类动态更新事务研究*)

张 仕[1,2] 赖会霞[1] 黄林鹏[2]

(福建师范大学数学与计算机科学学院 福州 350007)[1] (上海交通大学计算机科学与工程系 上海 200240)[2]

摘 要 针对软件单个类的动态更新操作存在许多限制,例如不允许删除类、方法等,提出类动态更新事务的方法,利用动态更新类集来克服这些限制,同时保证更新的安全性。基于此,对动态更新事务的一些属性,如 ACID 进行讨论,提出类型安全的类动态更新事务并进行了证明。最后,以 Java 语言为例,说明了如何构造动态更新程序,并进行了相应的实验,说明了方法的可行性。

关键词 软件更新,类动态更新,程序设计,更新事务

Research on Dynamic Update Transaction for Java Classes

ZHANG Shi[1,2] LAI Hui-xia[1] HUANG Lin-peng[2]

(School of Mathematics and Computer Science, Fujian Normal University, Fuzhou 350007, China)[1]

(Dept. of Computer Science and Engineering, Shanghai Jiaotong University, Shanghai 200240, China)[2]

Abstract Dynamic software updating is critical for many systems that must provide continuous service. Aiming to eliminating the restrictions on single class updating, such as not allowing deleting method that is invoke in other class, The paper proposed update transaction for dynamical updating class set, also discussed some properties about it, such as ACID. Proved the type-safety property formally. Proposed a new implementation method based on Java. The method makes use of Java class loading mechanism, which shows how to construct updateable program. An experiment was made for analyzing.

Keywords Software updating, Dynamic class update, Program design, Update transaction

1 引言

在软件维护阶段,为了改正软件的错误,提高软件性能以及适应外部环境的变化,也就需要进行软件的更新操作。一般来说,为了能够对正在运行的软件进行更新,需要停止软件的运行,然后载入新的软件并重新启动,这种软件更新方法对于通讯、银行等应用领域而言,会造成很大的损失。对于一些生命支持系统而言,则根本不允许软件重新启动进行更新操作。动态软件更新(Dynamic software updating, DSU)[4]是指在进行软件的新旧版本替换时,不需要中止原来程序的运行。动态软件更新能够保证软件提供持续不间断的运行,这对于许多关键的软件应用领域而言是非常重要的。

到目前为止,面向对象软件的动态更新主要研究如何对程序中的类进行更新[1,2,7,9,10],以及类的何种动态更新不会引起类型安全问题[2,3]。如果每次更新都只对其中的一个类进行,那就存在许多相关的更新限制,例如不允许任意删除某些方法/数据域,不允许对数据域的类型做某些修改等等。然而,在许多情况下,我们迫切需要以这种方式对某些类进行改变。对于这种情况,该如何解决呢? 问题的答案就是同时对多个相关类进行动态更新。例如,我们想要删除类 A 中的方法 m,由于类型安全方面的原因,这种修改是不允许的。但是,如果能够一次性对多个类进行动态更新,那么这种情况可能就会得到解决。

本文中提出更新事务的概念就是为了解决上述问题。对于动态软件更新事务,需要对它的原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)进行分析,同时还需要对更新事务的类型安全性进行研究。

对于 Java 语言,其动态软件更新可以通过对 JVM(Java Virtual Machine)进行修改来实现[2]。但是 JVM 修改可能使 Java 程序无法做到"一次编译,随处运行"。文献[10]中提出利用代理类(Proxy Class)的方法,该方法不需要 Java 语言和 JVM 的支持。该方法把一个类分为几个部分,分别是状态、接口和实现,并且利用一个 wrapper 来包装。然而,这种方法的限制在于接口是不允许被改变的,这也就在某种程度上大大减少了更新的灵活性。本文针对这些问题提出一种实现方法,该方法需要在新旧类上进行相应的修改,增加一定的程序量。同时,为了能够在一个运行的程序中装载相同类的不同实现,我们还对类装载进行扩展。

本文第 2 节叙述了有关更新事务的相关基本定义,主要包括更新事务、类型安全等等;第 3 节对如何设计一个动态更新程序的相关要点进行阐述;实验和相关的讨论在第 4 节;第 5 节介绍相关工作;最后总结本文。

2 动态更新事务

本节主要对一些基本概念进行形式化定义,并且讨论了类型安全性和其它相关属性。

*)本文得到国家自然科学基金(60673116),国家高技术研究发展计划(863)(2006AA01Z166),福建省自然科学基金计划资助项目(2007J0315)和福建省教育厅资助省属高校项目(2007F5037)资助。张 仕 讲师,博士研究生,CCF 会员,主要研究领域为程序设计语言、数据集成等;黄林鹏 教授,博士,博士生导师,主要研究领域为程序设计语言、数据集成。

2.1 基本定义

定义 1(继承关系,Inheritance $\leqslant$)[2] 如果 C 是从 CS 直接扩展得到,那么它们满足关系 $C \leqslant CS$。继承关系影响到一个类的复合,CS 可以是 C 的超类,也可以是 C 的接口定义。

如果类 C_1 包含对类 C_2 的引用,那么称 C_1 依赖于类 C_2。这里的引用可以包含方法调用、数据域存取和继承关系。在这种关系定义下,对类 C_2 的任意改变都会对类 C_1 产生影响[2,5]。

定义 2(依赖关系,Dependency $\propto$) 如果 C_1 依赖于类 C_2,那么称它们之间的依赖关系 $C_1 \propto C_2$ 成立。对于依赖关系,其满足传递性,依赖关系的传递关系表示为 $\propto^*$。依赖关系不但可以用于表示类之间的关系,而且可以被运用到方法和数据域上,表示它们的调用/存取/包含。

继承关系也满足传递性,可以表示为 $\leqslant^*$。本文限定不允许存在类之间的循环继承关系,所以可以得到 $C \leqslant^* C_S => \neg(C_S \leqslant^* C)$ 成立。也就有 $C \leqslant C_S => C \propto C_S$ 成立。继承关系也可以被运用到类和接口上,而且,依赖关系和继承关系也满足自反性,也就是有 $C \leqslant C$ 和 $C \propto C$ 成立。

对于方法之间的依赖关系,如果方法 $C_1.M$ 调用 $C_2.N$,那么它们之间满足关系 $C_1.M \propto C_2.N$。对于数据域上,也有类似的定义,如果数据域类型 $C_1.f$ 为 C_2,那么它们满足关系 $C_1.f \propto C_2$。我们也用 $C_1 \propto C_2$ 来表示 $C_1.M$ 依赖于 $C_2.N$,$C_1.f \propto C_2$,$C \leqslant C_S$。传递性可以被运用于数据域/方法之间的依赖关系上。如果数据域 $C_1.f$ 的类型为类 C_2,或者存在类 C_3 满足 $C_3 \propto^* C_2$(其中 C_3 是 $C_1.f$ 的类型),那么关系 $C_1.f \propto^* C_2$ 成立。对于数据域和继承性方面,则具有相同的解释和定义。

定义 3(类 C 的更新集,Class C's Updating Set) 如果类 C 被更新为 C_{new},那么类 C 的更新集 $Us(C)$ 定义为所有为了保证程序类型安全而需要更新的类的集合(包括直接的和间接的)。C 称为更新集 $Us(C)$ 的核。

本文中,我们假定 C,C_{new} 和 C_{old} 有相同的类名,并且 C_{new} 和 C_{old} 加上下标的目的就是为了区分类 C 的新定义和旧定义。类 C 的最大更新集定义为 $\mathrm{Max}Us(C)=\{C' \mid C' \propto^* C\}$,该集合包含所有和类 C 有依赖关系的类和接口,并且满足 $Us(C) \subseteq \mathrm{Max}Us(C)$。对于一个待更新类 A,图 1 中表示了几种可能的依赖关系。考虑图 1(a)中的情况,类 A 的更新集包含类 B 和 C;图 1(c)中是一个环形依赖关系。如果有一个节点需要更新,那么所有环中的类都需要加入到更新集 $Us(C)$ 中。

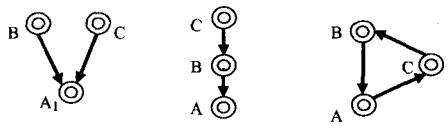

图 1 类的调用/存取关系图

在大多数的程序中,一个类的真实更新集总是比最大更新集要小很多,例如,如果 C 有一个方法 m,但是在新定义 C_{new} 中,该方法被删除,类 D 中包含有对 m 的调用,为了能够保证类型安全性,我们需要对类 D 的方法进行修改,以便能够适应 C 的新定义。这种情况下,类 D 属于更新集 $Us(C)$。

为了确保更新事务集,我们设计了如下的规则来定义。假设要更新类 X,初始化时设置更新集 $Us(X)=\{X\}$。

(1)仅仅修改方法的实现时,更新集 $Us(X)=Us(X) \cup \{X\}$。如果新的方法实现中存在对类 Y 的不存在的方法/数据域的调用,那么类 Y 就必须进行修改,这种情况下,更新集 $Us(X)=Us(X) \cup \{X\} \cup \{Y\}$。

(2)往类 X 中增加数据域或者方法时,更新集 $Us(X)=Us(X) \cup \{X\}$。

(3)从类 X 中删除一些数据域/方法时,那么更新集 $Us(X)=Us(X) \cup \bigcup_{i=1}^{n} Us(Y_i)$,其中 $Y_i \in \{Y \mid Y$ 调用/存取 X 中被删除掉的方法/数据域$\}$。

(4)对 X 中一些数据域/方法的类型加以修改,那么更新集 $Us(C)=Us(C) \cup \bigcup_{i=1}^{n} Us(B_i) \cup \bigcup_{i=1}^{n} \{D \mid Us(D) \wedge D \propto B_i\}$,其中 $B_i \in \{B \mid B$ 调用/存取 C 中类型被更新的方法/数据域$\}$。

如果对数据域/方法的名称进行修改,那么需要综合删除和加入的情况,也就是理解为删除一些数据域/方法,再加入新的数据域/方法。

在面向对象软件中,类之间的关系是一个复杂的有向图,对其中一个类进行更新可能会引起其它类产生错误。在执行一个程序时,类关系被转化为对象之间的复杂关系,所以我们必须确保把更新对程序造成的影响降到最小。本文仅仅对类型安全性(type-safety)进行讨论。如果有一些类由于方法的语义发生变化,那么这种修改也需要考虑加入到更新集 $Us(X)$ 中。

以 Java 语言为例,由类的动态改变所引起的类型错误主要有两种情况:静态类型错误和动态类型错误。静态类型错误是由于不可用的数据域/方法引用引起的。动态类型错误则主要是由于一些事件导致引用被绑定到不相容的类型上。JVM 平台在一些操作(如赋值、类型转换)前会进行动态类型检查,如果操作可能会引起动态类型错误,那么 JVM 将会抛出异常。

定义 4(类型安全,Type safety) 如果一个类 C 既没有静态类型错误,也没有违反 JVM 运行时类型检查能够发现的动态类型错误,那么就称类 C 是类型安全的。如果程序 P 的所有相关类都是类型安全的,那么就称程序 P 是类型安全的。

定义 5(更新事务,Update transaction) 更新事务是一个类更新序列,并且该更新序列中的所有类更新都必须一次执行完成,不能够中断。更新事务形如 $T=\{C_1 \rightarrow C_1', C_2 \rightarrow C_2', \cdots, C_n \rightarrow C_n'\}$,其中 $C_i \rightarrow C_i'(1 \leqslant i \leqslant n)$ 意味着把类 C_i 更新为新的类定义 C_i',并且把 C_i 类对象转化为符合新类定义 C_i' 的对象。

定义 6(良好的更新事务,Valid update transaction) 如果一个更新事务满足 ACID 性质,那么就称其为良好的更新事务,其中 ACID 性质定义如下:

原子性(Atomicity):更新事务中所有类的更新操作要么都执行,要么都不执行。

一致性(Consistency):程序在 $\{C_1', C_2', \cdots, C_n'\}$ 更新后是类型安全的。当然,在更新的过程中可能会存在类型不安全的状况。

隔离性(Isolation):作为一个整体,$\{C_1', C_2', \cdots, C_n'\}$ 要保证其对外的数据域/方法在类集外仍旧可以进行存取/调用,并且该整体对外进行数据域/方法的存取/调用必定存在一个类 D 中,其中类 $D \notin \{C_1', C_2', \cdots, C_n'\}$。

持久性(Durability):更新事务的操作在更新事务结束后能够被有效保存。

定义 7(类 C 的良好更新事务,Valid update transaction

for class C)　类事务 $T=\{C_1\rightarrow C_1', C_2\rightarrow C_2', \cdots, C_n\rightarrow C_n'\}$是良好的，并且$\{C_1, C_2, \cdots, C_n\}=U_S(C)$。类 C 的良好更新事务形如 $T_C=\{C_1\rightarrow C_1', C_2\rightarrow C_2', \cdots, C_n\rightarrow C_n'\}_C$。

定义 8(更新事务的良好性，Validity of update transaction)　更新事务 $T_C=\{C_1\rightarrow C_1', C_2\rightarrow C_2', \cdots, C_n\rightarrow C_n'\}_C$ 是良好的当且仅当下面的两个条件能够满足：

(1) 更新后，程序 P 中的任何类都不再依赖于从 C 中去除掉的数据域/方法。

(2) 如果存在类依赖于类 C 的类型集，那么元素 C 的类型集不能被去除。其中 C 的类型集是指类/接口可以成功实现对 C 的实例强制转化的类/接口的集合。

2.2　相关性质

定理 1(类型安全的更新，type-safe update)　利用良好的更新事务 $T_C=\{C_1\rightarrow C_1', C_2\rightarrow C_2', \cdots\}_C$ 把程序从 P 动态更新到 P'，如果 P 是类型安全的，那么最后得到的 P'也保持类型安全性。

证明：利用反证法，即存在一些情况使定理不成立。假设有良好的更新事务 $T_C=\{C_1\rightarrow C_1', C_2\rightarrow C_2', \cdots\}_C$ 对类型安全的程序 P 进行动态更新，最后得到的程序 P'却不是类型安全的。由上面定义可知，那么存在一个类 D 存取/调用类 E 中不存在的数据域/方法 $E.f/E.m$。对于 D 按照其具体情况可以分两种进行分析：

Case $D\notin\{C_1', C_2', \cdots, C_n'\}$

Subcase $E=C_i$：由隔离性定义知，$\{C_1', C_2', \cdots, C_n'\}$要保证其对外的数据域/方法在类集外仍旧可以进行存取/调用，所以不可能是 $E=C_i$，产生矛盾，排除这种情况。

Subcase $E\notin\{C_1', C_2', \cdots, C_n'\}$：从更新集的定义可得到，所有需要被更新的类都已经包含在$\{C_1', C_2', \cdots, C_n'\}$中，$D$ 和 E 都不属于$\{C_1', C_2', \cdots, C_n'\}$，且 P 是类型安全的，这种情况也不可能产生，所以可以排除。

Case $D\in\{C_1', C_2', \cdots, C_n'\}$

Subcase $E=C_i$：由一致性可知，程序在$\{C_1', C_2', \cdots, C_n'\}$更新后是类型安全的，这显然和前面的假设相矛盾，所以也不可能是这种情况。

Subcase $E\notin\{C_1', C_2', \cdots, C_n'\}$：从构造更新集可知，我们已经把所有需要修改的类包含到$U_S(C)$中，如果 E 需要修改，那么它必定在$\{C_1', C_2', \cdots, C_n'\}$中。但是从这种情况假设 $E\notin\{C_1', C_2', \cdots, C_n'\}$，可能产生矛盾。接着可由隔离性得到 D 不会调用/存取类 E 中不存在的方法/数据域。

由此，从上面的证明过程可以得出，如果 P 是类型安全的，并且更新事务是良好的，那么 P'也是类型安全的。　□

在更新事务 $T_C=\{C_1\rightarrow C_1', C_2\rightarrow C_2', \cdots\}_C$ 中，交换律可以被运用到类更新中，也就是说 $T_C=\{C_1\rightarrow C_1', C_2\rightarrow C_2'\}_C$等价于 $T_C=\{C_2\rightarrow C_2', C_1\rightarrow C_1'\}_C$。如果在更新事务中某个类产生更新错误，那么更新事务回滚，这可以保证整个程序的类型安全性。

定理 2(类型安全的更新，type-safe updates)　利用更新事务$\{T_{C_1}, T_{C_2}, \cdots, T_{C_n}\}$，把程序由 P 更新到 P'，如果 P 是类型安全的，并且 T_{C_i} 是以类 C_i 为核的良好的更新事务，其中 $T_{C_i}\in\{T_{C_1}, T_{C_2}, \cdots, T_{C_n}\}$　$(1\leqslant i\leqslant n)$，那么 P'是类型安全的。

证明：由定理 1 可以直接得到相关结论。

更新事务$\{T_{C_1}, T_{C_2}, \cdots, T_{C_n}\}$是顺序相关的，交换律对该更新事务是不可用的；所以，对于多个更新事务序列对程序进行更新的时候，其是顺序相关的，而且每一个更新事务都是可逆的。

3　Java 类动态更新

Java 语言在软件开发中得到广泛的运用，然而对系统的修改容易引入许多不确定行为。在动态更新中，需要把旧类定义替换为新的类定义，同时要把基于旧类定义的对象转换到基于新的类定义，在这一过程中需要保证运行中的方法和线程能够持续无误，需要解决的关键就是类型安全问题。

本节中对 Java 类装载机制进行分析，说明 Java 类动态更新的可行性，并简要叙述动态更新软件的设计。

3.1　Java 类装载机制

在 JVM 中，一个类一旦被装载到 JVM，那么相同的类就不会被再次装载。这里首先需要搞清楚相同的类的概念。在 Java 中，类是由其完整的有效类名所标识(fully qualified class name)。完整的有效类名由包名和类名所组成。在 JVM 中，类的唯一标识是由类的完整有效类名加上装载该类的 ClassLoader 实例所组成。举例说明，如果一个包 Pg 中的类 C_1 被装载器实例 kl_1 所装载，那么在 JVM 中的 C_1 类则由三元组 (C_1, Pg, kl_1)唯一标识。这就意味着如果两个类装载器实例 kl_1 和 kl_2 同时都装载了 Pg 包中的类 C_1，那么在 JVM 中，它们具有不同的标识，也就是类实例不同，主要原因在于它们是由不同的类装载器装载。

ClassLoader 类的方法 findClass 在父类装载器没有找到所需类定义时，则到当前的类装载器中查找并载入。所以，可以通过实例化类装载器的方法对 findClass 方法进行重载，例如 URLClassLoader 等等。

在 JVM 中定义了两种类装载器，系统类装载器和用户自定义类装载器。系统类装载器是 JVM 执行时的缺省类装载器，主要用来定位和装载系统类、用户自定义类等等。我们可以通过重载缺省类装载器的办法来定义自己的类装载器，如 NewClassLoader 就是通过对类 ClassLoader 重载得到，具体的 NewClassLoader 定义需要重载的方法如下。

```
public class NewClassLoader extends ClassLoader {
  public Class loadClass(String name,
        boolean resolve)
        throws ClassNotFoundException {…}
  public Class loadClass( String name))
    throws ClassNotFoundException {…}
    … …
}
```

由此也就实现了在程序运行过程中同时载入新旧类定义的问题。

3.2　可动态更新/可被动态更新类定义

通过利用用户自定义类装载器，我们利用下面的方法定义对象，对于非用户自定义的类，则可以和一般的方法一样对它进行操作。

```
public static Object getNewObject
  (String className, NewClassLoader cld)
  throws Exception{
      Class cla=cld. loadClass(className);
      return cla. newInstance();
}
```

对于方法调用和数据域的存取，则需要利用到 Java 的反射机制来实现，这些修改都是一些固定机械性的工作，可以通

过计算机程序自动进行，不过这方面的工作尚未展开。如下说明了如何进行方法调用。

```
public static Object invokeMethod(Object obj,
  String methodName,
  ArgumentHolder argHolder)
  throws Exception{
    //ArgumentHolder 类主要用来包装调用参数
  Method meth =obj. getClass(). getMethod
        (methodName,
        argHolder. getArgumentClasses());
  return meth. invoke(obj,argHolder. getArguments());
}
```

对于类是否需要被动态更新可以通过对 class 文件最近被修改的时间来判断。检查命令可以放在主方法中。每一个可更新类都必须实现一个用来发现是否需要动态更新和启动动态更新的方法。一个可动态更新类将定义如何更新类并且把旧对象上的状态转移到新的对象上去。

在新定义的类中，定义的更新方法主要实现检查其各个子对象是否需要进行动态更新。如果需要，那么首先必须载入子对象新的类定义，然后创建该子对象，并且利用旧的子对象和类装载器作为参数调用新建子对象的动态更新方法；对于其它情况，子对象类定义不需要更新，那么只要把旧子对象复制给相应变量，并且调用子对象的更新支持方法用于检查子对象内部是否有需要动态更新操作的类。

动态更新程序不但要在新类定义中定义更新方法用来转换对象状态，而且还要在被动态更新类定义中增加动态更新操作的动态更新支持方法。在该方法中，对于每一个子对象都进行如下操作：首先判断是否所包含的子对象需要动态更新，如果某些类最近被修改过，需要进行动态更新，那么该方法将会利用新的类定义和类装载器创建一个新的对象，并且以旧对象和类装载器为参数调用新对象的动态更新方法；否则，该方法将会调用子对象的更新支持方法。

在运行中的程序，需要创建一个类名和类装载器的映射关系，之后每一次动态更新都需要对这个映射关系进行维护，以便能够利用最新的类装载器装载特定的类。通过以上的定义，动态更新后的软件中所有需要更新的类对象都进行了更新，而不需要动态更新的类则仍旧保持旧定义，从而达到了动态更新的目的。

在主类中，首先判断是否有更新需求，若有，则创建一个新的类装载器，该类装载器将被用在这个动态更新事务上；然后，对主类中的各个子对象进行判断，如果需要更新则创建新子对象，并且以旧子对象和类装载器为参数调用新子对象的更新方法，否则直接以类装载器为参数调用旧子对象的更新支持方法。每次需要进行动态更新的时候，仅需要重新编译主类，当运行中的系统在检测点发现主类的变化后，它将启动动态更新。

主类不能够被重新装载，因此也就不能够被动态更新，这个问题可以通过对主类增加一层包装的方法来解决，也就是把主类放到一个包装类中，这种情况下可以实现对主类的动态更新，但是作为最外层的包装类却是当前情况下的真正主类，其仍旧不能更新，所以也就遇到了相同的问题。

4 实验

本节首先对多线程软件动态更新进行讨论，然后以多线程软件为例对本文方法实现的动态更新操作进行分析，说明其实现上的可行性和性能上的高效率。

4.1 多线程软件动态更新

每一个需要替换的类都可能拥有很多基于该类定义的对象，也可能有对象的方法正在运行。到目前为止，还没有任何方法可以适用于所有动态更新情况。为了能够进行动态更新，有许多的解决方法可以考虑[1,2,6,7]，例如：(1)等所有需要动态更新的对象生命期结束后，再载入新类定义新对象；(2)区分对待，选择部分需动态更新的对象进行更新，另外部分保持不变；(3)主动更新，通过在新旧对象数据、方法之间建立映射关系，把运行中的对象和方法进行迁移。

上述方法可以运用于不同的情况，例如，方法(1)要进行等待，而且一些运行时间较长的方法或者长期存在的对象则严重影响更新；方法(2)适用于部分情况，对正在运行的系统影响最小；对于方法(3)，程序员在编写程序的时候要做较多的工作，同时也需要对底层运行平台有一定的了解，所以很难实现，特别是对有些程序，动态更新前后的方法之间很难找到对应关系并且创建这种对应关系。

线程是一种特殊的对象，其动态更新操作方法同一般对象。本文保持正在运行的线程持续运行，利用自定义类装载器装载新的类定义。在更新操作结束后，所有新创建的线程都是基于新的类定义，并且使用新的类装载器。至于包含在线程中的对象，它们不需要利用上面提到的方法进行创建和调用，它能自动利用线程的类装载器装载最新的类。

在第 2 节中讨论了动态更新事务的 ACID 属性，它们能够确保类型安全性。在分析更新操作时，假设动态更新可以自动实施。如果软件是非线程软件，那么更新过程中不会被打断，因此动态更新事务能够满足执行的原子性，这就意味着动态更新不会对其它线程产生影响，然而在实际中，动态更新有可能被仍在运行的线程打断，这就容易引起公共数据的不一致性问题。

对动态更新需求的检查是在主函数中，所以更新事务也是从主类中的主函数开始，至于多线程软件，在进行动态更新的过程中仍旧有一些其它线程在运行。如果这些线程存取一个公共的需要进行动态更新的对象，那么不一致性情况就容易发生。例如，软件服务器正在运行，主函数循环不断对端口进行监听，一旦发现有客户端请求服务器的服务，主函数就创建一个连接，启动一个线程为该客户端服务。在动态更新共享对象时，由于非原子运行，因此不一致性可能会发生。

本文实验假设动态更新不涉及公共对象，在这个假设下，每一个线程都是封闭的，因此，动态更新事务对程序的更新是类型安全的，并且是一致的。

4.2 实验分析

本节主要描述一个实验系统，通过对该系统的实现和动态更新技术进行分析，结合实验结果进行讨论。

本文实验程序是一个客户服务器程序，服务器端负责提供服务给客户端。主类中的方法 main()中监听端口，在有客户端请求的情况下则创建一个线程为其服务。方法 main()也记录相应的各个连接和每一个连接请求。在每次服务器收到请求时，程序首先检查主类是否已经被更新，如果主类已被更新，那么调用更新方法开始进行动态更新。虽然主类不能被动态更新，我们重新编译它主要是为了能够把它作为一个更新信号。

客户端发送连接请求，然后提供相关参数给服务器。在收到服务器返回的结果后，客户端将把结果写入文件并且关

闭相关连接。我们使用循环把相关的操作进行包装，所以每一个客户端都能够保证对服务器的请求持续不断，这使得服务器能够维持在一个持续运行、建立线程的状态。

服务器端软件运行在 AMD Sempron™ processor 2500+ 1.41GHz，内存为 448MB。计算机间通过局域网连接，系统运行 Windows XP 系统。所有的代码都用 jdk-1.5.0 进行编译。动态更新事务包括对一个线程和三个相关类的动态更新，更新中包括增加和删除方法/数据域。

更新操作包括检查和装载类文件。动态更新事务总的执行时间少于 30ms。在实验中，通过使用 System.currentTimeMillis()函数来取得精确的时间。图 2 中，X 轴表示的时间序列从 49500ms 到 515000ms，Y 轴表示的时间是 3 个客户端通过局域网连接到服务器并执行完服务需要的时间。更新事务从 505343ms 开始，在 505359 结束。对于客户端而言，大多数的时间都用在通信上。从该图可以看出，更新事务对整个系统的性能影响非常小。主要的原因在于大多时间都是线程和客户端的简单通信等待。更新事务对客户端是完全透明的。

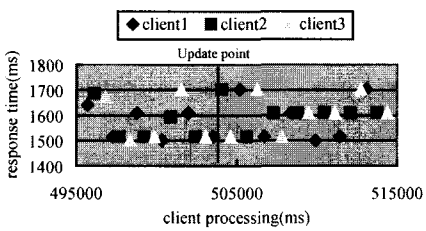

图 2　通过局域网访问的响应时间

图 3 中给出了和服务器运行在同一计算机上的客户端的响应时间。该实验从 1703ms 开始，到 4484ms 结束。大多数的服务请求响应时间都小于 100ms，除了受动态更新事务影响的几个服务，最多情况下需要的响应时间是 130ms 左右。更新服务总共花费了 31ms 的执行时间，其执行时间较长的原因主要在于 CPU 处理时间需要和多个线程共享，而且客户端也在同一台计算机上运行。

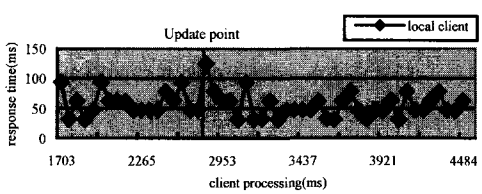

图 3　同一计算机上客户端响应时间

动态软件更新的目标是为了能够保证服务器在执行期间不会因为软件升级等原因中断运行、停止对外服务。通过对这些新的类进行在线更新，我们能够有效地保持应用程序的状态，保证连接持续开放并且可以对外提供服务。在我们的实验中，运行中的线程不会受到动态更新操作的影响。然而，服务器在发送完一个信号后不能够马上进行动态更新，它需要等待静止点，也就是达到检查点后才可以。为了最小化等待时间，对于一些较为复杂的程序，我们通过利用可以加入多个检查点的办法来解决这一问题。

5　相关工作

近 30 年来，在动态软件更新方面做了非常多的工作，同时也提出了各种各样的方法。本节中主要对相关的方法做一个简要的总结。

在 Java 语言方面，M. Dmitriev [1,9] 提出了一个运行时进化方法，该方法基于 HotSpot Java Virtual Machine。文章中还讨论了处理运行中的应用程序的活动方法的计划和策略。文献[2]分析了何种动态软件更新是安全的，并加以证明，该文献还通过对 classLoader 扩展的方法实现了新类的动态装载。文献[10]中提出了一种方法，通过对类进行包装的办法实现动态更新操作，这种方法不需要对 JVM 进行修改，但是它要求增加 wrapper，并且不允许对增加的接口类进行修改，不允许在类之外存取数据域，该方法最大的问题就是严格限制了接口类定义，这也就限制了某些动态更新操作。

在操作系统的动态更新方面，也有相关的一些研究。文献[6]对 Linux 核心模块进行动态更新，它用的是热交换方法(hot-swapping)。热交换方法需要新旧两个模块同时运行，然后把状态从旧模块中转换到新的模块上。K42 是一个面向对象的操作系统，其更新目标为处于静止状态下的对象[11]。

GinSeng[8] 是一个基于 C 程序的动态更新项目，其动态更新方法是生成补丁(patch)，在补丁中描述了新旧两个程序模块的变化。GinSeng 也通过扩展 load 操作对系统进行了形式化研究，并且证明了其类型安全性和 progress。在何时实现动态更新问题上，Ginseng 也要求在程序中有一个静态更新点，在这一更新点进行更新操作能够确保系统全局状态的一致性。每一个应用程序都是基于事件处理机制，在每一个事件处理的末尾，一般程序都处于一个较好的稳定状态，可以作为静态更新点。

结束语　本文提出了动态更新事务的概念，利用它对 Java 程序进行动态更新。在该方法下的动态软件更新不但允许进行类的替换、增加和删除，而且也允许对类中的方法/数据域进行替换、增加和删除。为了保证 Java“一次编译，随处运行”的性质，提出了一种新的动态 Java 软件更新方法，并且该动态更新是完全基于程序级定义的。本文提出的相关技术不需要运行系统的支持，所以可以应用到所有的 Java 程序中。

参 考 文 献

[1] Dmitriev M. Towards flexible and safe technology for runtime evolution of JAVA language applications // Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution. Florida, USA: ACM Press, 2001: 65-76

[2] Malabarba S, Pandey R, Gragg J, et al. Runtime support for type-safe dynamic Java classes // Proceedings of the European Conference on Object-Oriented Programming. Berlin, Germany: Springer, 2000: 337-361

[3] Zhang S, Huang L P. Formalizing Class Dynamic Software Updating // Proceedings of the 6th International Conference on Quality Software. BeiJing, China: IEEE CS, 2006: 403-409

[4] BiermanG, Hicks M, Sewell P, et al. Formalizing Dynamic Software Updating // Proceedings of the Second International Workshop on Unanticipated Software Evolution. Poland, 2003: 13-23

[5] Casais E. Managing class evolution in object-oriented systems // Object-Oriented Software Composition. USA: Prentice Hall, 1995: 133-195

[6] Lee Y F, Chang R C. Hotswapping Linux kernel modules. Journal of Srstems and Software, 2006, 79 (2): 163-175

[7] Gilsi H, Robert G. Dynamic C++ classes- A lightweight mechanism to update code in a running program // Proceedings of USENIX Annual Technical Conference. CA, USA: ACM Press, 1998: 65-76

[8] Neamtiu I, Hicks M, Stoyle G, et al. Practical Dynamic Software

Updating for C // Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation. Ottawa, Ontario, Canada: ACM Press, 2006: 72-83

[9] Dmitriev M. Safe Evolution of Large and Long - Lived Java Applications. PhD thesis. Scotland: University of Clasgow, 2001

[10] Orso A, Rao A, Harrold M J. A technique for dynamic updating of JAVA software // Proceedings of the International Conference on Software Maintenance (ICSM). Washington. D C, USA: IEEE CS, 2002: 649-658

[11] Baumann A, Heiser G. Providing dynamic update in an operating system // Proceedings of the USENIX Technical Conference. Anaheim, CA, USA: ACM Press, 2000: 279-291

(上接第 271 页)

的客体上下文和 *PC*(权限、客体上下文矩阵)可以得到系统活动权限集,系统活动权限集随着客体上下文的改变而改变。用户活动权限集是指由用户活动角色集继承的权限和直接分配的权限之和与系统活动权限集的交集,它随着用户活动角色集和系统活动权限集的改变而改变。

网格环境中基于上下文和角色的访问控制 CRBAC 模型原理如图 1 所示。

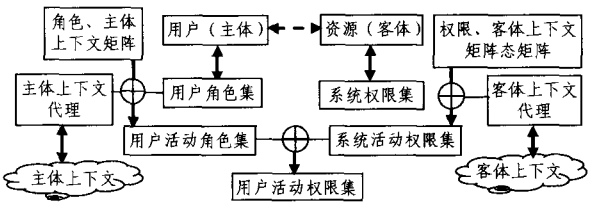

图 1 网格环境中基于上下文和角色的访问控制 CRBAC 模型

3 基于上下文和角色的访问控制 CRBAC 的案例分析

3.1 CRBAC 案例分析

在一个网格环境中有用户 $u1,u2,u3,u4$,角色 $r1,r2,r3,r4$,主体上下文 $c1,c2,c3$,权限 $p1,p2,p3,p4,p5$,客体上下文 $c1',c2',c3',c4',c5'$,*UR*(用户、角色矩阵)、*RC*(角色、主体上下文矩阵)、*RP*(角色、权限矩阵)、PC(权限、客体上下文矩阵)如下所示。

	r1	r2	r3	r4
u1	1	0	0	0
u2	0	1	0	0
u3	0	0	1	1
u4	0	0	0	1

UR(用户角色矩阵)

	c1	c2	c3
r1	0	1	1
r2	1	1	0
r3	1	0	1
r4	1	1	1

RC(角色主体上下文矩阵)

	p1	p2	p3	p4	p5
r1	1	0	1	0	1
r2	0	1	0	1	1
r3	1	1	1	0	0
r4	1	0	1	0	1

RP(角色权限矩阵)

	c1′	c2′	c3′	c4′	c5′	c6′
p1	0	1	1	0	1	1
p2	1	1	0	1	1	1
p3	1	0	1	1	0	1
p4	0	1	1	1	1	0
p5	1	1	1	1	1	1

PC(权限客体上下文矩阵)

若用户当前所处的主体上下文为 $c1$,客体上下文为 $c2'$ 和 $c4'$,系统的活动权限集为:

$SYS\_ACT\_PRMS(c2',c4')=\{p1,p2,p4,p5\}\cap\{p2,p3,p4,p5\}=\{p2,p4,p5\}$

用户 $u3$ 的活动角色集为:

$USER\_ACT\_ROLES(u3,c1)=\{r3,r4\}\cap\{r2,r3,r4\}=\{r3,r4\}$

用户 $u3$ 的活动权限集为:

$USER\_ACT\_PRMS(u3,c1,c2',c4')=\{\{p1,p2,p3\}\cup\{p1,p3,p5\}\}\cap\{p2,p4,p5\}=\{p2,p5\}$

当用户的主体上下文变为 $c2$ 客体上下文变为 $c3'$ 时,系统的活动权限集为:

$SYS\_ACT\_PRMS(c3')=\{p1,p3,p4,p5\}$

用户 $u3$ 的活动角色集为:

$USER\_ACT\_ROLES(u3,c2)=\{r3,r4\}\cap\{r1,r2,r4\}=\{r4\}$

用户 $u3$ 的活动权限集为:

$USER\_ACT\_PRMS(u3,c2,c3')=\{p1,p3,p5\}\cap\{p1,p3,p4,p5\}=\{p1,p3,p5\}$

从以上可以看出:主体上下文决定了用户的活动角色集,客体上下文决定了系统活动权限集,它们一起决定了用户的活动权限集。当用户的主体上下文和客体上下文改变时,用户活动角色集和系统活动权限集也随之改变,从而用户活动权限集也随之改变。

3.2 CRBAC 可行性研究

我们在标准 RBAC 模型的基础上加入了上下文的概念,将上下文分为主体上下文和客体上下文,主体上下文与用户主体环境相关,客体上下文与资源及服务的状态相关,没有冲突,主体上下文决定用户活动角色,客体上下文决定系统活动权限,两者一起决定用户活动权限集,是可行的。

结束语 基于上下文和角色的访问控制 CRBAC 模型在标准核心 RBAC 模型的基础上作了上下文方面的扩展,引入上下文后的访问控制 CRBAC 模型能根据环境上下文动态授权,解决网格环境中上下文敏感的访问控制。由于我们做的扩展是在核心 RBAC 模型的基础上,没有考虑角色继承,这方面还有很多工作需要进一步研究。

参 考 文 献

[1] Sandhu R, Conyne E J, Lfeinstein H, et al. Role - based access control models IEEE Computer, 1996, 29(2): 38-47

[2] 邓集波,洪帆. 基于任务的访问控制模型. 软件学报, 2003: 14(1): 76-82

[3] 陈伟鹤,殷新春,茅兵,等. 基于任务和角色的双重 web 访问控制模型. 计算机研究与发展, 2004: 41(9): 1466-1473

[4] 陈颖,杨寿保,郭磊涛,等. 网格环境下的一种动态跨域访问控制策略. 计算机研究与发展, 2006, 43(11): 1863-1869

[5] 黄建,卿斯汉,温红子. 带时间特性的角色访问控制. 软件学报, 2003, 14(11): 1944-1954

[6] Mossakowski T, Drouineaud M, Sohr K. A temporal-logic extension of role-based access control covering dynamic separation of duties Temporal Representation and Reasoning // 2003 Fourth International Conference on Temporal Logic, Proceedings. 10th International Symposium. 2003: 83-90

[7] Wullems C, Looi M, Clark A. Towards context-aware security: an authorization architecture for intranet environments // Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference. 2004: 132-137

[8] 陈华. 网格的访问控制模型. 微机发展, 2004, 14(8): 27-29

[9] 张纲,李晓林,游赣海,等. 基于角色的信息网格访问控制的研究. 计算机研究与发展, 2002, 39(8): 952-956

[10] 薛伟,怀进朋. 基于角色的访问控制模型的扩充和实现机制研究. 计算机研究与发展, 2003, 40(11): 1635-1642

[11] 孙为群,单保华,张程,等. 一种基于角色代理的服务网格虚拟组织访问控制模型. 计算机学报, 2006, 29(7): 1199-1208