Item 78 trong "Effective Java" của Joshua Bloch là "Synchronize access to shared mutable data". Trong item này, Bloch tập trung vào việc sử dụng đồng bộ hóa để bảo vệ dữ liệu có thể thay đổi (mutable data) được chia sẻ giữa các luồng thực thi trong môi trường đa luồng (multithreaded environment).

**Lý do sử dụng đồng bộ hóa:**

1. **Bảo vệ tính nhất quán**: Khi nhiều luồng cùng truy cập và thay đổi dữ liệu, việc không sử dụng đồng bộ hóa có thể dẫn đến tình trạng đọc/ghi không nhất quán, khiến dữ liệu trở nên không đáng tin cậy.
2. **Bảo vệ tính nhất quán qua thời gian (Temporal consistency)**: Đồng bộ hóa cũng đảm bảo rằng dữ liệu được đọc sau khi được ghi sẽ phản ánh các thay đổi mới nhất.

**Cách sử dụng đồng bộ hóa:**

1. **Sử dụng từ khóa synchronized**: Trong Java, từ khóa **synchronized** có thể được sử dụng để đồng bộ hóa một phương thức hoặc một khối mã. Khi một luồng nhập vào một khối mã được đồng bộ hóa, nó phải lấy khóa tương ứng trước khi thực thi khối mã đó.
2. **Sử dụng Locks (khóa)**: Locks cung cấp một cách linh hoạt hơn để đồng bộ hóa, với các phương thức như **lock()** và **unlock()**, cho phép kiểm soát rõ ràng hơn về thời gian một luồng có thể giữ một khóa.
3. **Sử dụng Atomic variables**: Atomic variables cung cấp các phép toán atomic như **getAndSet()** và **compareAndSet()**, loại bỏ nhu cầu sử dụng đồng bộ hóa bằng từ khóa **synchronized** cho các phép toán đơn giản trên dữ liệu nguyên tố.

///////////////////////////////////////////////

Item 79 trong "Effective Java" của Joshua Bloch là "Avoid excessive synchronization". Trong item này, Bloch cảnh báo về việc tránh sử dụng quá nhiều đồng bộ hóa trong mã Java vì nó có thể gây ra hiệu suất kém và tăng nguy cơ xảy ra các vấn đề như deadlock.

**Lí do tránh sử dụng quá nhiều đồng bộ hóa:**

1. **Hiệu suất kém**: Khi một phần lớn mã được đồng bộ hóa, các luồng khác phải chờ đợi để truy cập vào nó, dẫn đến hiệu suất giảm do cản trở và cạnh tranh giữa các luồng.
2. **Nguy cơ xảy ra deadlock**: Deadlock xảy ra khi hai hoặc nhiều luồng đợi lẫn nhau để giải phóng các khóa mà chúng cần. Sử dụng quá nhiều đồng bộ hóa tăng nguy cơ xảy ra deadlock vì mỗi luồng cố gắng giữ một hoặc nhiều khóa trong khi đợi để thực thi mã đồng bộ hóa.

**Cách tránh sử dụng quá nhiều đồng bộ hóa:**

1. **Giảm thiểu phạm vi đồng bộ hóa**: Thay vì đồng bộ hóa toàn bộ phương thức hoặc khối mã, hãy giảm thiểu phạm vi đồng bộ hóa càng nhiều càng tốt, chỉ đồng bộ hóa những phần của mã cần thiết để đảm bảo tính nhất quán và an toàn.
2. **Sử dụng cách tiếp cận lazy initialization hoặc copy-on-write**: Trong một số trường hợp, có thể sử dụng cách tiếp cận như lazy initialization hoặc copy-on-write để tránh việc phải đồng bộ hóa.
3. **Sử dụng cấu trúc dữ liệu không đồng bộ (non-blocking data structures)**: Sử dụng các cấu trúc dữ liệu không đồng bộ như ConcurrentHashMap có thể giúp tránh cần phải sử dụng đồng bộ hóa và giảm nguy cơ xảy ra deadlock.

//////////////////////////////////////

Item 80 trong "Effective Java" của Joshua Bloch là "Prefer executors, tasks, and streams to threads". Trong item này, Bloch khuyến nghị việc sử dụng các Executor, Task và Streams thay vì quản lý các luồng (threads) trực tiếp.

**Lí do ưu tiên sử dụng Executors, Tasks và Streams:**

1. **Tăng cường sự trừu tượng và mức độ tự động hóa**: Executors và Tasks cung cấp một cơ chế trừu tượng hóa cho việc quản lý và thực thi các nhiệm vụ (tasks). Thay vì quản lý các luồng một cách trực tiếp, chúng ta chỉ cần định nghĩa các nhiệm vụ và giao chúng cho Executors để thực thi.
2. **Tăng tính mở rộng**: Executors cho phép chúng ta dễ dàng điều chỉnh và mở rộng số lượng luồng được sử dụng trong quá trình thực thi nhiệm vụ.
3. **Tăng tính linh hoạt và tái sử dụng**: Executors và Tasks cho phép chúng ta tách biệt việc xác định nhiệm vụ và việc thực thi, giúp tạo ra mã linh hoạt và dễ tái sử dụng.
4. **Stream API giúp xử lý dữ liệu một cách tự nhiên và linh hoạt**: Stream API cung cấp một cách tiếp cận tự nhiên và linh hoạt cho việc xử lý dữ liệu trong Java, giúp chúng ta viết mã ngắn gọn và dễ hiểu.

**Cách sử dụng Executors, Tasks và Streams:**

1. **Executors và Tasks**:
   * Sử dụng Executors.newCachedThreadPool() cho các nhiệm vụ ngắn gọn hoặc không biết trước số lượng nhiệm vụ.
   * Sử dụng Executors.newFixedThreadPool(n) để xác định số lượng cố định của các luồng trong ThreadPool.
   * Sử dụng Executors.newSingleThreadExecutor() cho các nhiệm vụ cần đảm bảo tuân thủ thứ tự thực thi.
2. **Streams**:
   * Sử dụng Stream API cho các tác vụ xử lý dữ liệu, như lọc, ánh xạ, giảm số chiều, và tổng hợp.

/////////////////////////////////

Để trình bày chi tiết hơn về việc sử dụng các tiện ích đồng thời (**concurrency utilities**) thay vì **wait** và **notify** trong Java, chúng ta có thể xem xét một số điểm chính và cung cấp ví dụ cụ thể.

1. **An toàn và Linh hoạt**: Các tiện ích đồng thời trong **java.util.concurrent** cung cấp các cơ chế an toàn và linh hoạt hơn cho việc quản lý đồng bộ hóa. Chúng giúp tránh được các vấn đề như deadlock và race conditions.
2. **Locks và Conditions**: Thay vì sử dụng **synchronized** và **wait/notify**, chúng ta có thể sử dụng **ReentrantLock** và **Condition** từ gói **java.util.concurrent.locks**. Điều này cung cấp khả năng kiểm soát độc quyền truy cập và đợi/giảm thiểu một cách an toàn và linh hoạt.

Ví dụ:

java

Sao chép mã

ReentrantLock lock = new ReentrantLock(); Condition condition = lock.newCondition(); // Thread A lock.lock(); try { while (!conditionIsMet()) { condition.await(); } // Thực hiện hành động khi điều kiện được đáp ứng } finally { lock.unlock(); } // Thread B lock.lock(); try { // Cập nhật điều kiện condition.signalAll(); } finally { lock.unlock(); }

1. **Executors và ThreadPool**: Thay vì tạo và quản lý các luồng thủ công, chúng ta có thể sử dụng **ExecutorService** để quản lý các tác vụ và luồng một cách hiệu quả. Điều này giúp tránh được tạo ra quá nhiều luồng và tiêu tốn tài nguyên hệ thống một cách không cần thiết.

Ví dụ:

java

Sao chép mã

ExecutorService executor = Executors.newFixedThreadPool(5); for (int i = 0; i < 10; i++) { executor.submit(() -> { // Thực hiện công việc đồng thời }); } executor.shutdown();

1. **Các Cấu Trúc Dữ Liệu Đồng Thời**: **java.util.concurrent** cũng cung cấp các cấu trúc dữ liệu đồng thời như **ConcurrentHashMap**, **ConcurrentLinkedQueue** và **CopyOnWriteArrayList** giúp đảm bảo sự an toàn khi truy cập từ nhiều luồng.

Ví dụ:

java

Sao chép mã

Map<Integer, String> concurrentMap = new ConcurrentHashMap<>(); concurrentMap.put(1, "A"); concurrentMap.put(2, "B");

Tóm lại, việc sử dụng các tiện ích đồng thời từ package **java.util.concurrent** thay vì **wait** và **notify** giúp làm cho mã của bạn dễ đọc hơn, ít lỗi hơn và hiệu quả hơn trong việc quản lý đồng bộ hóa trong Java.

//////////////////////////////