**参数单例设计模式**

Bouglas Lyon 和 Franscisco Castellanos

**摘要**：

参数单例模式是使用参数的单例设计模式来确保一个类创建唯一的实例。这些实例被缓存到一个表中，在用户使用它们的参数来询问获取它们时，这个表将被检查并且实例将被有条件的创建。

参数单例设计模式的惰性实例化使用给定的参数创建实例，假如或者仅仅是假如，它没有进行标准初始化，然后，这个表通过非强制性的努力来产生这个实例。惰性实例化不是新的东西，或者因为一些原因，是单例设计模式中的。可是可以看出，参数的惰性实例化是新的而且对于参数单例模式来说也是新的。

我们使用我们的参数单例模式去重新获取绑定到给定的端口的RMI注册表。我们系统的目标是确保没有两个RMI注册表在相同的机子上去监听相同的socket并且在创建后使用RMI注册表。RMI注册表是在分布式服务器上使用的。

另外一些程序包括可通过数字计数的参数资源，在使用时被锁定同时在结束时被释放。例示包含一个文件系统，序列的端口，平行端口，视频摄像头，麦克风和音频输出流。

## 简介

单例设计模式确保只有一个类的单独实例存在，他同时也意味着去提供一个公共的访问接口。

根据对比，参数单例设计模式的目的是保证一个类在给定的一个系列参数值情况下拥有仅仅一个实例。就像是单例设计模式，他同样的提供了一个公共的可访问接口。

在接下来的例子中，我们将展示一些不知道的资源，更多的,我们将展示它可能在许多的端口上会有许多的注册表。未来，我们将说明单例设计模式的实现是负责联系他们。

## 2、动机

一个系统不能够容许使用唯一标识符的参数的多个类的多个实例存在。比如，你不能够拥有两个实例使用相同序列的端口，在同一时间。你不能够拥有两个实例去参数监听相同的socket连接。你不能够拥有两个实例在同一时间去向同一个文本文件结构写入。

操作系统通常有判断消耗型资源的责任（例如，录音驱动，端口号等）我们常常留下这些问题给操作系统区解决他们的竞争问题。

甚至对于简单的资源（如端口号）有一部分的标准来处理竞争。例如，在Linux上的一系列的端口都是使用一个“/var/lock”文件被锁定的，另一方面，使用 系统调用和终端的输入输出机制去保证没有其他的进程将打开那个端口，另外，锁定文件依赖于不一致的命名惯例。

一个好的解决方案是创建一个类负责联系从类中创建的实例。这个类被定义为final，所以他不能够被子类继承。这个类还需要有一个私有的构造器，所以其他的类不能够实例化他。这个新的设计模式被叫做参数单例设计模式同时他提供一条路径使用给定的参数列表去访问和创建实例。

## 3、适用性

在以下情况下使用参数单例设计模式：

1. 必须有一个通过给定的参数得到的准确的类的实例。
2. 这些实例对于客户端从一个良好的访问点必须是可访问的。

### 4、结构

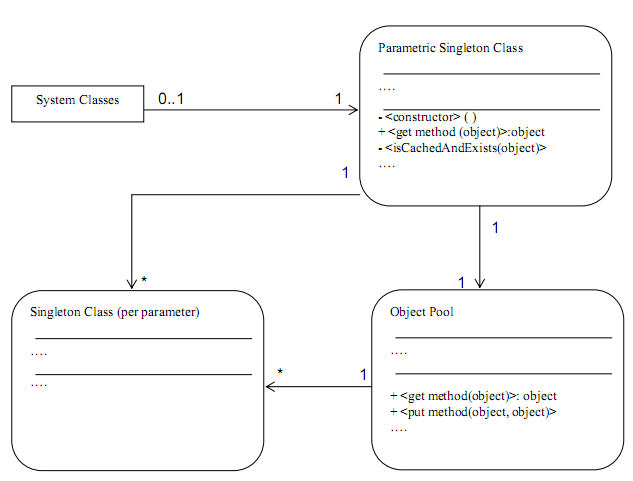


图4.1 一个参数单例设计模式的类图

## 5、参与者

这些参与者是需要参数单例设计模式实例的客户端

1. 参数单例设计模式定义了当从客户端请求时得到实例，假设这个实例还没有存在。
2. 参数单例设计模式返回实例给客户端。
3. 参数单例设计模式负责通过给定的参数创建唯一的实例。

## 6、合作

客户端仅仅通过参数单例模式获得一个指向参数单例模式实例的引用。假如实例处于一种不恰当的状态（比如，一系列的端口是打开的）则不是参数单例设计模式的责任去打开端口。这个任务是委托给系统的另外一个部分来处理的。

更多的，去检出资源也不是参数单例设计模式的责任。也就是说，多线程在同一时间能够有多个资源的引用。锁定也是委托给系统的一些其他部分来处理。

## 7、结论

参数单例设计模式有以下的一些好处：

1. 参数单例设计模式控制参数来定义实例的创建。参数单例设计模式类使用参数的惰性实例化去创建新的实例。
2. 减少命名空间。参数单例设计模式避免了全局变量去存储从一些相同参数创建的实例。

## 8、实现

这里有一些在使用参数单例设计模式时的实现问题要考虑：

1. 唯一的参数映射。参数单例设计模式需要相同结构的映射表去映射参数空间到实例空间，在8.1的例子中展示。

例子8.1

/\*\*

\* Single instance of the registry is instances per port. The instance

\* is put into a hash table.

\*/

private static Hashtable registry = new Hashtable(100);

…

private static void startRegistry(Integer port) throws RemoteException

{

Registry r = null;

try {

r = LocateRegistry.createRegistry(port.intValue());

registry.put(port, r);

} catch (Exception e) {

r =LocateRegistry.getRegistry(port.intValue());

registry.put(port, r);

}

}

2、保证唯一的实例。参数单例设计模式创建唯一的实例通过唯一的参数，在例子8.2中展示

例子 8.2

/\*\*

\* Checks whether a Registry has been started on this host for the

specified port.

\* Otherwise it starts one.

\*/

public static void restart(Integer port) {

if (isInCacheAndRunning(port))

return;

try {

startRegistry(port);

} catch (RemoteException e) {

In.message(e);

}

}

1. 为重复请求缓存实例。参数单例设计模式必须能够来查找实例，给出一系列的参数，同时处理一些数据结构。也就是说，必须有足够的空间去持有到所有程序需要的实例的引用。同样的，需要一种方法让客户端去安全快速的查询同时重新请求实例。这将在例子8.3中展示

例子8.3

private static Hashtable registry = new Hashtable(100);

private static void startRegistry(Integer port) throws RemoteException

{

Registry r = null

…

r =LocateRegistry.getRegistry(port.intValue());

registry.put(port, r);

…

}

public static Registry getRegistry(Integer port) {

restart(port);

return (Registry)registry.get(port);

}

## 9、例示代码

下面的代码使用参数单例模式使用RMI注册表去为每个端口创建一个并且是唯一一个实例。一个注册表需要缓存查询。假如在一个给定的端口上没有找到注册表，新的将被创建，所以我们使用惰性实例化机制。

public final class ParametricSingletonRmiRegistry {

private static Hashtable registry = new Hashtable(100);

// use singleton pattern and prevent instantiation of

RmiRegistryUtils.

private ParametricSingletonRmiRegistry() {

}

// the only instance of the Registry held by the

RmiRegistryUtils

public static void listNames(Integer port) {

if (!isInCacheAndRunning(port))

return;

Registry r = (Registry) registry.get(port);

System.out.println("registry on port: " + port);

System.out.println("registry: " + r);

System.out.println("names: ");

try {

print(r.list());

} catch (RemoteException e) {

In.message(e);

}

System.out.println("------------------------");

}

/\*\*

\* Checks whether a Registry has been started on this

host

\* in the specified port. If not, it tries to start

one.

\*/

public static void restart(int port) {

if (isInCacheAndRunning(port))

return;

try {

startRegistry(port);

} catch (RemoteException e) {

In.message(e);

}

}

/\*\*

\* @param port

\* @return true if registry is in cache and

running

\*/

private static boolean isInCacheAndRunning(Integer port) {

if (!isRegistryRunning(port.intValue())) return false;

if (!isRegistryInCache(port.intValue())) return false;

return true;

}

/\*\*

\* @param port

\* @return true if we are able to locate a local registry

\*/

private static boolean isRegistryRunning(int port) {

try {

Registry r = LocateRegistry.getRegistry(port);

if (r != null) return true;

} catch (RemoteException e) {

return false;

}

return false;

}

/\*\*

\* @param port

\* @return return true if registry is in

cache

\*/

private static boolean isRegistryInCache(int port) {

if (registry.get(new Integer(port)) == null) return false;

return true;

}

/\*\*

\* Restart the registry, if

needed.

\* Return the only instance of the registry for consistent

global

\* using by making use of the Singleton Design

Pattern

\*

\* @return the internally held registry

instance.

\*/

public static Registry getRegistry(Integer port) {

restart(port);

return (Registry) registry.get(port);

}

private static void startRegistry(Integer port) throws

RemoteException {

Registry r;

try {

r = LocateRegistry.createRegistry(port.intValue());

registry.put(port, r);

} catch (Exception e) {

r = LocateRegistry.getRegistry(port.intValue());

registry.put(port, r);

}

}

public static void main(String args[]) throws RemoteException {

//restart(port);

restart(5001);

restart(5002);

restart(5002);

Registry r = LocateRegistry.getRegistry(5002);

System.out.println("registry="+r);

}

}

## 10、相关工作

单例设计模式不是新的内容[Gamma et Al.]。可是，参数单例设计模式是新的内容，就像我们知道的一样。单例设计模式和工厂设计模式是构造型模式，单例模式保证你所要的实例是唯一的[Goldfedder]。对比一下，参数单例模式保证你通过给定的参数获取想要的实例。

有一些其他的单例设计模式的扩展，像是选项单例模式。可是，选项单例模式为子类创建了新的行为，而不是像参数单例模式限制行为[Maclean]。

单例设计模式的思想是派生类很难是新的。将类定义成为final将变成公共的做法。参数单例模式仅仅只是像缓存管理模式一样，不同的是，缓存管理模式不会限制通过给定的ID获取实例的数目。缓存管理模式这样设计是帮助加快实例的创建。事实上，他有许多与对象池设计模式的相似处。不同的是均衡性和一种check-in机制。参数单例模式没有这样的特性[Grand]

使用惰性实例化设计模式不是新的内容，同样的，事实上是一种在处理不知道缓存中实例数目的实际需要[Beck97]。

在单例设计模式中使用异步方式也不是新的内容，具有代表性的是被称为双检测锁定单例设计模式。这意味着实现参数单例设计模式是线程安全的[Shalloway]。异步关键词(synchronized)保证被调用的方法是原子性的操作（例如，他们将在一次全部发生或者不发生）。假如没有这种方式，一个竞争条件将在两个通过相同参数在同一时间需要相同的对象的线程之间发生，并且这会导致不可预料的行为。

## 11、总结

参数单例设计模式允许通过给定的一系列的参数来创建一个类的实例。他提供了一个简单的公共访问接口进入类，就像是单例设计模式[Cooper]。不同的是，对于不同的新参数，另外一个实例将被创建。

实例被存储在缓存中。这里没有一种机制去释放实例并且允许他们被垃圾回收机制回收，在目前。更多的就像是全局存储，假如使用者不正确的修改实例的值，则BUG将很难被跟踪发现。

参数单例设计模式的另外一个缺点是没有控制谁进入这个单例模式对象。就像是单例设计模式，参数单例设计模式是唯一的能够创建自身实例的类[Stelting]。

单例设计模式技术上的缺点是被认为等同于参数单例设计模式，这在单例的拷贝在运行在多虚拟机下，他们将为每一个虚拟机创建一个实例，我们的参数单例设计模式的实现将遭受这种缺陷，因为无论虚拟机是否在运行本地注册表调用将定位注册表。这是一个bug从而造成不可信任的参数单利设计模式（或者是非参数单例设计模式）实现者[Fox]。

参数单例设计模式留下了一些主题给未来的工作。介绍不纯类的check-in机制似乎十分有效。例如，消耗型资源（像一系列的端口）问题，我可能需要知道谁拥有这一系列的端口，并且使用一些机制去释放这些端口，一个锁定的资源能够引起一个进程的死锁。强制释放资源将在耗费资源而引起不可预料的行为时中断死锁。

我们发现已经运行的RMI注册表将引起一些简单的问题。正在中断死锁意味着杀死一个正在使用的注册表。当你杀死一个注册表时发生的问题的答案仍然是需要探讨的。所谓的探讨准确的来说是使用一种安全的机制去杀死这个注册表，一般来说，不同的注册表具有不同的类路径。重新开始注册将不仅仅保证一个类路径的改变，同样的在执行上下文中引起一个改变。在上下文中的改变需要重新载入类文件，确保代码的更新。一种自然的替代方式是去创建一个新的类加载器，但是实际上类加载器的数目是没有界限的，这可能是非常浪费内存的，。更糟糕的是，使用多个类加载器将使单例设计模式的实现失败[Geary]。

在新的注册表中设置代码对于一些使用者来说是重要的。默认的，基于注册表的代码是与参数单例设计模式一样的。在未来，一种改变这个的机制将是受欢迎的，例如：

// protocol + current directory (of the VM) + trailing slash

String codebase ="file:"+System.getProperty("user.dir")+"/";

System.setProperty( "java.rmi.server.codebase", codebase);

一些人声称使用锁文件将阻止多个单例设计模式的运行。我们将对此保持怀疑，因为多个在被删除锁定文件之前将死亡的程序将继续引起死锁。所有这个问题怎么去解决仍然是需要探讨的[JDF]。

在RXTX群组上有一些关于参数单例设计模式的单元测试的讨论。关于怎么实现这个单元测试的问题仍然是可以探讨的。一个（没有测试过的）注意是去创建一个外观模式在一个简单的接口表单中同时使用它的多个版本，来提供给单元测试一个实验模型[RXTX]。