前面讲完了很多关于多线程的内容，包括各种关键字和一些包的用法，还有类的加载等内容，那么下面将讲解一下一些常见的多线程的相关的设计模式。那么为什么需要讲设计模式呢？因为很多时候一些成员变量需要保证只实例化一次，或者线程的状态需要得到监控，甚至考虑一些不使用锁的机制来保证安全性等等，那么这些都是需要通过优秀的设计模式来实现的。那么首先第一个要讲的就是几种常见的单例设计模式，该模式在多线程下保证了实例的唯一性。而实现该模式的方法有很多，可以从线程安全、高性能和懒加载来评估。

1. 饿汉式(类加载过程中直接初始化)

```
     public class Singleton {
        //实例变量
        private byte[] data = new byte[1024];

       //在定义实例对象的时候直接初始化
        private static Singleton instance = new Singleton();

        //私有构造函数，不允许外部new
        private Singleton() {

        }

        public static Singleton getInstance() {
            System.out.println("******"+Singleton.class.getName());
            return instance;
        }
    }
```

这里注意构造方法那里加了private的话，那么就无法使用new来构造了，但是可以直接调用

```
     public class signalModeTest1 {

        public static void main(String[] args) {
            //测试饿汉式
        Singleton.getInstance();
        }
    }
```

然后这里的运行结果输出就是System的结果，这种方式就是因为instance作为类变量在类初始化的过程中会被收集进()方法中，该方法能够保证同步，因此就不会被实例化多次。但是这样就意味着如果这个变量长期不用的话，那么就会存放在内存当中，导致占用资源了，因此虽然性能不错，但是适用范围不大。

2. 懒汉式(使用时实例化)

2.1 无同步方法

```
  public class Signleton_lazy {
    //实例变量
    private byte[] data = new byte[1024];

   //在定义实例对象的时候不初始化
    private static Signleton_lazy instance = null;

    public Signleton_lazy() {    }

    public static Signleton_lazy getInstance() {
        System.out.println("******"+Signleton_lazy.class.getName());
        if(null == instance) {
            instance = new Signleton_lazy();
        }
        return instance;
    }
}
```

这个方法的特点就是修改了初始化，只有在调用getInstance的时候才实例化，这样就保证了使用时才实例化，但是如果在多线程的环境下还是有问题的，因为如果在new的时候，刚好也有另外一个线程进行判断，但是这时候还没实例化完成，所以会导致多次实例化，因此下面需要改进一下。

2.2同步方法

```
   public static synchronized Signleton_lazy getInstance() {
        System.out.println("******"+Signleton_lazy.class.getName());
        if(null == instance) {
            instance = new Signleton_lazy();
        }
        return instance;
    }
```

这里的区别就是给instance方法添加了一个同步关键字，这样就可以防止实例化多次了，但是这种会牺牲性能的，因此下面改进一下，采用Doucle-Check方法

2.3 Double-Check(在加锁代码块前后分别两次判断)

```
//final不允许被继承
public final class Signleton_Lazy_Double_Check {
    //实例变量
    private byte[] data = new byte[1024];

   //在定义实例对象的时候直接初始化
    private static Signleton_Lazy_Double_Check instance = null;
    int i;
    double j;

    public static synchronized Signleton_Lazy_Double_Check getInstance() {
        System.out.println("******"+Signleton_Lazy_Double_Check.class.getName());
        //当instance为Null时，进入同步代码块，同时该判断避免了每次都需要进入同步代码块，可以提高效率
        if(null == instance) {
            //只有一个线程能够获得Singleton.class关联的monitor
            synchronized(Singleton.class) {
                //判断如果instance为Null则创建
                if(null == instance) {
                    instance = new Signleton_Lazy_Double_Check(1,1.0);
                }
            }

        }
        return instance;
    }

    /*这里有一个风险就是，根据jvm指令重排序和Happends-Before规则
       这里如果instance最先被实例化，而i和j还没完成实例化，那么调用方法将会抛出空指针异常
    */
    public Signleton_Lazy_Double_Check(int i ,double j) {
        this.i = i;
        this.j =j;
    }


}
```

这里与前面不同的是，这里没有对类进行加锁，然后第一次判断的时候，如果有多个线程同时进来了，那么就会争夺monitor锁，争夺成功之后，还要进行多一次判断，这样后面就不需要争夺锁了。这里两个判断都不能少，第一个判断的作用在new以后，就可以避免争夺锁，那样能节省很多性能的。第二个判断是第一次争夺锁成功之后，那么需要进行一次判断。

但是这种方式其实还是有漏洞的，就是指令的重排序可能会导致实例化在getInstance方法之后，那么就会导致出错来了，所以这里最好还是加上volatile

> 下面我们开始说编译原理。所谓编译，就是把源代码“翻译”成目标代码——大多数是指机器代码——的过程。针对Java，它的目标代码不是本地机器代码，而是虚拟机代码。编译原理里面有一个很重要的内容是编译器优化。所谓编译器优化是指，在不改变原来语义的情况下，通过调整语句顺序，来让程序运行的更快。这个过程成为reorder。
>
> 要知道，JVM只是一个标准，并不是实现。JVM中并没有规定有关编译器优化的内容，也就是说，JVM实现可以自由的进行编译器优化。
>
> 下面来想一下，创建一个变量需要哪些步骤呢？一个是申请一块内存，调用构造方法进行初始化操作，另一个是分配一个指针指向这块内存。这两个操作谁在前谁在后呢？JVM规范并没有规定。那么就存在这么一种情况，JVM是先开辟出一块内存，然后把指针指向这块内存，最后调用构造方法进行初始化。
>
> 下面我们来考虑这么一种情况：线程A开始创建SingletonClass的实例，此时线程B调用了getInstance()方法，首先判断instance是否为null。按照我们上面所说的内存模型，A已经把instance指向了那块内存，只是还没有调用构造方法，因此B检测到instance不为null，于是直接把instance返回了——问题出现了，尽管instance不为null，但它并没有构造完成，就像一套房子已经给了你钥匙，但你并不能住进去，因为里面还没有收拾。此时，如果B在A将instance构造完成之前就是用了这个实例，程序就会出现错误了！

上面这段话就是真正说明了在多线程的状态下为什么Double Check的模式会有问题。所以这就是为什么建议使用volatile的作用了。

https://www.cnblogs.com/limingluzhu/p/5156659.html
Double-Check的参考模式

3.Holder方式

这种方式的区别就是把类的初始化不存放实例，在调用的时候直接初始化内部类，然后在内部类当中初始化实例，这样就可以保证只初始化一次了。类加载的初始化阶段() 方法是同步方法，在类中设置一个静态类，将唯一实例放在静态类中，第一次访问实例的时候内部类初始化，即初始化唯一实例。

4.枚举方式

```
  public enum  EnumSingleton {
    INSTANCE;

    private String value;

    EnumSingleton() {
        System.out.println("初始化");
    }

    public static void method() {
        System.out.println("method方法被调用了.........");
     }

    public static EnumSingleton getInstance() {
        return INSTANCE;
    }

    public static void main(String[] args) {
        EnumSingleton.method();
        System.out.println(EnumSingleton.INSTANCE == null);
        //EnumSingleton enumSingleton1 = EnumSingleton.getInstance();
        //EnumSingleton enumSingleton2 = EnumSingleton.getInstance();
        //System.out.println(enumSingleton1 == enumSingleton2);
    }
}
```

这里执行method方法的显示结果如下:

```
初始化
method方法被调用了.........
false
```

这里的结果可以看到静态方法被调用了，那么就会导致类被实例化了，所以这里也和前面提到的主动使用与被动使用有关联了。然后如果调用下面的注释代码，把method方法注释掉执行结果如下所示:

```
   public static void main(String[] args) {
    //EnumSingleton.method();
    //System.out.println(EnumSingleton.INSTANCE == null);
    EnumSingleton enumSingleton1 = EnumSingleton.getInstance();
    EnumSingleton enumSingleton2 = EnumSingleton.getInstance();
    System.out.println(enumSingleton1 == enumSingleton2);
}

   执行结果如下所示:

     初始化
     true
```

但是这里也有一点问题，就是没有办法懒加载特性，那么下面就改造一下。

```
  public class signletonEnum {
        //实例变量
        private byte[] data = new byte[1024];

        private signletonEnum() {

        }
        //使用枚举类型做Holder
        private enum EnumHolder{
            INSTANCE;
            private signletonEnum instance;

            EnumHolder(){
                this.instance = new signletonEnum();
            }

            private signletonEnum getSignleton() {
                return instance;
            }
        }

        public static signletonEnum getInstance() {
            return EnumHolder.INSTANCE.getSignleton();
        }

        public static void main(String[] args) {
            signletonEnum enumSingleton1 = signletonEnum.getInstance();
            signletonEnum enumSingleton2 = signletonEnum.getInstance();
            System.out.println(enumSingleton1 == enumSingleton2);
        }

}
```

这里使用了枚举类型做Holder,那么原理其实和上面的Holder类似的，当调用getInstance()方法的时候，就会初始化内部的枚举类型，那么就会初始化实例了。另外这里枚举类型是线程安全的且只能被实例化一次。