单例模式

1.单例模式（singleton pattern）：确保类只有一个实例，并提供一个全局访问点。

2.为什么要使用单例模式：有的时候，一个系统中，如果一个类有多个实例，会造成混乱和资源浪费。

3.使用场景：

- windows的Task Manager和Recycle bin
- 网站计数器
- 应用程序的日志应用
- Web应用的配置对象读取
- 数据库链接池
- 线程池
- 操作系统的文件系统

从以上的例子可以看出，一般使用的条件如下：

- 资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如上述中的日志文件，应用配置。
- 控制资源的情况下，方便资源之间的互相通信。如线程池等。

4.单例模式的经典实现

```
public class Singleton_classic {

 private static Singleton_classic uniqueInstance;
private Singleton_classic(){}

 public static Singleton_classic getInstance(){

 if(uniqueInstance == null){
    uniqueInstance = new Singleton_classic();
 }

 return uniqueInstance;

 }

}
```

这是最简单的实现，但是在多线程的环境下，有可能出现问题，比如线程A判断uniqueInstance为null，所以进入方法进行new，但是new是需要时间的，这时uniqueInstance还是null，线程B此时也通过判断进入方法，然后线程A已经new完了，线程B此时已经进入方法，所以也进行了new，所以此时一共new了两个对象，违背了单一实例。

5.通过同步getInstance方法来解决多线程问题

```
public class Singleton_synchronized {
private static Singleton_synchronized uniqueInstance;

 private Singleton_synchronized(){}

 public static synchronized Singleton_synchronized getInstance(){

 if(uniqueInstance == null){
 uniqueInstance = new Singleton_synchronized();
 }

 return uniqueInstance;

 }

}
```

为getInstance方法加上了synchronized关键字，所以此时不会发生多线程的问题，但是由于线程之间需要互相等待，所以会浪费很多时间，系统性能会下降。

6.通过急切初始化来解决多线程问题

```
public class Singleton_eagerlyInstantiaze {
private static Singleton_eagerlyInstantiaze uniqueInstance = new Singleton_eagerlyInstantiaze();

 private Singleton_eagerlyInstantiaze(){}

 public static Singleton_eagerlyInstantiaze getInstance(){

 return uniqueInstance;

 }

}
```

在定义uniqueInstance的时候已经进行初始化，而且是static的，保证只有一个，而且保证在使用之前已经初始化好了这一个，所以每一个线程都是在使用这一个实例。但是这会使系统启动的时候，就加载了这个类，会增加系统启动的时间，而且假如这个实例一直不被使用，但是却初始化好了，则会造成资源的浪费。

7.通过双重检查加锁的方法来解决多线程的问题

```
public class Singleton_doubleCheckedLocking {
private static volatile Singleton_doubleCheckedLocking uniqueInstance;
```

```
 private Singleton_doubleCheckedLocking(){}

 public static Singleton_doubleCheckedLocking getInstance(){

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

 return uniqueInstance;

 }

}
```

这种实现，依然是使用了延迟初始化的方法，也是对进行了同步机制，但是同步的单位不是在方法级别上，而是在代码块内，通过判断uniqueInstance是否是null值再进行同步，这就减少了同步的次数，不用每一个访问getInstance方法的时候，都进行同步，只有在第一次，uniqueInstance还是null值的时候才进行同步。

其中，进行了两次uniqueInstance == null的判断，原因是这样的。

假设在uniqueInstance还是null的时候，有两个线程进入了方法，此时遇到了同步机制，所以线程A进入同步代码块里面new了一个对象，但是线程B此时不知道线程A已经new了一个对象了，所以继续进去new了一个。

所以，需要在同步代码块里面继续进行uniqueInstance是否为null的判断，保证不会被实例化第二次。

这种方法虽然较好地克服了前两种实现在空间和时间上造成的资源浪费问题，但是还是使用到了同步机制，所以，还是有一定的时间上的开销。

注意uniqueInstance变量需要使用volatile关键字进行修饰。

8.现在有一个实现，叫做Initialization on Demand Holder，就是IoDH,实现如下：

```
public class Singleton_IoDH {
private Singleton_IoDH(){}

 private static class HolderClass{
 private final static Singleton_IoDH uniqueInstance = new Singleton_IoDH();
 }

 public static Singleton_IoDH getInstance(){
 return HolderClass.uniqueInstance;
 }

}
```

由于静态单例对象没有作为Singleton的成员变量直接实例化，因此类加载时不会实例化Singleton，第一次调用

getInstance()时将加载内部类HolderClass，在该内部类中定义了一个static类型的变量instance，此时会首先初始化这个成员变量，由Java虚拟机来保证其线程安全性，确保该成员变量只能初始化一次。由于getInstance()方法没有任何线程锁定，因此其性能不会造成任何影响。

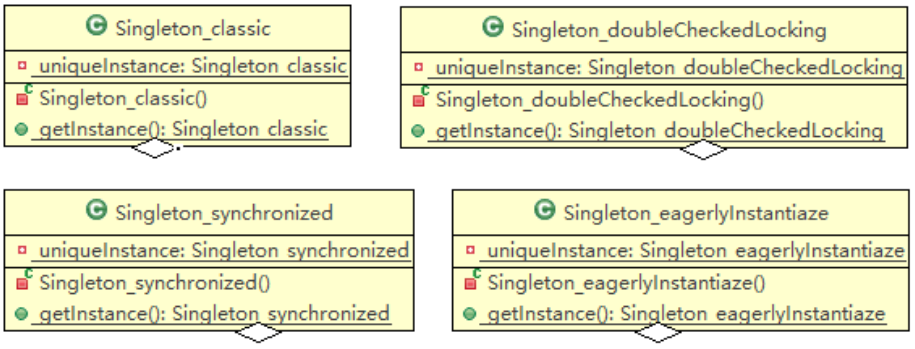
这种实现，是目前最好的单例模式的实现。

9.需要注意的问题：

如果程序存在多个类加载器，则有可能多个类加载器对同一个类进行加载，然后产生多个单例的实例。所以如果程序有多个类加载器，又想使用单例模式，则需要自行指定类加载器，并指定同一个类加载器。

10.UML图

正常的（包括解决多线程问题的）单例模式的UML图如下：



IoDH实现的单例模式UML如下：