本文主要介绍java的单例模式，以及详细剖析静态内部类之所以能够实现单例的原理。OK，废话不多说，进入正文。

首先我们要先了解下单例的四大原则：

1.构造私有。

2.以静态方法或者枚举返回实例。

3.确保实例只有一个，尤其是多线程环境。

4.确保反序列换时不会重新构建对象。

我们常用的单例模式有：

饿汉模式、懒汉模式、双重锁懒汉模式、静态内部类模式、枚举模式，我们来逐一分析下这些模式的区别。

1.饿汉模式：

public class SingleTon{

private static SingleTon INSTANCE = new SingleTon();

private SingleTon(){

}

public static SingleTon getInstance() {

return INSTANCE;

}

}

饿汉模式在类被初始化时就已经在内存中创建了对象，以空间换时间，故不存在线程安全问题。

2.懒汉模式：

public class SingleTon{

private static SingleTon INSTANCE = null;

private SingleTon(){}

public static SingleTon getInstance() {

if(INSTANCE == null){

INSTANCE = new SingleTon();

}

return INSTANCE；

}

}

懒汉模式在方法被调用后才创建对象，以时间换空间，在多线程环境下存在风险。

3.双重锁懒汉模式(Double Check Lock)

public class SingleTon{

private static SingleTon INSTANCE = null;

private SingleTon(){}

public static SingleTon getInstance(){if(INSTANCE == null){

synchronized(SingleTon.class){

if(INSTANCE == null){

INSTANCE = new SingleTon();

}

}

return INSTANCE;

}

}

}

DCL模式的优点就是，只有在对象需要被使用时才创建，第一次判断 INSTANCE == null为了避免非必要加锁，当第一次加载时才对实例进行加锁再实例化。这样既可以节约内存空间，又可以保证线程安全。但是，由于jvm存在乱序执行功能，DCL也会出现线程不安全的情况。具体分析如下：

INSTANCE = new SingleTon();

这个步骤，其实在jvm里面的执行分为三步：

   1.在堆内存开辟内存空间。

  2.在堆内存中实例化SingleTon里面的各个参数。

  3.把对象指向堆内存空间。

由于jvm存在乱序执行功能，所以可能在2还没执行时就先执行了3，如果此时再被切换到线程B上，由于执行了3，INSTANCE 已经非空了，会被直接拿出来用，这样的话，就会出现异常。这个就是著名的DCL失效问题。

不过在JDK1.5之后，官方也发现了这个问题，故而具体化了volatile，即在JDK1.6及以后，只要定义为private volatile static SingleTon  INSTANCE = null;就可解决DCL失效问题。volatile确保INSTANCE每次均在主内存中读取，这样虽然会牺牲一点效率，但也无伤大雅。

3.静态内部类模式：

public class SingleTon{

private SingleTon(){}

private static class SingleTonHoler{

private static SingleTon INSTANCE = new SingleTon();

}

public static SingleTon getInstance(){

return SingleTonHoler.INSTANCE;

}

}

静态内部类的优点是：外部类加载时并不需要立即加载内部类，内部类不被加载则不去初始化INSTANCE，故而不占内存。即当SingleTon第一次被加载时，并不需要去加载SingleTonHoler，只有当getInstance()方法第一次被调用时，才会去初始化INSTANCE,第一次调用getInstance()方法会导致虚拟机加载SingleTonHoler类，这种方法不仅能确保线程安全，也能保证单例的唯一性，同时也延迟了单例的实例化。

那么，静态内部类又是如何实现线程安全的呢？首先，我们先了解下类的加载时机。

类加载时机：JAVA虚拟机在有且仅有的5种场景下会对类进行初始化。

1.遇到new、getstatic、setstatic或者invikestatic这4个字节码指令时，对应的java代码场景为：new一个关键字或者一个实例化对象时、读取或设置一个静态字段时(final修饰、已在编译期把结果放入常量池的除外)、调用一个类的静态方法时。

2.使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没进行初始化，需要先调用其初始化方法进行初始化。

3.当初始化一个类时，如果其父类还未进行初始化，会先触发其父类的初始化。

4.当虚拟机启动时，用户需要指定一个要执行的主类(包含main()方法的类)，虚拟机会先初始化这个类。

5.当使用JDK 1.7等动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

这5种情况被称为是类的主动引用，注意，这里《虚拟机规范》中使用的限定词是"有且仅有"，那么，除此之外的所有引用类都不会对类进行初始化，称为被动引用。静态内部类就属于被动引用的行列。

我们再回头看下getInstance()方法，调用的是SingleTonHoler.INSTANCE，取的是SingleTonHoler里的INSTANCE对象，跟上面那个DCL方法不同的是，getInstance()方法并没有多次去new对象，故不管多少个线程去调用getInstance()方法，取的都是同一个INSTANCE对象，而不用去重新创建。当getInstance()方法被调用时，SingleTonHoler才在SingleTon的运行时常量池里，把符号引用替换为直接引用，这时静态对象INSTANCE也真正被创建，然后再被getInstance()方法返回出去，这点同饿汉模式。那么INSTANCE在创建过程中又是如何保证线程安全的呢？在《深入理解JAVA虚拟机》中，有这么一句话:

 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果在一个类的<clinit>()方法中有耗时很长的操作，就可能造成多个进程阻塞(需要注意的是，其他线程虽然会被阻塞，但如果执行<clinit>()方法后，其他线程唤醒之后不会再次进入<clinit>()方法。同一个加载器下，一个类型只会初始化一次。)，在实际应用中，这种阻塞往往是很隐蔽的。

故而，可以看出INSTANCE在创建过程中是线程安全的，所以说静态内部类形式的单例可保证线程安全，也能保证单例的唯一性，同时也延迟了单例的实例化。

那么，是不是可以说静态内部类单例就是最完美的单例模式了呢？其实不然，静态内部类也有着一个致命的缺点，就是传参的问题，由于是静态内部类的形式去创建单例的，故外部无法传递参数进去，例如Context这种参数，所以，我们创建单例时，可以在静态内部类与DCL模式里自己斟酌。

最后粗略的介绍下枚举类型的单例吧。

枚举单例：

public enum SingleTon{

INSTANCE;

public void method(){

//TODO

}

}

枚举在java中与普通类一样，都能拥有字段与方法，而且枚举实例创建是线程安全的，在任何情况下，它都是一个单例。我们可直接以

SingleTon.INSTANCE