Patrones de Concurrencia

[Introducción 2](#_Toc48897845)

[1. Sección Crítica (CRITICAL SECTION) 3](#_Toc48897846)

[1.1 Sección crítica con Fiabilidad de hilos (thread-safe) 4](#_Toc48897847)

[1.2 Sección crític con Inicialización temprana estática (Static early Initialization ) 4](#_Toc48897848)

[2. Orden de Bloqueo Consistente (CONSISTENT LOCK ORDER) 5](#_Toc48897849)

[3. Suspensión protegida (GUARDED SUSPENSION) 8](#_Toc48897850)

[4. Bloqueo de Lectura y Escritura (READ-WRITE LOCK) 11](#_Toc48897851)

[4.1 Aspectos destacados del diseño de la clase ReadWriteLock 12](#_Toc48897852)

[Bloqueos estáticos 12](#_Toc48897853)

[Métodos de bloqueo 12](#_Toc48897854)

[Liberación de Bloqueo 12](#_Toc48897855)

**Patrones de concurrencia:**

# Introducción

Los patrones de concurrencia tratan con:

* Formas de bloquear el código de clase y un orden de bloqueo de objetos para evitar que se produzcan condiciones de carrera y puntos muertos
* Los detalles de la optimización del acceso a un recurso de la aplicación para mejorar la capacidad de respuesta general de la aplicación.
* Los detalles de la ejecución del método mientras no se cumple una condición previa requerida

|  |  |
| --- | --- |
| Sección crítica | Forma más estricta de monitor. Se utiliza para bloquear el código a nivel de clase para evitar que varios subprocesos ejecuten el código bloqueado incluso en dos ejemplificaciones diferentes de la misma clase. |
| Orden de bloqueo consistente | Recomienda identificar y documentar un orden bien definido de objetos de bloqueo que se debe seguir de manera constante durante el diseño y el desarrollo de una aplicación para eliminar la posibilidad de que se produzca un abrazo mortal. |
| Suspensión protegida | Recomienda que se diseñe un método para suspender su ejecución hasta que el objeto esté en un estado que haga verdadera una condición previa requerida. |
| Bloqueo de lectura y escritura | Recomienda permitir operaciones de lectura simultáneas al tiempo que evita las actualizaciones simultáneas de los valores de un recurso de la aplicación para mejorar la capacidad de respuesta general de la aplicación. |
|  |  |

# Sección Crítica (CRITICAL SECTION)

* Una sección crítica es un segmento de código que debe ser ejecutado por un solo subproceso a la vez para producir los resultados esperados. Cuando se permite que más de un subproceso ejecute este segmento de código, podría producir resultados impredecibles

La siguiente es la lista de similitudes y diferencias entre monitores y secciones críticas:

|  |  |
| --- | --- |
| **Monitor** | **Sección Crítica** |
|  | Una sección crítica es una forma más estricta de monitor |
| Un monitor bloquea un solo objeto | sección crítica requiere un bloqueo en toda una clase de objet |
| **En Java** | |
| La implementación de un Monitor en un método requiere que el método se declare utilizando la palabra clave “sinchronized” | Se puede implementar una sección crítica utilizando la combinación de palabras clave “static” y “sinchronized”. |
| **No se permite** que dos subprocesos ejecuten el código sincronizado **en el mismo objeto** | No se permite que dos subprocesos ejecuten el código **sincronizado ningún objeto de la misma clase.** |
| **Dos subprocesos pueden** ejecutar el mismo código sincronizado en **dos objetos diferentes.** | No se permiten que dos subprocesos ejecuten el código en diferentes objetos de la misma clase (el código está bloqueado a nivel de clase). |

Para el patrón Singleton, se diseñó un registro de mensajes class FileLogger como singleton. La clase FileLogger mantiene un registrador de variables de clase del tipo FileLogger. Esta variable se utiliza para contener el ejemplar de FileLogger singleton. La clase FileLogger ofrece un método a nivel de clase get-FileLogger que puede ser utilizado por diferentes objetos cliente para acceder a la instancia de FileLogger del singleton. Como parte de la implementación del método getFileLogger, FileLogger comprueba si la instancia de singleton ya se ha creado. Si se encuentra que el registrador no está inicializado, se crea una instancia de FileLogger invocando su constructor privado y se asigna a la variable de clase del registrador. Esta implementación del método getFileLogger funciona bien en un entorno de un solo subproceso. En un entorno multiproceso, es posible que dos subprocesos ejecuten simultáneamente el método getFileLogger para ver si la variable de clase registrador es nula y, como resultado, inicializar el registrador dos veces. Esto significa que el constructor privado FileLogger se invoca dos veces.

Inicializar la variable del registrador dos veces en este ejemplo no da como resultado un error. Esto se debe a que el constructor privado FileLogger no realiza ninguna inicialización crítica compleja. Por el contrario, si el método del constructor singleton ejecuta operaciones como abrir una conexión de socket en un puerto en particular, ejecutar el constructor dos veces podría resultar en un error.

Para mejorar el diseño de la clase FileLogger para que sea adecuada para su uso en entornos multiproceso tenemos 2 enfoques:

## Sección crítica con Fiabilidad de hilos (thread-safe)

Esto implica convertir el método getFileLogger en una sección crítica para que solo un subproceso pueda ejecutarlo en un momento dado. Esto se puede lograr simplemente declarando que el método de nivel de clase getFileLogger como “synchronized”.

Este simple cambio convierte el método getFileLogger en “**CriticalSection”** y garantiza que nunca dos subprocesos ejecuten el método getFileLogger al mismo tiempo. Esto elimina por completo la posibilidad de que el constructor FileLogger sea invocado más de una vez dentro del método get-FileLogger.

## Sección crític con Inicialización temprana estática (Static early Initialization )

Los métodos de sincronización **tienen un efecto significativo en el rendimiento** general de la aplicación. En general**, los métodos sincronizados se ejecutan mucho más lento**, hasta 100 veces más lento que sus contrapartes no sincronizados. Como alternativa a declarar sincronizado el método getFileLogger, la variable del registrador se puede inicializar de manera temprana.

Esto evita la verificación e inicialización dentro del método getFile-Logger. Como resultado, getFileLogger se vuelve seguro para subprocesos sin tener que declararlo como sincronizado.

# Orden de Bloqueo Consistente (CONSISTENT LOCK ORDER)

Durante la discusión anterior sobre los patrones Monitor y la Sección Crítica, hemos visto que cuando la palabra clave sincronizada se usa para asegurar la ejecución de un solo subproceso de un bloque de código, un subproceso debe esperar mientras intenta adquirir el bloqueo asociado con el objeto especificado.

Considere un escenario en el que dos subprocesos se bloquean en dos objetos diferentes y cada uno está esperando un bloqueo en el objeto que está bloqueado por el otro subproceso. Ambos subprocesos estarán esperando eternamente y se dice que están en un estado de abrazo mortal.

En términos de implementación, este tipo de situación ocurre con mayor frecuencia debido a un orden inconsistente de bloqueo de objetos. Consideremos el segmento de código del Listado 42.1 para ilustrar cómo el bloqueo inconsistente en un entorno multiproceso puede causar un abrazo mortal. Considere un escenario en el que:

Considere un escenario donde:

* Dos subprocesos, A y B, invocan métodos simultáneamente, Method\_A y Method\_B, respectivamente en el mismo objeto SomeClass.
* El subproceso A adquiere un bloqueo en el objetoA y el subproceso B adquiere un bloqueo en el objetoB al mismo tiempo. En este punto, cada uno de los subprocesos espera un bloqueo en el objeto bloqueado por el otro subproceso y esto coloca al subproceso A y al subproceso B en una condición de abrazo mortal.

Para abordar estos problemas de interbloqueo, el patrón de orden de bloqueo coherente recomienda diseñar un orden de bloqueo de objetos (donde los objetos de una clase particular deben bloquearse antes de bloquear otros ejemplares de clase) para que se siga de forma coherente en una aplicación. Lo anterior puede eliminar el problema de interbloqueo asociado con el bloque de código de ejemplo. En otras palabras, al asegurarse de que los objetos estén bloqueados en un orden coherente en toda la aplicación, se puede solucionar el problema de los interbloqueos.

El bloque de código de ejemplo del Listado 42.1 puede modificarse para que los objetos ClassA se bloqueen antes de bloquear los objetos ClassB.

Este tipo de orden de bloqueo de objetos basado en el tipo de clase no funciona cuando los objetos que se van a bloquear son instancias de la misma clase. Puede que sea necesario un algoritmo más sofisticado para decidir el orden de bloqueo del objeto. El siguiente ejemplo ilustra uno de esos mecanismos.

Construyamos una clase de utilidad que ofrezca la funcionalidad de mover el contenido de un directorio a un directorio diferente en el sistema de archivos. Creemos un directorio de clases, cuyas instancias se pueden usar para representar directorios en el sistema de archivos.

La clase de utilidad FileSysUtil en su forma más simple se puede diseñar con un método para mover el contenido entre directorios.

La clase de utilidad FileSystem en su forma más simple se puede diseñar con un método para mover el contenido entre directorios.

Para mover el contenido de un directorio a otro, un objeto cliente o hilo necesita:

1. Crear objetos Directory correspondientes a los directorios de origen y destino.

2. Invocar el método moveContents pasando los objetos Directory creados en el Paso 1.

Como parte de su implementación del método moveContents, File-SysUtil bloquea los objetos Directory que representan los directorios de origen y destino en secuencia antes de mover realmente el contenido del directorio. Esto es para evitar que los subprocesos cambien o eliminen los directorios de origen o destino mientras el subproceso actual está en el proceso de mover el contenido del directorio de origen al directorio de destino. Para simplificar, la aplicación de ejemplo muestra un mensaje apropiado en lugar de mover realmente el contenido del directorio de origen.

Supongamos que existen dos directorios, dir1 y dir2, en el sistema de archivos. Para mover el contenido de dir1 a dir2, un hilo (p. Ej., Thread\_A) necesita crear dos objetos de directorio, objDir\_1 y objDir\_2, correspondientes a dir1 y dir2, respectivamente, y pasarlos como argumentos al método moveContents.

Mientras ejecuta el método moveContents, Thread\_A intenta adquirir bloqueos en objDir\_1 y objDir\_2 en secuencia. Al mismo tiempo, un hilo diferente (por ejemplo, Thread\_B) invoca el método moveContents en el mismo objeto FileSysUtil para mover el contenido de dir2 a dir1. Utilizando los mismos objetos Directory utilizados por Thread\_A, Thread\_B realiza una llamada de la siguiente manera: moveContents(objDir\_2, objDir\_1);

Similar a Thread\_A, mientras ejecuta el método moveContents, Thread\_B también intenta adquirir bloqueos en objDir\_1 y objDir\_2 pero en el orden inverso.

Si Thread\_A y Thread\_B adquieren bloqueos al mismo tiempo en objDir\_1 y objDir\_2, respectivamente, entonces cada subproceso continúa esperando un bloqueo en el objeto Directory bloqueado por el otro subproceso y esto causa un interbloqueo. Porque tanto objDir\_1 como objDir\_2 son del mismo tipo de clase Directory , definir un orden de bloqueo de objetos basado en el tipo de clase no funciona en este caso. Como alternativa, se puede utilizar el método hashCode de Java integrado para definir un orden de bloqueo de los objetos de directorio. El método hashCode se define en la clase superior java.lang.Object y es heredado por todas las clases en Java.

El método hashCode devuelve el ID único o el código hash asociado con un objeto. Se puede definir un esquema de bloqueo de objetos en función de algún tipo de orden de los códigos hash de los objetos a bloquear. Para eliminar la posibilidad de una situación de interbloqueo, se puede modificar el método moveContents para que los objetos que representan los directorios de origen y destino estén bloqueados. el orden ascendente de sus códigos hash asociados. Esto asegura que los objetos de directorio siempre estén bloqueados en el mismo orden, incluso si se pasan al método moveContents por dos subprocesos diferentes en diferente orden.

Con este cambio en su lugar, cuando dos subprocesos invocan el método moveContents al mismo tiempo para mover el contenido de dos directorios diferentes en direcciones opuestas, solo se otorga bloqueo a un subproceso en el primer objeto Director que se bloqueará. El segundo subproceso simplemente espera el bloqueo en el propio objeto firstDirectory. No surge la posibilidad de que el segundo hilo bloquee el objeto secondDirectory mientras que el primer hilo bloquea el primer objeto Directory.

La aplicación de ejemplo utiliza un mecanismo simple para definir el orden de bloqueo de los objetos de directorio. En el caso de una aplicación del mundo real, es necesario identificar y documentar una orden de bloqueo que sea adecuada para la aplicación. Este orden de bloqueo se puede seguir de forma coherente durante el diseño y el desarrollo de la aplicación.

# Suspensión protegida (GUARDED SUSPENSION)

En general, cada método de un objeto está diseñado para ejecutar una tarea específica. A veces, cuando se invoca un método en un objeto, es posible que el objeto deba estar en un estado determinado, lo cual es lógicamente necesario para que el método lleve a cabo la acción para la que está diseñado. En tales casos, el patrón de suspensión vigilada sugiere suspender la ejecución de método hasta que dicha condición previa se cumpla. En otras palabras, el requisito de que el objeto esté en **un estado particular se convierte en una condición previa para que el método ejecute su implementación de la tarea prevista**.

Cada clase en Java hereda los métodos de esperar, notificar y notificar a todos de la clase base java.lang.Object. Cuando un hilo invoca el método de espera de un objeto:

* Hace que el hilo libere el bloqueo de sincronización que mantiene en el objeto.
* El hilo permanece en estado de espera hasta que se le notifica que regrese mediante el método notificar o notificar a todos.

Con estos métodos build-in de espera, notificar y notificar a todos, el patrón de suspensión protegida se puede implementar en Java.

La estructura genérica de una clase Java cuando se aplica el patrón de suspensión protegida mediante los métodos incorporados esperar, notificar y notificar a todos se representa en el listado 43.1.

La clase SomeClass consta de dos métodos sincronizados: método — guarded-

Method y alterObjectStateMethod.

* GuardedMethod representa un método que requiere algún tipo de condición previa para convertirse en verdadero antes de continuar con su ejecución. Por lo tanto, verifica si la condición previa es verdadera “true” y mientras la condición previa no sea verdadera “true”, espera usando el método de espera.
* El método alterObjectStateMethod permite que diferentes objetos de cliente (subprocesos) cambien el estado de una instancia de SomeClass. Esto, a su vez, podría resultar en que la condición previa requerida se hiciera “true”.

Una vez que se cambia el estado del objeto, este método notifica a cualquier hilo en espera que esté esperando dentro del método protegido mediante el método de notificación. Si el cambio en el estado del objeto hace que la condición previa sea verdadera, el hilo de espera se reanuda con la ejecución del método guardado. De lo contrario, sigue esperando hasta que la precondition se convierta en verdadera. Tanto el método guardedMethod como el método alterObjectStateMethod están diseñados como métodos sincronizados para evitar condiciones de carrera (ejecución de acciones en una secuencia no esperada) en un entorno multiproceso.

Ejemplo

Construyamos una aplicación para simular el mecanismo de estacionamiento en un gimnasio. Un socio puede estacionar su automóvil si hay una plaza de estacionamiento vacía. Si no hay un espacio de estacionamiento vacío, un miembro debe esperar hasta que uno de los espacios de estacionamiento esté disponible.

Se puede diseñar una representación simple para el estacionamiento en la forma de la clase Estacionamiento que se muestra en el Listado 43.2.

El estacionamiento mantiene el número total de automóviles estacionados actualmente en su variable de instancia total de automóviles estacionados. Esto constituye el estado de un objeto ParkingLot.

La existencia de un espacio vacío es la condición previa para que un miembro proceda a estacionar su automóvil. Se puede ver que el método de estacionamiento primero verifica si se cumple esta condición previa. Si el número de miembros estacionados actualmente es mayor o igual al número total de espacios disponibles, se puede inferir que no hay un espacio de estacionamiento vacío disponible y el miembro debe esperar hasta que no exista esta condición.

Cuando un miembro abandona el estacionamiento, el número total de miembros actualmente estacionados se reduce y el método de salida leave notifica al azar a uno de los hilos en espera. Una vez que se recibe la notificación, el hilo notificado intenta bloquear el objeto. Una vez que se obtiene el bloqueo, verifica si se cumple la condición previa al volver a ingresar al ciclo while. Si se cumple la condición previa, se procede con la acción de estacionamiento.

El código de ejemplo simplemente muestra un mensaje e incrementa el número total de automóviles estacionados actualmente. Verificar la condición previa por parte del hilo notificado puede parecer redundante, pero es necesario en un entorno multiproceso. Esto se debe a la posibilidad de que un subproceso diferente altere el estado del objeto entre el momento en que el subproceso en espera intenta obtener un bloqueo en el objeto y el momento en que lo obtiene, de modo que la condición previa se vuelve falsa.

Uso **wait() and notify() en el diseño de clases ParkingLot** :

* El método park utiliza el método incorporado java.lang.Object wait () para mantener un hilo de miembro en espera mientras la condición previa no es cierta. Cuando se llama al método wait (), el subproceso actualmente ejecutado (en este caso un Miembro) se coloca en la cola de espera y se libera su bloqueo en el objeto ParkingLot (tenía un bloqueo en el objeto ParkingLot porque el parque está sincronizado). El siguiente hilo Member es libre de ingresar al método park y verifica si totalParkedCars> = MAX\_CAPACITY, que si es verdadero, también se coloca en la cola de espera.
* El método Leave utiliza el método de notificación java.lang.Object incorporado para notificar al azar a un único hilo en espera. La elección del hilo queda a discreción de la implementación específica de JVM. El hilo notificado relanza un bloqueo en el objeto ParkingLot y vuelve a ejecutarse en el método park donde se invocó el método wait (). Usando el método notifyAll integrado (el método leave también podría implementarse para notificar todos los hilos en espera a la vez). A continuación, los subprocesos en espera compiten para el bloqueo del objeto ParkingLot. Cualquier hilo que obtenga el bloqueo continúa la ejecución en el método park donde se llamó al método wait ().

La representación de un miembro se puede diseñar como un subproceso de Java (Listado 43.3) para facilitar la simulación de más de un miembro que busca estacionar sus autos al mismo tiempo.

Diseñemos un controlador de prueba GSTest para utilizar la clase Member para simular un escenario del mundo real en el que varios miembros intentan aparcar sus coches al mismo tiempo.

*Usando la analogía del ejemplo reflejado en la documentación y el código fuente  que se le anexa en esta sección, resuelva el problema de acceso de vehículos a un taller de servicios de mantenimiento periódicos que cuenta con una capacidad de atención máxima de cinco vehículos con un tiempo promedio de demora por revisión (simulado) de 800 milisegundos. Suponga que el taller al abrir ya tiene en cola una cantidad de ocho vehículos. Elabore el programa y los modelos funcionales, estructurales y dinámicos acordes a los trabajados durante el semestre académico. Se le podrá solicitar que explique el ejercicio y su conceptualización subyacente sobre la técnica de control de concurrencia utilizada.*

# Bloqueo de Lectura y Escritura (READ-WRITE LOCK)

Durante la discusión anterior sobre los patrones Monitor y la Sección Crítica, vimos que cuando varios subprocesos en una aplicación acceden simultáneamente a un recurso, podría resultar en un comportamiento impredecible. Por lo tanto, el recurso debe estar protegido para que solo un subproceso a la vez tenga acceso al recurso.

Aunque esto puede ser necesario en la mayoría de los casos, puede provocar una sobrecarga de CPU no deseada cuando algunos de los subprocesos que acceden al recurso están interesados ​​solo en leer los valores o el estado del recurso, pero no en cambiarlo. Debido a que la operación de lectura no altera los valores del recurso, se puede permitir con seguridad que varios subprocesos accedan al recurso al mismo tiempo si todos estos subprocesos solo están interesados ​​en leer los valores del recurso. Este tipo de diseño mejora la capacidad de respuesta general de la aplicación con una sobrecarga de CPU reducida.

Eso significa que cuando un hilo obtiene un bloqueo para simplemente leer los valores de un recurso, no debe evitar que otros hilos accedan al recurso para leer sus valores(se comparte un bloqueo de lectura).

Permitir que más de un subproceso actualice los valores de un recurso también podría generar resultados impredecibles (Si un subproceso puede leer los datos de un recurso mientras otro subproceso está actualizando el mismo recurso leería valores inconsistentes).

Para eliminar los problemas de concurrencia, cuando un hilo de este tipo necesita acceder al recurso para actualizar sus valores, debe obtener un bloqueo de escritura en el objeto que representa el recurso.

**Bloqueo de Lectura:**

* Cuando un hilo obtiene un bloqueo solo para leer los valores de un recurso, no debe evitar que otros hilos accedan al recurso para leer sus valores (se comparte un bloqueo de lectura).
* Se debe emitir un bloqueo de lectura si no hay un bloqueo de escritura emitido actualmente y no hay subprocesos esperando el bloqueo de escritura.

**Bloqueo de Escritura:**

* El bloqueo de escritura es un bloqueo exclusivo del objeto y evita que todos los demás subprocesos accedan al recurso al mismo tiempo.
* Si se solicita un bloqueo de lectura y escritura en un objeto al mismo tiempo, la solicitud de bloqueo de escritura debe concederse primero.
* Se debe emitir un bloqueo de escritura si no se ha emitido un bloqueo (lectura o escritura) en el objeto.

En Java, no hay una función disponible para implementar bloqueos de lectura y escritura, pero se puede construir una clase personalizada (Listado 44.1) con la responsabilidad de emitir bloqueos de lectura y escritura en un objeto a diferentes subprocesos de una aplicación.

## Aspectos destacados del diseño de la clase ReadWriteLock

### Bloqueos estáticos

ReadWriteLock mantiene diferentes estadísticas de bloqueo en un conjunto de variables de instancia de la siguiente manera:

* totalReadLocksGiven: para almacenar el número de bloqueos de lectura ya emitidos en el objeto.
* writeLockIssued: para indicar si se ha emitido un bloqueo de escritura o no.
* threadsWaitingForWriteLocks: para realizar un seguimiento del número de subprocesos que actualmente esperan un bloqueo de escritura.

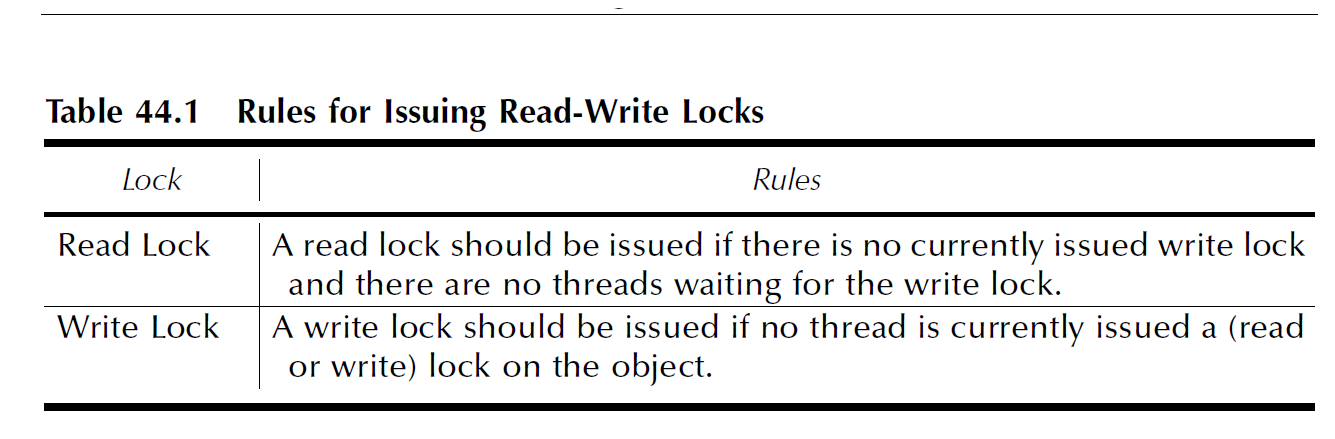
Estos valores, a su vez, son utilizados por los métodos de emisión de bloqueos: getReadLock y getWriteLock.

### Métodos de bloqueo

ReadWriteLock ofrece dos métodos, getReadLock y getWrite-Lock, que los objetos del cliente pueden usar para obtener bloqueos de lectura y escritura en un objeto, respectivamente. Como parte de la implementación de estos dos métodos, el ReadWriteLock emite bloqueos de lectura y escritura según las reglas enumeradas en la Tabla 44.1.

### Liberación de Bloqueo

El objeto cliente que actualmente tiene un bloqueo de lectura-escritura puede liberar el bloqueo invocando el método done. El método done actualiza las estadísticas de bloqueo apropiadas y permite que el bloqueo se emita a cualquier hilo en espera según las reglas enumeradas en la Tabla 44.1. La clase ReadWriteLock es una implementación genérica para emitir bloqueos de lectura-escritura y se puede usar fácilmente en cualquier aplicación.



Ejemplo

Aplicando el patrón de bloqueo de lectura-escritura, diseñemos una aplicación que permita a los miembros de una biblioteca:

* Ver detalles de diferentes elementos de la biblioteca.
* Echa un vistazo a un artículo si está disponible actualmente.

La aplicación debe garantizar que varios miembros puedan ver el estado de un artículo al mismo tiempo, pero que solo un miembro pueda retirar un artículo a la vez.

El diseño general de la aplicación se vuelve mucho más simple utilizando la clase ReadWrite-Lock diseñada anteriormente. La representación de un elemento de la biblioteca puede diseñarse en forma de una clase de elemento (Listado 44.2 **Item Class**) con métodos que permitan a los miembros verificar el estado de un elemento y registrar o retirar un elemento. Porque la verificación de estado de un elemento no implica cambia a su estado, el método getStatus adquiere un bloqueo de lectura.

* Esto permite que más de un subproceso invoque el método getStatus para verificar el estado de un artículo.
* Por el contrario, los métodos checkIn y checkOut implican cambios en el estado del artículo y, por lo tanto, adquieren un bloqueo de escritura antes de cambiar el estado del artículo. Esto asegura que solo un subproceso se permite alterar el estado del artículo aunque más de un hilo invoca el método checkIn / checkOut al mismo tiempo.

La clase Item hace uso de los servicios de un objeto ReadWriteLock para adquirir un bloqueo apropiado.

La representación de una transacción de un miembro se puede diseñar como un subproceso “Thread” de Java (Listado 44.3) representando un escenario del mundo real donde diferentes miembros acceden a un elemento simultáneamente. La clase MemberTransaction se configura con una operación para verificar el estado de un elemento o para registrar o verificar un elemento cuando se crea un ejemplar.

Para simular un escenario del mundo real, se diseña un programa de prueba RWTest para crear múltiples objetos MemberTransaction para realizar diferentes operaciones para leer el estado de un artículo o registrar o retirar un artículo.

Cuando se ejecuta RWTest, se mostrará el orden en que se emiten los diferentes bloqueos de lectura y escritura.