CAPÍTULO 19

El compilador de Java: javac

En este capítulo

- *Ejecución de javac.*
- *El compilador javac_g.*
- *Lecturas adicionales.*
- *Sumario.*

Ésta es la principal herramienta de desarrollo para los usuarios de SDK. El compilador de Java reúne archivos fuente .java en objetos *bytecode* .class, los ejecutables que entiende la Máquina Virtual de Java (JVM, *Java Virtual Machine*).

Ejecución de javac

Invocar a javac implica compilar los archivos de código fuente Java especificados. El compilador es en sí una aplicación Java, de modo que la invocación se traduce en lanzar una JVM que ejecute las clases compiladoras de Java.

Sintaxis:

```
javac [-green ¦ -native] [<opciones>] <archivos fuente>
javac [-green ¦ -native] [<opciones>] [<archivos fuente>] [@<archivos>]
(sólo en SDK1.2)
```

La versión SDK1.2 proporciona una utilidad para proyectos que contengan una larga lista de archivos fuente: la opción @ permite al compilador leer esta lista desde un archivo en lugar de atiborrar la línea de comando.

Plataformas: SDK1.2 y SDK1.1.

Opciones:

- -classpath <*ruta_clase*>. Establece la ruta de acceso a las clases donde hay que buscar durante la compilación. Si esta opción no está especificada, se usa $CLASSPATH.

  La manipulación del tiempo de compilación de la ruta de acceso a clases es similar a la manipulación en tiempo de ejecución: la ruta de acceso a clases en SDK1.1 debe incluir las clases centrales, en tanto que SDK1.2 se ocupa de las clases centrales por separado (véase la definición de –bootclasspath posteriormente en este capítulo).

  Observe que esta opción controla los lugares donde el compilador busca las clases. No regula el espacio donde la JVM que ejecuta el compilador busca las clases. Si es necesario, puede hacerse con la opción -J (será tratada posteriormente).

- -d <*directorio*>. Especifica un directorio al que van a ir los archivos .class. El compilador sitúa las clases en un subárbol jerárquico de este directorio basado en el nombre completo paquete+clase. Para más detalles, véase este tratamiento en el Capítulo 14, "Configuración del entorno SDK/JRE de Linux", en la sección "Clases cargadas desde los sistemas de ficheros".

  Si no especifica esta opción, los archivos .class se sitúan en el directorio actual. Si las clases son parte de un paquete, no va a poder ejecutarlas desde este directorio (también estudiado en el Capítulo 14, en la sección "Clases cargadas desde los sistemas de ficheros").

- -deprecation. Genera errores fatales si se utilizan clases despreciadas. Si no se especifica esta opción, el uso de este tipo de clases origina un aviso en tiempo de compilación.

- -encoding <*codificación*>. Especifica el carácter de codificación para los archivos fuente. Si no se especifica, javac emplea la codificación predefinida para el escenario actual.

- -g. Guarda la información íntegra del depurado en los archivos .class. Si no se especifica, el comportamiento predefinido es guardar alguna información (nombre de archivo fuente y números de línea, pero nunca información sobre variables locales).

- -g:none (sólo en SDK1.2). Impide que cualquier información de depurado se guarde en archivos .class.

- -g:<*lista de palabras clave*> (sólo en SDK1.2). Guarda, con criterio selectivo, información de depurado en el archivo .class. <lista de palabras clave> es una lista, separada por comas, de cualquiera de estas palabras o de todas ellas: source, lines o vars.

- -green. Obliga a que la JVM utilice el paquete de emulación de hilo green de Sun, que es una emulación del espacio de usuario del hilo *kernel*. Si se especifica, debe ser la primera opción en la línea de comando.

- -J<*opción*>. Especifica las opciones que se van a pasar al ejecutor de aplicaciones java que está ejecutando el compilador. Las opciones se pasan, desprovistas de -J, a la línea de comando de java.

- -native. Obliga a que la JVM utilice el hilo API de plataforma nativa. Si se especifica, debe ser la primera opción en la línea de comando.

- -nowarn. Anula los mensajes de aviso.

- -O. Optimiza el código para un mejor rendimiento en tiempo de ejecución: normalmente a costa del tamaño del código y de la depurabilidad. La documentación de SDK1.1 advierte de los riesgos del uso de esta opción; SDK1.2 ha eliminado algunos de los comportamientos de mayor riesgo.

  El consenso entre la comunidad de usuarios es que la mejor optimización la realizan los posprocesadores que operan en el proyecto entero después de la compilación. Estudiaremos uno de estos productos en el Capítulo 53, "DashO: optimización de aplicaciones para la distribución".

- -sourcepath <*ruta_acceso_fuente*> (sólo en SDK1.2). Especifica la ruta de acceso, separada por dos puntos, donde se buscarán los archivos fuente. Los componentes de la ruta de acceso pueden incluir directorios y archivos comprimidos ZIP y JAR.

- -verbose. Imprime mensajes informando de qué archivos de clases están compilándose y qué clases está cargando la JVM.

Opciones de compilación cruzada en SDK1.2

SDK1.2 proporciona estas opciones para soportar la compilación cruzada; la generación de clases compatibles con las plataformas anteriores de Java:

- -bootclasspath <*ruta_clases_arranque*>. Cambia la ruta de acceso de búsqueda que usa el compilador para resolver las referencias a clases centrales. Con esta opción podrá señalar la ruta de acceso a las clases de arranque en las librerías centrales de JDK1.1 durante una compilación para esa plataforma.

- -extdirs <*directorios*>. Emplea estos directorios de extensión (directorios que contienen archivos jar de extensiones) en lugar de la localización estándar para las extensiones SDK1.2/JRE1.2.

- -target <*versión*>. Especifica una JVM de destino: 1.1 ó 1.2. Si se especifica 1.1 (el predefinido), javac genera un código que se ejecutará en JDK1.1 y en las máquinas virtuales JDK1.2. Si se especifica 1.2, el código generado no ejecutará la máquina virtual en JDK1.1.

Opciones no estándar en SDK1.2:

- -X. Imprime una lista de opciones disponibles no estándar.

- -Xdepend. Lleva a cabo un análisis de las dependencias. Sustituye a la opción en tiempo de ejecución –checksource de java en JDK1.1, que es una opción de tiempo de compilación más acertada. Hace que javac busque todas las clases asequibles para los archivos fuente que son más recientes que sus objetos correspondientes y hace, también, que recompile las clases que provocan errores. El resultado de esto son las decisiones razonables, pero no totalmente robustas, según las cuales los archivos necesitan recompilarse.

  Desgraciadamente, esta opción ralentiza considerablemente la compilación. Estudiaremos un planteamiento alternativo a este problema en el Capítulo 48, "JmakeDepend: una utilidad para gestionar la construcción de un proyecto".

- -Xstdout. Envía mensajes a stdout (System.out) en vez de al predefinido stderr (System.err).

- -Xverbosepath. Genera mensajes de información describiendo dónde se encuentran los archivos de clases y de código fuente.

NOTAS

- Las clases que conforman el compilador de Java se encuentran en sitios diferentes en los dos entornos. En SDK1.1 están en el archivo classes.zip, que también acoge las clases centrales. En SDK1.2, se distribuyen en un tools.jar separado. Las personas dedicadas al desarrollo que confían en estas clases para crear sus propias herramientas necesitan incorporar el nuevo archivo jar en SDK1.2.

- Las opciones -d y -classpath son completamente independientes. Aunque -d especifica dónde se tienen que escribir las clases, no afecta a dónde se tienen que buscar éstas. Esto puede provocar alguna que otra sorpresa.

Considere un proyecto con dos clases, Foo y Bar, compilado en el directorio baz (javac -d baz ...), en el que la clase Foo contiene referencias a la clase Bar. Si tiene que cambiar y recompilar solamente Foo.java, debe especificar -d baz y también incluirse baz en la ruta de acceso a las clases. Esto indicará al compilador dónde debe poner Foo.class y dónde resolver la referencia a la clase Bar.

(Si ha recompilado los dos archivos de una vez, la modificación de la ruta de acceso a clases es innecesaria. Estudiaremos las complejidades del análisis de las dependencias en el proyecto JMakeDepend del Capítulo 48, "JMakeDepend: una utilidad para gestionar la construcción de un proyecto".)

El compilador javac_g

Esta versión del compilador es una parte normal sólo de SDK1.1. Hay una versión disponible para SDK1.2, aunque es muy poco probable que la vaya a necesitar.

Sintaxis:

```
javac_g [-green ¦ -native] [<opciones>] <archivos fuente>
```

Plataforma: SDK1.1.

Es una versión no optimizada del compilador empleada para la depuración del compilador en sí.

Lecturas adicionales

La documentación comercial de SDK1.1 y SDK1.2 dispone de información importante en las siguientes páginas:

```
javac   docs/tooldocs/solaris/javac.html
```

Sumario

Este capítulo ha presentado javac, el compilador comercializado con SDK de Sun. Esta herramienta no es la única opción en cuanto a compiladores; los capítulos siguientes echarán un vistazo a los compiladores alternativos de uso libre, entornos de desarrollo integrado, compiladores rápidos y compiladores que soportan extensiones de lenguaje.

CAPÍTULO 20

El depurador de Java: JDB

En este capítulo

SDK incluye un depurador no-GUI interactivo, jdb. Al igual que los ya conocidos depuradores de UNIX gdb y dbx, jdb proporciona una interfaz de comandos basada en textos que permite controlar la ejecución y examinar los contextos de las aplicaciones y las estructuras de los datos.

Ejecución de jdb

Tanto las aplicaciones como los *applets* se pueden depurar con jdb. También cabe la opción de ejecutar un *applet* desde jdb o adjuntarlo en una instancia ya existente que esté ejecutándose.

Sintaxis:

```
jdb [-green ¦ -native] [<opciones>] [<clases>] [<argumentos>]
jdb [-green ¦ -native] -host <nombre_host> -password <contraseña>
appletviewer [-green ¦ -native] -debug [<opciones>] <URL>
```

La primera forma de este comando es para ejecutar y depurar aplicaciones. La invocación es idéntica a ejecutar una aplicación con las opciones del ejecutor de aplicación java (véase el Capítulo 17, "Los ejecutadores de aplicaciones Java: java, jre, y oldjava"): un nombre de clase y unos argumentos que van a pasarse a main(). (Según la documentación de Sun, todas las opciones del ejecutor java se pueden utilizar aquí; la experiencia sugiere lo contrario.)

Si no se especifica <*clases*>, el depurador se inicia sin una clase cargada: puede cargarse después con un comando de depuración.

EJEMPLO

Para depurar una aplicación que normalmente se ejecuta con:

```
bash$ java com.foo.bar arg1 arg2
```

ejecute un depurador con:

```
bash$ jdb com.foo.bar arg1 arg2
```

La segunda forma del comando depura una aplicación que ya está ejecutándose. La aplicación original tiene que haberse ejecutado con la opción –debug; <*contraseña*> es la contraseña agente devuelta en la invocación de la aplicación.

EJEMPLO

Si utiliza jdb para adjuntar en una aplicación que ya está ejecutándose, tendrá que seguir ciertas prácticas cuando ejecute esa aplicación.

En SDK1.1 tiene que ejecutar la aplicación con java_g:

```
bash$ java_g -debug com.foo.bar arg1 arg2
Agent password=5k53pk
```

En SDK1.2 debe incluir tools.jar de SDK en la ruta de acceso a las clases cuando ejecute, y tiene que deshabilitar la compilación *just-in-time*:

```
bash$ java -debug -Djava.compiler= -classpath
$JAVA_HOME/lib/tools.jar:. com.foo.bar arg1 arg2
Agent password=5k53pk
```

Ahora ya está preparado para la depuración:

```
bash$ jdb -host localhost -password 5k53pk
```

La tercera forma del comando sirve para depurar *applets*. Véase appletviewer (Capítulo 18, "El visor de *applets* de Java: appletviewer") si desea la lista de opciones.

Plataformas: SDK1.2 y SDK1.1.

Opciones:

- -classpath. Se pasa al depurador de la JVM para utilizarla como ruta de acceso a las clases.
- -dbgtrace. Imprime información para la depuración del depurador.
- -D<*nombre_propiedad*>=<*valor_nuevo*>. Se pasa al depurador de la JVM para establecer valores de propiedad en la aplicación de destino.
- -green. Obliga a que la JVM emplee el paquete de emulación de hilo green de Sun, que es una emulación del espacio de usuario del hilo de *kernel*. Si se especifica, ésta debe ser la primera opción en la línea de comando. Observe que esto afecta a la operación del ejecutable del depurador, pero no a la JVM que ejecuta la aplicación que está depurándose. Utilice la variable de entorno THREADS_FLAG para afectar a esa JVM.
- -help. Imprime un mensaje de uso.
- -host <*nombre_host*>. *Host* en el que se están ejecutando procesos de depuración existentes.
- -native. Obliga a que la JVM utilice la API de hilos de plataforma nativa. Si se especifica, ésta debe ser la primera opción en la línea de comando.
- -password <*contraseña*>. Contraseña agente para los procesos existentes que van a depurarse.
- -version. Imprime la versión de jdb.
- -X<*opción java no estándar*> (sólo en SDK1.2). Se pasa a la JVM depuradora.

Comandos de jdb

Los comandos empleados en el depurador con interfaz textual son los siguientes:

Administración de hilos

- threads [<*grupo de hilos*>]. Incluye todos los hilos actuales en un grupo de hilos. Si no especifica el argumento, se usa el grupo de hilos predefinido (establecido por threadgroup, explicado posteriormente en esta lista). Un <threadgroup> de system deja ver todos los hilos.

  Para cada hilo, la lista muestra su nombre, el nombre de la clase de hilo (java.lang.Thread o clase derivada) y la situación actual del mismo. Para los comandos (estudiados a continuación) que necesitan un <*id_hilo*>, se usan los números ordinales (1, 2, 3, ...) que se muestran en la salida de los comandos.
- thread <*id_hilo*>. Establece el hilo predefinido que utilizarán los comandos específicos de los hilos, como los de navegación por la pila.
- suspend [<*ids_hilos*>]. Suspende uno o más hilos. El argumento opcional <*ids hilos*> es una lista (separada por espacios) de uno o más ID de hilos. Si no especifica ninguno, todos los hilos que no son del sistema se suspenden. Algunas operaciones de depuración, como la enumeración de variables locales, sólo se pueden llevar a cabo en hilos suspendidos.
- resume [<*ids_hilos*>]. Reanuda uno o más hilos. El argumento opcional <*ids hilos*> es una lista (separada por espacios) de uno o más números de hilos. Si no especifica ninguno, se reanudan todos los hilos.
- where [<*id_hilo*>]|[all]. Imprime un volcado de la pila para el <*id_hilo*> especificado, para todos los hilos o, si no especifica ningún argumento, para los hilos predefinidos.
- wherei [<*id_hilo*>]|[all]. Imprime un volcado de la pila, además de información del PC, para el <*id_hilo*> especificado, para todos los hilos o, si no especifica ningún argumento, para el hilo predefinido.
- threadgroups. Lista todos los grupos de hilos.
- threadgroup <*nombre*>. Establece el grupo de hilos predefinido para las diversas operaciones con hilos. El <*nombre*> se toma de la lista threadgroups, estudiado en el punto anterior.

Visualización de datos

- print <*id(s)*>. Imprime uno o más elementos, donde un elemento puede ser un local, una instancia, una variable de clase o un nombre de clase. Cuando se imprime una variable de clase, se necesita, aparentemente, cualificarla con el nombre de la clase (por ejemplo, classname.varname). Todos los elementos se escriben en su formato toString().
- dump <*id(s)*>. Imprime uno o más elementos, como en el comando anterior, pero en un formato detallado.
- locals. Imprime los nombres y los valores de todas las variables locales.
- classes. Incluye todas las clases e interfaces conocidas.
- methods <*id_clase*>. Enumera todos los métodos para una clase especificada.

Puntos de ruptura y navegación por la pila

- stop in <*id_clase*>.<*método*> y stop in <*id_clase*>.<*método*>[(<*tipo de argumento*>,...)] (sólo en SDK1.2). Fija un punto de ruptura en el método indicado. La versión SDK1.2 permite el acceso opcional al nombre del método con información íntegra de firma (tipos de argumento).
- stop at <*id_clase*>:<*línea*>. Establece el punto de ruptura en un número de línea especificado en la fuente <*id_clase*>.
- up [*n*]. Desplaza el número especificado de marcos hacia arriba en la pila de hilos. Si no especifica el argumento opcional, se predefine en 1.
- down [*n*]. Desplaza el número especificado de marcos hacia abajo en la pila de hilos. Si no especifica el argumento opcional, se predefine en 1.
- clear <*id_clase*>.<*método*>[(<*tipo_argumento*>,...)] (sólo en SDK1.2). Elimina el punto de ruptura en un método.
- clear <*id_clase*>:<*línea*>. Elimina el punto de ruptura del número de línea especificado en la fuente <*id_clase*>.
- step. Ejecuta la línea actual.
- step up. Ejecuta el resto del procedimiento actual hasta que vuelve al que llama.
- stepi. Ejecuta la instrucción actual del *bytecode*.
- next. Ejecuta la línea actual, ignorando las llamadas a métodos.
- cont. Continúa con la ejecución (actualmente parada).

Excepciones

- catch <*id_clase*>. Se rompe cuando tiene lugar una excepción especificada.
- ignore <*id_clase*>. Ignora la excepción especificada.

Fuente

- list [<*número de línea*>|<*método*>]. Lista el código fuente del número de línea especificado, para el método especificado o, si no lo ha especificado, próximo a la localización actual en la fuente.
- use [<*acceso_archivo_fuente*>]. Fija la ruta de acceso actual para encontrar archivos fuente. Si no especifica ningún argumento, imprime la ruta de acceso actual.

Recursos

- memory. Muestra el uso actual de la memoria: libre y total.
- gc. Obliga a que la recolección de residuos tenga lugar y recupere los objetos a los que no se ha hecho alusión.

Control de la depuración

- load <*nombre_clase*>. Carga una clase especificada para la depuración.
- run [<clase> [<*argumentos*>]]. Ejecuta la clase especificada con los argumentos especificados para main(). Si no especifica ningún argumento, ejecuta la clase y los argumentos especificados en la línea de comando de inicialización de jdb.
- !!. Repite el último comando de jdb ejecutado.
- help, ?. Muestra los comandos de jdb.
- exit, quit. Finaliza la sesión de depuración.

Previsión de futuro

El depurador jdb está creado sobre la antigua interfaz llamada *Java Debugger API*, que interactúa con una JVM en ejecución sobre una conexión de red. Sun describe jdb como concepto de prueba para esta API, cuya arquitectura se muestra en la Figura 20.1.

SDK1.2 introdujo un planteamiento nuevo de depuración *multitiered*, *Java Platform Debugging Architecture* (Arquitectura de Depuración de Plataforma Java), que

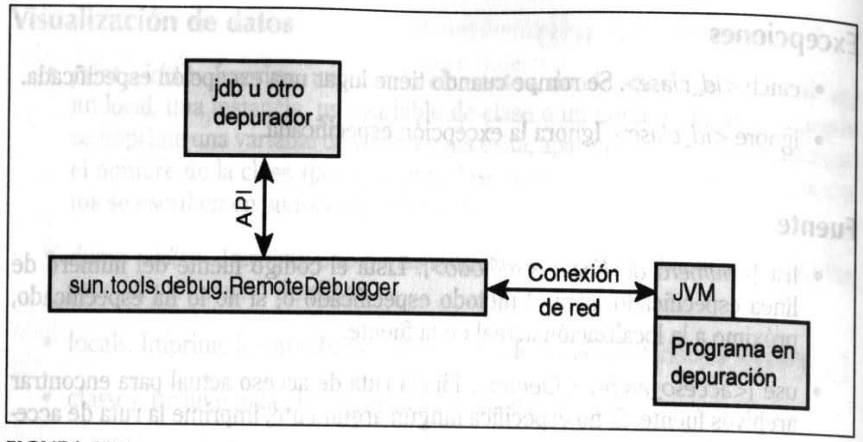

FIGURA 20.1
Depurador API de Java.

consta de una interfaz de código nativo de bajo nivel, un protocolo de red y una API de depuración (véase la Figura 20.2).

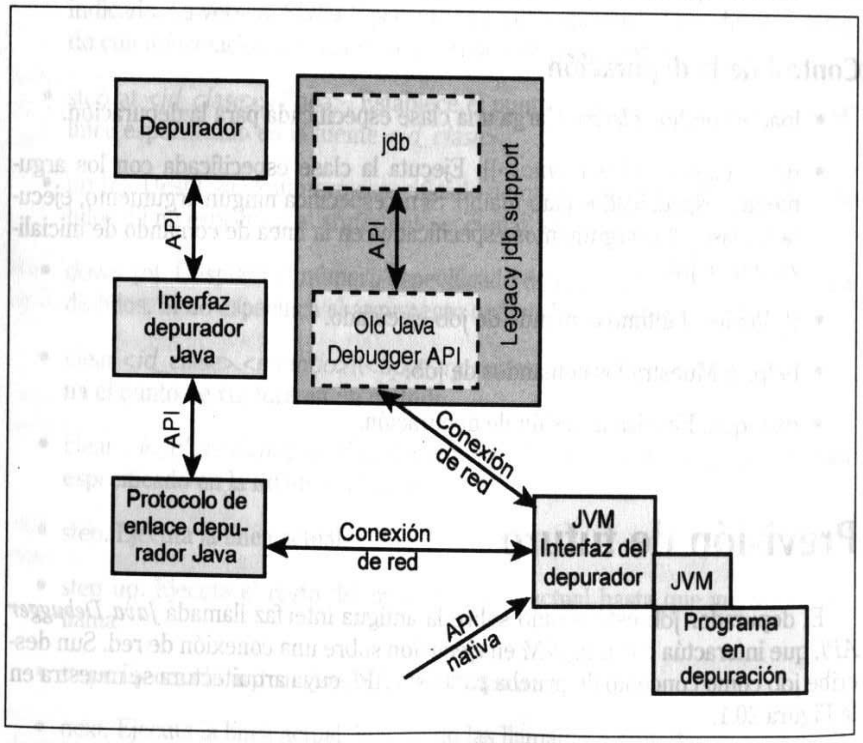

FIGURA 20.2
Arquitectura de depuración de plataforma Java (*Java Platform Debugging Architecture*).

La nueva arquitectura no se implementó del todo a tiempo para SDK1.2 y se ha programado para una versión posterior que, seguramente, contendrá otro depurador del tipo "concepto de prueba". El propósito a largo plazo de Sun es permitir a los desarrolladores de herramientas de terceros crear mejores depuradores. La interfaz de depuración de Java está pensada para que sea la API fundamental para los depuradores, aunque los implementadores tienen libertad para usar otras interfaces, incluida la interfaz nativa JVMDI de bajo nivel para una posible implementación de depuración en proceso.

Lecturas adicionales

La documentación comercial de SDK1.1 y SDK1.2 incluye información importante sobre las siguientes páginas:

```
jdb     docs/tooldocs/solaris/jdb.html
```

Sumario

Este capítulo ha explorado jdb, el depurador textual facilitado con SDK. Aunque jdb hace bien su cometido, no es ni la mejor ni la última palabra en depuradores de Java. Estudiaremos algunas alternativas: depuradores diferentes, así como también envolturas GUI para jdb, en el Capítulo 39, "El depurador Jikes", y en el Capítulo 40, "DDD: Data Display Debugger".

la variable frame en el panel Locals. Entonces podemos navegar por el panel Inspector para ver los contenidos de frame.

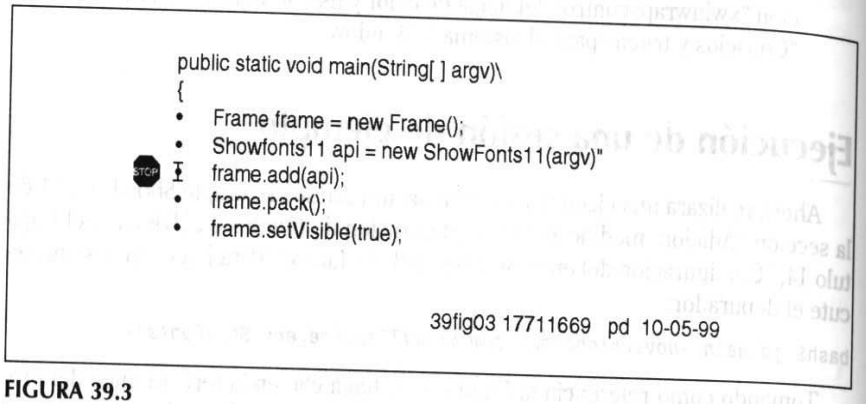

FIGURA 39.3
Inspeccionando la variable frame.

Puede continuar ejecutando y depurando utilizando los controles mencionados en esta sección.

Sumario

Este capítulo explica JD (*Jikes Debugger*) de AlphaWorks de IBM. Jikes todavía no es gratuito, y es un depurador GUI de primera clase. Desafortunadamente, todavía no ha migrado a JDK1.2 y parece no estar recibiendo más desarrollo.

CAPÍTULO 40

DDD: Data Display Debugger

En este capítulo

- *Cómo obtener DDD.*
- *Ejecución de DDD.*
- *La interfaz de DDD.*
- *Ejecución de una sesión de ejemplo.*
- *Calidad de DDD.*
- *Sumario.*

Data Display Debugger (DDD) es un frontal GUI gratuito para depuradores no-GUI como el depurador JDK de Sun.

Plataformas: SDK1.1 y SDK1.2.

DDD no es un depurador en sí mismo, pero consigue que los depuradores orientados a texto sean más sencillos de utilizar. Además de soportar depuradores de código nativo de UNIX clásicos como gdb, dbx y xdb, ha crecido en los últimos años para soportar Perl, Python y Java.

La magia de DDD es presentar un depurador GUI bien diseñado y traducir las acciones del usuario en comandos de texto reconocidos por la herramienta subyacente. Su habilidad para reconducir jdb de Sun (véase el Capítulo 20, "El depurador de Java: JDB") proporciona a los desarrolladores una poderosa depuración gráfica.

Cómo obtener DDD

DDD está disponible ampliamente en los almacenes Linux y UNIX. Está publicado en forma de código fuente y binario (para los sistemas libc5 y glibc), como RPM y *tarball*. Elija el método de instalación de su agrado y seleccione la distribución apropiada a su entorno.

Existen tres variantes de distribuciones binarias de DDD:

- ddd-dynamic. Una versión dinámicamente vinculada de DDD, con dependencias de muchas librerías X, y en la librería Motif (o su clónico Lesstif).
- ddd-semistatic. Una versión con la librería Motif vinculada estáticamente, eliminando la dependencia de una instalación Motif local. Esta versión todavía tiene dependencias de muchas librerías X.
- ddd-static. Un ejecutable vinculado estáticamente sin dependencias con librerías compartidas (ni siquiera con libc).

Para sistemas con Motif o Lesttif instalado, ddd-dynamic es la opción más sensata; para sistemas sin Motif o Lesstif, es más razonable ddd-semistatic.

Ejecución de DDD

El uso básico de DDD es directo, por lo que a continuación se enumeran sólo las opciones básicas y operaciones relacionadas con la depuración de Java. La funcionalidad de DDD es más extensa: su página *man* consta de 112 páginas (ejecute man ddd).

Sintaxis:

```
ddd --jdb [<opciones>] <clase>
```

Opciones:

- --font <*fuente*>. Especifica la fuente X que se utilizará para la interfaz.
- --fontsize <*tamaño*>. Especifica el tamaño de fuente, en unidades equivalentes a una décima de punto.
- --trace. Envía información de registro de la interacción de DDD<->jdb a stderr.
- --tty. Utiliza el terminal desde el que se lanzó ddd como consola de depuración en formato texto. El comportamiento predefinido es proporcionar la consola en una de las ventanas de la interfaz principal de DDD.
- --version. Informa del número de versión de DDD, y entonces sale.
- --configuration. Informa de la configuración de indicadores de DDD, y después sale.
- --manual. Ejecuta man ddd para mostrar la documentación.
- --license. Muestra la licencia de DDD.

Como programa X *Toolkit*, el ejecutable de ddd también soporta las opciones *toolkit* de X descritas en la página *man* del sistema X Window (man X).

Observe la ausencia de una opción --classpath. Debe configurar la ruta de acceso a las clases con la variable de entorno CLASSPATH.

Como con el depurador Jikes, DDD se hace difícil de utilizar si no es capaz de administrar la asignación de todos los colores que necesita. Los rodeos explicados en la sección "xwinwrap: : control del mapa de color y uso de visuales" del Capítulo 56, "Trucos y consejos para el sistema X Window", pueden ser útiles aquí.

La interfaz de DDD

La Figura 40.1 muestra la interfaz básica de DDD.

La Tabla 40.1 enumera las funciones disponibles con las ventanas que aparecen en la Figura 40.1.

Tabla 40.1 Ventanas de DDD y sus funciones

Ventana	Función
Ventana de fuente	Permite navegar por el código fuente. En ella también puede definir puntos de interrupción y examinar datos.
Ventana de consola	Muestra la interacción entre DDD y jdb. También puede introducir sus propios comandos jdb. Si ejecuta DDD con la opción --tty, el terminal desde el que lanzó el depurador actúa como consola.
Ventana de datos	Permite examinar en detalle las estructuras de datos (no aparece en la figura).
Cuadro de comandos	Dispone de comandos para ejecutar, detener, ejecución paso a paso, saltar y funciones similares.

El botón derecho del ratón despliega menús contextuales en la ventana del código fuente, permitiéndole administrar puntos de interrupción y examinar variables (véase la Figura 40.2). Estos menús son sensibles al contexto, por lo que las funciones disponibles dependen de la localización del ratón en el momento de pulsar el botón.

El resto de la extensa funcionalidad de DDD (configuración, búsqueda y control sobre la ejecución) queda recogida en los distintos menús desplegables de la barra de menús (véase la Figura 40.3).

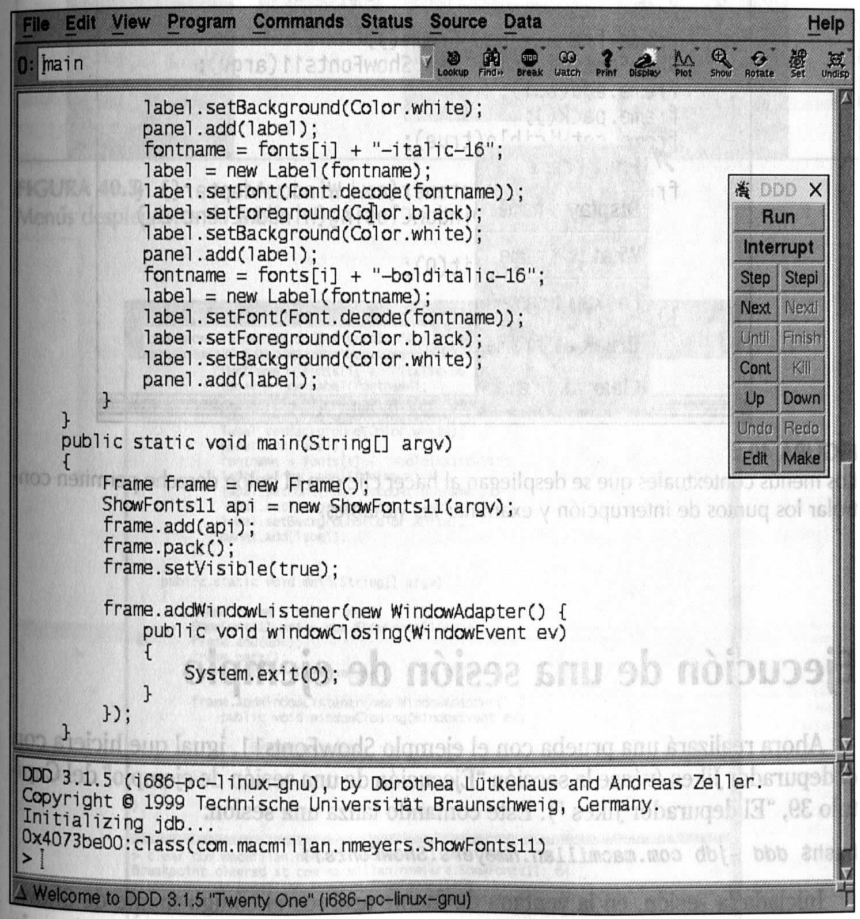

FIGURA 40.1
Interfaz de DDD, con una ventana para el código fuente, una ventana como consola y un cuadro de comandos.

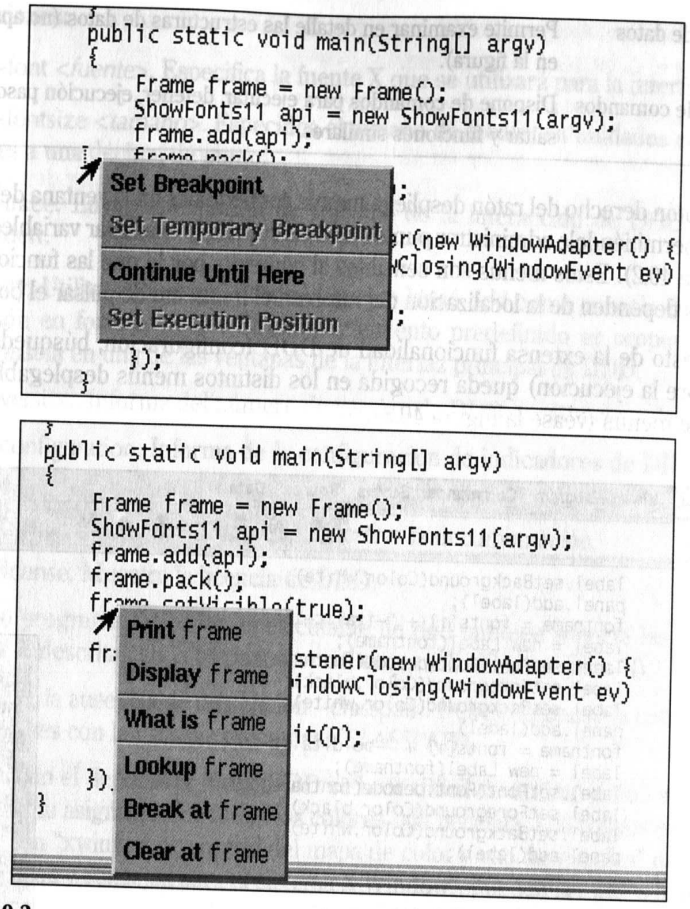

FIGURA 40.2
Los menús contextuales que se despliegan al hacer clic con el botón derecho permiten controlar los puntos de interrupción y examinar las variables.

Ejecución de una sesión de ejemplo

Ahora realizará una prueba con el ejemplo ShowFonts11, igual que hiciera con el depurador Jikes (véase la sección "Ejecución de una sesión de ejemplo" del Capítulo 39, "El depurador Jikes "). Este comando lanza una sesión:

```
bash$ ddd -jdb com.macmillan.nmeyers.ShowFonts11
```

Iniciada la sesión, en la ventana de fuente aparece el código fuente de ShowFonts11. Para fijar un punto de interrupción, haga clic con el botón derecho a la izquierda de la tercera línea de main() en esta ventana y seleccione Set Breakpoint desde la ventana desplegable (véase la Figura 40.4).

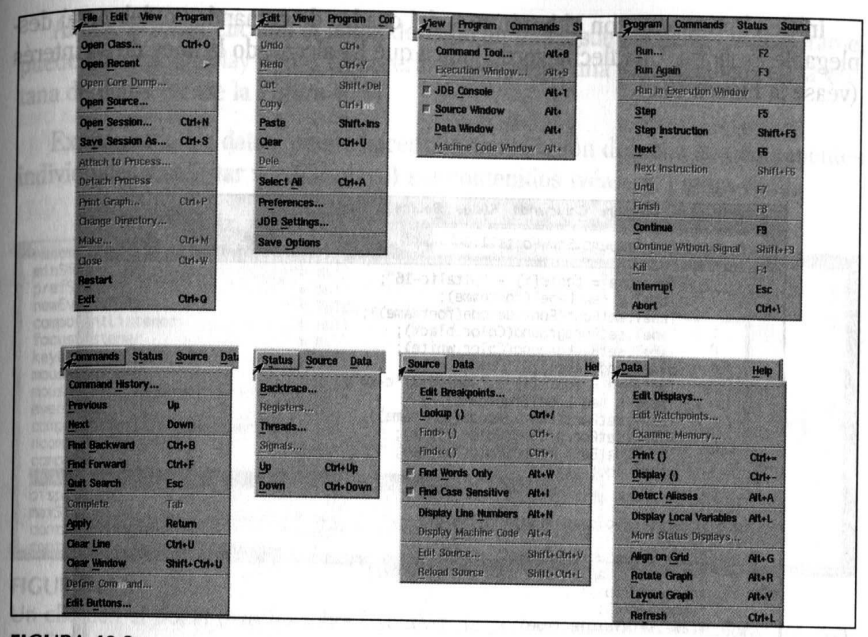

FIGURA 40.3
Menús desplegables de DDD.

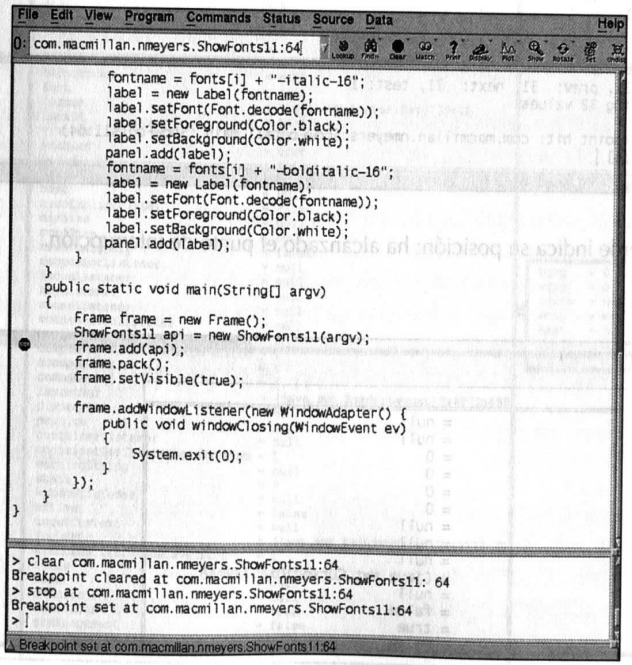

FIGURA 40.4
Punto de interrupción (indicado por una señal de stop) en la llamada frame.add().

Inicie el programa con el botón Run del cuadro de comandos o del menú desplegable Program. Una flecha verde indica que ha alcanzado la línea de su interés (véase la Figura 40.5).

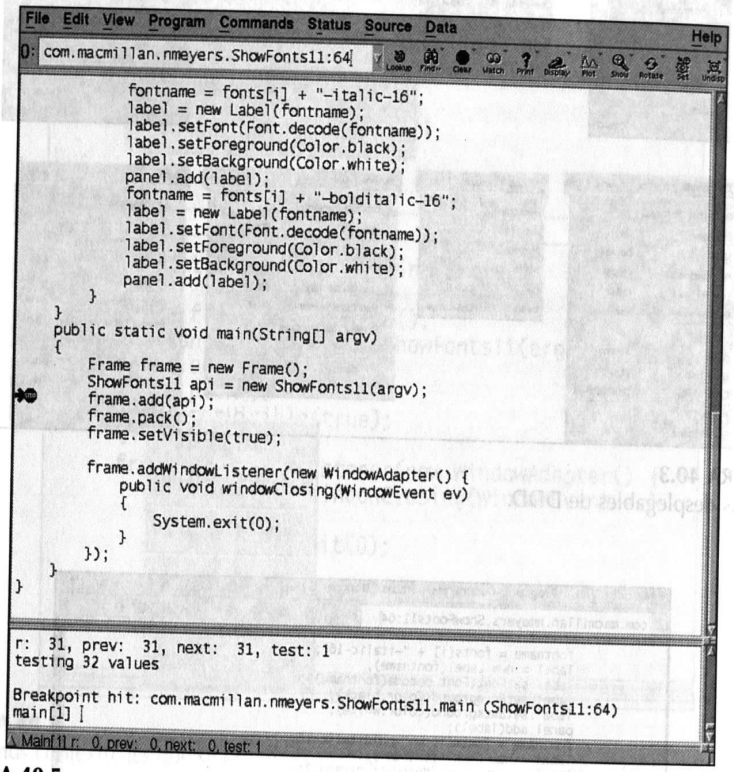

FIGURA 40.5
La flecha verde indica su posición: ha alcanzado el punto de interrupción.

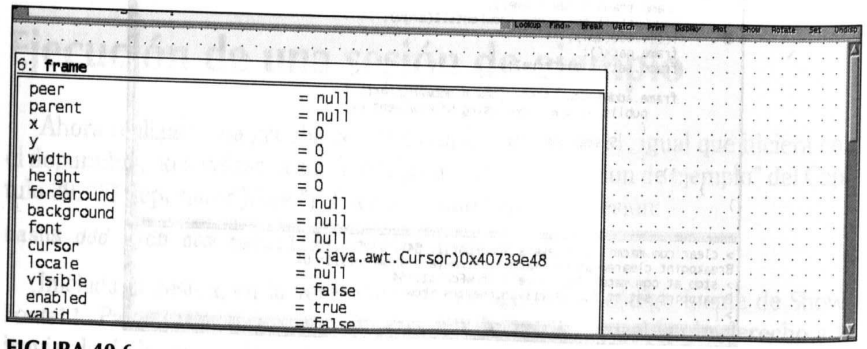

FIGURA 40.6
Detalles de la variable frame en la ventana de datos.

Ahora, al hacer clic con el botón derecho en una instancia de la variable frame, puede solicitar Display frame. Al hacerlo se despliega una vista detallada en la ventana de datos (véase la Figura 40.6).

Examinando los datos, puede hacer clic con el botón derecho en componentes individuales y solicitar ver (Display()) sus contenidos (véase la Figura 40.7).

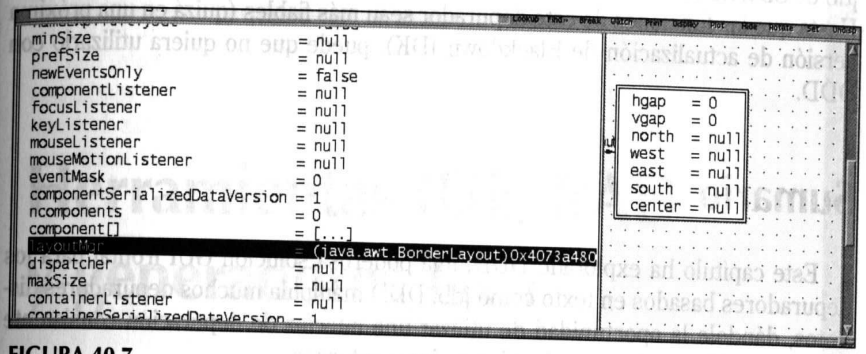

FIGURA 40.7
Un clic con el botón derecho sobre layoutMgr le permite descender y explorar sus contenidos.

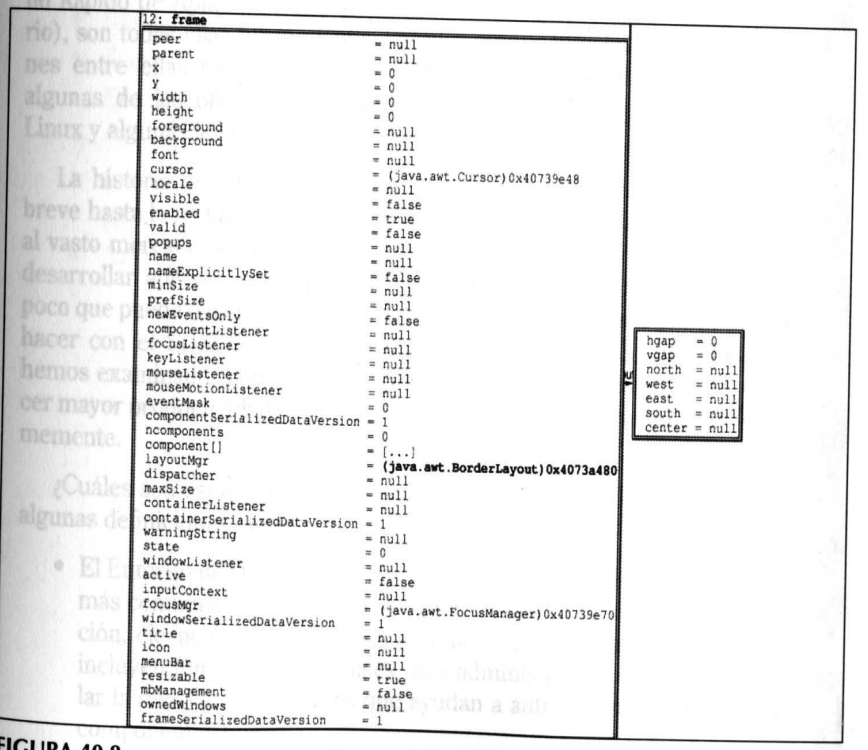

FIGURA 40.8
Traducción PostScript de los contenidos de la ventana de datos.

Calidad de DDD

DDD es una herramienta refinada. La experiencia demuestra que funciona bien con todos los depuradores, excepto con jdb de SDK1.2. El problema no es DDD, sino la inestabilidad de este depurador. Su comportamiento imprevisible hace que jdb de SDK1.2 sea difícil de utilizar por sí solo, y confunde de mala manera a DDD. Hasta que las funciones de este depurador sean más fiables (quizá en una próxima versión de actualización de Blackdown JDK), puede que no quiera utilizarlo con DDD.

Sumario

Este capítulo ha explorado DDD, una poderosa solución GUI frontal para los depuradores basados en texto como jdb. DDD manipula muchos depuradores distintos, dándole la oportunidad de utilizar una interfaz de depuración consistente con Java, C/C++ y otros muchos lenguajes y entornos.

PARTE IX

Herramientas IDE, RAD y generadores GUI

Las herramientas IDE (*Integrated Development Environments*, Entornos de Desarrollo Integrado), RAD (*Rapid Application Development*, Desarrollo Rápido de Aplicaciones) y generadores GUI (interfaz gráfica de usuario), son todas ellas herramientas basadas en una interfaz, y las distinciones entre ellas pueden ser poco claras. En esta parte examinaremos algunas de las ofertas disponibles, incluyendo aplicaciones nativas de Linux y algunas aplicaciones Java que se utilizan en Linux.

La historia de las herramientas de desarrollo avanzado en Linux es breve hasta la fecha, con la mayoría de las ofertas interesantes apuntando al vasto mercado de Microsoft Windows. Esto no significa que no pueda desarrollar aplicaciones Java importantes en Linux: realmente hay muy poco que pueda hacer con la herramienta RAD más avanzada que no pueda hacer con el SDK de Sun o con alguna de las otras herramientas que hemos examinado. Pero las herramientas avanzadas buenas pueden ofrecer mayor productividad, y su disponibilidad para Linux está mejorando firmemente.

¿Cuáles son estos productos y qué significan sus nombres? Aquí tiene algunas definiciones breves:

- El Entorno de Desarrollo Integrado (IDE) combina los componentes más comunes para el desarrollo de una aplicación (edición, compilación, ejecución y depuración) en una simple interfaz. A menudo se incluyen funciones adicionales para administrar el proyecto, controlar la revisión, y asistentes que ayudan a automatizar la creación de componentes nuevos.