Service

The Guava `Service` interface represents an object with an operational state, with methods to start and stop. For example, webservers, RPC servers, and timers can implement the `Service` interface. Managing the state of services like these, which require proper startup and shutdown management, can be nontrivial, especially if multiple threads or scheduling is involved. Guava provides some skeletons to manage the state logic and synchronization details for you.

Using a Service

The normal lifecycle of a `Service` is

- `Service.State.NEW` to
- `Service.State.STARTING` to
- `Service.State.RUNNING` to
- `Service.State.STOPPING` to
- `Service.State.TERMINATED`

A stopped service may not be restarted. If the service fails where starting, running, or stopping, it goes into state `Service.State.FAILED`.

A service can be started *asynchronously* using `startAsync()` if the service is `NEW`. So you should structure your application to have a unique place where each service is started.

Stopping the service is analogous, using the *asynchronous* `stopAsync()` method. But unlike `startAsync()`, it is safe to call this method multiple times. This is to make it possible to handle races that may occur when shutting down services.

Service also provides several methods to wait for service transitions to complete.

- *asynchronously* using `addListener()`. `addListener()` allows you to add a `Service.Listener` that will be invoked on every state transition of the service. N.B. if a service is not `NEW` when the listener is added, then any state transitions that have already occurred will *not* be replayed on the listener.
- *synchronously* using `awaitRunning()`. This is uninterruptible, throws no checked exceptions, and returns once the service has finished starting. If the service fails to start, this throws an `IllegalStateException`. Similarly, `awaitTerminated()` waits for the service to reach a terminal state (`TERMINATED` or `FAILED`). Both methods also have overloads that allow timeouts to be specified.

The `Service` interface is subtle and complicated. *We do not recommend implementing it directly*. Instead please use one of the abstract base classes in guava as the base for your implementation. Each base class supports a specific threading model.

Implementations

AbstractIdleService

The `AbstractIdleService` skeleton implements a `Service` which does not need to perform any action while in the "running" state -- and therefore does not need a thread while running -- but has startup and shutdown actions to perform. Implementing such a service is as easy as extending `AbstractIdleService` and implementing the `startUp()` and `shutDown()` methods.

```
protected void startUp() {
  servlets.add(new GcStatsServlet());
}
protected void shutDown() {}
```

Note that any queries to the `GcStatsServlet` already have a thread to run in. We don't need this service to perform any operations on its own while the service is running.

AbstractExecutionThreadService

An `AbstractExecutionThreadService` performs startup, running, and shutdown actions in a single thread. You must override the `run()` method, and it must respond to stop requests. For example, you might perform actions in a work loop:

```
public void run() {
  while (isRunning()) {
    // perform a unit of work
  }
}
```

Alternately, you may override in any way that causes `run()` to return.

Overriding `startUp()` and `shutDown()` is optional, but the service state will be managed for you.

```
protected void startUp() {
  dispatcher.listenForConnections(port, queue);
}
protected void run() {
  Connection connection;
  while ((connection = queue.take() != POISON)) {
    process(connection);
  }
}
protected void triggerShutdown() {
  dispatcher.stopListeningForConnections(queue);
  queue.put(POISON);
}
```

Note that `start()` calls your `startUp()` method, creates a thread for you, and invokes `run()` in that thread. `stop()` calls `triggerShutdown()` and waits for the thread to die.

AbstractScheduledService

An `AbstractScheduledService` performs some periodic task while running. Subclasses implement `runOneIteration()` to specify one iteration of the task, as well as the familiar `startUp` and `shutDown()` methods.

To describe the execution schedule, you must implement the `scheduler()` method. Typically, you will use one of the provided schedules from `AbstractScheduledService.Scheduler`, either `newFixedRateSchedule(initialDelay, delay, TimeUnit)` or

`newFixedDelaySchedule(initialDelay, delay, TimeUnit)`, corresponding to the familiar methods in `ScheduledExecutorService`. Custom schedules can be implemented using `CustomScheduler`; see the Javadoc for details.

AbstractService

When you need to do your own manual thread management, override `AbstractService` directly. Typically, you should be well served by one of the above implementations, but implementing `AbstractService` is recommended when, for example, you are modeling something that provides its own threading semantics as a `Service`, you have your own specific threading requirements.

To implement `AbstractService` you must implement 2 methods.

- `doStart()`: `doStart()` is called directly by the first call to `startAsync()`, your `doStart()` method should perform all initialization and then eventually call `notifyStarted()` if start up succeeded or `notifyFailed()` if start up failed.
- `doStop()`: `doStop()` is called directly by the first call to `stopAsync()`, your `doStop()` method should shut down your service and then eventually call `notifyStopped()` if shutdown succeeded or `notifyFailed()` if shutdown failed.

Your `doStart` and `doStop`, methods should be *fast*. If you need to do expensive initialization, such as reading files, opening network connections, or any operation that might block, you should consider moving that work to another thread.

Using ServiceManager

In addition to the `Service` skeleton implementations, Guava provides a `ServiceManager` class that makes certain operations involving multiple `Service` implementations easier. Create a new `ServiceManager` with a collection of `Services`. Then you can manage them:

- `startAsync()` will start all the services under management. Much like `Service#startAsync()` you can only call this method once, if all services are `NEW`.
- `stopAsync()` will stop all the services under management.
- `addListener` will add a `ServiceManager.Listener` that will be called on major state transitions.
- `awaitHealthy()` will wait for all services to reach the `RUNNING` state.
- `awaitStopped()` will wait for all services to reach a terminal state.

Or inspect them:

- `isHealthy()` returns `true` if all services are `RUNNING`.
- `servicesByState()` returns a *consistent* snapshot of all the services indexed by their state.
- `startupTimes()` returns a map from `Service` under management to how long it took for that service to start in milliseconds. The returned map is guaranteed to be ordered by startup time.

While it is recommended that service lifecycles be managed via `ServiceManager`, state transitions initiated via other mechanisms **do not impact the correctness** of its methods. For example, if the services are started by some mechanism besides `startAsync()`, the listeners will be invoked when appropriate and `awaitHealthy()` will still work as expected. The only requirement that `ServiceManager` enforces is that all `Services` must be `NEW` when `ServiceManager` is constructed.