

# 43 | 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？

王争 2020-02-10



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，这种设计模式就叫做单例设计模式，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，这也就是说，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`。

```
1 public class IdGenerator {
2     private AtomicLong id = new AtomicLong(0);
3
4     private static final ConcurrentHashMap<Long, IdGenerator> instances
5         = new ConcurrentHashMap<>();
6
7     private IdGenerator() {}
8
9     public static IdGenerator getInstance() {
10         Long currentThreadId = Thread.currentThread().getId();
11         instances.putIfAbsent(currentThreadId, new IdGenerator());
12         return instances.get(currentThreadId);
13     }
14
15     public long getId() {
16         return id.incrementAndGet();
17     }
18 }
```

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”就相当于是在进程内唯一、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：

```
1 public class IdGenerator {
2     private AtomicLong id = new AtomicLong(0);
3     private static IdGenerator instance;
4     private static SharedObjectStorage storage = FileSharedObjectStorage(/*入参省略*/);
5     private static DistributedLock lock = new DistributedLock();
6
7     private IdGenerator() {}
8
9     public synchronized static IdGenerator getInstance() {
10         if (instance == null) {
11             lock.lock();
12             instance = storage.load(IdGenerator.class);
13         }
14         return instance;
15     }
16
17     public synchronized void freeInstance() {
18         storage.save(this, IdGenerator.class);
19         instance = null; //释放对象
20         lock.unlock();
21     }
22
23     public long getId() {
24         return id.incrementAndGet();
25     }
26 }
27
28 // IdGenerator使用举例
29 IdGenerator idGenerator = IdGenerator.getInstance();
30 long id = idGenerator.getId();
31 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：

```
1 public class BackendServer {
2     private long serverNo;
3     private String serverAddress;
4
5     private static final int SERVER_COUNT = 3;
6     private static final Map<Long, BackendServer> serverInstances = new HashMap<>();
7
8     static {
9         serverInstances.put(1L, new BackendServer(1L, "192.134.22.138:8080"));
10        serverInstances.put(2L, new BackendServer(2L, "192.134.22.139:8080"));
11        serverInstances.put(3L, new BackendServer(3L, "192.134.22.140:8080"));
12    }
13
14    private BackendServer(long serverNo, String serverAddress) {
15        this.serverNo = serverNo;
16        this.serverAddress = serverAddress;
17    }
18
19    public BackendServer getInstance(long serverNo) {
20        return serverInstances.get(serverNo);
21    }
22
23    public BackendServer getRandomInstance() {
24        Random r = new Random();
25        int no = r.nextInt(SERVER_COUNT)+1;
26        return serverInstances.get(no);
27    }
28 }
```

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。

```
1 public class Logger {
2     private static final ConcurrentHashMap<String, Logger> instances
3         = new ConcurrentHashMap<>();
4
5     private Logger() {}
6
7     public static Logger getInstance(String loggerName) {
8         instances.putIfAbsent(loggerName, new Logger());
9         return instances.get(loggerName);
10    }
11
12    public void log() {
13        //...
14    }
15 }
16
17 //l1==l2, l1!=l3
18 Logger l1 = Logger.getInstance("User.class");
19 Logger l2 = Logger.getInstance("User.class");
20 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。