

昵称:清风崔

园龄:3年8个月

粉丝:15

关注:12

+加关注

< 2018年12月 >						
日	一	二	三	四	五	六
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

搜索

找找看

谷歌搜索

- 常用链接
- 我的随笔
- 我的评论
- 我的参与
- 最新评论
- 我的标签

- 最新随笔
1. 【JAVA集合】HashMap源码分析(转载)
2. Java单例模式深入详解
3. JetBrains WebStorm 注册码
4. 《Velocity java开发指南》中文版(下)转载
5. 《Velocity java开发指南》中文版(上)转载
6. 《VTL语法参考指南》中文版[转]
7. 《Velocity 模板使用指南》中文版[转载]
8. MYSQL常用命令集合(转载)
9. Spring(七)持久层
10. Spring(六)AOP切入方式

- 随笔分类(32)
- Ajax(2)
- Eclipse(1)
- Hadoop(1)
- Hibernate(2)
- Java(8)
- Linux(1)
- MySQL(1)
- Oracle
- Spring(11)
- Struts
- Velocity(4)

Java单例模式深入详解


原文地址:<http://www.cnblogs.com/hxsyl/>

仅作为笔记收藏.....


一.问题引入

偶然想想到的如果把Java的构造方法弄成private, 那里面的成员属性是不是只有通过static来访问呢;如果构造方法是private的话, 那么有什么好处呢;如果构造方法是private的话, 会不更好的封装该内呢?我主要是应用在使用普通类模拟枚举类型里, 后来发现这就是传说中的单例模式. 构造函数弄成private 就是单例模式, 即不想让别人用new 方法来创建多个对象, 可以在类里面先生成一个对象, 然后写一个public static方法把这个对象return出去. (eg:public 类名 getInstance(){return 你刚刚生成的那个类对象;}), 用static是因为你的构造函数是私有的, 不能产生对象, 所以只能用类名调用, 所有只能是静态函数.成员变量也可以写getter/setter供外界访问的.如果谁要用这个类的实例就用有兴趣的读者参看我的这一篇博文<http://www.cnblogs.com/hxsyl/archive/2013/03/18/2966360.html>.

第一个代码不是单例模式, 也就是说不一定只要构造方法是private的就是单例模式。



```
1 class A(){
2     private A(){}
3     public name;
4
5     pulbic static A creatInstance(){
6
7         return new A();
8     }
9
10 }
11
12 A a = A.createInstance();
13 a.name; //name 属性
```



按 Ctrl+C 复制代码

按 Ctrl+C 复制代码

二.单例模式概念及特点

java中单例模式是一种常见的设计模式, 单例模式分三种:懒汉式单例、饿汉式单例、登记式单例三种。

单例模式有一下特点:


- 1、单例类只能有一个实例。
- 2、单例类必须自己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。

单例模式确保某个类只有一个实例, 而且自行实例化并向整个系统提供这个实例。在计算机系统中, 线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机, 但只能有一个Printer Spooler, 以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口, 系统应当集中管理这些通信端口, 以避免一个通信端口同时被两个请求同时调用。总之, 选择单例模式就是为了避免不一致状态, 避免政出多头。

正是由于这个特点, 单例对象通常作为程序中的存放配置信息的载体, 因为它能保证其他对象读到一致的信息。例如在某个服务器程序中, 该服务器的配置信息可能存放在数据库或 文件中, 这些配置数据由某个单例对象统一读取, 服务进程中的其他对象如果要获取这些配置信息, 只需访问该单例对象即可。这种方式极大地简化了在复杂环境 下, 尤其是多线程环境下的配置管理, 但是随着应用场景的不同, 也可能带来一些同步问题。

三.典型例题

首先看一个经典的单例实现。



```
1 public class Singleton {
2
3     private static Singleton uniqueInstance = null;
4
5
6     private Singleton() {
7
8         // Exists only to defeat instantiation.
9
10    }
11
12
13
14
15     public static Singleton getInstance() {
16
17         if (uniqueInstance == null) {
18
19             uniqueInstance = new Singleton();
20
21         }
22
23         return uniqueInstance;
24
25     }
26
27     // Other methods...
```

设计模式(1)
随笔档案(33)
2016年3月 (2)
2015年8月 (1)
2015年5月 (5)
2015年4月 (25)

积分与排名
积分 - 13628
排名 - 35048

最新评论
1. Re: eclipse的使用、优化配置
第一次上这个，不知道能不能mark
--大宝键
2. Re: Java学习方法
不错，学习了
--莫问天机
3. Re:【转】Hibernate和ibatis的比较
能出活的就是好东西
--园林鸟

阅读排行榜
1. Ajax的原理和运行机制(3651)
2. Java学习方法(3313)
3. JetBrains WebStorm 注册码(2338)
4. Java单例模式深入详解(1226)
5. java面试题及答案(基础题122道，代码题19道)(469)

评论排行榜
1. 【转】Hibernate和ibatis的比较(1)
2. eclipse的使用、优化配置(1)
3. Java学习方法(1)

推荐排行榜
1. Java学习方法(3)
2. Java单例模式深入详解(1)
3. java面试题及答案(基础题122道，代码题19道)(1)
4. MYSQL常用命令集合(转载)(1)

28
29 }


Singleton通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton的唯一实例只能通过getInstance()方法访问。(事实上，通过Java反射机制是能够实例化构造方法为private的类的，那基本上会使所有的Java单例实现失效。此问题在此处不做讨论，姑且掩耳盗铃地认为反射机制不存在。)

但是以上实现没有考虑线程安全问题。所谓线程安全是指：如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。或者说：一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性，也就是说我们不用考虑同步的问题。显然以上实现并不满足线程安全的要求，在并发环境下很可能会出现多个Singleton实例。



```
1 public class TestStream {
2     private String name;
3     public String getName() {
4         return name;
5     }
6     public void setName(String name) {
7         this.name = name;
8     }
9     //该类只能有一个实例
10    private TestStream() {} //私有无参构造方法
11    //该类必须自行创建
12    //有2种方式
13    /*private static final TestStream ts=new TestStream();*/
14    private static TestStream ts1=null;
15    //这个类必须自动向整个系统提供这个实例对象
16    public static TestStream getTest() {
17        if(ts1==null){
18            ts1=new TestStream();
19        }
20        return ts1;
21    }
22    public void getInfo(){
23        System.out.println("output message "+name);
24    }
25 }
```





```
1 public class TestMain {
2     public static void main(String [] args){
3         TestStream s=TestStream.getTest();
4         s.setName("张孝祥");
5         System.out.println(s.getName());
6         TestStream s1=TestStream.getTest();
7         s1.setName("张孝祥");
8         System.out.println(s1.getName());
9         s.getInfo();
10        s1.getInfo();
11        if(s==s1){
12            System.out.println("创建的是同一个实例");
13        }else if(s!=s1){
14            System.out.println("创建的不是同一个实例");
15        }else{
16            System.out.println("application error");
17        }
18    }
19 }
```



运行结果：

张孝祥
张孝祥
output message 张孝祥
output message 张孝祥
创建的是同一个实例

结论：由结果可以得知单例模式为一个面向对象的应用程序提供了对象惟一的访问点，不管它实现何种功能，整个应用程序都会同享一个实例对象。

其次，下面是单例的三种实现。

1. 饿汉式单例类

飞哥下面这个可以不加final，因为静态方法只在编译期间执行一次初始化，也就是只会会有一个对象。



```
1 //饿汉式单例类，在类初始化时，已经自行实例化
2 public class Singleton1 {
3     //私有的默认构造子
4     private Singleton1() {}
5     //已经自行实例化
6     private static final Singleton1 single = new Singleton1();
7     //静态工厂方法
8     public static Singleton1 getInstance() {
9         return single;
10    }
11 }
```



2. 懒汉式单例类

那个if判断确保对象只创建一次。



```
1 //懒汉式单例类，在第一次调用的时候实例化
2 public class Singleton2 {
3     //私有的默认构造子
4     private Singleton2() {}
5     //注意，这里没有final
6     private static Singleton2 single=null;
7     //静态工厂方法
8     public synchronized static Singleton2 getInstance() {
9         if (single == null) {
10            single = new Singleton2();
11        }
12    }
13 }
```

```
11     }
12     return single;
13 }
14 }
```

3.登记式单例类

```
1 import java.util.HashMap;
2 import java.util.Map;
3 //登记式单例类。
4 //类似Spring里面的方法，将类名注册，下次从里面直接获取。
5 public class Singleton3 {
6     private static Map<String,Singleton3> map = new HashMap<String,Singleton3>();
7     static{
8         Singleton3 single = new Singleton3();
9         map.put(single.getClass().getName(), single);
10    }
11    //保护的默认构造子
12    protected Singleton3(){}
13    //静态工厂方法，返还此类惟一的实例
14    public static Singleton3 getInstance(String name) {
15        if(name == null) {
16            name = Singleton3.class.getName();
17            System.out.println("name == null"+"--->name="+name);
18        }
19        if(map.get(name) == null) {
20            try {
21                map.put(name, (Singleton3) Class.forName(name).newInstance());
22            } catch (InstantiationException e) {
23                e.printStackTrace();
24            } catch (IllegalAccessException e) {
25                e.printStackTrace();
26            } catch (ClassNotFoundException e) {
27                e.printStackTrace();
28            }
29        }
30        return map.get(name);
31    }
32    //一个示意性的商业方法
33    public String about() {
34        return "Hello, I am RegSingleton.";
35    }
36    public static void main(String[] args) {
37        Singleton3 single3 = Singleton3.getInstance(null);
38        System.out.println(single3.about());
39    }
40 }
```

四.单例对象作配置信息管理时可能会带来的几个同步问题

- 1.在多线程环境下，单例对象的同步问题主要体现在两个方面，单例对象的初始化和单例对象的属性更新。

本文描述的方法有如下假设：

- a. 单例对象的属性(或成员变量)的获取，是通过单例对象的初始化实现的。也就是说，在单例对象初始化时，会从文件或数据库中读取最新的配置信息。
- b. 其他对象不能直接改变单例对象的属性，单例对象属性的变化来源于配置文件或配置数据库数据的变化。

1.1单例对象的初始化

首先，讨论一下单例对象的初始化同步。单例模式的通常处理方式是，在对象中有一个静态成员变量，其类型就是单例类型本身；如果该变量为null，则创建该单例类型的对象，并将该变量指向这个对象；如果该变量不为null，则直接使用该变量。

这种处理方式在单线程的模式下可以很好的运行；但是在多线程模式下，可能产生问题。如果第一个线程发现成员变量为null，准备创建对象；这是第二个线程同时也发现成员变量为null，也会创建新对象。这就会造成在一个JVM中有多个单例类型的实例。如果这个单例类型的成员变量在运行过程中变化，会造成多个单例类型实例的不一致，产生一些很奇怪的现象。例如，某服务进程通过检查单例对象的某个属性来停止多个线程服务，如果存在多个单例对象的实例，就会造成部分线程服务停止，部分线程服务不能停止的情况。

1.2单例对象的属性更新

通常，为了实现配置信息的实时更新，会有一个线程不停检测配置文件或配置数据库的内容，一旦发生变化，就更新到单例对象的属性中。在更新这些信息的时候，很可能还会有其他线程正在读取这些信息，造成意想不到的后果。还是以通过单例对象属性停止线程服务为例，如果更新属性时读写不同步，可能访问该属性时这个属性正好为空(null)，程序就会抛出异常。

下面是解决方法

```
1 //单例对象的初始化同步
2 public class GlobalConfig {
3     private static GlobalConfig instance = null;
4     private Vector properties = null;
5     private GlobalConfig() {
6         //Load configuration information from DB or file
7         //Set values for properties
8     }
9     private static synchronized void syncInit() {
10        if (instance == null) {
11            instance = new GlobalConfig();
12        }
13    }
14    public static GlobalConfig getInstance() {
15        if (instance == null) {
16            syncInit();
17        }
18        return instance;
19    }
20    public Vector getProperties() {
21        return properties;
22    }
23 }
```

这种处理方式虽然引入了同步代码，但是因为这段同步代码只会在最开始的时候执行一次或多次，所以对整个系统的性能不会有影响。

单例对象的属性更新同步。

参照读者/写者的处理方式，设置一个读计数器，每次读取配置信息前，将计数器加1，读完后将计数器减1。只有在读计数器为0时，才能更新数据，同时要阻塞所有读属性的调用。


代码如下：




```
1 public class GlobalConfig {
2     private static GlobalConfig instance;
3     private Vector properties = null;
4     private boolean isUpdating = false;
5     private int readCount = 0;
6     private GlobalConfig() {
7         //Load configuration information from DB or file
8         //Set values for properties
9     }
10    private static synchronized void syncInit() {
11        if (instance == null) {
12            instance = new GlobalConfig();
13        }
14    }
15    public static GlobalConfig getInstance() {
16        if (instance==null) {
17            syncInit();
18        }
19        return instance;
20    }
21    public synchronized void update(String p_data) {
22        syncUpdateIn();
23        //Update properties
24    }
25    private synchronized void syncUpdateIn() {
26        while (readCount > 0) {
27            try {
28                wait();
29            } catch (Exception e) {
30            }
31        }
32    }
33    private synchronized void syncReadIn() {
34        readCount++;
35    }
36    private synchronized void syncReadOut() {
37        readCount--;
38        notifyAll();
39    }
40    public Vector getProperties() {
41        syncReadIn();
42        //Process data
43        syncReadOut();
44        return properties;
45    }
46 }
```



采用"影子实例"的办法具体说，就是在更新属性时，直接生成另一个单例对象实例，这个新生成的单例对象实例将从数据库或文件中读取最新的配置信息；然后将这些配置信息直接赋值给旧单例对象的属性。



```
1 public class GlobalConfig {
2     private static GlobalConfig instance = null;
3     private Vector properties = null;
4     private GlobalConfig() {
5         //Load configuration information from DB or file
6         //Set values for properties
7     }
8     private static synchronized void syncInit() {
9         if (instance = null) {
10             instance = new GlobalConfig();
11         }
12     }
13     public static GlobalConfig getInstance() {
14         if (instance = null) {
15             syncInit();
16         }
17         return instance;
18     }
19     public Vector getProperties() {
20         return properties;
21     }
22     public void updateProperties() {
23         //Load updated configuration information by new a GlobalConfig object
24         GlobalConfig shadow = new GlobalConfig();
25         properties = shadow.getProperties();
26     }
27 }
```



注意：在更新方法中，通过生成新的GlobalConfig的实例，从文件或数据库中得到最新配置信息，并存放放到properties属性中。上面两个方法比较起来，第二个方法更好，首先，编程更简单；其次，没有那么多的同步操作，对性能的影响也不大。

分类：[设计模式](#)

好文要顶

关注我

收藏该文







清风崔
关注 - 12
粉丝 - 15

[+加关注](#)


1

推荐

0

反对

[刷新评论](#) [刷新页面](#) [返回顶部](#)

 [注册用户](#)登录后才能发表评论, 请 [登录](#) 或 [注册](#), [访问网站首页](#)。

相关博文:

- [java模式:深入单例模式](#)
- [Java 单例模式详解](#)
- [单例模式详解](#)
- [【JAVA单例模式详解】](#)
- [JAVA单例模式](#)

最新新闻:

- [争议巨人史玉柱:不掺和互联网风口 曾靠脑白金救急](#)
 - [无处不在的“扫码红包”, 究竟是谁的“照妖镜”](#)
 - [被扔石头、扎轮胎、拔枪相向, 谷歌无人车遭过街喊打](#)
 - [巨头混战、水土不服 奈飞神话要在印度破灭了?](#)
 - [SQLite被曝存在漏洞 所有Chromium浏览器受影响](#)
- » [更多新闻...](#)