软件系统中代码克隆的检测技术

Techniques of Detecting Code Clones in Software Systems

叶青青 （杭州职业技术学院 浙江杭州 310018）

摘 要： 大型的软件系统常常有很多重复的克隆代码，给软件维护增加了很大的困难。如何利用工具检测这些重复代码是软件工程领域中一个重要的研究课题。本文首先引入了代码克隆的概念和定义，然后深入探讨自动检测克隆代码所面临的难点，并在此基础上详细阐述了利用后缀树检测代码克隆的一般方法。

关键词： 代码克隆 软件维护 后缀树

1 引言

软件系统常常含有大量的重复代码，有些甚至高达38%[2]，这些重复的代码通常被称为代码克隆。即使是业界公认的高质量的系统也不例外，如 Linux 含有 15-25% 左右的克隆代码[1]，GNU 系统有 9% 左右的克隆代码[2]，JDK 则有 21-29% 的代码代码[3]。

近年来，关于代码克隆的研究得到了广泛重视，已成为软件工程研究中的一个重要的前沿课题。各国的软件研究人员提出了多种检测代码克隆的方法，并开发了多种自动检测大规模软件系统中代码克隆的工具[2, 3, 5, 6]。本文着重介绍软件克隆的概念和检测时所面临的各种技术难关，并在此基础上详细阐述利用后缀树检测代码克隆的一般方法。

2 代码克隆的定义

一段程序代码 M 如果与另一段程序代码 N 有相当程度的相似性，我们把 M 称为 N 的代码克隆，M 和 N 构成一个代码克隆对。基于对相似性程度的不同定义，我们可以把代码克隆分为五类：完全克隆、格式克隆、参变克隆、断层克隆、和功能克隆。

(1) 完全克隆。如果 M 的文本序列和 N 的文本序列完全相同，M 是 N 的完全克隆。如图 1-1 中代码段 Q1(3-6 行)就是图 1-0 中代码段 P(3-6 行)的完全克隆。

(2) 格式克隆。如果排除程序代码在换行、空白、制表等方面的格式上的区别，以及注释句等方面的区别以后两个代码段 M 和 N 符合完全克隆的定义，那么 M 是 N 的格式克隆。如图 1-2 中的 Q2(3-6 行)是图 1-0 中 P 的格式克隆。

(3) 参变克隆。如果 M 和 N 之间除了常数值以及变量名、函数名等标识符之间的区别以外满足格式克隆的定义，那么 M 是 N 的参变克隆。如图 1-3 中的 Q3(3-7 行)就是图 1-0 中 P 的参变克隆，因为若把 Q3 中的变量 p 换成 a，q 换成 b，再把常数 3 换成 0 后，Q3 是 P 的格式克隆。

(4) 断层克隆。如果 M 和 N 之间除了少量行以外满足参变克隆的条件，M 是 N 的断层克隆。当软件开发人员在新程序里拷贝了 N 并删除 N 中若干行或插入新行时，就会造成断层克隆代码。如图 1-4 中的 Q4(3-8 行)是图 1-0 中 P 的断层克隆，Q4 比 P 多了第 6 行。

(5) 功能克隆。如果 M 和 N 在计算功能上完全相同，即 M 和 N 满足完全相同的程序前置条件和后置条件，那么不管他们在文面上有多么不同，他们是一对功能克隆。如图 1-5 中的 Q5(4-8 行)就是 P 的功能克隆。

3 代码克隆检测的难点

克隆检测就是发现软件系统中克隆的代码。最理想的检测是找出所有的功能克隆，但 由于目前尚没有技术对程序的语义进行快速的比较，还无法检测功能克隆。检测完全克隆相对来说比较容易，因为它其实是文本匹配问题的一个变种。代码克隆检测的主要研究集中在对格式克隆、参变克隆和断层克隆的检测。

检测代码克隆的方法和工具接受一组源程序代码的集合作为输入，经过计算输出一组克隆簇。每一个克隆簇含有多个代码段，其中的每一个代码段都和同簇的其他代码段构成克隆对。克隆检测通常分三个主要步骤：第一步分段，即把被检测对象的程序分割成代码片断；第二步逐段比较，即按相似性程度把代码片断归入克隆簇；第三步生成克隆报告。

统研究中的一个重要方面。代码克隆检测本身不是目的，而只是软件开发人员用以维护或理解软件系统的手段，软件开发人员希望检测工具只找出那些他们感兴趣的克隆。第四个问题是语言依存度。代码克隆检测方法和工具是否只能处理一种程序设计语言？它是否能处理同一种程序设计语言的各种方言？把它移植到其他程序设计语言上有多困难？第五个也是最重要的问

图 1　代码克隆例

克隆检测方法和工具必须要解决以下五个方面的问题。首先要避免漏报现象，即检测方法和工具应该致力于找出被检测的程序集合中存在的所有克隆。如果被检测的程序集合中有些代码克隆没有出现在检测结果中，这些没有被检测到的代码克隆就成了漏报克隆。其次要避免虚报现象，即输出结果里不该包括那些并非克隆的代码。如果输出结果中有实际上并不是克隆的代码，这些代码就构成了虚报克隆。漏报和虚报之间有相反的关系，一个方法在减少漏报的同时往往会导致虚报的增加，反之亦然。举极端的例子说，把所有的程序段对都列出来的话就不会有漏报，但会有太多的虚报而令结果毫无意义。同样地，如果什么输出都没有，就不会有虚报，但会有无数的漏报同样地使结果变得没有意义。第三、要检测和报告有意义的代码克隆。有些代码克隆，如变量定义、数组初始化等总具有相同的结构，他们的存在是必要的，不需要也不可能消除，检测和报告这些代码克隆并无实用价值。怎样从检测出的大量的克隆簇中遴选出对软件开发人员来说有意义的代码克隆，并以怎样的方式向软件开发人员报告是克隆检测系题是检测方法和工具的可扩展性，即能否迅速处理大规模的实际软件系统。代码克隆涉及到逐对比较，当输入的程序集合很大时，对计算速度和内存容量都有很大的挑战。

4　代码克隆的检测方法

基于字符串匹配的代码克隆检测方法是目前较为主要的检测技术[2, 3]，它把计算机程序当作由字符串组成的正文文本，利用字符串匹配的后缀树算法检测代码克隆。由于它不考虑程序的文法，对语言的依存度低，可容易地开发出处理多种语言的检测工具。字符串匹配技术的成熟度也使它具有优秀的可扩展性，可快速地处理大规模的软件系统。该克隆检测方法分三个基本过程：Token 序列生成、克隆对检测、和克隆报告生成（图 2）。

4.1　Token 序列生成

本过程对输入的源程序作简单的词法分析，生成 Token 序列。克隆检测的输入对象往往是由多个程序构

成的一个系统,所有的输入程序首先被合并为一个文件,然后再被转换成一个 Token 序列的正文文件。为了提高克隆检测的效果,这个过程需要对程序作一些适当的变形处理。

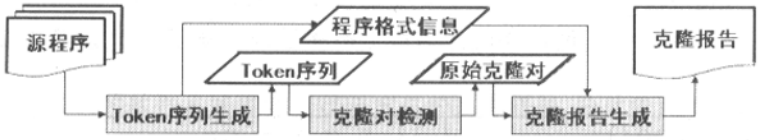

图 2 基于字符串匹配的代码克隆检测流程

最基本的变形处理是删除对程序功能没有影响的注释部分和冗余的空白字符(如重复的空格和制表符等),提高检测出格式克隆代码的能力,减少克隆漏报。有些方法还会删除换行符,以最大程度消除仅由于程序格式的不同而造成的区别,但换行符的删除也会增加系统的误报。在作这些删除处理时,要把被删除的内容保存在程序格式信息文件中,生成克隆报告时需要利用这个文件来重构源文件。

进一步的变形处理往往和具体的程序语言有关,它们的目的是缩短生成的 Token 序列,提高检测系统的效率和可扩展性。如果程序语言不区分大小写,所有的字母都应转换成小写或大写。有些预处理的编译指令,如 C 中的#include,Java 中的 import 等应删除,以减少 Token 序列的长度。程序语言中有些冗余的语句结构也可以缩短,如考虑删除 else、const、protected 等。很多程序语言含有数组初始化的结构,这些结构与代码克隆的检测无关,所以也应删除。通过这些处理生成的 Token 序列会更短,提高克隆检测的速度因为后续的克隆检测算法的运行时间和需要的内存量都取决于 Token 序列的长度。同样地,上述处理必须要同时把格式信息记入到程序格式信息文件中。

第三类的变形处理也称参量代入,即把程序中的各种标识符用一个特殊的符号代替。不同的检测工具所采用的参量代入的范围各有不同。比较简单的代入仅仅用同一个特殊符号,比如说用 $ 代替程序中所有的变量名。更大范围的代入则用同一个特殊符号同时代替变量名和函数名。最彻底的代入则用同一个特殊符号代替程序中所有的变量名、函数名、类型名、常量名和常量;这种参量代入后产生的 Token 序列中除了运算符和程序语句结构里的保留字(如 if, while)以外就只有特殊符号了。一般认为参量代入的彻底程度和漏报率与误报率有关系,参量代入范围的扩大可减少漏报,但会同时增加误报,相反,低范围的参量代入会导致漏报的增加但也会减少误报。根据检测工具的具体目的和用途,设计人员应采用相应范围的参量代入。经参量代入后生成的 Token 序列中每一个特殊符号所代表的本来的标识符都要记录在程序格式信息文件中。

比如说从图 1 – 0 中的程序 P 可以生成下列的 Token 序列:

$ = $; $ = $; if ($ > $) { $ = $ + $; $ = $; }

4.2 克隆检测

克隆检测过程的输入是如上所示的由 Token 构成的正文文件,输出是 Token 序列中互相匹配的 Token 子序列对。我们可以利用后缀树[7]匹配抽出其中的极长匹配子序列对。

后缀树是一种利用树来表示字符串的数据结构,可高速地处理各种字符串的操作。当用它来表示字符串 S 时,S 中的每一个后缀子串都有一条从根结点到叶子结点的路径。后缀树中的边代表一个字符,叶子结点代表子串的结束。叶子节点的个数与 S 中字符的个数相同。当两个子串共享从根开始的路径时,两个子串有相同的字符子序列,即有共同子串。

图 3 是 ABCDABCXD $ 的后缀树,其中 $ 表示字符串的结束。包括终结符 $,该字符串有 10 个字符,10 个后缀(包括其自身和仅有终结符的空串),因此它的后缀树有 10 个叶子结点。从后缀树中我们可以看到它有 4 个重复的子串:ABC、BC、C、D(图中用粗线表示)。

利用后缀树检测克隆代码时,我们首先把源程序生成的 Token 序列当作一个字符串,并构建出它的后缀树,然后从后缀树中抽出共同的子串。在构建后缀树时保存各边所代表的各个 Token 在源程序 Token 序列中的位置,我们就可以找出匹配的 Token 序列。比如说我

们把上例的 ABCDABCXD $ 表示成下列的形式：

A B C D A B C X D $
0 1 2 3 4 5 6 7 8 9

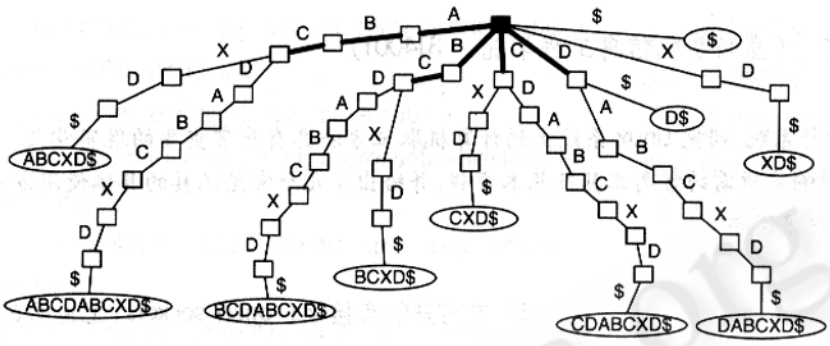

图 3　后缀树

那么输出的匹配对就是{(0,2; 4,7), (1,2; 5,6), (2,2; 6,6), (3,3; 8,8)}(分别对应 ABC、BC、C 和 D)。而这些匹配对就构成了我们要检测得代码克隆的原始克隆对。

很显然，如果只有一个 Token 匹配的克隆对是没有意义的。事实上，由于匹配的对象是 Token，而 Token 在程序中是最基本的单位，我们必须要限制克隆对的最小长度，否则这一过程输出的克隆对会太多，多得没有实际上的用处(所有的变量名都会是克隆)。经验表明 15 个以上的 Token 是比较好的缺省值，也就是说如果一对克隆的 Token 的长度不大于 15，我们就不把它们列在克隆对的结果中。因为一般的程序语句行平均有 5 个左右的 Token，选择 15 个 Token 意味着克隆代码大概会有 3 行左右。这一阀值应可由用户调节，当系统产生的克隆对太多的时候，用户们可增大这一阀值以减少克隆对的数目，相反，如果克隆对太少，则可适当的降低阀值。

4.3 克隆报告生成

上述的检测过程生成的只是原始数据和原始克隆对，必须要作进一步的处理才能够把生成让软件开发者看得懂的克隆报告。首先要把基于变形后的 Token 序列的克隆还原成代码。利用在 Token 序列生成过程中保存的程序格式信息文件，找出与原始克隆对中的 Token 位置相对应的源程序文件名和代码在该文件中的行列位置。如果原始克隆对的起始 Token 位置落在行中，要把它往前推移到下一行的开始位置，同样的，如果原始克隆对的终止 Token 位置落在行中，要把它往后移到前一行的结束位置。经这样变换后的克隆对如果只有一两行，这些克隆对要被舍弃。具体要报告多少行以上的克隆也应该可由用户在使用克隆检测工具时动态地指定。

最终的克隆报告可以采用如图 4 所示的下三角距阵方式报告克隆。横轴和纵轴都是源程序的行，如果 Fi 文件的 x 行和 Fj 文件的 y 行存在着克隆关系，就标上一个记号。图形报告方式有助于发现断层克隆，因为如果两个克隆对在图中很接近，它们可以被合并成一个克隆对。图 4 显示了对图 1 中的 P、Q3 和 Q4 进行克隆检测的结果。

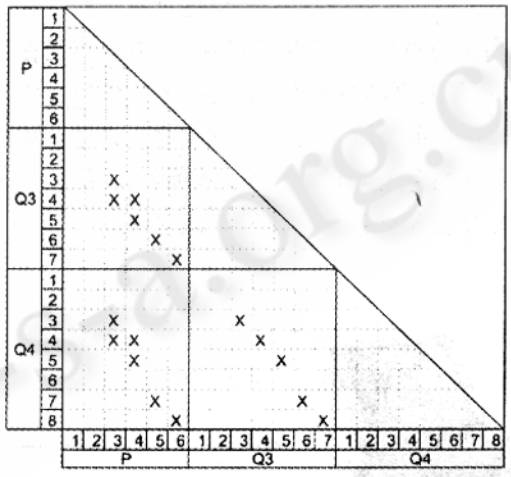

图 4　克隆报告

5　结束语

由于其对大规模软件系统维护的重要作用，克隆检测是近年来国际软件工程研究的一个重要热门课题。本文详细阐述了代码克隆的概念和代码克隆检测时面临的各种挑战，并描述了利用后缀树检测代码克隆的一般方法。

(下转第 77 页)

(上接第 97 页)

参考文献

1 Antoniol, G., et al., Analyzing Cloning Evolution in the Linux Kernel. Journal of Information and Software Technology, 2002. 44(13): p. 755 - 765.

2 Baker, B. S., On Finding Duplication and Near - Duplication in Large Software Systems, in Proceedings of 2nd IEEE Working Conference on Reverse Engineering. 1995. p. 86 - 95.

3 Kamiya, T., S. Kusumoto, and K. Inoue, CCFinder: A Multilinguistic Token - Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering, 2002. 28(6): p. 654 - 670.

4 Kim, M., et al., An Ethnographic Study of Copy and Paste Programming Practices in OOPL, in Proceedings of 2004 International Sympoisum on Empirical Software Engineering. 2004. p. 83 - 92.

5 Baxter, I. D., et al., Clone Detection Using Abstract Syntax Tree, in Proceedings of 1998 International Conference on Software Maintenance. 1998. p. 368 - 377.

6 Basit, H. A. and S. Jarzabek, Detecting Higher - Level Similarity Patterns in Programs. 2005: p. 156 - 165.

7 Weiner, P. Linear Pattern Matching Algorithm. in Proceedings of 14th IEEE Symposium on Switching and Automata Theory. 1973.