设计模式-单例模式

设计模式 单例模式

单例模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。
这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。
注意：
1、单例类只能有一个实例。
2、单例类必须自己创建自己的唯一实例。
3、单例类必须给所有其他对象提供这一实例。

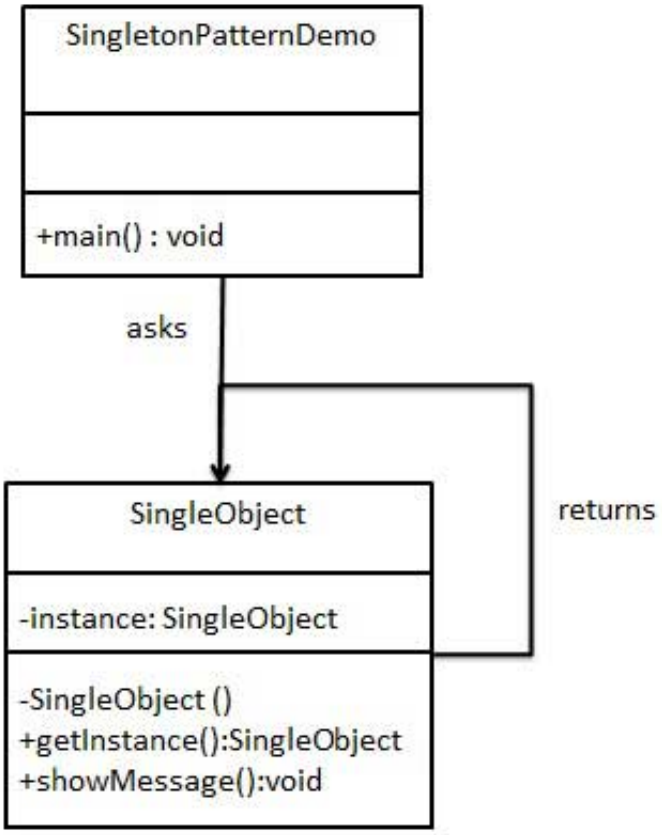
介绍

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
主要解决：一个全局使用的类频繁地创建与销毁。
何时使用：当您想控制实例数目，节省系统资源的时候。
如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。
关键代码：构造函数是私有的。
应用实例： 1、一个党只能有一个主席。 2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。 3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。
优点： 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。 2、避免对资源的多重占用（比如写文件操作）。
缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。
使用场景： 1、要求生产唯一序列号。 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。
注意事项：getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

实现

我们将创建一个 SingleObject 类。SingleObject 类有它的私有构造函数和本身的一个静态实例。

SingleObject 类提供了一个静态方法，供外界获取它的静态实例。SingletonPatternDemo，我们的演示类使用 SingleObject 类来获取 SingleObject 对象



setp1

创建一个Singleton类

SingleObject.java

```
public class SingleObject {

   //创建 SingleObject 的一个对象
   private static SingleObject instance = new SingleObject();

   //让构造函数为 private，这样该类就不会被实例化
   private SingleObject(){}

   //获取唯一可用的对象
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

setp2
从singleObject类获取唯一的对象
SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //不合法的构造函数
      //编译时错误：构造函数 SingleObject() 是不可见的
      //SingleObject object = new SingleObject();

      //获取唯一可用的对象
      SingleObject object = SingleObject.getInstance();

      //显示消息
      object.showMessage();
   }
}
```

setp3
验证输出

```
Hello world
```

单例模式的几种实现方式

懒汉式，线程不安全

- **描述**：这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。
  这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。
- **代码实例**：

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

    public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

懒汉式，线程安全

- **描述**：这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。
  优点：第一次调用才初始化，避免内存浪费。
  缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。
  getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。
- **代码实例**：

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}
    public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

饿汉式

- **描述**：这种方式比较常用，但容易产生垃圾对象。
  优点：没有加锁，执行效率会提高。
  缺点：类加载时就初始化，浪费内存。
  它基于 classloder 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。
- **代码实例**：

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
    return instance;
    }
}
```

双检锁/双重校验锁（DCL，即 double-checked locking）

- **描述**：这种方式采用双锁机制，安全且在多线程情况下能保持高性能。
  getInstance() 的性能对应用程序很关键。
- **代码实例**：

```
public class Singleton {
    private volatile static Singleton singleton;
    private Singleton (){}
    public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
        if (singleton == null) {
            singleton = new Singleton();
        }
        }
    }
    return singleton;
    }
}
```

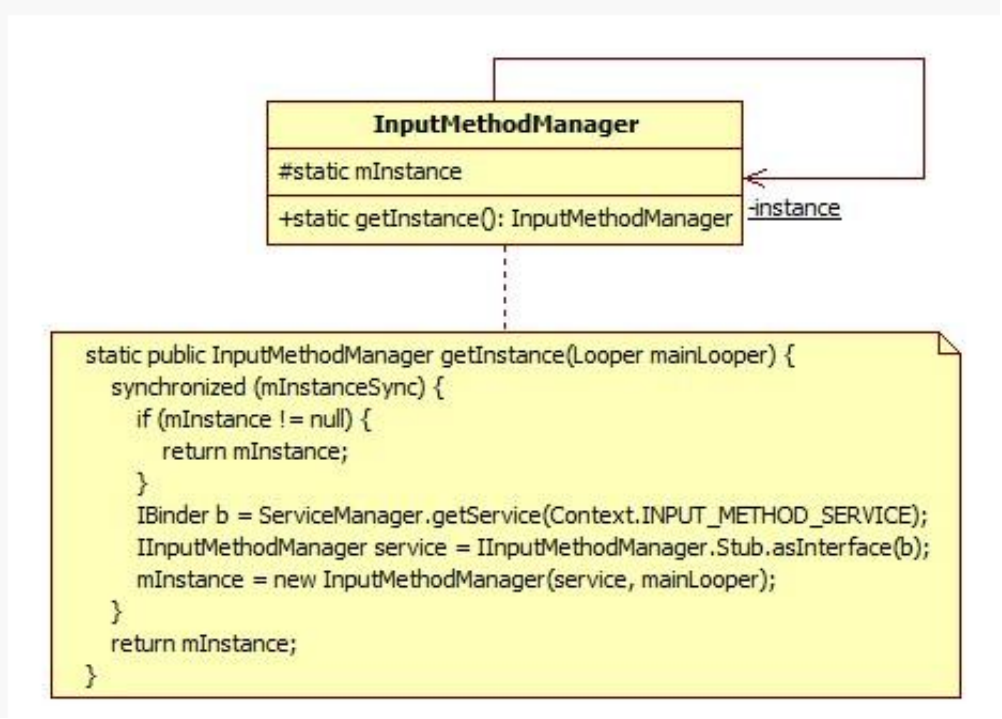
登记式/静态内部类

- **描述**：这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。
  这种方式同样利用了 classloder 机制来保证初始化 instance 时只有一个线程，它跟第 3 种方式不同的是：第 3 种方式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有显示通过调用 getInstance 方法时，才会显示装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第 3 种方式就显得很合理。
- **代码实例**：

```
public class Singleton {
    private static class SingletonHolder {
    private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
    return SingletonHolder.INSTANCE;
    }
}
```

Android实际开发中的运用

android中有很多系统级别的全局变量，如时间，输入法，账户，状态栏等等，android中对这些都直接或者有些间接用到了单例模式。
以输入法为例，把上图修改为实际情况：



非常的简单，但是有一点，从上面我们也看到了synchronized关键字，在多线程的环境下，单例模式为了保证自己实例数量的唯一，必然会做并发控制。

```
public final class InputMethodManager {
    static final Object mInstanceSync = new Object();//同步
    //内部全局唯一实例
    static InputMethodManager mInstance;

    //对外api
    static public InputMethodManager getInstance(Context context)
{
        return getInstance(context.getMainLooper());
    }

    /**
     * 内部api，供上面的外部api调用
     * @hide 系统隐藏的api
     */
    static public InputMethodManager getInstance(Looper
mainLooper) {
        synchronized (mInstanceSync) {
            if (mInstance != null) {
                return mInstance;
            }
            IBinder b =
ServiceManager.getService(Context.INPUT_METHOD_SERVICE);
            IInputMethodManager service =
IInputMethodManager.Stub.asInterface(b);
            mInstance = new InputMethodManager(service,
mainLooper);
        }
        return mInstance;
    }
}
```

客户端调用，比如contextimpl中的getSystemService()方法中如下调用：

```
class ContextImpl extends Context{
    @Override
    public Object getSystemService(String name) {
        if (WINDOW_SERVICE.equals(name)) {
            //... ... 省略下面n个if，else if
        } else if (INPUT_METHOD_SERVICE.equals(name)) {
            //获取输入法管理者唯一实例
            return InputMethodManager.getInstance(this);
        }  else if (KEYGUARD_SERVICE.equals(name)) {
             //... ... 省略下面n个if，else if
        } else if (ACCESSIBILITY_SERVICE.equals(name)) {
            //又见单例，无处不在
            return AccessibilityManager.getInstance(this);
        } else if (LOCATION_SERVICE.equals(name)) {
            //... ... 省略下面n个if，else if
        }  else if (NFC_SERVICE.equals(name)) {
            return getNfcManager();
        }
        return null;
    }
}
```