43讲单例模式（下）：如何设计实现一个集群环境下的分布式单例模式

上两节课中，我们针对单例模式，讲解了单例的应用场景、几种常见的代码实现和存在的问题，并粗略给出了替换单例模式的方法，比如工厂模式、IOC容器。今天，我们再进一步扩展延伸一下，一块讨论一下下面这几个问题：

- 如何理解单例模式中的唯一性?
- 如何实现线程唯一的单例?
- 如何实现集群环境下的单例?
- 如何实现一个多例模式?

今天的内容稍微有点“烧脑”，希望你在看的过程中多思考一下。话不多说，让我们正式开始今天的学习吧！

如何理解单例模式中的唯一性?

首先，我们重新看一下单例的定义：“一个类只允许创建唯一一个对象（或者实例），那这个类就是一个单例类，这种设计模式就叫作单例设计模式，简称单例模式。”

定义中提到，“一个类只允许创建唯一一个对象”。那对象的唯一性的作用范围是什么呢？是指线程内只允许创建一个对象，还是指进程内只允许创建一个对象？答案是后者，也就是说，单例模式创建的对象是进程唯一的。这里有点不好理解，我来详细地解释一下。

我们编写的代码，通过编译、链接，组织在一起，就构成了一个操作系统可以执行的文件，也就是我们平时所说的“可执行文件”（比如Windows下的exe文件）。可执行文件实际上就是代码被翻译成操作系统可理解的一组指令，你完全可以简单地理解为就是代码本身。

当我们使用命令行或者双击运行这个可执行文件的时候，操作系统会启动一个进程，将这个执行文件从磁盘加载到自己的进程地址空间（可以理解操作系统为进程分配的内存存储区，用来存储代码和数据）。接着，进程就一条一条地执行可执行文件中包含的代码。比如，当进程读到代码中的User user = new User();这条语句的时候，它就在自己的地址空间中创建一个user临

时变量和一个User对象。

进程之间是不共享地址空间的，如果我们在一个进程中创建另外一个进程（比如，代码中有一个fork()语句，进程执行到这条语句的时候会创建一个新的进程），操作系统会给新进程分配新的地址空间，并且将老进程地址空间的所有内容，重新拷贝一份到新进程的地址空间中，这些内容包括代码、数据（比如user临时变量、User对象）。

所以，单例类在老进程中存在且只能存在一个对象，在新进程中也会存在且只能存在一个对象。而且，这两个对象并不是同一个对象，这也就说，单例类中对象的唯一性的作用范围是进程内的，在进程间是不唯一的。

如何实现线程唯一的单例?

刚刚我们讲了单例类对象是进程唯一的，一个进程只能有一个单例对象。那如何实现一个线程唯一的单例呢?

我们先来看一下，什么是线程唯一的单例，以及“线程唯一”和“进程唯一”的区别。

“进程唯一”指的是进程内唯一，进程间不唯一。类比一下，“线程唯一”指的是线程内唯一，线程间可以不唯一。实际上，“进程唯一”还代表了线程内、线程间都唯一，这也是“进程唯一”和“线程唯一”的区别之处。这段话听起来有点像绕口令，我举个例子来解释一下。

假设IdGenerator是一个线程唯一的单例类。在线程A内，我们可以创建一个单例对象a。因为线程内唯一，在线程A内就不能再创建新的IdGenerator对象了，而线程间可以不唯一，所以，在另外一个线程B内，我们还可以重新创建一个新的单例对象b。

尽管概念理解起来比较复杂，但线程唯一单例的代码实现很简单，如下所示。在代码中，我们通过一个HashMap来存储对象，其中key是线程ID，value是对象。这样我们就可以做到，不同的线程对应不同的对象，同一个线程只能对应一个对象。实际上，Java语言本身提供了ThreadLocal工具类，可以更加轻松地实现线程唯一单例。不过，ThreadLocal底层实现原理也是基于下面代码中所示的HashMap。

```
public class IdGenerator {
  private AtomicLong id = new AtomicLong(0);

  private static final ConcurrentHashMap<Long, IdGenerator> instances
          = new ConcurrentHashMap<>();

  private IdGenerator() {}

  public static IdGenerator getInstance() {
    Long currentThreadId = Thread.currentThread().getId();
    instances.putIfAbsent(currentThreadId, new IdGenerator());
    return instances.get(currentThreadId);
  }

  public long getId() {
    return id.incrementAndGet();
  }
}
```

如何实现集群环境下的单例?

刚刚我们讲了“进程唯一”的单例和“线程唯一”的单例，现在，我们再来看下，“集群唯一”的单例。

首先，我们还是先来解释一下，什么是“集群唯一”的单例。

我们还是将它跟“进程唯一”“线程唯一”做个对比。“进程唯一”指的是进程内唯一、进程间不唯一。“线程唯一”指的是线程内唯一、线程间不唯一。集群相当于多个进程构成的一个集合，“集群唯一”就相当于是进程内唯一、进程间也唯一。也就是说，不同的进程间共享同一个对象，不能创建同一个类的多个对象。

我们知道，经典的单例模式是进程内唯一的，那如何实现一个进程间也唯一的单例呢？如果严格按照不同的进程间共享同一个对象来实现，那集群唯一的单例实现起来就有点难度了。

具体来说，我们需要把这个单例对象序列化并存储到外部共享存储区（比如文件）。进程在使用这个单例对象的时候，需要先从外部共享存储区中将它读取到内存，并反序列化成对象，然后再使用，使用完成之后还需要再存储回外部共享存储区。

为了保证任何时刻，在进程间都只有一份对象存在，一个进程在获取到对象之后，需要对对象加锁，避免其他进程再将其获取。在进程使用完这个对象之后，还需要显式地将对象从内存中删除，并且释放对对象的加锁。

按照这个思路，我用伪代码实现了一下这个过程，具体如下所示：

```
public class IdGenerator {
  private AtomicLong id = new AtomicLong(0);
  private static IdGenerator instance;
  private static SharedObjectStorage storage = FileSharedObjectStorage(/*入参省略，比如文件地址*/);
  private static DistributedLock lock = new DistributedLock();

  private IdGenerator() {}

  public synchronized static IdGenerator getInstance()
    if (instance == null) {
      lock.lock();
      instance = storage.load(IdGenerator.class);
    }
    return instance;
  }

  public synchroinzed void freeInstance() {
    storage.save(this, IdGeneator.class);
    instance = null; //释放对象
    lock.unlock();
  }

  public long getId() {
    return id.incrementAndGet();
  }
}

// IdGenerator使用举例
IdGenerator idGeneator = IdGenerator.getInstance();
long id = idGenerator.getId();
IdGenerator.freeInstance();
```

如何实现一个多例模式?

跟单例模式概念相对应的还有一个多例模式。那如何实现一个多例模式呢?

“单例”指的是，一个类只能创建一个对象。对应地，“多例”指的就是，一个类可以创建多个对象，但是个数是有限制的，比如只能创建3个对象。如果用代码来简单示例一下的话，就是下面这个样子：

```
public class BackendServer {
  private long serverNo;
  private String serverAddress;

  private static final int SERVER_COUNT = 3;
  private static final Map<Long, BackendServer> serverInstances = new HashMap<>();

  static {
    serverInstances.put(1L, new BackendServer(1L, "192.134.22.138:8080"));
    serverInstances.put(2L, new BackendServer(2L, "192.134.22.139:8080"));
    serverInstances.put(3L, new BackendServer(3L, "192.134.22.140:8080"));
  }

  private BackendServer(long serverNo, String serverAddress) {
    this.serverNo = serverNo;
    this.serverAddress = serverAddress;
  }

  public BackendServer getInstance(long serverNo) {
    return serverInstances.get(serverNo);
  }

  public BackendServer getRandomInstance() {
    Random r = new Random();
    int no = r.nextInt(SERVER_COUNT)+1;
    return serverInstances.get(no);
  }
}
```

实际上，对于多例模式，还有一种理解方式：同一类型的只能创建一个对象，不同类型的可以创建多个对象。这里的“类型”如何理解呢?

我们还是通过一个例子来解释一下，具体代码如下所示。在代码中，logger name就是刚刚说的“类型”，同一个logger name获取到的对象实例是相同的，不同的logger name获取到的对象实例是不同的。

```
public class Logger {
  private static final ConcurrentHashMap<String, Logger> instances
          = new ConcurrentHashMap<>();

  private Logger() {}

  public static Logger getInstance(String loggerName) {
    instances.putIfAbsent(loggerName, new Logger());
    return instances.get(loggerName);
  }

  public void log() {
    //...
  }
}

//l1==l2, l1!=l3
Logger l1 = Logger.getInstance("User.class");
Logger l2 = Logger.getInstance("User.class");
Logger l3 = Logger.getInstance("Order.class");
```

这种多例模式的理解方式有点类似工厂模式。它跟工厂模式的不同之处是，多例模式创建的对象都是同一个类的对象，而工厂模式创建的是不同子类的对象，关于这一点，下一节课中就会讲到。实际上，它还有点类似享元模式，两者的区别等到我们讲到享元模式的时候再来分析。除此之外，实际上，枚举类型也相当于多例模式，一个类型只能对应一个对象，一个类可以创建多个对象。

重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要掌握的重点内容。

今天的内容比较偏理论，在实际的项目开发中，没有太多的应用。讲解的目的，主要还是拓展你的思路，锻炼你的逻辑思维能力，加深你对单例的认识。

1.如何理解单例模式的唯一性?

单例类中对象的唯一性的作用范围是“进程唯一”的。“进程唯一”指的是进程内唯一，进程间不唯一；“线程唯一”指的是线程内唯一，线程间可以不唯一。实际上，“进程唯一”就意味着线程内、线程间都唯一，这也是“进程唯一”和“线程唯一”的区别之处。“集群唯一”指的是进程内唯一、进程间也唯一。

2.如何实现线程唯一的单例?

我们通过一个HashMap来存储对象，其中key是线程ID，value是对象。这样我们就可以做到，不同的线程对应不同的对象，同一个线程只能对应一个对象。实际上，Java语言本身提供了ThreadLocal并发工具类，可以更加轻松地实现线程唯一单例。

3.如何实现集群环境下的单例?

我们需要把这个单例对象序列化并存储到外部共享存储区（比如文件）。进程在使用这个单例对象的时候，需要先从外部共享

存储区中将它读取到内存，并反序列化成对象，然后再使用，使用完成之后还需要再存储回外部共享存储区。为了保证任何时刻在进程间都只有一份对象存在，一个进程在获取到对象之后，需要对对象加锁，避免其他进程再将其获取。在进程使用完这个对象之后，需要显式地将对象从内存中删除，并且释放对对象的加锁。

4.如何实现一个多例模式?

“单例”指的是一个类只能创建一个对象。对应地，“多例”指的就是一个类可以创建多个对象，但是个数是有限制的，比如只能创建3个对象。多例的实现也比较简单，通过一个Map来存储对象类型和对象之间的对应关系，来控制对象的个数。

课堂讨论

在文章中，我们讲到单例唯一性的作用范围是进程，实际上，对于Java语言来说，单例类对象的唯一性的作用范围并非进程，而是类加载器（Class Loader），你能自己研究并解释一下为什么吗?

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言

小晏子

要回答这个课后问题，要理解classloader和JDK8中使用的双亲委派模型。
classloader有两个作用：1. 用于将class文件加载到JVM中；2. 确认每个类应该由哪个类加载器加载，并且也用于判断JVM运行时的两个类是否相等。
双亲委派模型的原理是当一个类加载器接收到类加载请求时，首先会请求其父类加载器加载，每一层都是如此，当父类加载器无法找到这个类时（根据类的全限定名称），子类加载器才会尝试自己去加载。
所以双亲委派模型解决了类重复加载的问题， 比如可以试想没有双亲委派模型时，如果用户自己写了一个全限定名为java.lang.Object的类，并用自己的类加载器去加载，同时BootstrapClassLoader加载了rt.jar包中的JDK本身的java.lang.Object，这样内存中就存在两份Object类了，此时就会出现很多问题，例如根据全限定名无法定位到具体的类。有了双亲委派模型后，所有的类加载操作都会优先委派给父类加载器，这样一来，即使用户自定义了一个java.lang.Object，但由于BootstrapClassLoader已经检测到自己加载了这个类，用户自定义的类加载器就不会再重复加载了。所以，双亲委派模型能够保证类在内存中的唯一性。
联系到课后的问题，所以用户定义了单例类，这样JDK使用双亲委派模型加载一次之后就不会重复加载了，保证了单例类的进程内的唯一性，也可以认为是classloader内的唯一性。当然，如果没有双亲委派模型，那么多个classloader就会有多个实例，无法保证唯一性。

2020-02-10 12:07

下雨天

课堂讨论
Java中，两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。单例类对象的唯一性前提也必须保证该类被同一个类加载器加载！

2020-02-10 06:17

Ken张云忠

实际上，对于 Java 语言来说，单例类对象的唯一性的作用范围并非进程，而是类加载器（Class Loader），你能自己研究并解释一下为什么吗?
因为Jvm中类加载时采用的是双亲委派模式，对于类的唯一性的确定是通过类全名和类加载器实例一起来实现的，jdk8可以支持多个Java应用共享jre下的很多类实例就是通过扩展类加载器实现的，所以这里所说单例类实例唯一性的作用范围是类加载器指的就是即使类全名相同的类文件也必须要保证被同个类应用类加载器加载。

2020-02-10 08:10

aoe

老师讲的多例模式应该就是享元模式，常量池、数据库连接池经常使用。

2020-02-10 18:24

LJK

对于集群下的单例实现加锁有点迷惑，对象第一次实例化之后再通过getInstance就不会加锁了直接返回实例，由此有两个问题不太明白：

1. 这时如果多个进程都拿到了这个实例，save操作需要做并发控制吗？（还是就是synchronize就行了？对java不是很熟悉）
2. 这时该进程没有锁，但是freeInstance会释放一把锁，会有重复释放锁的问题吗？

2020-02-10 02:55

小喵喵

IdGenerator.freeInstance(); 应该是idGenerator.freeInstance();

2020-02-14 14:45

唐龙

不是很懂为什么需要多例模式，什么情况下需要用到多例模式。

2020-02-10 11:00

Snway

对于类加载器，可以简单理解：不同类加载器之间命名空间不一样，不同类加载器加载出来的类实例是不一样的，所以如果使用多个类加载器，可能会导致单例失效而产生多个实例

2020-02-10 09:12

黄林晴

打卡

2020-02-10 01:26

李小四

设计模式_43:

作业

Java的类加载有一个双亲委托的机制(递归地让父加载器在cache中寻找，如果都找不到才会让当前加载器去加载)，这个机制保证了有诸多好处，与今天的内容相关的就是：不管类名是否相同，不同加载器，加载的一定是不同的类。

1. 如果两个加载器是父子关系，那么只会被加载一次。
2. 如果两个加载器无父子关系，即使加载类名相同的类也会按照不同的类处理。

综上，Java的单例对象对象是类加载器唯一的。

感想

今天的内容，有一个感想: 程序员的头脑中，要能够想象程序运行的过程中，内存中发生了什么，我们要对底层多一些研究，否则真的不知其所以然。

2020-02-22 22:28

kylexy_0817

集群间单例其实有点类似于文件锁，应用通过锁文件，确保只能启动一个应用进程

2020-02-22 21:55

乾坤瞬间

课后练习回答，类加载器是在代码执行前的一步加载操作而提出来的一种实现。类加载器根据类在不同作用命名空间(域)下，分为root类加载器，扩展类加载器，应用类加载器。并且在不同作用空间下，加载同一份class文件所存储的静态空间是不一样的。同时，类静态空间存储的代码是进程共享的，所以，即使针对同一份class文件，即使在同一进程下，由于类加载器的分层委托特性，可能会存储多分class文件

2020-02-21 17:09

whistleman

yeah，打卡完成

2020-02-19 07:44

，

深入理解JAVA虚拟机第三版 总结:

大前提:每一个类加载器,都有一个独立的类名称空间(通俗的解释:两个类只有在同一个类加载器加载的前提下,才能比较它们是否"相等")

启动类加载器:加载JAVA_HOME\lib目录下的类库

↑

扩展类加载器:加载JAVA_HOME\lib\ext目录下的类库,是java SE 扩展功能, jdk9 被模块化的天然扩展能力所取代

↑

应用程序加载器:加载用户的应用程序
↑
用户自定义的加载器:供用户扩展使用,加载用户想要的内容

这个类加载器的层次关系被称为类的"双亲委派模型"

双亲委派模型工作流程:
如果一个类加载器收到了加载请求,那么他会把这个请求委派给父类去完成,每一层都是如此,所以他最后会被委派到启动类加载器中,只有父类反馈自己无法完成这个加载请求时,子类才会尝试自己去加载
类不会重复的原因:
比如一个类,java.lang.Object,存放在JAVA_HOME/lib/rt.jar中,无论哪个类加载器想要加载他,最终都会被委派给启动类加载器去加载
反之,如果没有双亲委派机制,用户自己编写一个java.lang.Object类,那么如果他被其他类加载器加载,内存中就会出现两个ava.lang.Object类
2020-02-17 09:59


桂城老托尼
感谢分享
1，线程唯一性一节，threadlocal也好，记录线程id的方式也好，如果是线程池的话，需要确认下是否需要clean。另外如果线程能被销毁，再创建的线程id是否会重复？如果重复在某些场景下可能会有问题。
2，分布式环境下的单例一节，针对id生成器这种频繁访问的服务，如果频繁加锁效率比较低，可以考虑常用的sequnce方案，每个进程持有一段id range，保证每个(分布式)进程某时间区间不重复即可。
3，尝试回答下课堂讨论，不同的classloader实例加载的class天然不属于一个，new出来的对象应该也不是一个，classloader是类的隔离级别。

2020-02-16 08:13


守拙
课堂讨论:

简单了解了下classLoader机制.
单例模式理论上应该是ApplicationClassLoader内的单例.
由JVM的双亲委派模型保证了类不会被重复的加载.
2020-02-15 23:14


FIGNT
类加载时采用双亲委派的机制，优先级排序启动加载器>扩展加载器>应用加载器>自定义加载器。这种机制保证相同的类只能加载一次，而且已java类库的类优先加载，而自定义的后加载。比如自己实现一个String类,类库中也有String类，加载哪个呢?有个优先级，只有高优先级没有加载时低优先级的才能加载。从这个角度看其实单例的唯一性作用于ClassLoader。只不过双亲委派机制保证只有一个类加载器加载。如果没有双亲委派机制，那么实例在类加载器中唯一，在类加载器间不唯一。所以准确说单例类对象的唯一性的作用范围并非进程，而是类加载器。
2020-02-15 17:41


L
不同的类加载器类全限定名不一样, 全限定名不一样就不是一个类, 而 jdk 的双亲委派模型会先加载父类, 如果父类没有, 则加载自定义类, 这样才能保证单例
2020-02-14 13:48


每天晒白牙
看到留言大家都在讨论类加载器知识点，这是我整理的类加载器知识点，大家可以参考下https://mp.weixin.qq.com/s/KseFpfXaaGDGoUdqJNzg8w
2020-02-14 11:05


杨杰
在多例的伪代码里面:

```
public BackendServer getInstance(long serverNo) {
return serverInstances.get(serverNo);
}
```

是不是应该改成：

```
public static BackendServer getInstance(long serverNo) {
return serverInstances.get(serverNo);
}
```

2020-02-12 19:39