Android APIs API level: 17

android.net.wifi.p2p.ns
android.nfc
android.nfc.tech
android.opengl
android.os
android.os.storage
android.preference
android.provider
android.renderscript
android.sax
android.security
android.service.dreams
android.service.textser
android.service.wallpa

AsyncTask
BatteryManager
Binder

Use Tree Navigation

public abstract class

AsyncTask

extends Object

Summary: Nested Classes | Fields | Ctors | Methods | Protected Methods | Inherited Methods | [Expand All]

Added in API level 3

java.lang.Object
    android.os.AsyncTask<Params, Progress, Result>

Class Overview

AsyncTask enables proper and easy use of the UI thread. This class allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

AsyncTask is designed to be a helper class around `Thread` and `Handler` and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the `java.util.concurrent` pacakge such as `Executor`, `ThreadPoolExecutor` and `FutureTask`.

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called `Params`, `Progress` and `Result`, and 4 steps, called `onPreExecute`, `doInBackground`,

`onProgressUpdate` and `onPostExecute`.

Developer Guides

For more information about using tasks and threads, read the Processes and Threads developer guide.

Usage

AsyncTask must be subclassed to be used. The subclass will override at least one method (`doInBackground(Params...)`), and most often will override a second one (`onPostExecute(Result)`.)

Here is an example of subclassing:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }
```

```
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

Once created, a task is executed very simply:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

AsyncTask's generic types

The three types used by an asynchronous task are the following:

1. **`Params`**, the type of the parameters sent to the task upon execution.
2. **`Progress`**, the type of the progress units published during the background computation.
3. **`Result`**, the type of the result of the background computation.

Not all types are always used by an asynchronous task. To mark a type as unused, simply use the type **`Void`**:

```
private class MyTask extends AsyncTask<Void, Void, Void> { ... }
```

The 4 steps

When an asynchronous task is executed, the task goes through 4 steps:

1. `onPreExecute()`, invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
2. `doInBackground(Params...)`, invoked on the background thread immediately after `onPreExecute()` finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use `publishProgress(Progress...)` to publish one or more units of progress. These values are published on the UI thread, in the `onProgressUpdate(Progress...)` step.
3. `onProgressUpdate(Progress...)`, invoked on the UI thread after a call to `publishProgress(Progress...)`. The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.
4. `onPostExecute(Result)`, invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Cancelling a task

A task can be cancelled at any time by invoking `cancel(boolean)`. Invoking this method will cause subsequent calls to `isCancelled()` to return true. After invoking this method,

`onCancelled(Object)`, instead of `onPostExecute(Object)` will be invoked after `doInBackground(Object[])` returns. To ensure that a task is cancelled as quickly as possible, you should always check the return value of `isCancelled()` periodically from `doInBackground(Object[])`, if possible (inside a loop for instance.)

Threading rules

There are a few threading rules that must be followed for this class to work properly:

- The AsyncTask class must be loaded on the UI thread. This is done automatically as of `JELLY_BEAN`.
- The task instance must be created on the UI thread.
- `execute(Params...)` must be invoked on the UI thread.
- Do not call `onPreExecute()`, `onPostExecute(Result)`, `doInBackground(Params...)`, `onProgressUpdate(Progress...)` manually.
- The task can be executed only once (an exception will be thrown if a second execution is attempted.)

Memory observability

AsyncTask guarantees that all callback calls are synchronized in such a way that the following operations are safe without explicit synchronizations.

- Set member fields in the constructor or `onPreExecute()`, and refer to them in `doInBackground(Params...)`.
- Set member fields in `doInBackground(Params...)`, and refer to them in

`onProgressUpdate(Progress...)` and `onPostExecute(Result)`.

Order of execution

When first introduced, AsyncTasks were executed serially on a single background thread. Starting with `DONUT`, this was changed to a pool of threads allowing multiple tasks to operate in parallel. Starting with `HONEYCOMB`, tasks are executed on a single thread to avoid common application errors caused by parallel execution.

If you truly want parallel execution, you can invoke `executeOnExecutor(java.util.concurrent.Executor, Object[])` with `THREAD_POOL_EXECUTOR`.

Summary

Nested Classes		
enum	AsyncTask.Status	Indicates the current status of the task.

Fields		
public static final Executor	SERIAL_EXECUTOR	An `Executor` that executes tasks one at a time in serial order.
public static final Executor	THREAD_POOL_EXECUTOR	An `Executor` that can be used to execute tasks in parallel.

Public Constructors

	AsyncTask() Creates a new asynchronous task.

Public Methods	
final boolean	cancel (boolean mayInterruptIfRunning) Attempts to cancel execution of this task.
static void	execute (Runnable runnable) Convenience version of `execute(Object)` for use with a simple Runnable object.
final AsyncTask<Params, Progress, Result>	execute (Params... params) Executes the task with the specified parameters.
final AsyncTask<Params, Progress, Result>	executeOnExecutor (Executor exec, Params... params) Executes the task with the specified parameters.
final Result	get (long timeout, TimeUnit unit) Waits if necessary for at most the given time for the computation to complete, and then retrieves its result.
final Result	get () Waits if necessary for the computation to complete, and then retrieves its result.
final AsyncTask.Status	getStatus () Returns the current status of this task.
final boolean	isCancelled () Returns `true` if this task was cancelled before it completed normally.

Protected Methods	
abstract Result	doInBackground (Params... params) Override this method to perform a computation on a background thread.
void	onCancelled (Result result) Runs on the UI thread after `cancel(boolean)` is invoked and `doInBackground(Object[])` has finished.
void	onCancelled () Applications should preferably override `onCancelled(Object)`.
void	onPostExecute (Result result) Runs on the UI thread after `doInBackground(Params...)`.
void	onPreExecute () Runs on the UI thread before `doInBackground(Params...)`.
void	onProgressUpdate (Progress... values) Runs on the UI thread after `publishProgress(Progress...)` is invoked.
final void	publishProgress (Progress... values) This method can be invoked from `doInBackground(Params...)` to publish updates on the UI thread while the background computation is still running.

Inherited Methods	[Expand]
▸ From class java.lang.Object	

Fields

public static final Executor **SERIAL_EXECUTOR**

Added in API level 11

An `Executor` that executes tasks one at a time in serial order. This serialization is global to a particular process.

public static final Executor **THREAD_POOL_EXECUTOR**

Added in API level 11

An `Executor` that can be used to execute tasks in parallel.

Public Constructors

public **AsyncTask** ()

Added in API level 3

Creates a new asynchronous task. This constructor must be invoked on the UI thread.

Public Methods

public final boolean **cancel** (boolean mayInterruptIfRunning)

Added in API level 3

Attempts to cancel execution of this task. This attempt will fail if the task has already completed, already been cancelled, or could not be cancelled for some other reason. If successful, and this task has not started when `cancel` is called, this task should never run. If the task has already started, then the `mayInterruptIfRunning` parameter determines whether the thread executing this task should be interrupted in an attempt to stop the task.

Calling this method will result in `onCancelled(Object)` being invoked on the UI thread after `doInBackground(Object[])` returns. Calling this method guarantees that `onPostExecute(Object)` is never invoked. After invoking this method, you should check the value returned by `isCancelled()` periodically from `doInBackground(Object[])` to finish the task as early as possible.

Parameters

mayInterruptIfRunning	`true` if the thread executing this task should be interrupted; otherwise, in-progress tasks are allowed to complete.

Returns

`false` if the task could not be cancelled, typically because it has already completed normally; `true` otherwise

See Also

`isCancelled()`

`onCancelled(Object)`

public static void **execute** (Runnable runnable)

Added in API level 11

Convenience version of `execute(Object)` for use with a simple Runnable object. See `execute(Object[])` for more information on the order of execution.

See Also

`execute(Object[])`

`executeOnExecutor(java.util.concurrent.Executor, Object[])`

public final AsyncTask<Params, Progress, Result> **execute** (Params...

params) Added in API level 3

Executes the task with the specified parameters. The task returns itself (this) so that the caller can keep a reference to it.

Note: this function schedules the task on a queue for a single background thread or pool of threads depending on the platform version. When first introduced, AsyncTasks were executed serially on a single background thread. Starting with `DONUT`, this was changed to a pool of threads allowing multiple tasks to operate in parallel. Starting `HONEYCOMB`, tasks are back to being executed on a single thread to avoid common application errors caused by parallel execution. If you truly want parallel execution, you can use the `executeOnExecutor(Executor, Params...)` version of this method with `THREAD_POOL_EXECUTOR`; however, see commentary there for warnings on its use.

This method must be invoked on the UI thread.

Parameters

params The parameters of the task.

Returns

This instance of AsyncTask.

Throws

IllegalStateException If `getStatus()` returns either `RUNNING` or `FINISHED`.

See Also

`executeOnExecutor(java.util.concurrent.Executor, Object[])`

`execute(Runnable)`

public final AsyncTask<Params, Progress, Result> **executeOnExecutor** (Executor exec, Params... params) Added in API level 11

Executes the task with the specified parameters. The task returns itself (this) so that the caller can keep a reference to it.

This method is typically used with `THREAD_POOL_EXECUTOR` to allow multiple tasks to run in parallel on a pool of threads managed by AsyncTask, however you can also use your own `Executor` for custom behavior.

Warning: Allowing multiple tasks to run in parallel from a thread pool is generally *not* what one wants, because the order of their operation is not defined. For example, if these tasks are used to modify any state in common (such as writing a file due to a button click), there are no guarantees on the order of the modifications. Without careful work it is possible in rare cases for the newer version of the data to be over-written by an older one, leading to obscure data loss and stability issues. Such changes are best executed in serial; to guarantee such work is serialized regardless of platform version you can use this function with `SERIAL_EXECUTOR`.

This method must be invoked on the UI thread.

Parameters

exec	The executor to use. `THREAD_POOL_EXECUTOR` is available as a convenient process-wide thread pool for tasks that are loosely coupled.
params	The parameters of the task.

Returns

This instance of AsyncTask.

Throws

IllegalStateException	If `getStatus()` returns either `RUNNING` or `FINISHED`.

See Also

`execute(Object[])`

public final Result **get** (long timeout, TimeUnit unit)

Added in API level 3

Waits if necessary for at most the given time for the computation to complete, and then retrieves its result.

Parameters

timeout	Time to wait before cancelling the operation.
unit	The time unit for the timeout.

Returns

The computed result.

Throws

CancellationException	If the computation was cancelled.
ExecutionException	If the computation threw an exception.
InterruptedException	If the current thread was interrupted while waiting.
TimeoutException	If the wait timed out.

public final Result **get** ()

Added in API level 3

Waits if necessary for the computation to complete, and then retrieves its result.

Returns

The computed result.

Throws

CancellationException	If the computation was cancelled.
ExecutionException	If the computation threw an exception.
InterruptedException	If the current thread was interrupted while waiting.

public final AsyncTask.Status **getStatus** ()

Added in API level 3

Returns the current status of this task.

Returns

The current status.

public final boolean **isCancelled** ()

Added in API level 3

Returns `true` if this task was cancelled before it completed normally. If you are calling `cancel(boolean)` on the task, the value returned by this method should be checked periodically from `doInBackground(Object[])` to end the task as soon as possible.

Returns

`true` if task was cancelled before it completed

See Also

`cancel(boolean)`

Protected Methods

protected abstract Result **doInBackground** (Params... params)

Added in API level 3

Override this method to perform a computation on a background thread. The specified parameters are the parameters passed to `execute(Params...)` by the caller of this task. This method can call `publishProgress(Progress...)` to publish updates on the UI thread.

Parameters

params The parameters of the task.

Returns

A result, defined by the subclass of this task.

See Also

`onPreExecute()`

`onPostExecute(Result)`

`publishProgress(Progress...)`

protected void **onCancelled** (Result result)

Added in API level 11

Runs on the UI thread after `cancel(boolean)` is invoked and `doInBackground(Object[])` has finished.

The default implementation simply invokes `onCancelled()` and ignores the result. If you write your own implementation, do not call `super.onCancelled(result)`.

Parameters

result The result, if any, computed in `doInBackground(Object[])`, can be null

See Also

`cancel(boolean)`

`isCancelled()`

protected void **onCancelled** ()

Added in API level 3

Applications should preferably override `onCancelled(Object)`. This method is invoked by the default implementation of `onCancelled(Object)`.

Runs on the UI thread after `cancel(boolean)` is invoked and `doInBackground(Object[])` has finished.

See Also

`onCancelled(Object)`

`cancel(boolean)`

`isCancelled()`

protected void **onPostExecute** (Result result)

Added in API level 3

Runs on the UI thread after `doInBackground(Params...)`. The specified result is the value returned by `doInBackground(Params...)`.

This method won't be invoked if the task was cancelled.

Parameters

result The result of the operation computed by `doInBackground(Params...)`.

See Also

`onPreExecute()`

`doInBackground(Params...)`

`onCancelled(Object)`

protected void **onPreExecute** ()

Added in API level 3

Runs on the UI thread before `doInBackground(Params...)`.

See Also

`onPostExecute(Result)`

`doInBackground(Params...)`

protected void **onProgressUpdate** (Progress... values)

Added in API level 3

Runs on the UI thread after `publishProgress(Progress...)` is invoked. The specified values are the values passed to `publishProgress(Progress...)`.

Parameters

values The values indicating progress.

See Also

`publishProgress(Progress...)`

`doInBackground(Params...)`

protected final void **publishProgress** (Progress... values)

Added in API level 3

This method can be invoked from `doInBackground(Params...)` to publish updates on the UI thread while the background computation is still running. Each call to this method will trigger the execution of `onProgressUpdate(Progress...)` on the UI thread.

`onProgressUpdate(Progress...)` will note be called if the task has been canceled.

Parameters

values The progress values to update the UI with.

See Also

`onProgressUpdate(Progress...)`

`doInBackground(Params...)`

Android 4.2 r1 — 15 Jan 2013 0:04