Java Daemon API Specification Request

Thomas Kopp, 29 November 2000

Table of Content

1. Introduction

Enterprise computing is based on server processes. The Java language offers a wide range of APIs for implementing server processes such as Java servlets or enterprise Java beans. These frameworks always rely on containers for hosting enterprise components. Containers are mostly realized as modules of independently running system level services as database systems or web servers, which have been written in a native platform-specific language or in the Java language.

Currently, there is a gap for deploying and running system level services written in the Java language in a uniform fashion. On Unix platforms, the problem can be solved by manually writing a suitable rc init script for calling the Java application launcher in order to start or stop a Java service. On Windows platforms, the problem is harder to solve, e.g. by writing a native wrapper application, which supplies a functionality similar to an rc init script.

This proposal introduces Java daemons, which should fill the gap. Java daemons are independently running system level services written in the Java language according to a daemon API and hosted from daemon containers. A daemon container may be written in a native language or in the Java language. In order to do so, the Java daemon framework also provides for an SPI to implement daemon containers. In addition, a simple command line utility similar to the Java application launcher is supplied on each platform for installing and running Java daemons in a uniform fashion.

The behavior and installation of a Java daemon is independent of the current host platform. Thus, a developer should no longer work around platform-specific details for deploying and running system level services written in Java.

2. Daemon Life Cycle

In order to bring a daemon in a state for fulfilling its service at runtime, it has to be *initialized*. Initialization comprises tasks like reading configuration parameters or allocating remote resources. After initialization a daemon may run until it is told to terminate servicing or until it runs in an unrecoverable error, which both result in the *destruction* of the current daemon instance for releasing allocated resources again.

During servicing a daemon is in a so-called *active* state. Otherwise it is considered to be inactive, meaning that no service can be supplied by a daemon when it is inactive.

Apart from this scenario, there may be daemons, which once being active, can be stopped and re-started again in order to re-load parameters or to provide for other administration purposes without the need to run through the whole initialization cycle.

Thus, some daemons may be temporarily *paused*. A daemon is not paused when it is active or when it has been shutdown. As an example, a web server may be paused for saving log files, which requires flushing the log buffers. Saving log files, however, would result in unnecessarily resetting the web server’s volatile resource cache when completely shutting the server down. Temporarily stopping the service would be a suitable way for solving problems like this.

3. Daemon Interfaces

Several interfaces can be used for realizing daemons.

The API consists of the *Daemon* and *Pausable* interfaces, which supply hooks being called during life cycle changes described in the previous section. The former interface has to be implemented by all daemons, the latter can optionally be implemented by daemons that can temporarily stop and re-start.

The SPI consists of the *DaemonControl* and *PausableControl* interfaces for managing a daemon's life cycle accordingly. In addition to the life cycle controlling interfaces, there is another SPI for controlling resources used by a daemon. This *DaemonContext* interface defines methods for setting configuration attributes and other resources.

Available resources can be accessed by a daemon using the complementary *DaemonConfig* API. In addition, there are *Logging* and *ExtendedLogging* interfaces for writing output to a log resource if applicable.

Life cycles may be observed by other components, e.g. for handling resource problems or for notifying administration requirements. Thus, there is also a *DaemonListener* interface for life cycle change notifications.

All interfaces are linked together by the *GenericDaemon*, which is the base class for all Java daemons. This base class is extended by the *PausableDaemon* class, which supplies extended functionality in order to be sub-classed by daemons that might temporarily stop.

A *DaemonException* is a new subclass of Exception for indicating a problem during daemon initialization or startup.

4. Daemon States

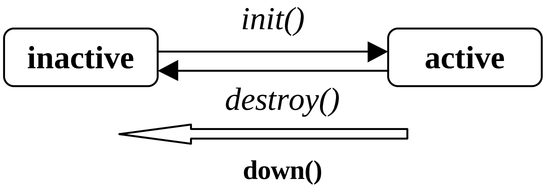
The state of a daemon is changed using a control method called via one of the above-cited control interfaces. Life cycle management comprises the *init* and *destroy* methods for daemon initialization and destruction and the *stop* and *start* methods for temporarily stopping and re-starting. The latter two methods are also called when destroying or initializing pausable daemons respectively.

State updates are performed via the daemon container. In addition, a daemon can also manage state itself but it can only switch from active to pause or inactive states. A complementary control action is required for going the other way again. For managing its own state a daemon uses the *down* and *pause* methods.

All state changes are notified via the listener interface if applicable.

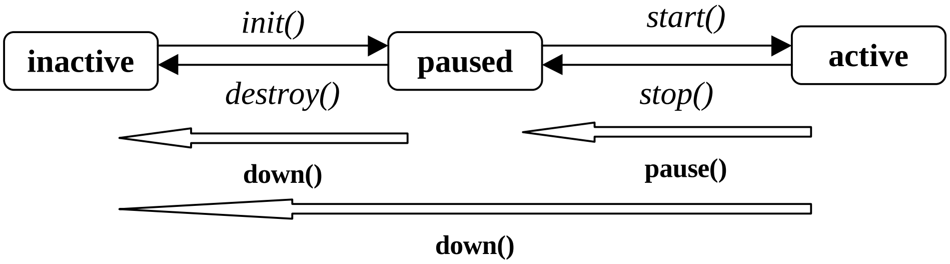
The following chart describes the various states and transitions with italic labeling indicating transitions caused by control actions and bold face labeling indicating transitions invoked by the daemon itself.

- ordinary daemons



Calling down does not interrupt a running init method call but it prevents the daemon from reaching the active state. Since down may be called from a separate thread started during init this situation may happen in practice.

- pausable daemons



Calling pause does not interrupt a running start method call but it prevents the daemon from reaching the active state. Since pause may be called from a separate thread started during start this situation may happen in practice.

Notes:

- The pause and down methods result in state updates without passing the corresponding stop or destroy methods. Life cycle hooks are only passed when state is managed via the control interface.
- Reaching a pause state is only notified via a listener when the daemon was active before.

As an implementation detail, the pause and down methods are postponed until the end of a running control action, i.e. state computation after a control action performed is also based on pause or down calls occurred since the begin of the corresponding control action.

5. Synchronization Issues

Most daemon interfaces are singletons because state is kept in the corresponding daemon instance itself or in a one-to-one relationship to that instance. Access to these singletons is hence synchronized via the basic daemon instance.

Calling life cycle methods via the DaemonControl interface is synchronized via the current control instance.

Life cycle updates are synchronized via the current DaemonEventHandler instance, which controls bound listeners. Updates are not synchronized together with life cycle control actions because a daemon managing its own life cycle could be in deadlock conflict with control methods called from a different thread.

6. Security Issues

There are currently no specific security issues for Java daemons beyond standard Java security.

7. Daemon/Container Contract

A daemon container supplies attributes for a daemon and manages life cycles via the above-cited interfaces. In addition it can supply a logging resource for a daemon or make use of the daemon listener interface for watching daemon life states.

Attributes should only be modified when the daemon is inactive or in pause state. This frees a daemon from the obligation to synchronize access to its DaemonConfig attribute name collection (cf. API documentation for details).

A daemon should use the pause or down methods only for service stops or error halts if no useful servicing is possible without an administration action performed. Thus, life state management should be left up to the controlling instance, meaning the container/daemon relationship is a master/servant relationship as long as the daemon can do what it has been told to do.

The Pausable interface should only be implemented by daemons that can react in a suitable way on start or stop methods calls. Thus, all daemon life cycle methods of the basic daemon classes are defined as abstract methods (cf. API documentation for details).

If a daemon decides to call pause or down, a possibly running init, start, stop or destroy method will not terminate prior to reach its end. This rule does not apply if a daemon throws a DaemonException during initialization or startup.

A normal daemon behavior would consist in creating a separate thread during initialization or startup and continue running in that thread until it is paused or shutdown via a control method (cf. examples for details).

8. Implementation Issues

The Java daemon API could be made available via a simple command line launcher and installer (cf. reference implementation for details). This installer would provide for deploying Java daemons including configuration attribute setting in a uniform fashion on each JDK platform. It would also provide for hosting daemons at runtime. Thus, this utility would be similar to the current Java launcher.

Daemon containers, which offer a richer functionality, could be realized using the Java language on top of the basic JDK level daemon containers, i.e. high level containers would be implemented as Java daemons themselves on behalf of the daemon SPI.

High-level containers could be supplied by third-party vendors and would not be part of the basic daemon framework. They might supply features such as

- graphical user interfaces for managing daemon state of whole daemon groups
- graphical user interfaces for managing daemon listeners and daemon dependencies
- graphical user interfaces for watching daemons and scheduling timer driven daemons
- switchboards for basic Java services like RMI registry, remote compilers or other
- remote administration interfaces for distributed daemons linked via network protocols
- load balancing or messaging utilities for distributed daemons

9. Reference Implementations

Reference implementations for the basic JDK level utilities would use platform-specific features if applicable. They would have to make a maximum effort for supplying a uniform Java-like interface and using different platform-specific features for realizing this interface. They would even have to emulate platform-specific features if missing for realizing a basic daemon container.

As an example an NT reference installation would make use of the NT service layer. A Unix reference installation would supply a utility for automatically generating rc init scripts and handling signals supplied by these scripts for managing daemons.