Package org.apache.hadoop.service.launcher

This package contains classes, interfaces and exceptions to launch YARN services from the command line.

See: Description

Interface Summary

Interface	Description
IrqHandler.Interrupted	Callback issues on an interrupt
LaunchableService	An interface which services can implement to have their execution managed by the ServiceLauncher.
LauncherExitCodes	Common Exit codes.

Class Summary

Class	Description
AbstractLaunchableService	Subclass of `AbstractService` that provides basic implementations of the new methods
InterruptEscalator.ServiceForcedShutdown	forced shutdown runnable.
IrqHandler.InterruptData	Interrupt data to pass on.
ServiceLauncher<S extends Service>	A class to launch any service by name.

Exception Summary

Exception	Description
ServiceLaunchException	A service launch exception that includes an exit code.

Package org.apache.hadoop.service.launcher Description

This package contains classes, interfaces and exceptions to launch YARN services from the command line.

Key Features

General purpose YARN service launcher:

The `ServiceLauncher` class parses a command line, then instantiates and launches the specified YARN service. It then waits for the service to finish, converting any exceptions raised or exit codes returned into an exit code for the (then exited) process.

This class is designed be invokable from the static `ServiceLauncher.main(String[])` method, or from `main(String[])` methods implemented by other classes that which to provide their own endpoints.

Extended YARN Service Interface:

The `LaunchableService` interface extendss `Service` with methods to pass down the CLI arguments, to execute an operapation without having to spawn a thread in the `Service.start()` phase.

Standard Exit codes:

`LauncherExitCodes` defines a set of exit codes that can be used by services to standardize exit causes.

Escalated shutdown:

The `ServiceShutdownHook` shuts down any service via the hadoop shutdown mechanism. The `InterruptEscalator` can be registered to catch interrupts, triggering the shutdown -and forcing a JVM exit if it times our or a second interrupt is received.

Tests:

test cases include interrupt handling and lifecycle failures.

Launching a YARN Service

The Service Launcher can launch *any YARN service*. It will instantiate the service classname provided, by zero-args constructor. Or, if none is available, falling back to a constructor that takes a `String` as its parameter, on the assumption that the parameter is the service name.

The launcher will initialize the service via `Service.init(Configuration)`, then start it via its `Service.start()` method. It then waits indefinitely for the service to stop.

After the service has stopped, a non-null value of `Service.getFailureCause()` is interpreted as a failure, and, if it didn't happen during the stop phase (i.e. when `Service.getFailureState()` is not `STATE.STOPPED`, escalated into a non-zero return code.

To view the worklow in sequence, it is:

1. (prepare configuration files -covered later)
2. instantiate service via its empty or string constructor
3. call `Service.init(Configuration)`
4. call `Service.start()`
5. call `Service.waitForServiceToStop(long)`
6. If an exception was raised: propagate it
7. If an exception was recorded in `Service.getFailureCause()` while the service was running: propagate it.

For a service to be fully compatible with this launch model, it must

- Start worker threads, processes and executors in its `Service.start()` method
- Terminate itself via a call to `Service.stop()` in one of these asynchronous methods.

If a service does not stop itself, *ever*, then it can be launched as a long-lived daemon. The service launcher will never terminate, but neither will the service. The service launcher does register signal handlers to catch `kill` and control-C signals -calling `stop()` on the service when signalled. This means that a daemon service *may* get some warning and time to shut down.

To summarize: provided a service launches its long-lived threads in its Service `start()` method, the service launcher can create it, configure it and start it -triggering shutdown when signalled. What these services can not do is get at the command line parameters or easily propagate exit codes (there is way covered later). These features require some extensions to the base `Service` interface: *the Launchable Service*.

Launching a Launchable YARN Service

A Launchable YARN Service is a YARN service which implements the interface `LaunchableService`.

It adds two methods to the service interface -and hence two new features:

1. Access to the command line passed to the service launcher

2. A blocking `int execute()` method which can return the exit code for the application.

This design is ideal for implementing services which parse the command line, and which execute short-lived applications. For example, end user commands can be implemented as such services, so integrating with YARN's workflow and `YarnClient` client-side code.

It can just as easily be used for implementing long-lived services that parse the command line -it just becomes the responsibility of the service to decide when to return from the `execute()` method. It doesn't even need to `stop()` itself; the launcher will handle that if necessary.

The `LaunchableService` interface extends `Service` with two new methods.

`LaunchableService.bindArgs(Configuration, List)` provides the `main(String args[])` arguments as a list, after any processing by the Service Launcher to extract configuration file references. This method *is called before* *`Service.init(Configuration)`*. This is by design: it allows the arguments to be parsed before the service is initialized, so allowing services to tune their configuration data before passing it to any superclass in that `init()` method. To make this operation even simpler, the `Configuration` that is to be passed in is provided as an argument. This reference passed in is the initial configuration for this service; the one that will be passed to the init operation. In `LaunchableService.bindArgs(Configuration, List)`, a Launchable Service may manipulate this configuration by setting or removing properties. It may also create a new Configuration instance -which may be needed to trigger the injection of HDFS or YARN resources into the default resources of all Configurations. If the return value of the method call is a configuration reference (as opposed to a null value), the returned value becomes that passed in to the `init()` method.

After the `bindArgs()` processing, the service's `init()` and `start()` methods are called, as usual.

At this point, rather than block waiting for the service to terminate (as is done for a basic service), the method `LaunchableService.execute()` is called. This is a method expected to block until completed, returning the intended application exit code of the process when it does so.

After this `execute()` operation completes, the service is stopped and exit codes generated. Any exception raised during the `execute()` method takes priority over any exit codes returned by the method --so services may signal failures simply by returning exceptions with exit codes.

To view the worklow in sequence, it is:

1. (prepare configuration files --covered later)
2. instantiate service via its empty or string constructor
3. call `LaunchableService.bindArgs(Configuration, List)`
4. call `Service.init(Configuration)` with the existing config, or any new one returned by `LaunchableService.bindArgs(Configuration, List)`
5. call `Service.start()`
6. call `LaunchableService.execute()`
7. call `Service.stop()`
8. The return code from the `LaunchableService.execute()` becomes the exit code of the process, unless overridden by an exception.
9. If an exception was raised in this workflow: propagate it
10. If an exception was recorded in `Service.getFailureCause()` while the service was running: propagate it.

Exit codes and Exceptions

For a basic service, the return code is 0 unless an exception was raised.

For a `LaunchableService`, the return code is the number returned from the `LaunchableService.execute()` operation, again, unless overridden an exception was raised.

Exceptions are converted into exit codes -but rather than simply have a "something went wrong" exit code , exceptions >*may* provide exit codes which will be extracted and used as the return code. This enables LaunchableServices to use exceptions as a way of returning error codes to signal failures -and for normal Services to return any error code at all.

Any exception which implements the `ExitCodeProvider` interface is considered be a provider of the exit code: the method `ExitCodeProvider.getExitCode()` will be called on the caught exception to generate the return code. This return code and the message in the exception will be used to generate an instance of `ExitUtil.ExitException` which can be passed down to `ExitUtil.terminate(ExitUtil.ExitException)` to trigger a JVM exit. The initial exception will be used as the cause of the `ExitUtil.ExitException`.

If the exception is already an instance or subclass of `ExitUtil.ExitException`, it is passed directly to `ExitUtil.terminate(ExitUtil.ExitException)` without any conversion. One such subclass, `ServiceLaunchException`

may be useful: it includes formatted exception message generation. It also declares that it extends `LauncherExitCodes` interface listing common exception codes. These are exception codes that can be raised by the `ServiceLauncher` itself to indicate problems during parsing the command line, creating the service instance and such like. There are also some common exit codes for Hadoop/YARN service failures, such as `LauncherExitCodes.EXIT_CONNECTIVITY_PROBLEM`. Note that `ExitUtil.ExitException` itself implements `ExitCodeProvider.getExitCode()`

If an exception does not implement `ExitCodeProvider.getExitCode()`, it will be wrapped in an `ExitUtil.ExitException` with the exit code `LauncherExitCodes.EXIT_EXCEPTION_THROWN`.

To view the exit code extraction in sequence, it is:

1. If no exception was triggered by a basic service, a `ServiceLaunchException` with an exit code of 0 s created.
2. For a LaunchableService, the exit code is the result of `execute(). Again, a ServiceLaunchException with a return code of 0 is created.`
3. Otherwise, if the exception is an instance of `ExitException`, it is returned as the service terminating exception.
4. If the exception implements `ExitCodeProvider`, its exit code and `getMessage()` value become the exit exception.
5. Otherwise, it is wrapped as a `ServiceLaunchException` with the exit code `LauncherExitCodes.EXIT_EXCEPTION_THROWN` to indicate that an exception was thrown.
6. This is finally passed to `ExitUtil.terminate(ExitUtil.ExitException)`, by way of `ServiceLauncher.exit(ExitUtil.ExitException)`; a method designed to allow subclasses to override for testing.
7. The `ExitUtil` class then terminates the JVM with the specified exit code, printing the `toString()` value of the exception if the return code is non-zero.
   This process may seem convoluted, but it is designed to allow any exception in the Hadoop exception hierarchy to generate exit codes, and to minimise the amount of exception wrapping which takes place.

Interrupt Handling

The Service Launcher has a helper class, the `InterruptEscalator` to handle the standard SIGKILL signal and control-C signals. This class Registers for signal callbacks from these signals, and, when received, attempts to stop the service in a limited period of time, then triggers a JVM shutdown by way of `ExitUtil.terminate(int, String)`

If a second signal is received, the `InterruptEscalator` reacts by triggering an immediate JVM halt, invoking `ExitUtil.halt(int, String)`. This escalation process is designed to address the situation in which a shutdown-hook can block, yet the caller (such as an init.d daemon) wishes to kill the process. The shutdown script should repeat the kill signal after a chosen time period, to trigger the more aggressive process halt. The exit code will always be `LauncherExitCodes.EXIT_INTERRUPTED`.

The `ServiceLauncher` also registers a `ServiceShutdownHook` with the Hadoop shutdown hook manager, unregistering it afterwards. This hook will stop the service if a shutdown request is received -so ensuring that if the JVM is exited by any thread, an attempt to shut down the service will be made.

Configuration class creation

The Configuration class used to initialize a service is a basic `Configuration` instance. As the launcher is the entry point for an application, this implies that the HDFS, YARN or other default configurations will not have been forced in through the constructors of `HdfsConfiguration` or `YARNConfiguration`.

There are three strategies for dealing with this

Subclass the Service launcher and override its `ServiceLauncher.createConfiguration()` method with one that creates the right configuration. This is good if a single launcher can be created for all services launched by a module, such as HDFS or YARN. It does imply a dedicated script to invoke the custom main() method.

In the `LaunchableService.bindArgs(Configuration, List)`, a new configuration is created:

```
public Configuration bindArgs(Configuration config, List args) throws
Exception {
  Configuration newConf = new YARNConfiguration(config);
  return newConf;
}
```

This guarantees a configuration of the right type is generated for all instances created via the service launcher. It does imply that this is expected to be only way that services will be launched.

```
protected void serviceInit(Configuration conf) throws Exception {
  super.serviceInit(new YARNConfiguration(conf));
```

```
}
```

This is a strategy used by many existing YARN services, and is ideal for services which to not implement the LaunchableService interface. Its one weakness is that the configuration is now private to that instance. Some YARN services use a single shared configuration instance as a way of propagating information between peer services in a `CompositeService`. While a dangerous practice, it does happen.

Configuration file loading

Before the service is bound to the CLI, the ServiceLauncher scans through all the arguments after the first one, looking for instances of the argument `ServiceLauncher.ARG_CONF` argument pair: `--conf <file>`. This must refer to a file in the local filesystem which exists. It will be loaded into the Hadoop configuration class (the one created by the `ServiceLauncher.createConfiguration()` method. If this argument is repeated multiple times, all configuration files are merged -with the latest file on the command line being the last one to be applied. All the `--conf <file>` argument pairs are stripped off the argument list provided to the instantiated service; they get the merged configuration, but not the commands used to create it.

Utility Classes

- `IrqHandler`: registers interrupt handlers using `sun.misc` APIs.
- `ServiceLaunchException`: a subclass of `ExitUtil.ExitException` which takes a String-formatted format string and a list of arguments to create the exception text.