单例模式，真不简单

原创 苏三说技术 苏三说技术 今天

收录于话题
#设计模式

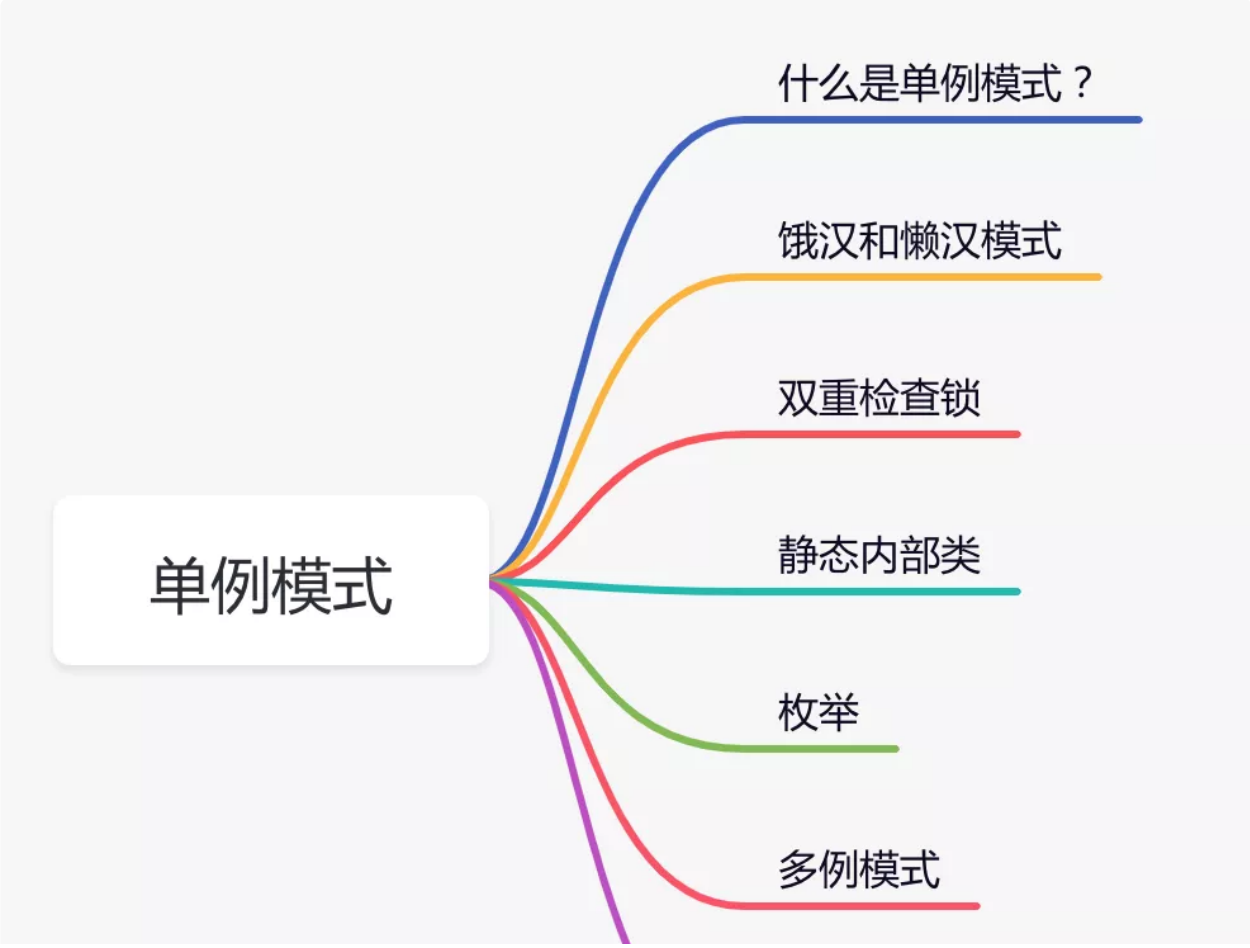
1个

大家好，我是苏三，又跟大家见面了。

前言

单例模式无论在我们面试，还是日常工作中，都会面对的问题。但很多单例模式的细节，值得我们深入探索一下。

这篇文章透过单例模式，串联了多方面基础知识，非常值得一读。



1 什么是单例模式？

单例模式 是一种非常常用的软件设计模式，它定义是单例对象的类 只能允许一个实例存在 。

该类负责创建自己的对象，同时确保只有一个对象被创建。一般常用在工具类的实现或创建对象需要消耗资源的业务场景。

单例模式的特点：

- 类构造器私有
- 持有自己类的引用
- 对外提供获取实例的静态方法

我们先用一个简单示例了解一下单例模式的用法。

```
public class SimpleSingleton {
    //持有自己类的引用
    private static final SimpleSingleton INSTANCE = new SimpleSingleton();

    //私有的构造方法
    private SimpleSingleton() {
    }
    //对外提供获取实例的静态方法
    public static SimpleSingleton getInstance() {
        return INSTANCE;
    }

    public static void main(String[] args) {
        System.out.println(SimpleSingleton.getInstance().hashCode());
        System.out.println(SimpleSingleton.getInstance().hashCode());
    }
}
```

打印结果：

```
1639705018
1639705018
```

我们看到两次获取SimpleSingleton实例的hashCode是一样的，说明两次调用获取到的是同一个对象。

可能很多朋友平时工作当中都是这么用的，但我要说这段代码是有问题的，你会相信吗？

不信，我们一起往下看。

2 饿汉和懒汉模式

在介绍单例模式的时候，必须要先介绍它的两种非常著名的实现方式： 饿汉模式 和 懒汉模式 。

2.1 饿汉模式

实例在初始化的时候就已经建好了，不管你有没有用到，先建好了再说。具体代码如下：

```
public class SimpleSingleton {
    //持有自己类的引用
    private static final SimpleSingleton INSTANCE = new SimpleSingleton();

    //私有的构造方法
    private SimpleSingleton() {
    }
    //对外提供获取实例的静态方法
    public static SimpleSingleton getInstance() {
        return INSTANCE;
```

```
    }
}
```

饿汉模式，其实还有一个变种：

```
public class SimpleSingleton {
    //持有自己类的引用
    private static final SimpleSingleton INSTANCE;
    static {
       INSTANCE = new SimpleSingleton();
    }

    //私有的构造方法
    private SimpleSingleton() {
    }
    //对外提供获取实例的静态方法
    public static SimpleSingleton getInstance() {
        return INSTANCE;
    }
}
```

使用静态代码块的方式实例化INSTANCE对象。

使用饿汉模式的好处是： 没有线程安全的问题 ，但带来的坏处也很明显。

```
private static final SimpleSingleton INSTANCE = new SimpleSingleton();
```

一开始就实例化对象了，如果实例化过程非常耗时，并且最后这个对象没有被使用，不是白白造成资源浪费吗？

还真是啊。

这个时候你也许会想到，不用提前实例化对象，在真正使用的时候再实例化不就可以了？

这就是我接下来要介绍的： 懒汉模式 。

2.2 懒汉模式

顾名思义就是实例在用到的时候才去创建，“比较懒”，用的时候才去检查有没有实例，如果有则返回，没有则新建。具体代码如下：

```
public class SimpleSingleton2 {

    private static SimpleSingleton2 INSTANCE;

    private SimpleSingleton2() {
    }

    public static SimpleSingleton2 getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new SimpleSingleton2();
        }
        return INSTANCE;
    }
}
```

示例中的INSTANCE对象一开始是空的，在调用getInstance方法才会真正实例化。

嗯，不错不错。但这段代码还是有问题。

2.3 synchronized关键字

上面的代码有什么问题？

答：假如有多个线程中都调用了getInstance方法，那么都走到 if (INSTANCE == null) 判断时，可能同时成立，因为INSTANCE初始化时默认值是null。这样会导致多个线程中同时创建INSTANCE对象，即INSTANCE对象被创建了多次，违背了只创建一个INSTANCE对象的初衷。

那么，要如何改进呢？

答：最简单的办法就是使用 `synchronized` 关键字。

改进后的代码如下：

```
public class SimpleSingleton3 {
    private static SimpleSingleton3 INSTANCE;

    private SimpleSingleton3() {
    }

    public synchronized static SimpleSingleton3 getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new SimpleSingleton3();
        }
        return INSTANCE;
    }
    public static void main(String[] args) {
        System.out.println(SimpleSingleton3.getInstance().hashCode());
        System.out.println(SimpleSingleton3.getInstance().hashCode());
    }
}
```

在getInstance方法上加 `synchronized` 关键字，保证在并发的情况下，只有一个线程能创建INSTANCE对象的实例。

这样总可以了吧？

答：不好意思，还是有问题。

有什么问题？

答：使用synchronized关键字会消耗getInstance方法的性能，我们应该判断当INSTANCE为空时才加锁，如果不为空不应该加锁，需要直接返回。

这就需要使用下面要说的双重检查锁了。

2.4 饿汉和懒汉模式的区别

but，在介绍双重检查锁之前，先插播一个朋友们可能比较关心的话题：饿汉模式 和 懒汉模式 各有什么优缺点？

- 饿汉模式 ：优点是没有线程安全的问题，缺点是浪费内存空间。

- 懒汉模式 ：优点是没有内存空间浪费的问题，缺点是如果控制不好，实际上不是单例的。

好了，下面可以安心的看看双重检查锁，是如何保证性能的，同时又保证单例的。

3 双重检查锁

双重检查锁顾名思义会检查两次：在加锁之前检查一次是否为空，加锁之后再检查一次是否为空。

那么，它是如何实现单例的呢？

3.1 如何实现单例？

具体代码如下：

```
public class SimpleSingleton4 {

    private static SimpleSingleton4 INSTANCE;

    private SimpleSingleton4() {
    }

    public static SimpleSingleton4 getInstance() {
        if (INSTANCE == null) {
            synchronized (SimpleSingleton4.class) {
                if (INSTANCE == null) {
                    INSTANCE = new SimpleSingleton4();
                }
            }
        }
        return INSTANCE;
    }
}
```

在加锁之前判断是否为空，可以确保INSTANCE不为空的情况下，不用加锁，可以直接返回。

为什么在加锁之后，还需要判断INSTANCE是否为空呢？

答：是为了防止在多线程并发的情况下，只会实例化一个对象。

比如：线程a和线程b同时调用getInstance方法，假如同时判断INSTANCE都为空，这时会同时进行抢锁。

假如线程a先抢到锁，开始执行synchronized关键字包含的代码，此时线程b处于等待状态。

线程a创建完新实例了，释放锁了，此时线程b拿到锁，进入synchronized关键字包含的代码，如果没有再判断一次INSTANCE是否为空，则可能会重复创建实例。

所以需要在synchronized前后两次判断。

不要以为这样就完了，还有问题呢？

3.2 volatile关键字

上面的代码还有啥问题？

```
public static SimpleSingleton4 getInstance() {
      if (INSTANCE == null) {//1
          synchronized (SimpleSingleton4.class) {//2
              if (INSTANCE == null) {//3
                  INSTANCE = new SimpleSingleton4();//4
              }
          }
      }
      return INSTANCE;//5
  }
```

getInstance方法的这段代码，我是按1、2、3、4、5这种顺序写的，希望也按这个顺序执行。

但是java虚拟机实际上会做一些优化，对一些代码指令进行重排。重排之后的顺序可能就变成了：1、3、2、4、5，这样在多线程的情况下同样会创建多次实例。重排之

后的代码可能如下：

```
public static SimpleSingleton4 getInstance() {
    if (INSTANCE == null) {//1
       if (INSTANCE == null) {//3
           synchronized (SimpleSingleton4.class) {//2
                INSTANCE = new SimpleSingleton4();//4
            }
        }
    }
    return INSTANCE;//5
}
```

原来如此，那有什么办法可以解决呢？

答：可以在定义INSTANCE是加上 volatile 关键字。具体代码如下：

```
public class SimpleSingleton7 {

    private volatile static SimpleSingleton7 INSTANCE;

    private SimpleSingleton7() {
    }

    public static SimpleSingleton7 getInstance() {
        if (INSTANCE == null) {
            synchronized (SimpleSingleton7.class) {
                if (INSTANCE == null) {
                    INSTANCE = new SimpleSingleton7();
                }
            }
        }
        return INSTANCE;
    }
}
```

volatile 关键字可以保证多个线程的可见性，但是不能保证原子性。同时它也能禁止指令重排。

双重检查锁的机制既保证了线程安全，又比直接上锁提高了执行效率，还节省了内存空间。

除了上面的单例模式之外，还有没有其他的单例模式？

4 静态内部类

静态内部类顾名思义是通过静态的内部类来实现单例模式的。

那么，它是如何实现单例的呢？

4.1 如何实现单例模式？

具体代码如下：

```
public class SimpleSingleton5 {

    private SimpleSingleton5() {
    }

    public static SimpleSingleton5 getInstance() {
        return Inner.INSTANCE;
    }

    private static class Inner {
        private static final SimpleSingleton5 INSTANCE = new SimpleSingleton5();
    }
}
```

我们看到在SimpleSingleton5类中定义了一个静态的内部类Inner。在SimpleSingleton5类的getInstance方法中，返回的是内部类Inner的实例INSTANCE对象。

只有在程序第一次调用getInstance方法时，虚拟机才加载Inner并实例化INSTANCE对象。

java内部机制保证了，只有一个线程可以获得对象锁，其他的线程必须等待，保证对象的唯一性。

4.2 反射漏洞

上面的代码看似完美，但还是有漏洞。如果其他人使用 反射 ，依然能够通过类的无参构造方式创建对象。例如：

```
Class<SimpleSingleton5> simpleSingleton5Class = SimpleSingleton5.class;
try {
    SimpleSingleton5 newInstance = simpleSingleton5Class.newInstance();
    System.out.println(newInstance == SimpleSingleton5.getInstance());
} catch (InstantiationException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
}
```

上面代码打印结果是false。

由此看出，通过反射创建的对象，跟通过getInstance方法获取的对象，并非同一个对象，也就是说，这个漏洞会导致SimpleSingleton5非单例。

那么，要如何防止这个漏洞呢？

答：这就需要在无参构造方式中判断，如果非空，则抛出异常了。

改造后的代码如下：

```
public class SimpleSingleton5 {

    private SimpleSingleton5() {
        if(Inner.INSTANCE != null) {
           throw new RuntimeException("不能支持重复实例化");
       }
    }

    public static SimpleSingleton5 getInstance() {
```

```
        return Inner.INSTANCE;
    }

    private static class Inner {
        private static final SimpleSingleton5 INSTANCE = new SimpleSingleton5();
        }
    }

}
```

如果此时，你认为这种静态内部类，实现单例模式的方法，已经完美了。

那么，我要告诉你的是，你错了，还有漏洞。。。

4.3 反序列化漏洞

众所周知，java中的类通过实现 `Serializable` 接口，可以实现序列化。

我们可以把类的对象先保存到内存，或者某个文件当中。后面在某个时刻，再恢复成原始对象。

具体代码如下：

```
public class SimpleSingleton5 implements Serializable {

    private SimpleSingleton5() {
        if (Inner.INSTANCE != null) {
            throw new RuntimeException("不能支持重复实例化");
        }
    }

    public static SimpleSingleton5 getInstance() {
        return Inner.INSTANCE;
    }

    private static class Inner {
        private static final SimpleSingleton5 INSTANCE = new SimpleSingleton5();
    }

    private static void writeFile() {
        FileOutputStream fos = null;
```

```
    ObjectOutputStream oos = null;
    try {
        SimpleSingleton5 simpleSingleton5 = SimpleSingleton5.getInstance();
        fos = new FileOutputStream(new File("test.txt"));
        oos = new ObjectOutputStream(fos);
        oos.writeObject(simpleSingleton5);
        System.out.println(simpleSingleton5.hashCode());
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (oos != null) {
            try {
                oos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (fos != null) {
            try {
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

    }
}

private static void readFile() {
    FileInputStream fis = null;
    ObjectInputStream ois = null;
    try {
        fis = new FileInputStream(new File("test.txt"));
        ois = new ObjectInputStream(fis);
        SimpleSingleton5 myObject = (SimpleSingleton5) ois.readObject();

        System.out.println(myObject.hashCode());
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
```

```
            if (ois != null) {
                try {
                    ois.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        writeFile();
        readFile();
    }
}
```

运行之后，发现序列化和反序列化后对象的hashCode不一样：

```
189568618
793589513
```

说明，反序列化时创建了一个新对象，打破了单例模式对象唯一性的要求。

那么，如何解决这个问题呢？

答：重新readResolve方法。

在上面的实例中，增加如下代码：

```
private Object readResolve() throws ObjectStreamException {
    return Inner.INSTANCE;
}
```

运行结果如下：

```
290658609
290658609
```

我们看到序列化和反序列化实例对象的hashCode相同了。

做法很简单，只需要在readResolve方法中，每次都返回唯一的Inner.INSTANCE对象即可。

程序在反序列化获取对象时，会去寻找readResolve()方法。

- 如果该方法不存在，则直接返回新对象。
- 如果该方法存在，则按该方法的内容返回对象。
- 如果我们之前没有实例化单例对象，则会返回null。

好了，到这来终于把坑都踩完了。

还是费了不少劲。

不过，我偷偷告诉你一句，其实还有更简单的方法，哈哈哈。

纳尼。。。

5 枚举

其实在java中枚举就是天然的单例，每一个实例只有一个对象，这是java底层内部机制保证的。

简单的用法：

```
public enum  SimpleSingleton7 {
    INSTANCE;
```

```
    public void doSamething() {
        System.out.println("doSamething");
    }
}
```

在调用的地方：

```
public class SimpleSingleton7Test {

    public static void main(String[] args) {
        SimpleSingleton7.INSTANCE.doSamething();
    }
}
```

在枚举中实例对象INSTANCE是唯一的，所以它是天然的单例模式。

当然，在枚举对象唯一性的这个特性，还能创建其他的单例对象，例如：

```
public enum  SimpleSingleton7 {
    INSTANCE;

    private Student instance;

    SimpleSingleton7() {
       instance = new Student();
    }

    public Student getInstance() {
       return instance;
    }
}

class Student {
}
```

jvm保证了枚举是天然的单例，并且不存在线程安全问题，此外，还支持序列化。

在java大神Joshua Bloch的经典书籍《Effective Java》中说过：

> 单元素的枚举类型已经成为实现Singleton的最佳方法。

6 多例模式

我们之前聊过的单例模式，都只会产生一个实例。但它其实还有一个变种，也就是我们接下来要聊的： 多例模式 。

多例模式顾名思义，它允许创建多个实例。但它的初衷是为了控制实例的个数，其他的跟单例模式差不多。

具体实现代码如下：

```
public class SimpleMultiPattern {
    //持有自己类的引用
    private static final SimpleMultiPattern INSTANCE1 = new SimpleMultiPattern();
    private static final SimpleMultiPattern INSTANCE2 = new SimpleMultiPattern();

    //私有的构造方法
    private SimpleMultiPattern() {
    }
    //对外提供获取实例的静态方法
    public static SimpleMultiPattern getInstance(int type) {
        if(type == 1) {
          return INSTANCE1;
        }
        return INSTANCE2;
    }
}
```

为了看起来更直观，我把一些额外的安全相关代码去掉了。

有些朋友可能会说：既然多例模式也是为了控制实例数量，那我们常见的 池技术 ，比如：数据库连接池，是不是通过多例模式实现的？

答：不，它是通过 享元模式 实现的。

那么，多例模式和享元模式有什么区别？

- 多例模式：跟单例模式一样，纯粹是为了控制实例数量，使用这种模式的类，通常是作为程序某个模块的入口。
- 享元模式：它的侧重点是对象之间的衔接。它把动态的、会变化的状态剥离出来，共享不变的东西。

7 真实使用场景

最后，跟大家一起聊聊，单例模式的一些使用场景。我们主要看看在java的框架中，是如何使用单例模式，给有需要的朋友一个参考。

7.1 Runtime

jdk提供了 `Runtime` 类，我们可以通过这个类获取系统的运行状态。

比如可以通过它获取cpu核数：

```
int availableProcessors = Runtime.getRuntime().availableProcessors();
```

`Runtime` 类的关键代码如下：

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    public static Runtime getRuntime() {
        return currentRuntime;
    }

    private Runtime() {}
    ...
}
```

从上面的代码我们可以看出，这是一个单例模式，并且是饿汉模式。

但根据文章之前讲过的一些理论知识，你会发现Runtime类的这种单例模式实现方式，显然不太好。实例对象既没用 `final` 关键字修饰，也没考虑对象实例化的性能消耗问题。

不过它的优点是实现起来非常简单。

7.2 NamespaceHandlerResolver

spring提供的DefaultNamespaceHandlerResolver是为需要初始化默认命名空间处理器，是为了方便后面做标签解析用的。

它的关键代码如下：

```
@Nullable
private volatile Map<String, Object> handlerMappings;

private Map<String, Object> getHandlerMappings() {
  Map<String, Object> handlerMappings = this.handlerMappings;
  if (handlerMappings == null) {
   synchronized (this) {
    handlerMappings = this.handlerMappings;
    if (handlerMappings == null) {
     if (logger.isDebugEnabled()) {
      logger.debug("Loading NamespaceHandler mappings from [" + this.handlerMappingsL
     }
     try {
      Properties mappings =
        PropertiesLoaderUtils.loadAllProperties(this.handlerMappingsLocation, this.cl
      if (logger.isDebugEnabled()) {
       logger.debug("Loaded NamespaceHandler mappings: " + mappings);
      }
      handlerMappings = new ConcurrentHashMap<>(mappings.size());
      CollectionUtils.mergePropertiesIntoMap(mappings, handlerMappings);
      this.handlerMappings = handlerMappings;
     }
     catch (IOException ex) {
      throw new IllegalStateException(
        "Unable to load NamespaceHandler mappings from location [" + this.handlerMapp
     }
```

```
    }
   }
  }
  return handlerMappings;
 }
```

我们看到它使用了双重检测锁，并且还定义了一个局部变量handlerMappings，这是非常高明之处。

使用局部变量相对于不使用局部变量，可以提高性能。主要是由于 volatile 变量创建对象时需要禁止指令重排序，需要一些额外的操作。

7.3 LogFactory

mybatis提供 `LogFactory` 类是为了创建日志对象，根据引入的jar包，决定使用哪种方式打印日志。具体代码如下：

```
public final class LogFactory {

  public static final String MARKER = "MYBATIS";

  private static Constructor<? extends Log> logConstructor;

  static {
    tryImplementation(new Runnable() {
      @Override
      public void run() {
        useSlf4jLogging();
      }
    });
    tryImplementation(new Runnable() {
      @Override
      public void run() {
        useCommonsLogging();
      }
    });
    tryImplementation(new Runnable() {
      @Override
      public void run() {
        useLog4J2Logging();
```

```
    }
  });
  tryImplementation(new Runnable() {
    @Override
    public void run() {
      useLog4JLogging();
    }
  });
  tryImplementation(new Runnable() {
    @Override
    public void run() {
      useJdkLogging();
    }
  });
  tryImplementation(new Runnable() {
    @Override
    public void run() {
      useNoLogging();
    }
  });
}

private LogFactory() {
  // disable construction
}

public static Log getLog(Class<?> aClass) {
  return getLog(aClass.getName());
}

public static Log getLog(String logger) {
  try {
    return logConstructor.newInstance(logger);
  } catch (Throwable t) {
    throw new LogException("Error creating logger for logger " + logger + ".  Cause
  }
}

public static synchronized void useCustomLogging(Class<? extends Log> clazz) {
  setImplementation(clazz);
}

public static synchronized void useSlf4jLogging() {
  setImplementation(org.apache.ibatis.logging.slf4j.Slf4jImpl.class);
}
```

```
public static synchronized void useCommonsLogging() {
  setImplementation(org.apache.ibatis.logging.commons.JakartaCommonsLoggingImpl.cla
}

public static synchronized void useLog4JLogging() {
  setImplementation(org.apache.ibatis.logging.log4j.Log4jImpl.class);
}

public static synchronized void useLog4J2Logging() {
  setImplementation(org.apache.ibatis.logging.log4j2.Log4j2Impl.class);
}

public static synchronized void useJdkLogging() {
  setImplementation(org.apache.ibatis.logging.jdk14.Jdk14LoggingImpl.class);
}

public static synchronized void useStdOutLogging() {
  setImplementation(org.apache.ibatis.logging.stdout.StdOutImpl.class);
}

public static synchronized void useNoLogging() {
  setImplementation(org.apache.ibatis.logging.nologging.NoLoggingImpl.class);
}

private static void tryImplementation(Runnable runnable) {
  if (logConstructor == null) {
    try {
      runnable.run();
    } catch (Throwable t) {
      // ignore
    }
  }
}

private static void setImplementation(Class<? extends Log> implClass) {
  try {
    Constructor<? extends Log> candidate = implClass.getConstructor(String.class);
    Log log = candidate.newInstance(LogFactory.class.getName());
    if (log.isDebugEnabled()) {
      log.debug("Logging initialized using '" + implClass + "' adapter.");
    }
    logConstructor = candidate;
  } catch (Throwable t) {
    throw new LogException("Error setting Log implementation.  Cause: " + t, t);
  }
```

```
  }
}
```

这段代码非常经典，但它却是一个不走寻常路的单例模式。因为它创建的实例对象，可能存在多种情况，根据引入不同的jar包，加载不同的类创建实例对象。如果有一个创建成功，则用它作为整个类的实例对象。

这里有个非常巧妙的地方是：使用了很多tryImplementation方法，方便后面进行扩展。不然要写很多，又臭又长的if...else判断。

此外，它跟常规的单例模式的区别是，LogFactory类中定义的实例对象是Log类型，并且getLog方法返回的参数类型也是Log，不是LogFactory。

最关键的一点是：getLog方法中是通过构造器的newInstance方法创建的实例对象，每次请求getLog方法都会返回一个新的实例，它其实是一个多例模式。

7.4 ErrorContext

mybatis提供 `ErrorContext` 类记录了错误信息的上下文，方便后续处理。

那么它是如何实现单例模式的呢？关键代码如下：

```
public class ErrorContext {
  ...
  private static final ThreadLocal<ErrorContext> LOCAL = new ThreadLocal<ErrorContext

  private ErrorContext() {
  }

  public static ErrorContext instance() {
    ErrorContext context = LOCAL.get();
    if (context == null) {
      context = new ErrorContext();
      LOCAL.set(context);
    }
    return context;
  }
  ...
}
```

我们可以看到，ErrorContext跟传统的单例模式不一样，它改良了一下。它使用了饿汉模式，并且使用 `ThreadLocal` ，保证每个线程中的实例对象是单例的。这样看来，ErrorContext类创建的对象不是唯一的，它其实也是多例模式的一种。

7.5 spring的单例

以前在spring中要定义一个bean，需要在xml文件中做如下配置：

```
<bean id="test" class="com.susan.Test" init-method="init" scope="singleton">
```

在bean标签上有个 `scope` 属性，我们可以通过指定该属性控制bean实例是单例的，还是多例的。如果值为 `singleton` ，代表是单例的。当然如果该参数不指定，默认也是单例的。如果值为 `prototype` ，则代表是多例的。

在spring的 `AbstractBeanFactory` 类的 `doGetBean` 方法中，有这样一段代码：

```
if (mbd.isSingleton()) {
    sharedInstance = getSingleton(beanName, () -> {
      return createBean(beanName, mbd, args);
  });
  bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
} else if (mbd.isPrototype()) {
    Object prototypeInstance = createBean(beanName, mbd, args);
    bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
} else {
    ....
}
```

这段代码我为了好演示，看起来更清晰，我特地简化过的。它的主要逻辑如下：

1. 判断如果scope是singleton，则调用getSingleton方法获取实例。
2. 如果scope是prototype，则直接创建bean实例，每次会创建一个新实例。
3. 如果scope是其他值，则允许我们自定bean的创建过程。

其中getSingleton方法主要代码如下：

```
public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
  Assert.notNull(beanName, "Bean name must not be null");
  synchronized (this.singletonObjects) {
   Object singletonObject = this.singletonObjects.get(beanName);
   if (singletonObject == null) {
          singletonObject = singletonFactory.getObject();
         if (newSingleton) {
           addSingleton(beanName, singletonObject);
        }
   }
   return singletonObject;
  }
}
```

有个关键的singletonObjects对象，其实是一个ConcurrentHashMap集合：

```
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);
```

getSingleton方法的主要逻辑如下：

1. 根据beanName先从singletonObjects集合中获取bean实例。
2. 如果bean实例不为空，则直接返回该实例。
3. 如果bean实例为空，则通过getObject方法创建bean实例，然后通过addSingleton方法，将该bean实例添加到singletonObjects集合中。
4. 下次再通过beanName从singletonObjects集合中，就能获取到bean实例了。

在这里spring是通过ConcurrentHashMap集合来保证对象的唯一性。

最后留给大家几个小问题思考一下：

1. 多例模式 和 多对象模式有什么区别？
2. java框架中有些单例模式用的不规范，我要参考不？
3. spring的单例，只是结果是单例的，但完全没有遵循单例模式的固有写法，它也算是单例模式吗？

欢迎大家给我留言，说出你心中的答案。