链滴

深入理解几种单例模式的实现方式

作者：zyjImmortal
原文链接：https://ld246.com/article/1586486644999
来源网站：链滴

饿汉式

饿汉式的单例实现方式就是说在类加载的时候就已经创建并初始化好了，所以实例的创建过程是线程全的

```
public class IdGenerator {
    private AtomicLong id = new AtomicLong(0);
    private static final IdGenerator instance = new IdGenerator();
    private IdGenerator(){}
    public static IdGenerator getInstance(){
        return instance;
    }
    public long getId(){
        return id.incrementAndGet();
    }
}
```

但是饿汉式是有一个缺点的，它不支持延时加载，也就是说实例在使用之前就已经创建好了，如果占资源过多，但又不使用，会造成一定的浪费，还是需要根据具体情况确定要不要使用这种方式。不过有人说，如果耗时长，那在使用的时候再加载，就会影响性能，也是难为了饿汉了。

AtomicLong是一个原子变量类型，提供了线程安全的原子操作，这样可以保证在多线程环境下，获取d的时候不会出现重复ID的情况。

代码中的构造函数通过private修饰符进行修饰，保证了外部代码不能通过构造函数初始化IdGenerato类。

上面是一个简单的唯一递增ID号码生成器，采用的饿汉式的单例实现模式，instance实例定义成了一静态常量，我们这知道在运行一个类的时候，先要加载到JVM中，类加载的过程分为三个阶段，分别：加载、链接和初始化，其中链接阶段又分为三个步骤分别是：验证、准备和解析，其中在准备的步中就会创建类或接口的静态变量，并初始化静态变量的初始值。

懒汉式

上面我们说到饿汉式是不支持延时加载到的，那懒汉式就支持延时加载了，懒汉式的实现方式是给获实例的方法加了锁

```
public class IdGenerator {
    private AtomicLong id = new AtomicLong(0);
    private static IdGenerator instance;
    private IdGenerator(){}

    public static synchronized IdGenerator getInstance(){
        if (instance == null){
            instance = new IdGenerator();
        }
        return instance;
    }
    public long getId(){
        return id.incrementAndGet();
    }
}
```

加锁的结果就是性能降低，如果这个单例被频繁的使用的话，那性能问题就会比较严重，需要考虑换方式实现了。

双重检测

双重检测的单例实现方式弥补了上面饿汉式和懒汉式的缺点：不能延时加载和性能低的问题，具体实方式就是在获取的实例的时候先判断是否已经创建过了，如果是就直接返回,这是第一重检测，没有的，就进入同步块，进入同步块后再进行判断实例是否已经存在，如果存在就直接返回，这是第二重检，如果不存在的话就在同步的情况下创建一个实例。

```
public class IdGenerator {
    private AtomicLong id = new AtomicLong(0);
    private static IdGenerator instance;
    private IdGenerator(){
        // 初始化代码
    }
    public static IdGenerator getInstance(){
        if (instance == null){
            synchronized (IdGenerator.class){ // 这里指明synchronized保护的是当前的类对象
                if (instance == null){
                    instance = new IdGenerator();
                }
            }
        }
        return instance;
    }
    public long getId(){
        return id.incrementAndGet();
    }
}
```

懒汉式的实现方式每次获取实例的时候都要进同步代码，这样就会造成多次的获取锁释放锁，造成性

损耗，但是双重检测实际上只需要同步一次创建实例就可以了，再获取实例的时候是不用进同步块代的，这样就大大提高了性能。

静态内部类

静态内部类的实现方式是一种比双重检测更加简单的一种实现方式，而且既保证了性能又做到了延时载

```
public class IdGenerator {
    private AtomicLong id = new AtomicLong(0);
    private IdGenerator(){
        // 初始化代码
    }
    private static class SingletonHolder{
        private static final IdGenerator instance = new IdGenerator();
    }

    public static IdGenerator getInstance(){
        return SingletonHolder.instance;
    }
    public long getId(){
        return id.incrementAndGet();
    }
}
```

SingletonHolder是一个静态内部类，使用privat可以使内部类完全对外隐藏，当外部类IdGenerator载的时候，并不会创建实例，只有当调用getInstance方法的时候才会加载SingletonHolder，并创实例，这样就具备了饿汉式的安全特性，同时也具备了延迟加载的特性。

枚举

枚举的实现方式是利用了java枚举类型本身的特性，保证了实例创建的线程安全和实例的唯一性

```
public enum IdGenerator {
    INSTANCE;
    private AtomicLong id = new AtomicLong(0);
    public long getId(){
        return id.incrementAndGet();
    }
    public static void main(String[] args) {
        IdGeneratorEnum.INSTANCE.getId();
    }
}
```

可以看出枚举的实现方式是最简洁的，由jvm保证线程安全和单一实例。还可以有效防止序列化和反列化造成多个实例和利用反射创建多个实例的情况。枚举类型在编译成class文件后，再反编译，

```
public final class IdGenerator extends java.lang.Enum<IdGenerator> {
  public static final IdGenerator INSTANCE;
  public static IdGenerator[] values();
  public static IdGenerator valueOf(java.lang.String);
  public long getId();
  public static void main(java.lang.String[]);
```

```
  static {};
}
```

反编译后的枚举类其实继承了Enum这个类，INSTANCE是一个静态常量了，似乎又回到了饿汉式的种那些，但是比它多出了延迟加载而且更加简洁的特性