

6.6 Mẫu thiết kế SINGLETON

6.6.1 Mở đầu

Đôi khi rất quan trọng đối với một số lớp là chỉ có chính xác một khởi tạo. Có rất nhiều đối tượng mà chúng ta chỉ cần một khởi tạo của chúng và nếu chúng ta khởi tạo nhiều lần, chúng ta sẽ gặp nhiều vấn đề như hành vi chương trình không đúng, lạm dụng tài nguyên hoặc các kết quả không tương thích.

Bạn có thể yêu cầu chỉ có một đối tượng cho một lớp, chẳng hạn, khi bạn tạo một ngữ cảnh xác định của một ứng dụng hoặc một bể luồng quản trị được, thiết lập đăng ký, một driver kết nối với một thiết bị đầu vào và đầu ra. Nhiều hơn một đối tượng kiểu này rõ ràng sẽ sinh ra sự không tương thích cho hệ thống của bạn.

Mẫu Singleton đảm bảo rằng, một lớp chỉ có một khởi tạo, và cung cấp một điểm truy cập tổng thể đến nó. Tuy nhiên, mặc dù mẫu Singleton là đơn giản nhất theo sơ đồ lớp, vì ở đây chỉ có một lớp duy nhất, cài đặt của nó là hơi rắc rối.

Trong bài này chúng ta sẽ thử các cách khác nhau để tạo nên một đối tượng duy nhất cho một lớp và sẽ nhìn thấy cách này tốt hơn cách kia như thế nào.



6.6.2 Tạo một lớp sử dụng mẫu Singleton như thế nào.

Có nhiều cách để tạo kiểu lớp như vậy, nhưng chúng ta sẽ thấy một cách sẽ tốt hơn cách khác như thế nào.

Bắt đầu bởi cách đơn giản nhất.

Cái gì sẽ xảy ra, nếu ta cung cấp một biến tổng thể mà tạo nên một đối tượng truy cập được. Chẳng hạn:

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
    public static SingletonEager sc = new SingletonEager();
}
```

Như chúng ta biết, ở đây chỉ có một bản sao của biến tĩnh của một lớp, chúng ta có thể áp dụng nó. Code của client sử dụng biến `sc` tĩnh này có vẻ là tốt. Nhưng, nếu client sử dụng thao tác `new SingletonEager()`, ở đây có thể có khởi tạo mới của lớp này.

Để không cho phép lớp có thể được khởi tạo ngoài lớp, ta sẽ làm cho hàm tạo là *private*.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
    public static SingletonEager sc = new SingletonEager();
    private SingletonEager() {}
}
```

Bằng cách giữ hàm tạo là *private*, không lớp nào có thể khởi tạo được lớp này. Chỉ có một cách lấy được đối tượng của lớp là sử dụng biến tĩnh *sc* mà đảm bảo chỉ có một đối tượng ở đây.

Nhưng như chúng ta biết, cung cấp truy cập trực tiếp đến thành viên của lớp không phải là ý tưởng hay. Chúng ta sẽ cung cấp một phương thức mà biến *sc* sẽ được nhận không trực tiếp.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
    private static SingletonEager sc = new SingletonEager();
    private SingletonEager() {}
    public static SingletonEager getInstance() {
        return sc;
    }
}
```

Như vậy, đây là lớp singleton mà đảm bảo chỉ một đối tượng của một lớp được tạo và ngay cả nếu có một số yêu cầu, thì trả về cùng một đối tượng đã được khởi tạo.

Có một vấn đề với cách tiếp cận này là, đối tượng sẽ được tạo ngay khi lớp được tải vào máy ảo JVM. Nếu đối tượng không được yêu cầu, thì có thể đối tượng là không có ích bên trong bộ nhớ.

Luôn luôn là cách tiếp cận tốt nếu đối tượng được tạo khi nó được yêu cầu. Vì vậy chúng ta sẽ tạo đối tượng ở lời gọi đầu tiên và sẽ trả về cùng đối tượng đó trong các lời gọi tiếp theo.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazy {

    private static SingletonLazy sc = null;
    private SingletonLazy() {}
    public static SingletonLazy getInstance() {
        if(sc==null) {
            sc = new SingletonLazy();
        }
        return sc;
    }
}
```

Trong phương thức *getInstance()*, ta kiểm tra nếu biến *sc* là *null*, thì ta sẽ khởi tạo đối tượng và trả về nó. Như vậy trong lần gọi đầu khi *sc* có thể là *null*, đối tượng được tạo và trong các lời gọi tiếp theo sẽ trả về cùng đối tượng đó.

Điều này trông có vẻ tốt? Tuy nhiên code này sẽ lỗi trong môi trường đa luồng. Tưởng tượng rằng, có hai luồng song song truy cập đến lớp, một luồng t1 đưa ra lời gọi đầu tiên đến phương thức *getInstance()*, nó kiểm tra nếu biến tĩnh *sc* là null và sau đó bị ngắt vì một lý do nào đó. Luồng thứ hai t2 gọi phương thức *getInstance()*, qua lệnh kiểm tra *if* thành công và khởi tạo đối tượng. Sau đó luồng t1 quay trở lại và nó cũng tạo đối tượng mới. Trong thời điểm này, có thể có hai đối tượng của lớp đó.

Để tránh điều này, chúng ta sẽ sử dụng từ khóa *synchronized* cho phương thức *getInstance()*. Bằng cách này, chúng ta buộc mỗi luồng phải chờ đến lượt nó trước khi sử dụng phương thức. Vì vậy không có hai luồng sẽ sử dụng phương thức cùng một thời điểm. Phương thức đồng bộ được trả giá, là nó giảm hiệu năng, nhưng nếu lời gọi *getInstance()* sẽ không phát sinh thêm gánh nặng cho ứng dụng, thì không sao. Phương án khác là chuyển đến cách tiếp cận khởi tạo sớm như đã nêu trong ví dụ trước.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazyMultithreaded {

    private static SingletonLazyMultithreaded sc = null;
    private SingletonLazyMultithreaded() {}
    public static synchronized SingletonLazyMultithreaded getInstance() {
        if(sc==null) {
            sc = new SingletonLazyMultithreaded();
        }
        return sc;
    }
}
```

Nhưng nếu bạn muốn sử dụng đồng bộ, có một kỹ thuật khác là “khóa kiểm tra hai lần” để giảm việc sử dụng đồng bộ. Với “khóa kiểm tra hai lần”, trước hết ta kiểm tra xem một khởi tạo đã được tạo chưa, nếu chưa, thì ta đồng bộ. Cách này, ta chỉ sử dụng đồng bộ lần đầu.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazyDoubleCheck {

    private volatile static SingletonLazyDoubleCheck sc = null;
    private SingletonLazyDoubleCheck() {}
    public static SingletonLazyDoubleCheck getInstance() {
        if(sc==null) {
            synchronized(SingletonLazyDoubleCheck.class) {
                if(sc==null) {
                    sc = new SingletonLazyDoubleCheck();
                }
            }
        }
        return sc;
    }
}
```

Ngoài ra còn một số cách khác mà bề mẫu singleton.

- Nếu lớp là *Serializable*
- Nếu nó là sử dụng được clone

- Và cũng, nếu lớp đó được tải bằng bộ tải đa lớp
- Ví dụ sau chỉ ra bạn có thể bảo vệ lớp của bạn khỏi việc khởi tạo hơn một lần như thế nào

```
package com.javacodegeeks.patterns.singletonpattern;

import java.io.ObjectStreamException;
import java.io.Serializable;

public class Singleton implements Serializable{

    private static final long serialVersionUID = -1093810940935189395L;
    private static Singleton sc = new Singleton();
    private Singleton(){

        if(sc!=null){
            throw new IllegalStateException("Already created.");
        }
    }
    public static Singleton getInstance(){
        return sc;
    }

    private Object readResolve() throws ObjectStreamException{
        return sc;
    }

    private Object writeReplace() throws ObjectStreamException{
        return sc;
    }

    public Object clone() throws CloneNotSupportedException{
        throw new CloneNotSupportedException("Singleton, cannot be cloned");
    }

    private static Class getClass(String classname) throws ClassNotFoundException {
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        if(classLoader == null)
            classLoader = Singleton.class.getClassLoader();
        return (classLoader.loadClass(classname));
    }
}
```

- Cài đặt các *readResolve()* và *writeReplace()* trong lớp *singleton* của bạn và trả về cùng một đối tượng qua chúng.
- Bạn cần cài đặt phương thức *clone()* và đưa ra ngoại lệ sao cho *singleton* không thể được *clone*.
- Để ngăn cản *singleton* được khởi tạo từ các bộ tải lớp khác nhau, bạn có thể cài đặt phương thức *getClass()*. Phương thức *getClass()* trên đây liên kết với bộ tải với luồng hiện tại, nếu bộ tải là *null*, phương thức sẽ sử dụng cùng một bộ tải mà tải lớp *singleton*.

Mặc dù chúng ta có thể sử dụng mọi kỹ thuật trên, có một cách đơn giản và dễ dàng để tạo lớp singleton. Như JDK 1.5, bạn có thể tạo lớp singleton sử dụng enums. Hằng số Enum là tĩnh ẩn và là không thay đổi (*final*) và bạn không thể thay đổi giá trị của chúng khi được tạo ra.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEnum {

    public enum SingleEnum{
        SINGLETON_ENUM;
    }
}
```

Bạn sẽ có lỗi trong thời gian dịch khi bạn thử khởi tạo một đối tượng *Enum*. *Enum* sẽ được tải tĩnh, như vậy nó là an toàn luồng. Phương thức *clone* trong *Enum* là *final* nên đảm bảo rằng các hằng số *Enum* không bao giờ được *clone*. Enum là kế thừa từ *serializable*, cơ chế tuần tự đảm bảo khởi tạo kép sẽ không bao giờ được tạo.

6.6.3 Khi nào sử dụng Singleton

- Khi cần chính xác một khởi tạo của một lớp và nó cần được truy cập từ client từ một điểm truy cập được biết đến.
- Khi khởi tạo gốc có thể được mở rộng bởi các lớp con, và client có thể sử dụng khởi tạo mở rộng mà không cần thay đổi code của họ.