链滴

设计模式 -- 单例模式（Singleton pattern）及应用

作者：james

原文链接：https://ld246.com/article/1515032436626

来源网站：链滴

<h2 id="单例模式">单例模式</h2>

参考文档:

菜鸟教程单例模式介绍了单例模式的 6 种实现
该文仅介绍 spring 的单例模式：spring 的单例模式
介绍原理：Spring 的单例模式底实现
参考书籍：漫谈设计模式：从面向对象开始-刘济华.pdf

<h3 id="1--单例模式解析">1. 单例模式解析</h3>

单例所指的就是单个实例，也就是说要保证一个类仅有一个实例。
单例模式有以下的特点:

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

<h3 id="2--单例模式示例">2. 单例模式示例</h3>

以下示例都是在同一个 JVM 中实现的，在分布式环境下如何保证分布在不同机器上的整个应用有一个实例是更复杂的话题。

<h4 id="2-1-简单单例模式">2.1 简单单例模式</h4>

为了实现上面的特点，有以下基本实现：

<pre><code class="highlight-chroma">/**
 * 单例模式示例:
 * - 简单单例模式
 */
public class Singleton{
    //变量是私有的,界无法访问
    private static Singleton singleton = new Singleton();
    //other fields...

    //Singleton类有一个构造方法，被private修饰的， 客户对象无法创建该类实例。
    private Singleton(){

```
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //实现全局访问

</span></span><span class="highlight-line"><span class="highlight-cl">    public static Si
gleton getInstance(){
</span></span><span class="highlight-line"><span class="highlight-cl">        return single
on;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    //other mothed
...
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<ul>
<li>如果该实例需要比较复杂的初始化过程时，可以把这个过程应该写在 static{……}代码块中</li>
<li>此实现是线程安全的，当多个线程同时去访问该类的 getInstance（） 方法时，不会初始化多
不同的对象，这是因为，JVM 在加载此类时，对于 static 属性的初始化只能由一个线程执行且仅一
。</li>
</ul>
<h4 id="2-2-延迟创建的单例模式">2.2 延迟创建的单例模式</h4>
<ul>
<li>上面的单例模式会在类加载的时候就初始化实例，如果这样的单例较多，会使程序初始化效率低
。</li>
<li>为了在类的实例第一次使用时才初始化，我们可以把单例的实例化过程放到 getInstance（）方
中，而不在加载类时预先创建。</li>
</ul>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl"> * 单例模式示例:
</span></span><span class="highlight-line"><span class="highlight-cl"> * - 延迟创建的单
模式
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span><span class="highlight-line"><span class="highlight-cl">public class UnThr
adSafeSingleton{
</span></span><span class="highlight-line"><span class="highlight-cl">    //变量是私有的,
界无法访问
</span></span><span class="highlight-line"><span class="highlight-cl">    private static U
ThreadSafeSingleton singleton = null;
</span></span><span class="highlight-line"><span class="highlight-cl">    //other fields...
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //UnThreadSaf
Singleton类只有一个构造方法，被private修饰的， 客户对象无法创建该类实例。
</span></span><span class="highlight-line"><span class="highlight-cl">    private UnThre
dSafeSingleton(){
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //实现全局访问

</span></span><span class="highlight-line"><span class="highlight-cl">    public static Un
hreadSafeSingleton getInstance(){
</span></span><span class="highlight-line"><span class="highlight-cl">        //在这里实现
迟创建，但是下面的三行不是线程安全的，
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">        //在高并发环
下会生成多个不同的该类实例
</span></span><span class="highlight-line"><span class="highlight-cl">        //在没有自动
存回收机制的语言平台中，容易内存泄露
</span></span><span class="highlight-line"><span class="highlight-cl">        if(singleton
=null){
</span></span><span class="highlight-line"><span class="highlight-cl">            singleton
 new UnThreadSafeSingleton();
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        return single
on;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    //other mothed
...
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span></code></pre>
<ul>
<li>为了解决上面线程安全的问题，我们可以给此方法加 synchronized 关键字，来保证多线程访问只会实例化一次。</li>
</ul>
<h4 id="Double-Check-Locking"><strong>Double-Check Locking</strong></h4>
<ul>
<li>为了提高并发性能，可以只对上面导致线程不安全的三行代码进行同步</li>
<li>参考：<a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.cnblogs.com%2Figongsi%2Farchive%2F2012%2F04%2F01%2F2429166.html" target="_blank" rel="nofollow uc">java 中 volatile 关键字的含义</a></li>
</ul>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl"> * 单例模式示例：
</span></span><span class="highlight-line"><span class="highlight-cl"> * - 线程安全的延
创建的单例模式
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span><span class="highlight-line"><span class="highlight-cl">public class Doubl
CheckSingleton{
</span></span><span class="highlight-line"><span class="highlight-cl">    //变量是私有的,
界无法访问
</span></span><span class="highlight-line"><span class="highlight-cl">    //volatile具有sy
chronized的可见性特点，也就是说线程能够自动发现volatile变量的最新值。
</span></span><span class="highlight-line"><span class="highlight-cl">    //这样，如果ins
atnce实例化成功，其他线程便能立即发现
</span></span><span class="highlight-line"><span class="highlight-cl">    private volatile
tatic DoubleCheckSingleton singleton = null;
</span></span><span class="highlight-line"><span class="highlight-cl">    //other fields...
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //DoubleCheckS
ngleton类只有一个构造方法，被private修饰的， 客户对象无法创建该类实例。
</span></span><span class="highlight-line"><span class="highlight-cl">    private Double
heckSingleton(){
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //实现全局访问
```

```
    public static DoubleCheckSingleton getInstance(){
        //两次检查是了防止两个线程都进行到同步块的时候，可能会创建两个实例
        if(singleton=null){
            synchronized (DoubleCheckSingleton.class){
                if(singleton ==null){ //这里如果不检查还是会生成两个实例
                    singleton = new DoubleCheckSingleton();
                }
            }
        return singleton;
    }
    //other mothed...
}

```

2.3 Initialization on demond holder

- 另一种实现延迟加载且线程安全的实例化方法

```
/**
 * 单例模式示例:
 * - 延迟加载单例式
 */
public class LazyLoadedSingleton{
    //other fields...

    //LazyLoadedSingleton类只有一个构造方法，被private修饰的， 客户对象无法创建该类实例。
    private LazyLoadedSingleton(){

    }

    private static class LazyHolder{
        //变量是私有,外界无法访问
        //holds zhe sngleton class
        private static LazyLoadedSingleton singleton = new LazyLoadedSingleton();

    }
```

    /**
     * 只有当第一次用该方法时，JVM才会加载LazyHolder类，然后才初始化static的singleton
     * 这样即保证了程安全，又实现了延迟加载
     */
    public static LazyLoadedSingleton getInstance(){
        return LazyHolder.singleton;
    }
    //other mothed ...
}

</code></pre>
<h3 id="2--spring的单例模式">2. spring 的单例模式</h3>

参考文档：

该文仅介绍 spring 的单例模式：spring 的单例模式
介绍原理：Spring 的单例模式底实现
对 spring 中单例模式线程安全的讨：Spring 中 Singleton 模式的线程安全
针对上面的单例模式有以下说法：

延迟加载的单例模式可以称为懒汉式单例
非延迟加载的称为饿汉式单例

饿汉式单例在自己被加载时就将自己实例化，如果从资源利用效率角度来讲，比懒汉式单例类稍些。但是从速度和反应时间角度来讲，则比懒汉式要稍好些。

解释参考文档上的一句话：由于构造函数是私有的，因此该类不能被继承

参考：There is o default constructor available in xxx 错误引发

上面的单例模式都不能被继承，所以 spring 采用的是另一种单例模式，称为单例注册表

<h4 id="2-1-单例注册表方法">2.1 单例注册表方法</h4>

Spring 对单例的底层实现，到底是饿汉式单例还是懒汉式单例呢？呵呵，都不是。Spring 框架单例的支持是采用单例注册表的方式进行实现的。

以下参考文章分析的源代码应该是 spring 比较老的版本，我参考现在比较新的 4.3.2 版本，有

不同，但是原理应该是一样的。下面的源代码是 spring-beans-4.3.2-REALEASE 版本的。
详情参考：<a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.cnblogs.co
%2Flixuwu%2Fp%2F5676119.html" target="_blank" rel="nofollow ugc">Spring 的单例模式底
实现
Spring 对 bean 实例的创建是采用单例注册表的方式进行实现的，而这个注册表的缓存是 Hash
ap 对象，如果配置文件中的配置信息不要求使用单例，Spring 会采用新建实例的方式返回对象实例</
i>

<pre><code class="highlight-chroma"><span class="highlight
cl">public abstract class AbstractBeanFactory extends FactoryBeanRegistrySupport implemen
s ConfigurableBeanFactory {

.....
 public Object get
ean(String name) throws BeansException {
  return this.d
GetBean(name, (Class)null, (Object[])null, false);
  }
.....
 protected <T&
t; T doGetBean(String name, Class<T> requiredType, final Object[] args, boolean typeCh
ckOnly) throws BeansException {
  //对传入的Be
n name稍做处理，防止传入的Bean name名有非法字符(或者做转码)
  final String b
anName = this.transformedBeanName(name);
  /** Cache of s
ngleton objects: bean name --> bean instance */
  /**
  * private fina
 Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object&gt
(256);
  */
  Object share
Instance = this.getSingleton(beanName);

  Object bean;
  //从单例注册
中查找，如果存在，就获得了单例
  if(sharedInst
nce != null && args == null) {
  .....
  bean = this.
etObjectForBeanInstance(sharedInstance, name, beanName, (RootBeanDefinition)null);
  } else {
  .....
  //否则
  try {
  //取得b
an的定义
  final Ro
tBeanDefinition ex1 = this.getMergedLocalBeanDefinition(beanName);

```
                ......
                //如果单例，做如下处理
                if(ex1.isingleton()) {
                    shareInstance = this.getSingleton(beanName, new ObjectFactory() {
                        ......
                    );
                    bean = this.getObjectForBeanInstance(sharedInstance, name, beanName, ex1);
                } else if(x1.isPrototype()) {
                ......
                //如果是rototype。创建一个bean
                Object ar25;
                    try {
                        this.beforePrototypeCreation(beanName);
                        var5 = this.createBean(beanName, ex1, args);
                    } finaly {
                        this.afterPrototypeCreation(beanName);
                    }
                    ......
                    bean = this.getObjectForBeanInstance(var25, name, beanName, ex1);
                } else {
                ......
                bean = his.getObjectForBeanInstance(var28, name, beanName, ex1);
                }
            } catch (BensException var23) {
                this.cleaupAfterBeanCreationFailure(beanName);
                throw va23;
            }
        }
        if(requiredTye != null &amp;&amp; bean != null &amp;&amp; !requiredType.isAssignableFrom(bean.getClss())) {
            ......
        } else {
            return bea;
        }

}
```

</code></pre>