单例模式的终极实现方案

 G9YH 发布于 2017-10-14

单例模式（Singleton）是一种使用率非常高的设计模式，其主要目的在于保证某一类在运行期间仅被创建一个实例，并为该实例提供了一个全局访问方法，通常命名为getInstance()方法。单例模式的本质简言之即是：

控制实例数目

以Java为例，单例模式通常可分为**饿汉式**和**懒汉式**两种常规实现方式

饿汉式单例实现

饿汉式顾名思义，就是对类实例（食物？）的需求非常强烈，因此，在装载该单例类的时候就会创建类实例。如下

```
public class Singleton {
    /**
     * 装载时即创建类实例，并保存在类变量instance中
     * 加上static关键词使得该变量能在getInstance()静态方法中使用
     */
    private static Singleton instance = new Singleton();

    /**
     * 私有化构造方法，使外部无法通过构造方法构造除instance外的类实例
     * 从而达到单例模式控制类实例数目的目的
     */
    private Singleton() {
    }

    /**
     * 类实例的全局访问方法
     * 加上static关键词使得外部可以通过类名直接调用该方法获取类实例
     * @return 单例类实例
     */
    public static Singleton getInstance() {
        //  由于类实例在类装载时已被创建并保存在instance中，因此可直接返回
```

```
    }
}
```

事实上，在Android开发中，Android Studio提供了一个直接创建单例类的功能（File->new->Singleton），该功能自动生成的单例类正是采用了饿汉式的实现方式

懒汉式单例实现

说到懒，我们自然而然会想到拖延症这一恶习，这一点和懒汉式的单例实现方式相似，这一实现方式会一直等到真正需要使用对象实例的时候再去创建该实例。如下

```
     * 装载时不创建类实例，但需要利用一个类变量去保存后续创建的类实例
     * 添加static关键词使得该变量能在getInstance()静态方法中使用
     */
    private static Singleton instance = null;

    /**
     * 私有化构造方法，使外部无法通过构造方法构造除instance外的类实例
     * 从而达到单例模式控制类实例数目的目的
     */
    private Singleton() {
    }

    /**
     * 类实例的全局访问方法
     * 添加static关键词使得外部可以通过类名直接调用该方法获取类实例
     * @return 单例类实例
     */
    public static Singleton getInstance() {
        //  如果instance未被初始化，则初始化该类实例
        if (instance == null) {
            instance = new Singleton();
        }

        return instance;
    }
}
```

事实上，虽然我们前面拿拖延症来与懒汉式做类比，但懒汉式的拖延却是实际开发中的一种较为常见的节省资源的方式，即**延迟加载**思想。这一思想的核心在于直到需要使用某些资源或数据时再去加载该资源或获取该数据，这样可以尽可能地节省使用前的内存空间

不难分析出，当外部多个线程同时想要获取单例类实例时，上述懒汉式实现方式便很容易导致并发问题。通常有如下几种改进方式

添加synchronized关键词

```
....
public static synchronized Singleton getInstance() {
....
```

这种改进方式是最简单的，但由于外部每次调用getInstance()方法时均需进行判断，因此该方式也是效率较低的

利用双重检查加锁机制

双重检查加锁机制分为如下两重检查

- 在程序每次调用getInstance()方法时先不进行同步，而是在进入该方法后再去检查类实例是否存在，若不存在则进入接下来的同步代码块
- 进入同步代码块后将再次检查类实例是否存在，若不存在则创建一个新的实例

这样一来，就只需要在类实例初始化时进行一次同步判断即可，而非每次调用getInstance()方法时都进行同步判断，大大节省了时间，具体实现如下

```
     * 从而达到单例模式控制类实例数目的目的
     */
    private Singleton() {
    }

    /**
     * 类实例的全局访问方法
     * 添加static关键词使得外部可以通过类名直接调用该方法获取类实例
     * @return 单例类实例
     */
    public static Singleton getInstance() {

        //  第一重检查：如果instance未被初始化，则进入同步代码块
        if (instance == null) {
            //  同步代码块，保证线程安全
            synchronized (Singleton.class) {
                //  第二重检查：如果instance未被初始化，则初始化该类实例
                if (instance == null) {
                    instance = new Singleton();
                }
```

```
        return instance;
    }
}
```

利用Java缓存思想实现的单例实现

```
    private static Map<String, Singleton> map = new HashMap<>();

    /**
     * 私有化构造方法，使外部无法通过构造方法构造除instance外的类实例
     * 从而达到单例模式控制类实例数目的目的
     */
    private Singleton() {
    }

    /**
     * 类实例的全局访问方法
     * 添加static关键词使得外部可以通过类名直接调用该方法获取类实例
     * @return 单例类实例
     */
    public static Singleton getInstance() {
        //  尝试从缓存容器中获取类实例
        Singleton instance = map.get(KEY);
        //  未能获取类实例，则初始化该实例，并将其缓存至容器中
        if (instance == null) {
            instance = new Singleton();
            map.put(KEY, instance);
        }

        return instance;
    }
}
```

上述实现方式暂未考虑线程安全问题。事实上，利用缓存来实现的单例模式其最大的优点在于对单例模式进行扩展。我们自然而然地可以想到这么一种情况，既然在实际开发中经常需要保证某个类只能被创建一个实例，那么，会不会出现保证某个类只能被创建两个或多个实例这种需求呢？对于这项需求，我们首先可以想到，上述实现方式中所建立的缓存容器是可以存储多个类实例的，利用这一特点，只需考虑一个问题，即外部调用时到底需要为其返回哪一个实例，便可实现**“双例模式”**以及**“多例模式”**（原谅我为它们取了一些奇怪的名字）了，具体实现如下

```
    private Singleton() {
    }
```

```
     * @return 单例类实例
     */
    public static Singleton getInstance() {
        //  尝试从缓存容器中获取第index个类实例
        String key = KEY + index;
        Singleton instance = map.get(key);
        //  未能获取类实例，则初始化该实例，并将其缓存至容器相应index中
        if (instance == null) {
            instance = new Singleton();
            map.put(key, instance);
        }

        //  这里以最基本的顺序调用为例，其他复杂调度方式不加讨论，具体调用方式如下
        //  index++，以在下一次调用中获取下一个类实例，当达到类实例数上限时，重新获取第一个
        if ((++index) > MAX) {
            index = 1;
        }

        return instance;
    }
```

单例模式的最佳实现

综合而言，上述实现方式都或多或少地存在诸如线程不安全、无法做到延迟加载等小缺陷。这里给出一个可以称得上完美的最佳解决方案

Lazy Initialization Holder Class 模式

这一方案的核心在于Java的**类级内部类**（即使用static关键词修饰的内部类，否则称之为对象级内部类）以及**多线程缺省同步锁**，先来看看具体实现

```
public class Singleton {
    /**
     * 类级内部类，用于缓存类实例
     * 该类将在被调用时才会被装载，从而实现了延迟加载
     * 同时由于instance采用静态初始化的方式，因此JVM能保证其线程安全性
     */
    private static class Instance {
        private static Singleton instance = new Singleton();
    }

    /**
     * 私有化构造方法，使外部无法通过构造方法构造除instance外的类实例
```

```
    private Singleton() {
    }

    /**
     * 类实例的全局访问方法
     * 添加static关键词使得外部可以通过类名直接调用该方法获取类实例
     * @return 单例类实例
     */
    public static Singleton getInstance() {
        return Instance.instance;
    }
}
```

在前面提到的饿汉式实现方式中，我们利用Java的静态初始化、借由JVM实现了线程安全，因此这里同样采用了这种方式。而另一方面，为了避免饿汉式实现中无法进行延迟加载的缺陷，我们

断资源等优势，可以说是单例模式的最佳实现了

java android 设计模式 单例模式 线程安全

阅读 4.5k • 发布于 2017-10-14

G9YH

程序员小菜鸡的成长日记

43 声望 **4** 粉丝

关注作者

撰写评论 ...

提交评论

鲸鱼与巨浪：不是编译Instance 的时候就创建了Singleton实例吗?

• 回复 • 2021-10-24

继续阅读

解决ScrollView嵌套RecyclerView的显示及滑动问题

项目中时常需要实现在ScrollView中嵌入一个或多个RecyclerView。这一做法通常会导致如下几个问题 页面...

G9YH 赞 14 阅读 70.2k 评论 4

重写GridView实现仿今日头条的频道编辑页(1)

本文旨在通过重写GridView，配合系统弹窗实现仿今日头条的频道编辑页面 注：由于代码稍长，本文仅列出...

G9YH 阅读 1.9k

单例模式

1. 什么是单例 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。适用于： 当类只能有一个实例而...

martinwangjun 赞 1 阅读 1k

单例模式

单例模式，是一种常用的软件设计模式，在它的核心结构中只包含一个被称为单例的特殊类，通过单例模式...

java部落 赞 1 阅读 830

单例模式

定义：确保一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。说白了就是：就算你重复调...

不会写jsx的辣鸡 阅读 332

单例模式

以上代码中instance作为类变量在初始化的过程中会被收集进<clinit>()方法中，该方法能够确保同步，...

Smile3k 阅读 1.1k

3 4 1

单例模式的应用场景： 注册表对象 日志对象 为什么要使用单例： 防止资源使用过度 程序运行结果出现不一...

Gatlin 阅读 677

单例模式

简单说几句 本文首发公众号【一名打字员】 说起这个单例，简直是人名中的张伟、代码圈中的八阿哥，已经...

wslongchen 阅读 1k