第2章 单例模式

提出问题

> 问题描述：设计一个打印池模块，用于接收并执行各种模块发送过来的打印请求。

设计打印池之前先设计一个请求类，包含请求ID和内容两个信息：

```
public class Request {
    private String id;
    private String content;
}
```

打印池的请求采用队列的方式管理，设计`add()`和`cancel()`两个方法实现对打印请求的增/删操作，设计`print()`方法模拟打印操作。具体实现如下：

```
public class Spooler {
    private Queue<Request> requestList;

    public Spooler() {
        requestList = new LinkedList<Request>();
    }

    public void print() {
        Request r = requestList.poll();
        if(r != null) {
            System.out.println("ID:" + r.getId());
            System.out.println("Content:" + r.getContent());
        }
    }

    public void cancel(String id) {
        Request remove = null;
        for(Request r:requestList) {
            if(id.equals(r.getId())) {
                remove = r;
            }
        }
        if(remove != null)
            requestList.remove(remove);
    }

    public void add(Request r) {
        requestList.offer(r);
    }
}
```

客户端采用两个线程模拟打印请求发送和打印处理两个过程。打印请求线程每秒随机产生并发送2个打印任务，打印线程每秒处理1个打印任务。

```
public class Test {
    public static void main(String[] args) {
        Spooler sp = new Spooler();
        Thread taskThread = new Thread(new Runnable() {
            @Override
            public void run() {
                while(true) {
                    sp.add(new Request(UUID.randomUUID().toString(), "" + Math.random()));
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }}, "CreateTask");

        taskThread.start();

        Thread printThread = new Thread(new Runnable() {
            @Override
            public void run() {
                while(true) {
                    sp.print();
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }, "Print");
        printThread.start();
    }
}
```

通过分析，整个程序生命周期中只需要一个打印池实例，而且为该实例提供全局访问。思考如何设计更合理?

模式名称

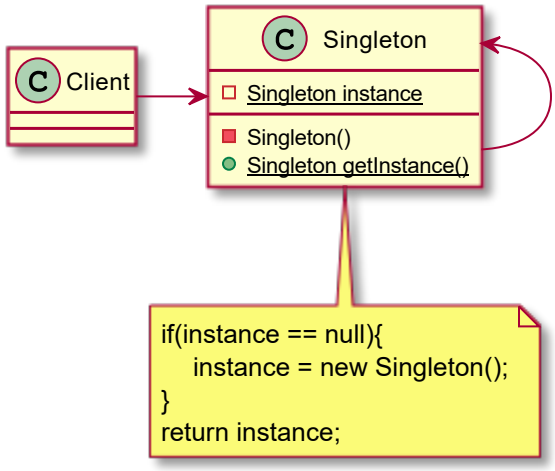
单例模式：Singleton

设计意图

确保一个类只有一个实例，并能在全局访问。

> Ensure a class only has one instance, and provide a global point of access to it.

设计结构



实现方式

根据单例创建时机，主要分为饿汉模式和懒汉模式两种实现方式。饿汉模式是在类加载后直接创建实例，懒汉模式是按需在调用`getInstance()`方法的时候根据情况创建。

饿汉模式

饿汉模式有两种具体实现方式：（1）直接赋初值；（2）静态代码赋值。
（1）直接赋初值的方式适合简单的单例对象。假如对象创建过程比较复杂，而且面向资源管理功能（例如文件系统等）时，需要一定的逻辑代码和异常处理代码。

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return instance;
    }
}
```

（2）静态代码赋值的方式。

```
public class Singleton {
    private static Singleton instance;
    // 静态代码，类加载后执行一次
    static {
        try {
            instance = new Singleton();
        }catch(Exception e) {
            throw new RuntimeException("Exception occured in creating singleton instance");
        }
    }
    private Singleton() {}
    public static Singleton getInstance() {
        return instance;
    }
}
```

懒汉模式

（1）简单初始化方式。在`getInstance()`中判断当前实例是否已经被创建，如果为空则调用构造器创建对象。这种方式适合单线程程序，多线程可能同时调用`getInstance()`方法创建多个对象，破坏单例结构。

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

（2）线程安全的方式。

- 方法同步。这种方式线程安全，但是效率较低，每次只允许一个线程访问单例对象。

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- 双重检查。

```
public static Singleton getInstance() {
    if(instance == null) {
        synchronized(Singleton.class) {
            if(instance == null) {
                instance = new Singleton();
            }
        }
    }
    return instance;
}
```

单例与序列化

反序列化过程一般会读取一个独立对象，可能破坏单例结构。例如以下代码会打印两个不同的哈希码。

```
public class Singleton implements Serializable{

    public static void main(String[] args) throws Exception {
        Singleton instanceOne = Singleton.getInstance();
        ObjectOutput out = new ObjectOutputStream(new FileOutputStream("filename.ser"));
        out.writeObject(instanceOne);
        out.close();

        ObjectInput in = new ObjectInputStream(new FileInputStream("filename.ser"));
        Singleton instanceTwo = (Singleton) in.readObject();
        in.close();

        System.out.println("instanceOne hashCode="+instanceOne.hashCode());
        System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());
    }

    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            synchronized(Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

根据对象读取的规范，可以通过实现`readResolve()`方法来替代输入流中的对象。

> For `Serializable` classes, the readResolve method allows a class to replace or resolve the object read from the stream before it is returned to the caller. The `readResolve` method is called when `ObjectInputStream` has read an object from the stream and is preparing to return it to the caller.

具体实现方式如下：

```
protected Object readResolve() {
    return getInstance();
}
```

单例与反射

Java反射是一种动态访问信息的机制，在运行状态下能获取任意类的结构（方法定义和字段声明），访问任意对象的方法和属性。即便在编译时定义为不可访问的信息，通过反射机制也能访问。

> Java Reflection provides ability to inspect and modify the runtime behavior of application. Reflection in Java is one of the advance topic of core java. Using java reflection we can inspect a class, interface, enum, get their structure, methods and fields information at runtime even though class is not accessible at compile time. We can also use reflection to instantiate an object, invoke its methods, change field values.

通过反射机制调用私有构造器创建新对象，从而破坏单例结构。具体如下：

```
public class Singleton{

    public static void main(String[] args) throws Exception {
        Singleton instanceOne = Singleton.getInstance();
        Singleton instanceTwo = null;
        try {
            Constructor<?>[] constructors = Singleton.class.getDeclaredConstructors();
            for (Constructor<?> constructor : constructors) {
                //创建新对象，破坏单例结构
                constructor.setAccessible(true);
                instanceTwo = (Singleton) constructor.newInstance();
                break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(instanceOne.hashCode());
        System.out.println(instanceTwo.hashCode());
    }
    // 其他代码略
```

单例与克隆

通过`Object`类的`clone()`函数实现对象复制会破坏单例结构。

```
public class Singleton implements Cloneable{

    public static void main(String[] args) throws Exception {
        Singleton one = Singleton.getInstance();
        Singleton two = (Singleton) one.clone();

        System.out.println(one.hashCode());
        System.out.println(two.hashCode());
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

解决问题

根据单例模式进行修改，打印池类修改如下：

```
public class Spooler {
    private Queue<Request> requestList;

    // 增加引用自身的静态字段
    private static Spooler instance;

    // 增加getInstance方法
    public static Spooler getInstance() {
        if(instance == null) {
            synchronized(Spooler.class) {
                if(instance == null) {
                    instance = new Spooler();
                }
            }
        }
        return instance;
    }

    // 访问性修改为private
    private Spooler() {
        requestList = new LinkedList<Request>();
    }

    // 其他不变
}
```

客户端程序代码修改如下：

```
public class Test {
    public static void main(String[] args) {

        Thread taskThread = new Thread(new Runnable() {
            @Override
            public void run() {
                while(true) {
                    Request r = new Request(UUID.randomUUID().toString(), ""+Math.random());
                    // 通过静态方法访问
                    Spooler.getInstance().add(r);
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }}, "CreateTask");
        taskThread.start();

        Thread printThread = new Thread(new Runnable() {
            @Override
            public void run() {
                while(true) {
                    // 通过静态方法访问
                    Spooler.getInstance().print();
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }, "Print");
        printThread.start();
    }
}
```

效果与适用性

效果

- 只创建一个实例，强化对独占实例的访问控制；

- 支持全局访问，优化共享资源访问，节省内存开销。

适用性

- 当类只能有一个实例而且客户可以从全局访问；
- 需要频繁实例化的类，实例化耗时较长，占用资源多，经常使用；
- 频繁访问数据库和文件等外部资源的对象。

扩展案例

JDK中的RunTime

`RunTime`类单例设计：

```
public class Runtime {
    private static final Runtime currentRuntime = new Runtime();
    private static Version version;
    public static Runtime getRuntime() {
        return currentRuntime;
    }
}
```

> Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method. An application cannot create its own instance of this class.