单例设计模式

Singleton

郭嘉

知识点:

1.模式定义/应用场景/类图分析

2.字节码知识/字节码指令重排序

3.类加载机制

4.JVM序列化机制

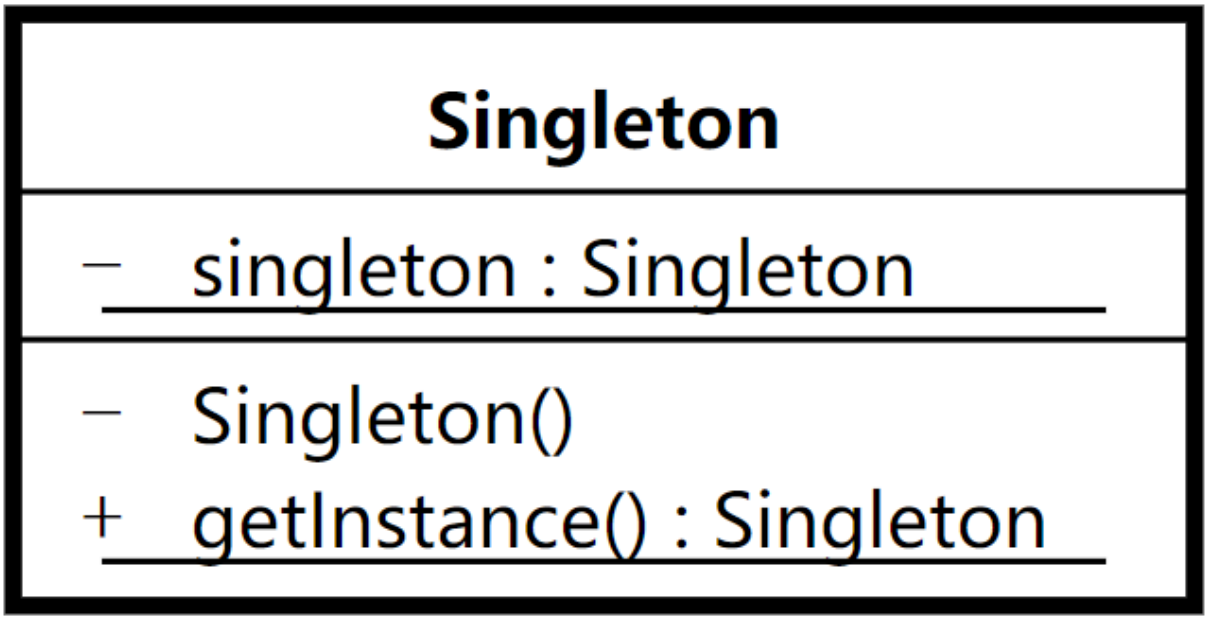
5.单例模式在Spring框架 & JDK源码中的应用

模式定义:

保证一个类只有一个实例，并且提供一个全局访问点

场景:

重量级的对象，不需要多个实例，如线程池，数据库连接池。



1.懒汉模式：延迟加载， 只有在真正使用的时候，才开始实例化。

1）线程安全问题

2）double check 加锁优化

3）编译器(JIT),CPU 有可能对指令进行重排序，导致使用到尚未初始化的实例，可以通过添加volatile 关键字进行修饰，

对于volatile 修饰的字段，可以防止指令重排。

```
class LazySingleton{
 private volatile static LazySingleton instance;
 private LazySingleton(){

 }
 public static LazySingleton getInstance() {
 if (instance==null){
 synchronized (LazySingleton.class){
 if (instance==null){
  instance=new LazySingleton();
  // 字节码层
  // JIT ， CPU 有可能对如下指令进行重排序
  // 1 .分配空间
  // 2 .初始化
  // 3 .引用赋值
  // 如重排序后的结果为如下
```

```
17    // 1 .分配空间
18    // 3 .引用赋值 如果在当前指令执行完，有其他线程来获取实例，将拿到尚未初始化好的
实例
19    // 2 .初始化
20
21
22    }
23    }
24
25    }
26    return instance;
27    }
28  }
```

2.饿汉模式：

类加载的 初始化阶段就完成了 实例的初始化 。本质上就是借助于jvm类加载机制，保证实例的唯一性（初始化过程只会执行一次）及线程安全（JVM以同步的形式来完成类加载的整个过程）。

类加载过程：

1，加载二进制数据到内存中， 生成对应的Class数据结构，

2，连接： a. 验证， b.准备（给类的静态成员变量赋默认值），c.解析

3，初始化： 给类的静态变量赋初值

只有在真正使用对应的类时，才会触发初始化 如（ 当前类是启动类即main函数所在类，直接进行new 操作，访问静态属性、访问静态方法，用反射访问类，初始化一个类的子类等.）

```
1  // 饿汉模式
2
3  class HungrySingleton{
4   private static HungrySingleton instance=new HungrySingleton();
5   private HungrySingleton(){
6   }
7   public static HungrySingleton getInstance() {
8   return instance;
```

```
9  }
10 }
```

3.静态内部类

1).本质上是利用类的加载机制来保证线程安全

2).只有在实际使用的时候，才会触发类的初始化，所以也是懒加载的一种形式。

```
1  class InnerClassSingleton{
2
3   private static class InnerClassHolder{
4   private static InnerClassSingleton instance= new InnerClassSingleton();
5   }
6   private InnerClassSingleton(){
7
8   }
9   public static InnerClassSingleton getInstance(){
10   return InnerClassHolder.instance;
11   }
12 }
```

4.反射攻击实例：

```
1  Constructor<InnerClassSingleton> declaredConstructor=InnerClassSingleton.c
lass.getDeclaredConstructor();
2  declaredConstructor.setAccessible( true );
3  InnerClassSingleton innerClassSingleton=declaredConstructor.newInstance();
4
5  InnerClassSingleton instance=InnerClassSingleton.getInstance();
6  System.out.println(innerClassSingleton==instance);
```

静态内部类防止反射破坏

```
1  class InnerClassSingleton {
2
3
4   private static class InnerClassHolder{
5   private static InnerClassSingleton instance= new InnerClassSingleton();
6   }
7   private InnerClassSingleton(){
8
```

```
 if (InnerClassHolder.instance!=null){
  throw new RuntimeException( " 单例不允许多个实例 " );
  }

  }
  public static InnerClassSingleton getInstance(){
  return InnerClassHolder.instance;
  }
}
```

5.枚举类型

1）天然不支持反射创建对应的实例，且有自己的反序列化机制

2）利用类加载机制保证线程安全

```
public enum EnumSingleton {
 INSTANCE;

 public void print(){
 System.out.println(this.hashCode());
 }
}
```

6.序列化

1）可以利用 指定方法来替换从反序列化流中的数据 如下

```
ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;
```

```
class InnerClassSingleton implements Serializable{

 static final long serialVersionUID = 42L;

 private static class InnerClassHolder{
 private static InnerClassSingleton instance= new InnerClassSingleton();
 }
 private InnerClassSingleton(){

  if (InnerClassHolder.instance!=null){
  throw new RuntimeException( " 单例不允许多个实例 " );
  }
```

```
13
14  }
15  public static InnerClassSingleton getInstance(){
16  return InnerClassHolder.instance;
17  }
18
19  Object readResolve() throws ObjectStreamException{
20  return InnerClassHolder.instance;
21  }
22
23 }
```

源码中的应用

```
1 // Spring & JDK
2 java.lang.Runtime
3 org.springframework.aop.framework.ProxyFactoryBean
4 org.springframework.beans.factory.support.DefaultSingletonBeanRegistry
5 org.springframework.core.ReactiveAdapterRegistry
6 // Tomcat
7 org.apache.catalina.webresources.TomcatURLStreamHandlerFactory
8 // 反序列化指定数据源
9 java.util.Currency
```