and cancel the task and timer.
        if (this.counterTask != null)
            this.counterTask.cancel();
        if (this.counterTimer != null)
            this.counterTimer.cancel();

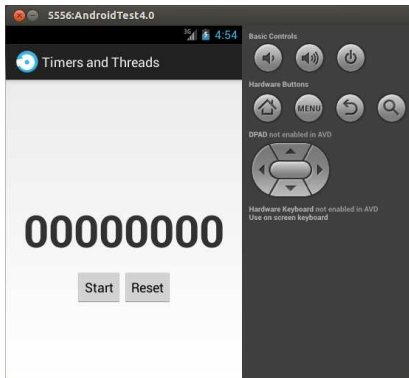
        // Reset the labels, counter and the running flag.
        this.counterTxt.setText(R.string.lblCounterStart);
        this.startStopBtn.setText(R.string.lblStart);
        this.isRunning = false;
        this.counter = 0;
    }

    class CounterTask extends TimerTask {
        @Override
        public void run() {
            counterTxt.post(new Runnable() {
                public void run() {
                    counter++;
                    counterTxt.setText(String.format("%08d", counter));
                }
            });
        }
    }
}

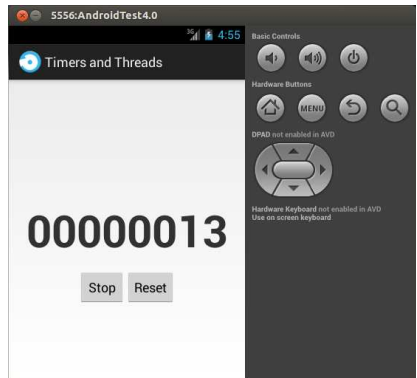
```

Listing : Timer einsetzen

Screenshots



(a) Der Counter vor dem Start



(b) Der laufende Counter

Abbildung: Ein Timer basierter Counter

Anmerkungen

- ▶ Timer ist sehr einfach zu verwenden
- ▶ Probleme bei zeitkritischen Aufgaben
- ▶ Ausführung da sequentiell recht langsam
- ▶ Oftmals Verwendung von *ScheduledThreadPoolExecutor* sinnvoll

Allgemeines

- ▶ Verwendung durch Ableitung
- ▶ Ausführung einmaliger oder wiederkehrender Aufgaben
- ▶ Verteilung der Aufgaben auf einen Thread-Pool
- ▶ Aufgaben werden im *One-Shot*-Modus zu absolutem oder relativem Zeitpunkt ausgeführt
- ▶ Regelmäßig wiederkehrende Aufgaben werden in festem Intervall oder mit festgelegtem zeitlichen Abstand ausgeführt
- ▶ Aufgaben können einzeln verwaltet werden ohne Pool zu beeinflussen

Vergleich zu Timer

- ▶ Verwaltung eines Thread-Pools statt sequentieller Ausführung in einem Thread
→ Performanter
- ▶ Verwendung von *Runnables* und *Callables* anstatt eigener *TimerTask*-Klasse
→ Flexibler in der Entwicklung
- ▶ *Callables* zusätzlich zu *Runnables* (*TimerTask* implementiert *Runnable*)
→ Erlaubt Rückgabewerte und werfen von Exceptions
- ▶ Beide garantieren das Aufgaben nicht vor angestrebten Zeitpunkt ausgeführt werden, sonst allerdings nichts

Zeitablaufsteuerung

Methode	Beschreibung
<i>schedule()</i>	Führt Aufgabe ein einziges mal aus.
<i>scheduleAtFixedRate()</i>	Erlaubt eine wiederholte Ausführung einer Aufgabe, dessen Ausführung keiner zeitlichen Abweichung unterliegen darf.
<i>scheduleWithFixedDelay()</i>	Erlaubt die wiederholte Ausführung einer Aufgabe, wobei das Zeitintervall zwischen den Ausführungen fest ist.
<i>execute()</i> & <i>submit()</i>	Führen die Aufgabe direkt aus.

Rückgabewerte

Sowohl die *schedule*()*, als auch die *submit()* Methoden liefern ein *ScheduledFuture*- bzw. *Future*-Objekt zurück, das dazu verwendet werden kann, die Aufgabe mit *cancel()* abubrechen, mit *get()* das Ergebnis der Aufgabe zu lesen und mit *isDone()* zu überprüfen, ob die Aufgabe bereits erledigt wurde.

Aufgaben beenden

Ausführung durch `ThreadPoolExecutor` kann mit `shutdown()` oder auch `shutdownNow()` beendet werden. `shutdownNow()` Versucht alle Aufgaben zu beenden, `shutdown()` richtig sich nach Policies:

Policy		Beschreibung
<i>ExecuteExistingDelayed-</i> <i>downPolicy</i>	<i>TasksAfterShut-</i>	Falls diese Policy ausgewählt wird (auf <i>true</i> gesetzt), dann werden die einmalig ausgeführten Aufgaben nicht aus der Warteschlange gelöscht.
<i>ContinueExistingPeriodic-</i> <i>downPolicy</i>	<i>TasksAfterShut-</i>	Falls diese Policy ausgewählt wird (auf <i>true</i> gesetzt), dann werden alle periodisch wiederholten Aufgaben nicht aus der Warteschlange gelöscht.

Implementierung

```
public class PoolExecutorActivity extends Activity {  
    // Place the attributs here!  
  
    @Override  
5    protected void onCreate(Bundle savedInstanceState) {  
        // Do the setup here!  
    }  
  
    public void onStartStopClicked(View v) {  
10        if(!this.isRunning) {  
            // Setup the timer and task and start the scheduling.  
            this.counterTimer = new ScheduledThreadPoolExecutor(1);  
            this.counterTimer.scheduleAtFixedRate(this.counterTask, 1000, 1000, TimeUnit.MILLISECONDS);  
  
15            // Change the button label and the running state.  
            this.startStopBtn.setText(R.string.lblStop);  
            this.isRunning = true;  
        } else {  
            // Cancel the current task.  
20            this.counterTimer.shutdownNow();  
  
            // Change the button label and the running state.  
            this.startStopBtn.setText(R.string.lblStart);  
            this.isRunning = false;  
25        }  
    }  
    ...  
}
```

Listing : ScheduledThreadPoolExecutor einsetzen

Implementierung II

```

public class PoolExecutorActivity extends Activity {
    ...

    public void onResetClicked(View v) {
        // Make a toast ...
        Toast.makeText(this, "Canceled timer!", Toast.LENGTH_SHORT).show();

        // ... and cancel the task and timer.
        if (this.counterTimer != null) {
            this.counterTimer.shutdownNow();
        }

        // Reset the labels, counter and the running flag.
        this.counterTxt.setText(R.string.lblCounterStart);
        this.startStopBtn.setText(R.string.lblStart);
        this.isRunning = false;
        this.counter = 0;
    }

    private Runnable counterTask = new Runnable() {
        @Override
        public void run() {
            counterTxt.post(
                new Runnable() {
                    public void run() {
                        counter++;
                        counterTxt.setText(String.format("%08d", counter));
                    }
                }
            );
        }
    };
}

```

Listing : ScheduledThreadPoolExecutor einsetzen