

# 单例 (Singleton)

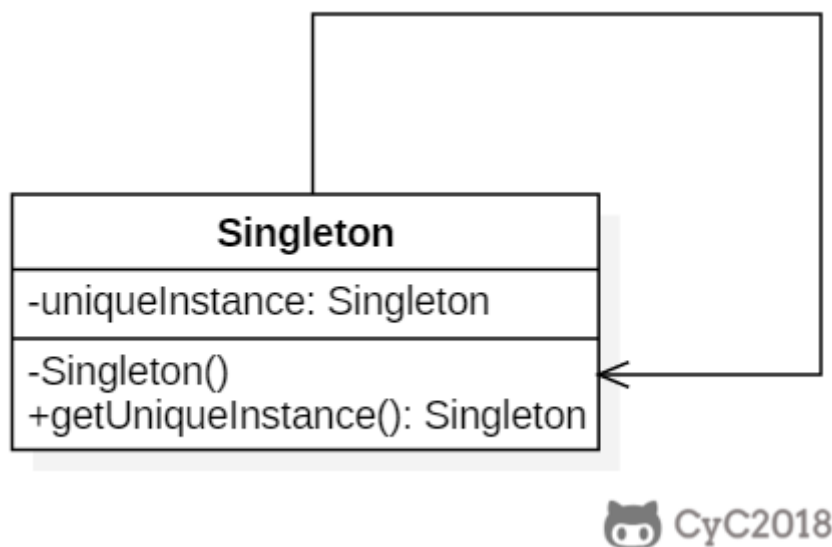
## Intent

确保一个类只有一个实例，并提供该实例的全局访问点。

## Class Diagram

使用一个私有构造函数、一个私有静态变量以及一个公有静态函数来实现。

私有构造函数保证了不能通过构造函数来创建对象实例，只能通过公有静态函数返回唯一的私有静态变量。



CyC2018

## Implementation

### I 懒汉式-线程不安全

以下实现中，私有静态变量 `uniqueInstance` 被延迟实例化，这样做的好处是，如果没有用到该类，那么就不会实例化 `uniqueInstance`，从而节约资源。

这个实现在多线程环境下是不安全的，如果多个线程能够同时进入 `if (uniqueInstance == null)`，并且此时 `uniqueInstance` 为 `null`，那么会有多个线程执行 `uniqueInstance = new Singleton();` 语句，这将导致实例化多次 `uniqueInstance`。

```
public class Singleton {

    private static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
    }
}
```

```
    }  
    return uniqueInstance;  
}  
}
```

## II 饿汉式-线程安全

线程不安全问题主要是由于 uniqueInstance 被实例化多次，采取直接实例化 uniqueInstance 的方式就不会产生线程不安全问题。

但是直接实例化的方式也丢失了延迟实例化带来的节约资源的好处。

```
private static Singleton uniqueInstance = new Singleton();
```

## III 懒汉式-线程安全

只需要对 getUniqueInstance() 方法加锁，那么在一个时间点只能有一个线程能够进入该方法，从而避免了实例化多次 uniqueInstance。

但是当一个线程进入该方法之后，其它试图进入该方法的线程都必须等待，即使 uniqueInstance 已经被实例化了。这会让线程阻塞时间过长，因此该方法有性能问题，不推荐使用。

```
public static synchronized Singleton getUniqueInstance() {  
    if (uniqueInstance == null) {  
        uniqueInstance = new Singleton();  
    }  
    return uniqueInstance;  
}
```

## IV 双重校验锁-线程安全

uniqueInstance 只需要被实例化一次，之后就可以直接使用了。加锁操作只需要对实例化那部分的代码进行，只有当 uniqueInstance 没有被实例化时，才需要进行加锁。

双重校验锁先判断 uniqueInstance 是否已经被实例化，如果没有被实例化，那么才对实例化语句进行加锁。

```
public class Singleton {  
  
    private volatile static Singleton uniqueInstance;  
  
    private Singleton() {  
    }  
  
    public static Singleton getUniqueInstance() {  
        if (uniqueInstance == null) {  
            synchronized (Singleton.class) {  

```

```
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
    }
}
return uniqueInstance;
}
```

考虑下面的实现，也就是只使用了一个 if 语句。在 `uniqueInstance == null` 的情况下，如果两个线程都执行了 if 语句，那么两个线程都会进入 if 语句块内。虽然在 if 语句块内有加锁操作，但是两个线程都会执行 `uniqueInstance = new Singleton();` 这条语句，只是先后的问题，那么就会进行两次实例化。因此必须使用双重校验锁，也就是需要使用两个 if 语句：第一个 if 语句用来避免 `uniqueInstance` 已经被实例化之后的加锁操作，而第二个 if 语句进行了加锁，所以只能有一个线程进入，就不会出现 `uniqueInstance == null` 时两个线程同时进行实例化操作。

```
if (uniqueInstance == null) {
    synchronized (Singleton.class) {
        uniqueInstance = new Singleton();
    }
}
```

`uniqueInstance` 采用 `volatile` 关键字修饰也是很有必要的，`uniqueInstance = new Singleton();` 这段代码其实是分为三步执行：

1. 为 `uniqueInstance` 分配内存空间
2. 初始化 `uniqueInstance`
3. 将 `uniqueInstance` 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1>3>2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程  $T_1$  执行了 1 和 3，此时  $T_2$  调用 `getUniqueInstance()` 后发现 `uniqueInstance` 不为空，因此返回 `uniqueInstance`，但此时 `uniqueInstance` 还未被初始化。

使用 `volatile` 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

## V 静态内部类实现

当 `Singleton` 类被加载时，静态内部类 `SingletonHolder` 没有被加载进内存。只有当调用 `getUniqueInstance()` 方法从而触发 `SingletonHolder.INSTANCE` 时 `SingletonHolder` 才会被加载，此时初始化 `INSTANCE` 实例，并且 JVM 能确保 `INSTANCE` 只被实例化一次。

这种方式不仅具有延迟初始化的好处，而且由 JVM 提供了对线程安全的支持。

```
public class Singleton {

    private Singleton() {
```

```
}

private static class SingletonHolder {
    private static final Singleton INSTANCE = new Singleton();
}

public static Singleton getUniqueInstance() {
    return SingletonHolder.INSTANCE;
}
}
```

## VI 枚举实现

```
public enum Singleton {

    INSTANCE;

    private String objName;

    public String getObjName() {
        return objName;
    }

    public void setObjName(String objName) {
        this.objName = objName;
    }

    public static void main(String[] args) {

        // 单例测试
        Singleton firstSingleton = Singleton.INSTANCE;
        firstSingleton.setObjName("firstName");
        System.out.println(firstSingleton.getObjName());
        Singleton secondSingleton = Singleton.INSTANCE;
        secondSingleton.setObjName("secondName");
        System.out.println(firstSingleton.getObjName());
        System.out.println(secondSingleton.getObjName());

        // 反射获取实例测试
        try {
            Singleton[] enumConstants = Singleton.class.getEnumConstants();
            for (Singleton enumConstant : enumConstants) {
                System.out.println(enumConstant.getObjName());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
}  
}
```

```
firstName  
secondName  
secondName  
secondName
```

该实现可以防止反射攻击。在其它实现中，通过 `setAccessible()` 方法可以将私有构造函数的访问级别设置为 `public`，然后调用构造函数从而实例化对象，如果要防止这种攻击，需要在构造函数中添加防止多次实例化的代码。该实现是由 JVM 保证只会实例化一次，因此不会出现上述的反射攻击。

该实现在多次序列化和序列化之后，不会得到多个实例。而其它实现需要使用 `transient` 修饰所有字段，并且实现序列化和反序列化的方法。

## Examples

- Logger Classes
- Configuration Classes
- Accesing resources in shared mode
- Factories implemented as Singletons

## JDK

- [java.lang.Runtime#getRuntime\(\)](#)
- [java.awt.Desktop#getDesktop\(\)](#)
- [java.lang.System#getSecurityManager\(\)](#)