# 设计模式-单例模式

**1、简介**

**意图：**确保类只有一个实例，并提供全局访问。

**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。

**三要素：**

私有的构造方法；

指向自己实例的私有静态引用；

以自己实例为返回值的静态的公有方法。

**2、使用场景以及优缺点**

1）Windows的Task Manager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~

2）windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

3）网站的计数器，一般也是采用单例模式实现，否则难以同步。

4）应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。

5）Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。

6）数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。

7）多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。

8）操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。

9）HttpApplication 也是单位例的典型应用。熟悉ASP.Net(IIS)的整个请求生命周期的人应该知道HttpApplication也是单例模式，所有的HttpModule都共享一个HttpApplication实例.

总结以上，不难看出：

单例模式应用的场景一般发现在以下条件下：

1）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。

2）控制资源的情况下，方便资源之间的互相通信。如线程池等。

**优点：**  
    1.在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就 防止其它对象对自己的实例化，确保所有的对象都访问一个实例   
    2.单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。   
    3.提供了对唯一实例的受控访问。   
    4.由于在系统内存中只存在一个对象，因此可以 节约系统资源，当 需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。   
    5.允许可变数目的实例。   
    6.避免对共享资源的多重占用。   
**缺点：**  
    1.不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。   
    2.由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。   
    3.单例类的职责过重，在一定程度上违背了“单一职责原则”。   
    4.滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

**3、七种单例**

**1） 饿汉模式**

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

**2）懒汉模式（线程不安全）**

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

**3）懒汉模式（线程安全）**

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

这种写法能够在多线程中很好的工作，但是每次调用getInstance方法时都需要进行同步，造成不必要的同步开销，而且大部分时候我们是用不到同步的，所以不建议用这种模式。

**4）双重检查模式 （DCL）**

|  |
| --- |
| public class Singleton {  private volatile static Singleton singleton;  private Singleton (){  }  public static Singleton getInstance() {  if (instance== null) {  synchronized (Singleton.class) {  if (instance== null) {  instance= new Singleton();  }  }  }  return singleton;  }  } |

**5） 静态内部类单例模式**

|  |
| --- |
| public class Singleton {  private Singleton(){  }  public static Singleton getInstance(){  return SingletonHolder.sInstance;  }  private static class SingletonHolder {  private static final Singleton sInstance = new Singleton();  }  } |

**6） 枚举单例**

|  |
| --- |
| public enum Singleton {  INSTANCE;  public void doSomeThing() {  }  }  private Object readResolve() throws ObjectStreamException{  return singleton;  } |

**7） 使用容器实现单例模式**

|  |
| --- |
| public class SingletonManager {  　　private static Map<String, Object> objMap = new HashMap<String,Object>();  　　private Singleton() {  　　}  　　public static void registerService(String key, Objectinstance) {  　　　　if (!objMap.containsKey(key) ) {  　　　　　　objMap.put(key, instance) ;  　　　　}  　　}  　　public static Object getService(String key) {  　　　　return objMap.get(key) ;  　　}  } |

**4、单例在spring中应用**

spring的Bean默认的是单例的，Bean的作用域可以通过Bean标签的scope属性进行设置，Bean的作用域包括：  
默认情况下scope="singleton"，那么该Bean是单例，任何人获取该Bean实例的都为同一个实例；  
scope="prototype"，任何一个实例都是新的实例；  
scope="request"，在WEB应用程序中，每一个实例的作用域都为request范围；

scope="session",在WEB应用程序中，每一个实例的作用域都为session范围；

注意：在默认情况下，Bean实例在被Spring容器初始化的时候，就会被实例化，默认调用无参数的构造方法。在其它情况下，Bean将会在获取实例的时候才会被实例化。

在Spring中，bean可以被定义为两种模式：prototype（多例）和singleton（单例）

|  |
| --- |
| <bean id="hi" class="com.test.Hi" init-method="init" scope="singleton"> |

**Spring源码示例：**

public abstract class AbstractBeanFactory implements ConfigurableBeanFactory{

/\*\*

\* 充当了Bean实例的缓存，实现方式和单例注册表相同

\*/

private final Map singletonCache=new HashMap();

public Object getBean(String name)throws BeansException{

return getBean(name,null,null);

}

...

public Object getBean(String name,Class requiredType,Object[] args)throws BeansException{

//对传入的Bean name稍做处理，防止传入的Bean name名有非法字符(或则做转码)

String beanName=transformedBeanName(name);

Object bean=null;

//手工检测单例注册表

Object sharedInstance=null;

//使用了代码锁定同步块，原理和同步方法相似，但是这种写法效率更高

synchronized(this.singletonCache){

sharedInstance=this.singletonCache.get(beanName);

}

if(sharedInstance!=null){

...

//返回合适的缓存Bean实例

bean=getObjectForSharedInstance(name,sharedInstance);

}else{

...

//取得Bean的定义

RootBeanDefinition mergedBeanDefinition=getMergedBeanDefinition(beanName,false);

...

//根据Bean定义判断，此判断依据通常来自于组件配置文件的单例属性开关

//<bean id="date" class="java.util.Date" scope="singleton"/>

//如果是单例，做如下处理

if(mergedBeanDefinition.isSingleton()){

synchronized(this.singletonCache){

//再次检测单例注册表

sharedInstance=this.singletonCache.get(beanName);

if(sharedInstance==null){

...

try {

//真正创建Bean实例

sharedInstance=createBean(beanName,mergedBeanDefinition,args);

//向单例注册表注册Bean实例

addSingleton(beanName,sharedInstance);

}catch (Exception ex) {

...

}finally{

...

}

}

}

bean=getObjectForSharedInstance(name,sharedInstance);

}

//如果是非单例，即prototpye，每次都要新创建一个Bean实例

//<bean id="date" class="java.util.Date" scope="prototype"/>

else{

bean=createBean(beanName,mergedBeanDefinition,args);

}

}

...

return bean;

}

}

源码地址分析2：

https://www.jianshu.com/p/552a7e509afb