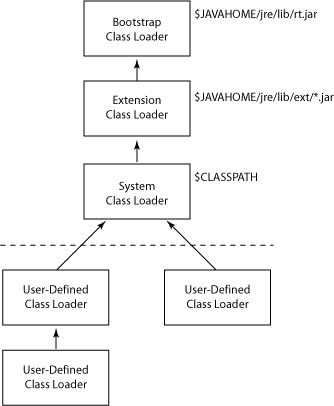
[自定义ClassLoader实现java应用核心逻辑模块热部署](http://blog.csdn.net/zhangdaiscott/article/details/23378023)

2014-04-10 19:48 2043人阅读 [评论](http://blog.csdn.net/zhangdaiscott/article/details/23378023#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhangdaiscott/article/details/23378023#report)

 本文主要是根据classloader的特性，结合实际产品环境中遇到的问题，来探讨下JAVA应用中局部模块热部署的可行性。

     我们知道，一些web应用提供了自动检测装载webapp的功能，但大部分的时候，就是相当于重新启动了一遍Webapp，存储在内存中的数据也会丢失，并不能灵活地满足需要。而OSGI框架，虽然也提供了模块的热部署，但为了用热部署而将应用限制在OSGI的框框中，有些时候得不偿失。于是想根据实际需要来定制classloader，灵活地指定哪些类重载，哪些类不需要。

     言归正传，进行我们的实践，这里先简单介绍下JAVA的classloader机制：



 从上图可以看出虚拟机中的Classloader的层次结构， 由最外层的Classloader去load全限定名指定的class，如果load不到，则委托给该classloader的父classloader，直至到root classloader。

Java代码



1. **protected** **synchronized** Class<?> loadClass(String name, **boolean** resolve)
2. **throws** ClassNotFoundException
3. {
4. // First, check if the class has already been loaded
5. Class c = findLoadedClass(name);
6. **if** (c == **null**) {
7. **try** {
8. **if** (parent != **null**) {
9. c = parent.loadClass(name, **false**);
10. } **else** {
11. c = findBootstrapClass0(name);
12. }
13. } **catch** (ClassNotFoundException e) {
14. // If still not found, then invoke findClass in order
15. // to find the class.
16. c = findClass(name);
17. }
18. }
19. **if** (resolve) {
20. resolveClass(c);
21. }
22. **return** c;
23. }

**[java]** [view plain](http://blog.csdn.net/zhangdaiscott/article/details/23378023) [copy](http://blog.csdn.net/zhangdaiscott/article/details/23378023)

1. **protected** **synchronized** Class<?> loadClass(String name, **boolean** resolve)
2. **throws** ClassNotFoundException
3. {
4. // First, check if the class has already been loaded
5. Class c = findLoadedClass(name);
6. **if** (c == **null**) {
7. **try** {
8. **if** (parent != **null**) {
9. c = parent.loadClass(name, **false**);
10. } **else** {
11. c = findBootstrapClass0(name);
12. }
13. } **catch** (ClassNotFoundException e) {
14. // If still not found, then invoke findClass in order
15. // to find the class.
16. c = findClass(name);
17. }
18. }
19. **if** (resolve) {
20. resolveClass(c);
21. }
22. **return** c;
23. }

    这里我们的自定义Classloader可以通过重载loadClass方法，仅对指定package下的类进行加载，其余全部委托父Classloader来加载 ， 这里需要用到classloader的defineClass方法，以便我们的classloader可以载入任意指定位置的class文件。

Java代码



1. **public** **class** MyClassLoader **extends** ClassLoader{
2. **public** **static** ConcurrentHashMap<String, Class<?>> classes = **new** ConcurrentHashMap<String, Class<?>>();
4. **public** **static** MqClassLoader instance = **new** MyClassLoader();
5. //构造自定义Classloader, 并指定父Classloader
6. **public** MyClassLoader() {
7. **super**(Thread.currentThread().getContextClassLoader());
8. }
10. **public** Class<?> load(String name, **byte**[] data, **boolean** resolve) {
11. Class<?> klass = defineClass(name, data, 0, data.length);
12. **if** (resolve)
13. resolveClass(klass);
14. classes.put(name, klass);
15. **return** klass;
16. }
18. **public** Class<?> loadClass(String name, **boolean** resolve) **throws** ClassNotFoundException {
19. Object value = classes.get(name); // 检查缓存
20. **if** (value != **null** && value != INVALID) {
21. Class<?> klass = (Class<?>) value;
22. **if** (resolve)
23. resolveClass(klass);
24. **return** klass;
26. } **else** { // 缓存中不存在
27. **byte**[] data = read(findClassFile(name)); // 读取类文件
28. **if** (data == **null**)
29. **return** **super**.loadClass(name, resolve); // 交由父classloader去load类文件
30. **else** {
31. **try** {
32. lock.lock();
33. Object cc = classes.get(name); // 检查缓存
34. **if** (cc != **null**) {
35. **return** (Class<?>) cc;
36. } **else**
37. **return** instance.load(name, data, resolve); // 自己load类文件
38. } **finally** {
39. lock.unlock();
40. }
41. }
42. }
43. }
44. }

**[java]** [view plain](http://blog.csdn.net/zhangdaiscott/article/details/23378023) [copy](http://blog.csdn.net/zhangdaiscott/article/details/23378023)

1. **public** **class** MyClassLoader **extends** ClassLoader{
2. **public** **static** ConcurrentHashMap<String, Class<?>> classes = **new** ConcurrentHashMap<String, Class<?>>();
4. **public** **static** MqClassLoader instance = **new** MyClassLoader();
5. //构造自定义Classloader, 并指定父Classloader
6. **public** MyClassLoader() {
7. **super**(Thread.currentThread().getContextClassLoader());
8. }
10. **public** Class<?> load(String name, **byte**[] data, **boolean** resolve) {
11. Class<?> klass = defineClass(name, data, 0, data.length);
12. **if** (resolve)
13. resolveClass(klass);
14. classes.put(name, klass);
15. **return** klass;
16. }
18. **public** Class<?> loadClass(String name, **boolean** resolve) **throws** ClassNotFoundException {
19. Object value = classes.get(name); // 检查缓存
20. **if** (value != **null** && value != INVALID) {
21. Class<?> klass = (Class<?>) value;
22. **if** (resolve)
23. resolveClass(klass);
24. **return** klass;
26. } **else** { // 缓存中不存在
27. **byte**[] data = read(findClassFile(name)); // 读取类文件
28. **if** (data == **null**)
29. **return** **super**.loadClass(name, resolve); // 交由父classloader去load类文件
30. **else** {
31. **try** {
32. lock.lock();
33. Object cc = classes.get(name); // 检查缓存
34. **if** (cc != **null**) {
35. **return** (Class<?>) cc;
36. } **else**
37. **return** instance.load(name, data, resolve); // 自己load类文件
38. } **finally** {
39. lock.unlock();
40. }
41. }
42. }
43. }
44. }

    上面的代码描述了自定义classloader的载入逻辑，findClassFile() 就是自己定义的从哪里找需要的类文件方法。

     defineClass 方法可以灵活地用来实现分布式动态计算，hadoop mapreduce应该就是使用了这一方法来保证服务器集群间的处理类的传输和运行。

Java代码



1. /\*\*
2. \* 重新初始化，以便实现重载所指定的类
3. \*/
4. **public** **static** **void** reset() {
5. instance = **new** MyClassLoader();
6. classes.clear();
7. }

**[java]** [view plain](http://blog.csdn.net/zhangdaiscott/article/details/23378023) [copy](http://blog.csdn.net/zhangdaiscott/article/details/23378023)

1. /\*\*
2. \* 重新初始化，以便实现重载所指定的类
3. \*/
4. **public** **static** **void** reset() {
5. instance = **new** MyClassLoader();
6. classes.clear();
7. }

   其实每次reset都会产生一个新的classloader实例， 该实例会在所有这个实例装载的的类全部被回收后才被回收，这里比较奔放的全部reset掉，经过测试尚未发现内存溢出问题。

   调用这些类的方法：

Java代码



1. **public** **static** **void** invoke(String method ,Object[] obj, Class<?>[] parameterTypes){
2. **try** {
3. Object cls = MyClassLoader.instance.loadClass("类全限定名", **true**).newInstance();
4. cls.getClass().getMethod(method,parameterTypes).invoke(cls ,obj);
5. } **catch** (Exception e) {
6. logger.error("reloadable error " + method, e);
7. }
8. }

**[java]** [view plain](http://blog.csdn.net/zhangdaiscott/article/details/23378023) [copy](http://blog.csdn.net/zhangdaiscott/article/details/23378023)

1. **public** **static** **void** invoke(String method ,Object[] obj, Class<?>[] parameterTypes){
2. **try** {
3. Object cls = MyClassLoader.instance.loadClass("类全限定名", **true**).newInstance();
4. cls.getClass().getMethod(method,parameterTypes).invoke(cls ,obj);
5. } **catch** (Exception e) {
6. logger.error("reloadable error " + method, e);
7. }
8. }

  这里只能通过反射的方式调用，由于classloader的安全机制，同样的类，父classloader装载的类 和 子classloader装载的类不能互相转换。

   需要强调的是，被重载的类重载后所有的变量都会被重新初始化，因此一些重要数据变量还是得交由父Classloader来管理。

    该方式存在一些缺点，把核心逻辑归并到一起，将变量分离出去，从而影响了应用本身的结构。

   目前这种方式正应用在一个消息服务中，主要避免由于一些微小的改动而重新启动服务，重启消息服务这对于大型系统来说是很麻烦的 ，是否值得还很难说，权当对java的深入学习吧。