ORACLE®

JavaOne™

ORACLE®

Enhanced Process APIs

Roger Riggs
Consulting Member of Technical Staff
Java Products Group, Oracle
October 27, 2015

JavaOne ORACLE

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.

Program Agenda

1. Creating and Working with Processes
2. Information about Processes
3. Asynchronous Process Management
4. Efficient Handling of Process Output
5. Summary

Many Use Cases

- Running arbitrary commands
  - Collecting, filtering, and redirecting output
  - Connecting heterogeneous commands and shells
- Test execution
  - Run a series of tests
  - Log the output
  - Clean up left over processes
- Monitoring
  - Monitor long running processes and re-spawn if they die
  - Collect usage statistics

ProcessBuilder – The basics

- ProcessBuilder Basics
  - Command and arguments
  - Environment variables and working directory
  - Redirection
    - Standard input, standard output, standard error
    - Inherit from invoking process or discard output
    - Send to or read from Files
    - Send to OutputStreams or read from InputStreams
- Create a process

```
Process p = new ProcessBuilder(“date”).start();
```

Process – Controls a spawned process

- waitFor(), waitFor(timeout, units) – wait for the process to exit
- isAlive(), getPid(), info(), exitValue() – information about the process
- Redirecting output and input to I/O Streams
  - getErrorStream(), getInputStream(), getOutputStream()
- children(), allChildren() – the direct and indirect children
- destroy(), destroyForcibly(), supportsNormalTermination()
- onExit() – a ComputableFuture for process exit

ProcessHandle – A native process

- allProcesses() – All OS processes*
- getCurrent(), of(pid), parent() – get ProcessHandles
- isAlive(), getPid(), info() – information about the process
- children(), allChildren() – streams of direct and indirect children
- destroy(), destroyForcibly()
- onExit() – a ComputableFuture for process exit

* Limited by the native system access controls

ProcessBuilder redirecting to a file

```
File outFile = new File("out.tmp");

Process p = new ProcessBuilder(“ls”, “-lt”)
    .directory(new File(“/home/duke”))
    .redirectOutput(outFile)
    .redirectError(Redirect.INHERIT)
    .start();

int status = p.waitFor();
if (status == 0) {
    p = new ProcessBuilder("cat" , outFile.toString())
            .inheritIO()
            .start();
    p.waitFor();
}
```

Program Agenda

1 Creating and Working with Processes

2 Information about Processes

3 Asynchronous Process Management

4 Efficient Handling of Process Output

5 Summary

ProcessBuilder can supply environment variables

```
ProcessBuilder pb = new ProcessBuilder("printenv", "horse", "dog“, “LANG”)
                        .inheritIO();

pb.environment().put("horse", "oats");
pb.environment().put("dog", "treats");

pb.start().waitFor();
```

Output from printenv

```
oats
treats
en_US.UTF-8
```

Information about Processes

ProcessHandle.Info

- Information about processes is controlled by the OS
- Values are wrapped in Optional to indicate if the value is not available
- The user – Optional<String>
- The command – Optional<String>
- The arguments – Optional<String[]>
- The start time – Optional<Instant>
- The cputime – Optional<Duration>

Information about Processes

```
    static void showProcess(ProcessHandle ph) {
        ProcessHandle.Info info = ph.info();
        log.printf(“pid: %d, parent: %s, user: %s, cmd: %s%n”,
            ph.getPid(), ph.getParent(),
            info.user().orElse(“none”), info.command().orElse(“none”);
    }
```

```
% showProcess(ProcessHandle.current());
pid: 2909, parent: Optional[2650], user: duke, cmd: /opt/jdk1.9.0/bin/java
```

Filter Processes using Streams

```
Optional<String> currUser = ProcessHandle.current().info().user();
ProcessHandle.allProcesses()
    .filter(p1 -> p1.info().user().equals(currUser))
    .sorted(CodeSamples::parentComparator)
    .forEach(CodeSamples::showProcess);


static int parentComparator(ProcessHandle p1, ProcessHandle p2) {
    return Long.compare(p1.parent().get().getPid(),
                        p2.parent().get().getPid());
}
```

Sensitive Process Information

- Process information may contain sensitive info, userids, paths, arguments to commands
- Process control is sensitive, destroying a process may be detrimental
- When running as a normal application a ProcessHandle has the same OS privileges to information about other processes as a native application; information about system processes may not be available
- When a SecurityManager is in use, security policy must grant
  - `RuntimePermission(“manageProcess”)`

Program Agenda

1. Creating and Working with Processes
2. Information about Processes
3. Asynchronous Process Management
4. Efficient Handling of Process Output
5. Summary

OnExit – Flexible handling of process exit

- onExit returns a ComputableFuture<Process>
- ComputableFuture is multi-faceted handle to the Process / ProcessHandle
- As a Future the use is synchronous
  - isDone(), get(), get(timeout, units)
- As a ComputableFuture can schedule actions when the process exits
  - thenApply, thenAccept, thenRun,
    thenApplyAsync, thenAcceptAsync, thenRunAsync, etc.
  - Actions run in a thread provided by the ForkJoinPool commonPool

Example using Process.onExit

- Set of commands to run repeatedly
- Parallelism – run <n> of them in parallel
- Keep track of the results

Start the process again

```
    Semaphore count = new Semaphore(11);
    CountDownLatch end = new CountDownLatch(1);

    static void start(ProcessBuilder pb, Semaphore count, CountDownLatch end) {
        try {
            if (count.tryAcquire()) {
                Process p = pb.start();
                p.onExit()
                        .thenAccept(CodeSamples::logExit)
                        .thenRun(() -> start(pb, count, end));
            } else {
                end.countDown();
            }
        } catch (IOException ioe) {
            throw new RuntimeException("Process start failed", ioe);
        }
    }

    static void logExit(Process p) {
        log.printf("exit: %d, status: %d%n", p.getPid(), p.exitValue());
    }
```

Count each run

```
Semaphore count = new Semaphore(11);
CountDownLatch end = new CountDownLatch(1);

static void start(ProcessBuilder pb, Semaphore count, CountDownLatch end) {
    try {
        if (count.tryAcquire()) {
            Process p = pb.start();
            p.onExit()
                    .thenAccept(CodeSamples::logExit)
                    .thenRun(() -> start(pb, count, end));
        } else {
            end.countDown();
        }
    } catch (IOException ioe) {
        throw new RuntimeException("Process start failed", ioe);
    }
}

static void logExit(Process p) {
    log.printf("exit: %d, status: %d%n", p.getPid(), p.exitValue());
}
```

Finish when all have been started

```
Semaphore count = new Semaphore(11);
CountDownLatch end = new CountDownLatch(1);

static void start(ProcessBuilder pb, Semaphore count, CountDownLatch end) {
    try {
        if (count.tryAcquire()) {
            Process p = pb.start();
            p.onExit()
                    .thenAccept(CodeSamples::logExit)
                    .thenRun(() -> start(pb, count, end));
        } else {
            end.countDown();
        }
    } catch (IOException ioe) {
        throw new RuntimeException("Process start failed", ioe);
    }
}

static void logExit(Process p) {
    log.printf("exit: %d, status: %d%n", p.getPid(), p.exitValue());
}
```

OnExit – Complete

```
Semaphore count = new Semaphore(11);
CountDownLatch end = new CountDownLatch(1);

static void start(ProcessBuilder pb, Semaphore count, CountDownLatch end) {
    try {
        if (count.tryAcquire()) {
            Process p = pb.start();
            p.onExit()
                    .thenAccept(CodeSamples::logExit)
                    .thenRun(() -> start(pb, count, end));
        } else {
            end.countDown();
        }
    } catch (IOException ioe) {
        throw new RuntimeException("Process start failed", ioe);
    }
}

static void logExit(Process p) {
    log.printf("exit: %d, status: %d%n", p.getPid(), p.exitValue());
}
```

Repeat command and wait for them to be done

```
void repeat(ProcessBuilder pb, int total, int parallelism) throws Exception {
    Semaphore count = new Semaphore(total);
    CountDownLatch end = new CountDownLatch(1);

    for (int i = 0; i < parallelism; i++)      // Start the first n
        start(pb1, count, end);

    end.await();                     // wait until there are no more to be started

    ProcessHandle.current()      // wait for each of the active children to exit
        .children().forEach(CodeSamples::waitForExit);
}

static void waitForExit(ProcessHandle p) {
    try { p.onExit().get();} catch (Exception e) { … }
}

repeat(new ProcessBuilder("sh", "-c", "sleep 1;exit 1"), 11, 2);
```

Process Diagnostics and Progressive cleanup

- Commands don’t always terminate when expected
- Before destroying the process it is helpful to log some diagnostics
- Simply requesting the process to terminate normally may not succeed
- Harsher measures may be needed

Monitor the last moments of the Process

```
class TimeoutMonitor implements Runnable {
    public static void schedule(Process process, int delay, int rate) {
        new TimeoutMonitor(process).scheduledFuture
           = timeoutExecutor.scheduleAtFixedRate(ts, delay, rate, TimeUnit.SECONDS);
    }
    public synchronized void run() {
        if (process.isAlive()) {
            log.printf("Timeout countdown: %d%n", --countdown);
            showProcess(process.toHandle());
            process.allChildren().forEach(CodeSamples::showProcess);
            if (countdown == 1) {
                log.printf("Destroy process: %d%n", process.getPid());
                process.destroy();
            } else if (countdown == 0) {
                log.printf("Forcibly destroy process: %d%n", process.getPid());
                process.allChildren().forEach(ProcessHandle::destroyForcibly);
                process.destroyForcibly();
            }
        } else {
            scheduledFuture.cancel(false);
        }
    }
}
```

Start monitor for timeout

```
    Semaphore count = new Semaphore(11);
    CountDownLatch end = new CountDownLatch(1);

    static void start(ProcessBuilder pb, Semaphore count, CountDownLatch end) {
        try {
            if (count.tryAcquire()) {
                Process p = pb.start();
                p.onExit()
                        .thenAccept(CodeSamples::logExit)
                        .thenRun(() -> start(pb, count, end));
                TimeoutMonitor.schedule(p, 120, 5);
            } else {
                end.countDown();
            }
        } catch (IOException ioe) {
            throw new RuntimeException("Process start failed", ioe);
        }
    }

    static void logExit(Process p) {
        log.printf("exit: %d, status: %d%n", p.getPid(), p.exitValue());
    }
```

Program Agenda

1 Creating and Working with Processes

2 Information about Processes

3 Asynchronous Process Management

4 Efficient Handling of Process Output

5 Summary

Pipeline Output between Processes

Candidate for JDK 9

- Pipelines are a familiar tool for shell users
- Previously, no direct way to send the output of one process to another
- New ProcessBuilder.startPipe(ProcessBuilder… builders)
  - Launches one process for each builder
  - The standard output of each is directed to the standard input of the next
  - Input to the first builder and output of the last builder can be redirected
  - Returns a List of the processes created

Pipelining Processes

```
ProcessBuilder pb1 = new ProcessBuilder("ls");
ProcessBuilder pb2 = new ProcessBuilder("fgrep", “duke")
                         .redirectOutput(Redirect.INHERIT);
List<Process> processes = ProcessBuilder.startPipe(pb1, pb2);

processes.forEach(p -> {
    try {
        int status = p.waitFor();
        log.printf("status: %d%n", status);
    } catch (InterruptedException ie) {
    }
});
```

Pipe Channels for Process Output

Candidate for JDK 9

- Handling output of Processes via IO streams requires a thread per process
- NIO Channels for Pipes support bulk data transfers
- Pipes Are SelectableChannels

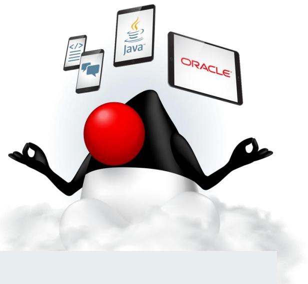
Copy Process Output via Pipe Channel to File

```
Process p = new ProcessBuilder("ls", "-ltGh")
               .redirectOutput(ProcessBuilder.Redirect.PIPE_CHANNEL)
               .start();

 // Copy from the channel to a file
 Path path = Paths.get("out.tmp");
 try (WritableByteChannel out = Files.newByteChannel(path, …  );
      Pipe.SourceChannel chan = p.getInputChannel()) {

     ByteBuffer bb = ByteBuffer.allocate(4096);
     while (chan.read(bb) > 0) {
         bb.flip();
         out.write(bb);
         bb.clear();
     }
 }
 p.waitFor();
```

NIO Selector can handle many channels in a Single Thread

- Create a Selector
- Associate a Channel specific function to consume the data
- Register Channel with the Selector
- Run the Selector to dispatch ready channels
- Wait for it to complete

Setup the Selector and Consumers of Process Output



```
Selector selector = Selector.open();

ProcessBuilder pb = new ProcessBuilder("ls", "-ltGh");
startTally(pb, selector);

ProcessBuilder pb2 = new ProcessBuilder("ls", "-l", "/tmp")
    .redirectOutput(ProcessBuilder.Redirect.PIPE_CHANNEL);
startTally(pb2, selector);

ProcessBuilder pb3 = new ProcessBuilder("ls", "-l", "/xxx")
        .redirectOutput(ProcessBuilder.Redirect.PIPE_CHANNEL);
startTally(pb3, selector);

runSelector(selector);
```

Create the consumer and register the channel

```
void startTally(ProcessBuilder pb, Selector selector) throws IOException {
    int[] tally = new int[1];

    pb.redirectOutput(ProcessBuilder.Redirect.PIPE_CHANNEL);

    Process p = pb.start();

    Consumer<SelectionKey> tallyFunc = (SelectionKey k) -> tallySize(k, p, tally);

    Pipe.SourceChannel chan = p.getInputChannel();
    chan.configureBlocking(false);
    chan.register(selector, SelectionKey.OP_READ, tallyFunc);
}
```

Run the Selector to dispatch ready channels

```
void runSelector(Selector selector) throws IOException {
    while (selector.selectNow() > 0 ||
            (selector.keys().size() > 0 && selector.select() > 0)) {

        Iterator<SelectionKey> it = selector.selectedKeys().iterator();
        while (it.hasNext()) {
            SelectionKey key = it.next();
            it.remove();

            ((Consumer<SelectionKey>) key.attachment())
                    .accept(key);              // Invoke the consumer
        }
    }
}
```

Channel Consumer to tally output size

```
void tallySize(SelectionKey key, Process p, int[] tally) {
    ReadableByteChannel chan = (ReadableByteChannel) key.channel();
    ByteBuffer bb = ByteBuffer.allocate(4096);
    try {
        int len;
        while ((len = chan.read(bb)) > 0) {
            tally[0] += len;
        }
        if (len < 0) {      // EOF
            log.printf("pid: %d, exit: %d, size: %d%n",
                              p.getPid(), p.exitValue(), tally[0]);
            closeChannel(chan);
        }
    } catch (IOException ioe) {
        closeChannel(chan);
    }
}
```

NIO Pipe Channel Summary

Candidate for JDK 9

- Create a Selector
- Register Channel and a Consumer with the Selector
- Run the Selector to dispatch ready channels
- Wait for everything to complete

Process and Process Handle Recap

- Information about native processes
  - Process id, user, command, arguments, cpu time, start time
- Monitor and Control
  - isAlive
  - destroy, destroyForcibly
- Process hierarchy
  - Streams of ProcessHandles
  - For all Processes, children and descendents

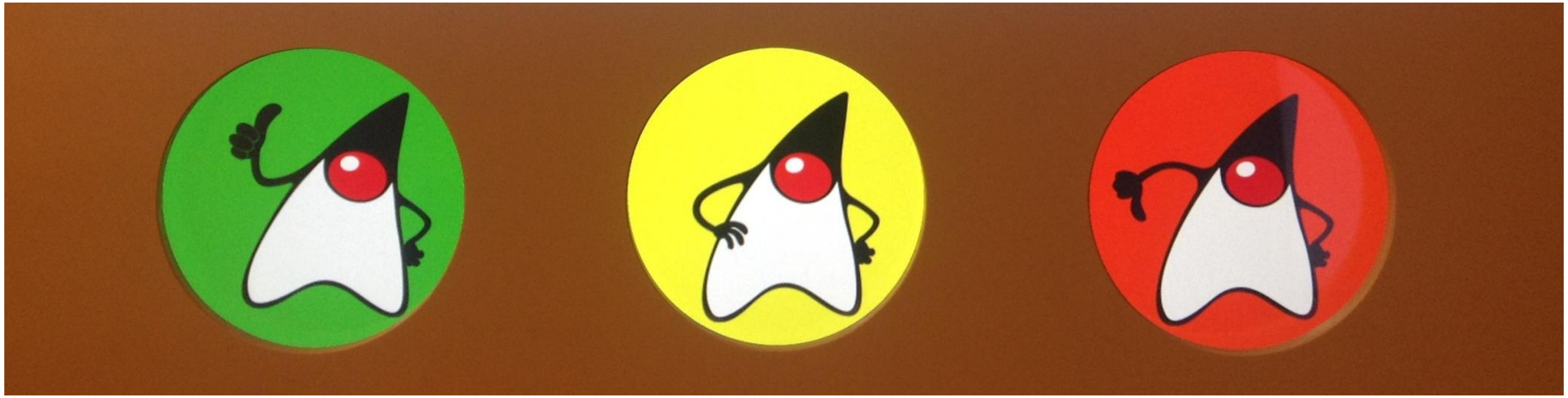
Process Enhancements Summary

- Monitoring process and task oriented results processing
  - onExit – Link tasks using ComputableFuture and the J.U.C. common pool
- Process handling of output
  - Pipeline output between processes *
  - Selectable Pipe Channels – scalable processing of output from processes *

* Candidate for JDK 9

Questions?

Comments!



Session Surveys

Help us help you!!

- Oracle would like to invite you to take a moment to give us your session feedback. Your feedback will help us to improve your conference.
- Please be sure to add your feedback for your attended sessions by using the Mobile Survey or in Schedule Builder.

JavaOne™

ORACLE®

ORACLE®