1 单例模式核心

- 单例模式的核心是一个类在其整个生命周期中只有一个实例对象

2 饿汉式

- 优点：简单明了，运行时速度快
- 缺点：在第一次加载类到内存中时就会初始化，不管有没有使用，可能会造成资源浪费

```
public class Teacher {
    private String name;
    private Integer age;
    //防止通过构造器初始化
    private Teacher() {}
    private Teacher(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    private static final Teacher YU_QIAN = new Teacher("于谦",54);
    public Teacher getYuQian() {
        return YU_QIAN;
    }
}
static ExecutorService executor = Executors.newFixedThreadPool(6);
@Test
public void test() throws ExecutionException, InterruptedException {
    for (int i = 0; i < 10000; i++) {
        Future f1 =executor.submit(()->{
            Teacher.getYuQian();
        });
        Future f2 =executor.submit(()->{
            Teacher.getYuQian();
        });
        Teacher t1 =(Teacher) f1.get();
        Teacher t2 =(Teacher) f2.get();
        Assert.assertTrue(t1 == t2);
    }
}
```

3 双重检验锁

- 优点：为懒加载模式，可能节约资源
- 缺点：代码较为复杂，有锁消耗，依赖 JDK 版本

```
// Java 5 以前的 JMM （Java 内存模型）是存在缺陷的，即时将变量声明成 volatile 也不能完全避免重排序
//主要是 volatile 变量前后的代码仍然存在重排序问题。这个 volatile 屏蔽重排序的问题在 Java 5 中才得以修复，所以在这之后才可以放心使用 volatile
public class TeacherDoubleCheck {
    private String name;
    private Integer age;
    private TeacherDoubleCheck() {}
    private TeacherDoubleCheck(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    private volatile static TeacherDoubleCheck YU_QIAN;
    public static TeacherDoubleCheck getTeacher() {
        if (YU_QIAN == null) {                         //Single Checked
            synchronized (Teacher.class) {
                if (YU_QIAN == null) {                 //Double Checked
                    YU_QIAN = new TeacherDoubleCheck("于谦",54);
                }
            }
        }
        return YU_QIAN ;
    }
}
```

4 静态内部类

- 优点：静态内部类只有在第一次调用它的时候才初始化，比双重检验锁代码简单
- 缺点：和双重检验锁一样，初始化静态内部类时会加锁，后面调用时就不会在初始化了，所以也有锁开销

```
public class TeacherStaticNested {
    private String name;
    private Integer age;
    private TeacherStaticNested() {}
    private TeacherStaticNested(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    //定义一个静态内部类
    private static class TeacherHolder {
        private static final TeacherStaticNested YU_QIAN = new TeacherStaticNested("于谦",54);
    }
    public static final TeacherStaticNested getYuQian() {
        return TeacherHolder.YU_QIAN;
    }
}
```

5 枚举

- 优点：第一次调用任意一个枚举实例时，初始化所有的枚举实例，代码简单，序列化安全等
- 缺点：单元素的枚举类型已经成为实现Singleton的最佳方法 ——《Effective Java》

```
public enum Pool {

 ORACLE_POOL("oracle",DataSourceBuilder.create().type(HikariDataSource.class).build()),

 JVM_POOL("jvm",DataSourceBuilder.create().type(ComboPooledDataSource.class).build()),
    MYSQL_POOL("mysql",DataSourceBuilder.create().type(DruidDataSource.class).build());
    private String name;
    private DataSource dataSource;
    Pool (String name,DataSource dataSource) {
        this.name = name;
        this.dataSource = dataSource;
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
static ExecutorService executor = Executors.newFixedThreadPool(6);
@Test
public void test() throws ExecutionException, InterruptedException {
   Teacher t0 = null;
   for (int i = 0; i < 10000; i++) {
       Future f1 =executor.submit(()->{
           Pool.ORACLE_POOL.getDataSource();//第一次调用的时候才初始化，构造器加锁
       });
       Future f2 =executor.submit(()->{
           Pool.ORACLE_POOL.getDataSource();
       });

       Teacher t1 =(Teacher) f1.get();
       Teacher t2 =(Teacher) f2.get();
       Assert.assertTrue(t1 == t2);
       if (i == 9999) t0 = t1;
   }
   Assert.assertFalse(t0 == Pool.JVM_POOL.getDataSource());
}
```

参考

为什么要用枚举实现单例模式（避免反射、序列化问题） 单例模式有五种写法 java枚举类型的实现原理 枚举实现单例原理 Java 利用枚举实现单例模式 为什么我墙裂建议大家使用枚举来实现单例 github 源码地址