基于抽象语法树分析的版本控制分支合并算法①

李 郑[1], 李 姝[2], 王 俊[1], 刘士进[1]

[1](国网电力科学研究院, 南京 211100)
[2](国网上海市电力公司信息通信公司, 上海 200000)

摘 要: 在软件开发中, 并行开发已经成为了现实中标准的开发模式, 因此软件开发的版本控制在开发过程中得到了非常广泛的应用. 虽然大多数的版本控制工具都能完成分支合并以及将不同版本的更改合并起来, 但这些工具往往是以一行一行的代码为单位进行文本比较的合并, 在遇到某行被同时修改时, 则不能给出满意的合并结果. 给出了一种新的合并算法, 用来解决文本比较分支合并中的问题, 它在传统工具的文本比较出现冲突的代码行, 对该代码块建立抽象语法树, 根据抽象语法树的分析比较, 完成不同分支版本代码的自动合并, 减少分支合并中的代码冲突及手动合并的工作量, 提高开发效率.

关键词: 抽象语法树; 版本控制; 分支合并

Revision Control Branch Merging Algorithm Based on AST

LI Zheng[1], LI Shu[2], WANG Jun[1], LIU Shi-Jin[1]

[1](State Grid Electric Power Research Institute, Nanjing 210003, China)
[2](State Grid Information and Communications Co of Shanghai Electric Power Co. Ltd, Shanghai 200000, China)

Abstract: Software Development, where developers work together parallelly is the standard develop mode actually, needs the revision control tool to assistance the development widely. Most of the tools today could merge the code changes from different branches edited by different developers. These tools are based on code text differ comparation line by line, and the merged code isn't always the prediction of the developers. This paper presents a new branch merging algorithm which intends to provide a better result. We build an abstract syntax tree of the code block where the traditional merge gives a code confliction. Then we merge this code block of different revisions automately based on AST nodes comparative analysis. In this way, the code conflictions and manual work in branch merging could be reduced. At the same time, the development efficiency would be improved.

Key words: revision control; AST; branch merging

由于软件开发规模的扩大, 软件开发的过程也更为复杂, 软件开发的方式已经转变为集团化, 分布式, 团队协作的开发方式. 在这种开发方式下, 开发过程往往需要将软件的开发版本恢复到某一历史状态, 以及如果多个开发人员同时修改了程序该做如何处理, 软件配置管理工具能够协助我们解决这些问题. 其中版本控制是软件配置管理的基本要求, 它可以保证在任何时刻恢复任何一个版本、版本控制还记录每个配置项的发展历史, 这样就保证了版本之间的可追踪性, 也为查找错误提供了帮助, 版本控制也是支持并行开发的基础.

1 版本控制

版本控制用于维护文件的所有版本, 随着时间的推移, 系统不断产生新的版本. 使用版本控制系统, 人们可以返回到各个文件以前的修订版本, 还可以比较任意两个版本以查看它们之间的变化. 通过这种方式, 版本控制可以保留一个文件修订的可检索的准确历史日志, 甚至将整个项目都回退到过去某个时间点的状态. 更重要的是, 版本控制系统有助于多个人(甚

① 收稿时间:2014-06-24;收到修改稿时间:2014-08-11

至位于完全不同的地理位置)通过 Internet 或专用网将各自的更改合并到同一个源存储库, 从而进行协同开发项目.

版本控制工具中的关键操作有签入, 签出, 提交, 存储库等等, 本文中讨论的重点在版本控制的分支, 合并以及代码冲突.

1.1 版本控制的分支合并

分支是指目录和文件的现有原始版本发展路径(代码主干)的副本. 分支的生命周期是从某次提交的版本副本开始的, 并从此副本处移动, 形成自己的历史. 合并是指将某分支上的更改融入到代码主干或另一个分支.

假设我们的项目代码是提交到代码库的主干来进行维护的, 此时接到了一个任务, 完成这个任务需要三四个开发人员的合作, 且开发人员间需要共享代码, 这样就可以在代码库创建一个专为这次任务的分支, 参与此次任务的人员在该分支上做开发, 等完成之后再合并到代码主干上, 分支建立避免了将实现了一半的不完整功能提交到主干上, 影响主干代码的正常工作. 同时,由于一段时间在分支上独立开发, 可能主干代码在此时已经被其他开发人员修改, 通常叫代码的并行更改, 此情况下当分支代码合并回主干时就会出现较多的代码冲突. 因此为了减少代码冲突, 我们需要对上述分支合并的过程进行深入研究.

版本控制的分支合并, 就是把在版本控制中的不同分支的文件集合中的多处更改进行整合的操作. 当前的版本控制工具大多提供了分支功能, 分支合并也是其中核心问题, 完成分支文件内容合并的合并结果影响着软件并行开发的效率.

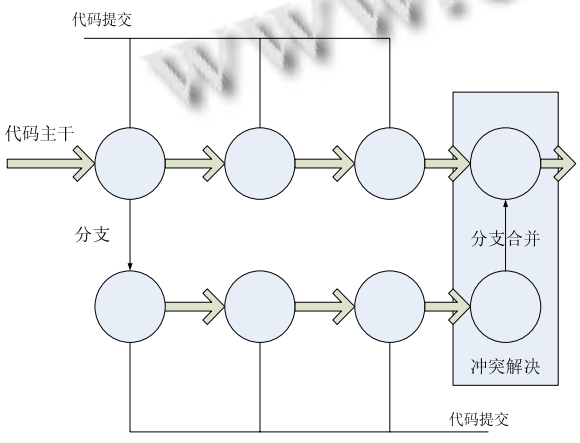

图 1　版本控制的分支合并

在某些情况下, 合并可以被自动执行, 因为有足够的版本历史信息来重构代码变化信息, 并且这些变化信息也不会发生冲突. 但在另一些情况下, 必须手动的决定分支合并过程中到底哪些代码是合并的文件应该包含的内容.

自动进行合并是版本控制工具对同时发生(在逻辑意义上)变化的文件进行的合并的操作, 它提高了软件开发过程中的开发效率, 也是分支合并力求最好的结果.

手动合并是在代码产生冲突的过程中, 不得不有开发人员(可能通过合并工具辅助)进行手动的修改他们需要整合的差异文件. 当自动运行合并遇到更改冲突的时候, 工具就会标记冲突发生, 需要进行手动合并, 当前的版本控制工具很少有自动合并工具可以合并两个同时修改了同一行的代码(比方说, 一个改变了参数名, 另一个添加了注释). 在这些情况下, 当前的版本控制工具需要用户手动修改预期的合并结果.

分支合并算法是一个较为活跃的研究领域, 也有许多不同的算法支持更好的自动合并, 他们之间有着细微的差别. 常见的合并算法包括三路合并, 递归的三路合并, 模糊的补丁程序, 编织合并, 补丁转换.

1.2 三路合并算法

我们假设有两个人对同一个文件进行修改, 如图 2 所示, 对比发现两个文件在第 30 行不同, 此时如果另外一个人丙去看甲和乙修改过后的两个文件, 那么他如何判断是甲还是乙对代码的第 30 行做了修改? 或者是甲和乙都做了修改? 如果此时丙能获得甲和乙修改的原始版本, 那么上述问题即得到解决, 甲对代码第 30 行做出了修改, 而乙没有修改, 而此时对甲和乙的文件版本进行合并时, 即判定接受甲的修改操作, 将修改后的代码保留在最终合并的文件中. 这种合并方式即为三路合并(three-way merging).

三路合并算法是当前用的最广泛的分支合并算法, 它是在文本差异自动分析后, 对两个新的文件版本执行合并操作, 同时也参考其共同父节点或共同的祖先节点, 在上例中即供参考的原始版本. 它是一个粗略的合并方法, 因为它仅需要找到一个共同的祖先来完成合并更改.

三路合并使用祖先节点的文件版本, 以确定派生版本的内容中哪些是已被更改的代码块. 如果没有任何一方发生改变, 则原样保留. 若只有一个分支版本

发生改变则使用已经改变的块. 如果一个代码块在两个派生版本中均被改变, 即同一个代码块在两个版本均被编辑, 且变化不同, 则它被标记为一个冲突的情况, 交给用户来手动编辑.

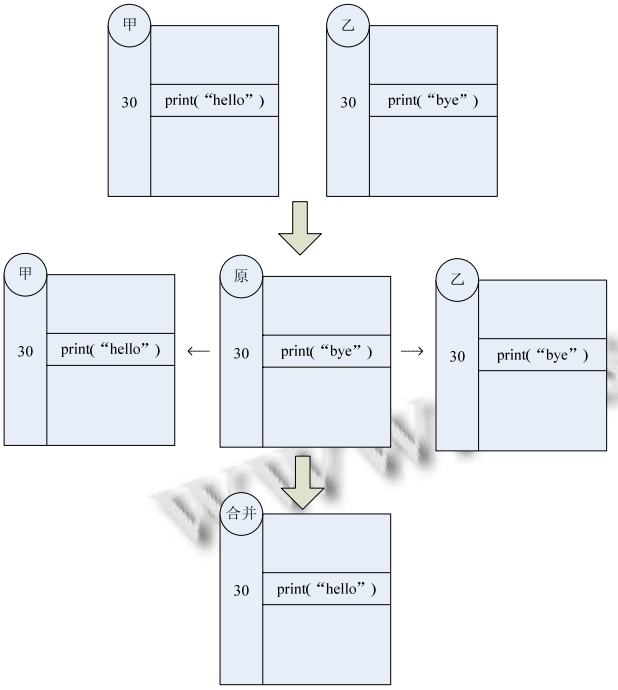

图 2 三路合并

我们假设现在有两次提交分别记为 *A* 和 *B*, 这两次提交并且处于不同分支, 如图 3, 那么 *A* 与 *B* 的合并 $A\vee B$ 就可以表示为$[A\vee B]=[A]+[B]-[C]$, 这里 *C* 为 *AB* 共同的祖先节点, 也就是合并的基础节点. 之所以要减去[*C*], 是因为它的内容必然在[*A*] [*B*]中均已包含, *A*+*B*-*C* 就是三路合并的算法, 你也可以把它看做是将 *B*-*C* 的补丁作用于 *A*, 或者是将 *A*-*C* 的补丁作用于 *B*[8-10].

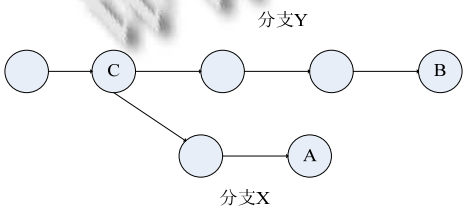

图 3 分支合并图例

三路合并算法需要找到两个合并节点各自分支的共同祖先, 这里我们如果遍历 *A* 或 *B* 节点版本中的代码, 假设其中有代码块 *w*, 我们根据 *w* 是否在 *A*/*B*/*C* 三个节点中的状态进行讨论, 依据三路合并算法的合并规则, 其分支合并的结果如表 1.

表 1 三路合并算法规则[7,8,10]

代码块 *w* 存在于 *A* 中	代码块 *w* 存在于 *B* 中	代码块 *w* 存在于共同祖先 *C* 中	代码块 *w* 存在于合并结果中
是	是	是	是
是	否	否	是
否	是	否	是
否	是	是	否
是	否	是	否
是	是	否	是
否	否	是	否

当然这里的合并规则并非是完美的解决方式, 他有可能造成部分代码错误. 因为实际情况远比上述表格中的情况复杂的多, 例如在 *A* 和 *B* 的共同前置祖先节点不唯一的情况下, *A* 和 *B* 不一定只有一个共同祖先节点, 它的祖先节点有可能是一个集合 $C_1, \ldots, C_k$, 对于 Git 而言它会首先需要构造一个虚拟的祖先节点, 也就是对 *C* 进行合并 $C=C_1\vee\ldots\vee C_k$, 然后再进行三路合并算法进行分支合并计算[*A*]+[*B*]-[*C*], 这种算法被称为递归三路合并的算法

三路合并算法是通过 diff3 程序[17, 18] 实现的, diff3 是 Linux 中文件比较的命令, 他通过比较 3 个文件(通常其中两个文件都是由另一个文件修改后得到)diff3 可以给出两个更改版本分别做出了哪些更改, 也可以用 *diff3 -m* 和*-merge* 将两个开发人员所作不同的更改合并到一个新的程序文件中去. 它的优点在于允许从文件锁定基础的版本控制系统转化到已分支合并为基础的版本控制系统. 它被广泛使用在版本控制系统中(CVS、SVN)

1.3 建立抽象语法树

语法树是描述程序语法结构的一种图形表示, 语法树描绘了如何从文法的开始符开始推导出它的语言中的一个语句, 形象的说它具有如下特性:

1. 树根标记为开始符号
2. 每个叶节点由记号标记
3. 每个内节点由一个非终结符标记
4. 如果 *A* 是某个内节点的非终结符号标记, $X_1, X_2\ldots, X_n$, 是该节点从左到右排列的所有子节点的标记, 则 $A\rightarrow, X_1, X_2\ldots, X_n$,是一个产生式, 这里 $X_1, X_2\ldots, X_n$, 是一个终结符或非终结符[1,5].

要建立一个抽象语法树，必须以抽象语法树的文法为前提，文法是决定怎样将语言的元素组合起来的规则的集合，给定一个编程语言后，可以根据语言的特点定义出抽象语法树的文法，构造语法树.

抽象语法树的每一个节点都是一个类对象，因此在代码解析器中，节点类包含了对所有文法规则对应的基本类.

解析器通过扫描源代码，一一匹配从而去创建一个个节点类对应的对象，这一步是词法分析器的任务，它逐字逐句的对源程序进行扫描，产生单词符号，返回一个标识号，用于判断哪一个节点类需要创建. 然后是语法分析，的是在词法分析识别出单词符号的基础上，建立一棵与输入串相匹配的语法分析树，在解析类中，每个节点类都对应一个调用方法，用于创建这个节点类对象.

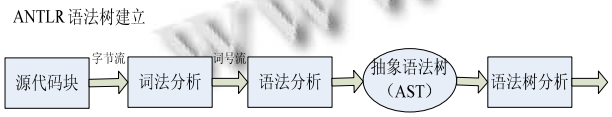

图 4　ANTLR 语法树的建立

本文中使用 ANTLR 开源语法分析工具完成代码到抽象语法树的转换，使用 ANTLR 生成抽象语法树分为两步，第一步是读取代码，然后根据文法规则，ANTLR 生成响应词法和语法分析器，利用词法分析将代码转换为短语流，作为语法分析器的输入从而得出抽象语法树.

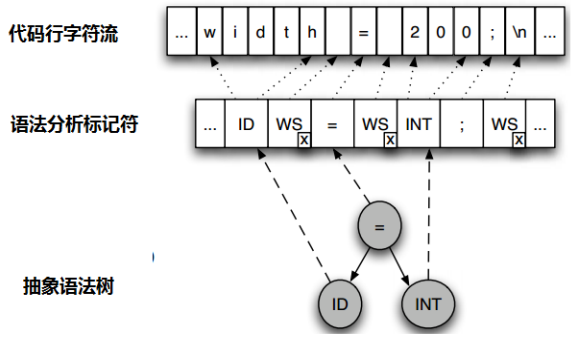

图 5　生成抽象语法树

1.4 基于抽象语法树的分支合并

通常而言在软件并行开发中，两个开发者同时修改代码的情况非常的常见，好在这两个开发者通常是在不同的分支上进行修改，根据现有的以代码文本“行”为单位的代码合并算法而言，大多数的单方修改都能够很容易的被识别，并完成自动分支合并.

我们这里的考虑的情况主要集中在现有的版本控制工具中被认为是产生冲突的情况， 当然也并非是说我们提出的算法可以很好的彻底的解决任何的冲突，任何的算法都不可能解决所有的问题，本算法是通过必要的分析对原本不应该产生冲突的地方进行分支合并，以减少分支合并的冲突.

根据 1.1 节的分析我们假设需要对分支 *X* 上的节点 *A*，分支 *Y* 上的节点 *B* 进行合并，如图 3，其最近共同祖先为 *C*：*A* 节点中有 *CodeBlockA*，*B* 节点中有 *CodeBlockB*，而在 *C* 中并没有这两段代码，在现有以文本比较为基础的合并工具中，该出会被认为是一种冲突，需要用户手工修改.

该代码片断在现有的版本控制工具中，由于 *A,B* 中方法的实现体不同,因此会被认定为冲突 ，其实我们看到在 *A* 和 *B* 中，分别定义了两个无关的参数以及两个无关的函数体，如果对 *A,B* 进行语法分析，很容易的完成 *A, B* 的合并，那么究竟我们是如何做到这样的合并的，首先我们需要对上述 *A, B, C* 中的代码建立抽象语法树，然后根据语法树比较，对其树子节点进行三路合并.

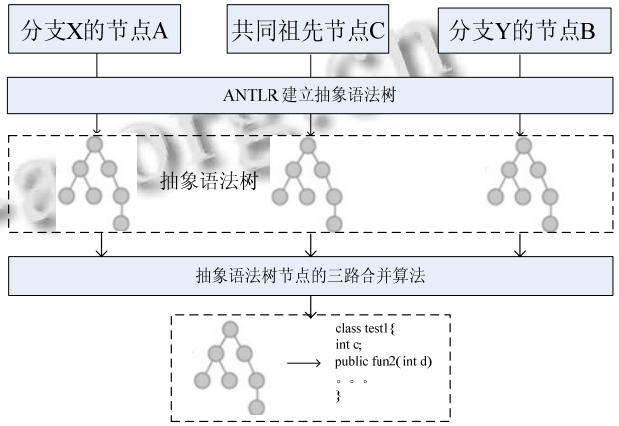

图 6　基于抽象语法树的三路合并

根据抽象语法树进行节点的三路合并主要有以下步骤:

① 依次遍历树 *A* 根节点，然后在 *B* 树和 *C* 树中查找该节点，将其状态纪录在一个数组中

② 以先根序列遍历 *A* 树中每个节点的子节点，将其状态纪录在一个数组中.

③ 如果对树中节点比较, *ABC* 三树当前节点相同,

但有节点其中有一个或多个存在子节点则继续遍历子节点.

④ 若当前节点相同, 均存在子节点且不同, 则在该节点的上级节点标注冲突

⑤ 最后根据维护数组中的状态计算得到合并的结果集并输出合并代码.

表 2 算法的伪码实现

```
AST treeA;
AST treeB;
AST treeC;
function ASTMerging()
{
if($rootA!=null)
{
    Map map = new HashMap();
    if(GetthreeWayMerging($rootA)!="remove")
    map.put($rootA, GetthreeWayMerging($rootA))
    if(nodeExistOnB())
    nodeB = searchTreeB($rootA.type,$rootA.value);
    if($rootA.hasChild())
     if(nodeB.hasChild() )
     {for(child in $rootA.children)
          { if(child notExistOnB)
               map.put(child, "conflict");
          }
          ASTMerging(child);
     }
     else{map.put($rootA.children, "MergingIn")
      }
     else{
     if(nodeB.hasChild())
     for(child in nodeB.children)
      map.put(nodeB.children, "MergingIn")
     }
     }
    return map;
}
}
function ThreeWayMerging(node)
{  if(nodeExistOnTreeA()==false  &&nodeExistOnB()  ==  true&&
nodeExistOnC()== true)
    return "remove"
    else if (nodeExistOnTreeA()==true &&nodeExistOnB() == false&&
nodeExistOnC()== true)
    return "remove";
    else if (nodeExistOnTreeA()==false &&nodeExistOnB() == false&&
nodeExistOnC()== true)
    return "remove";
    else{ return true;}
}
```

2 实例场景分析

下面我们来看具体的一个合并场景, 其中有两个分支 *X* 和 *Y*, 共同的祖先节点为 *C*, 分支 *X* 和 *Y* 上分别有衍生版本 *A* 和 *B* 节点, 其代码如表 3.

表 3 分支合并代码示例

A 节点代码	B 节点代码	C 节点代码

class test
{ int f;
int num1;
float num2;
int getf()
{ int a =
getAValue();
int b= getBValue();
if(a!=NumberBase.A
default)
return a +
b;
 	
else{ return
getSymbal(a) ; }
}
}
	class test
	{ int f ; float num2;
	int num1 = 0;
	int getf()
	{ int a =
	getAValue();
	int b= getBValue();
	if(a!=NumberBase.Ade
	fault)
	return a -
	b;
	
	else{ return
	getSymbal(a) ; }
	}
	}
		class test
		{ int f;
		int getf()
		{ int a =
		getAValue();
		if(a!=NumberBase.Ade
		fault)
		return a;
		
		else{ return
		getSymbal(a) ; }
		}
		}

由语法分析工具分别对三个节点的 class test 代码进行语法解析, 并且分别建立抽象语法树, 语法树的图形表示如图 7 所示.

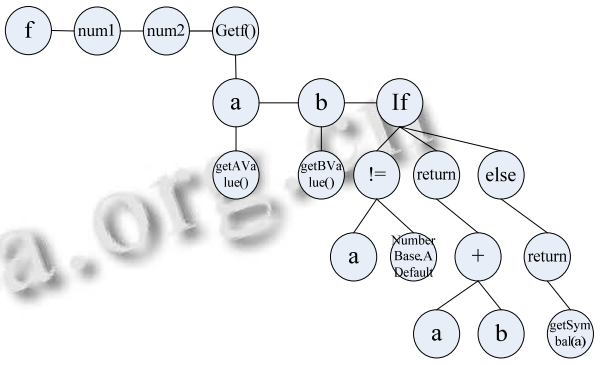

图 7 A 节点代码抽象语法树图示

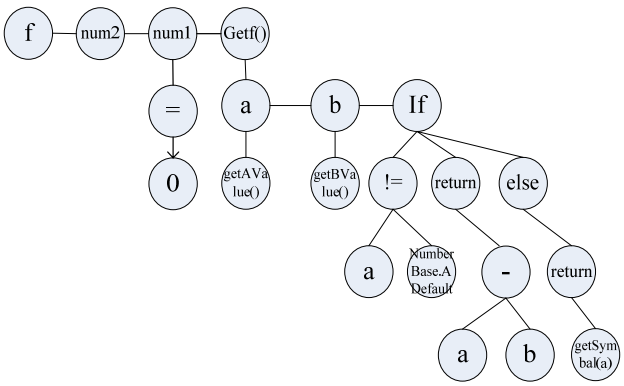

图 8 B 节点抽象语法树图示

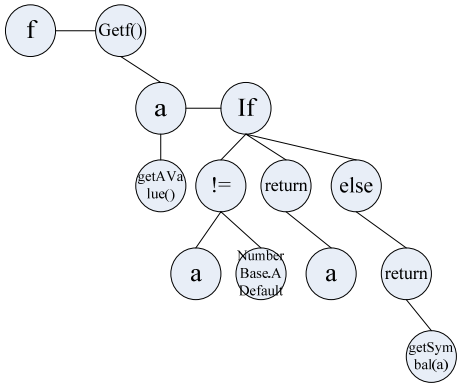

图 9　C 节点抽象语法树图示

根据建立的抽象语法树进行分析如下

首先遍历抽象语法树 *TreeA* ,简写为 *A*, 获取第一层的节点 *num1*, 遍历搜索 *treeB*, 和 *treeC*, 简写为 *B*, *C*, 我们根据遍历分析维护下面一张表, 前三列分别为树 *A*, *B*, *C* 上对应的节点, 最后一列为根据三路合并算法的规则表格 1 哪个节点会出现在合并结果中.

表 4　语法树节点对比信息表 1

TreeA	TreeB	TreeC	Result
A: (int)f	B: (int)f	C: (int)f	(int)f 合并到结果集
A: (int)num1(无子节点)	B: (int)num1(有子节点)	不存在	遍历子节点
不存在	B: (int)num1: =0	不存在	treeB: (int)num1: =0 合并到结果集
A: (float)Num2	B: (float)Num2	不存在	(float)Num2 合并到结果集
A: (int)Getf()(有子节点)	B: (int)Getf()(有子节点)	C: (int)Getf()(有子节点)	遍历子节点

遍历分析函数 *getf()*, 已知 *getf* 节点类型和参数均相同, 同样的方式对 *getf()*方法的抽象语法树进行分析, 这里我们方便, 在写法上均省略了*(int)getf()*:

表 5　语法树节点对比信息表

TreeA	TreeB	TreeC	Result
A: (int)a(有子节点)	B: (int)a(有子节点)	C: (int)a(有子节点)	遍历子节点
A: (int)a: =: getDefaultA	B: (int)a: =: getDefaultA	C: (int)a: =: getDefaultA	(int)a: =: getDefaultA 合并到结果集
A: (int)b(有子节点)	B: (int)b(有子节点)	不存在	遍历子节点
A: (int)b: =getDefaultB	B: (int)b: =getDefaultB	不存在	(int)b: =getDefaultB 合并到结果集
A: If 语句(有子节点)	B: If 语句(有子节点)	C: If 语句(有子节点)	遍历子节点
A: If 语句: !=: a: NmberBase.ADefault	B: If 语句: !=: a: NmberBase.ADefault	C: If 语句: !=: a: NmberBase.ADefault	If 语句: !=: a: NmberBase.ADefault 合并到结果集
A: If 语句: return: a + b	B: If 语句: return: a - b	C: If 语句: return: a -	标记为冲突
A: If 语句: else(有子节点)	B: If 语句: else(有子节点)	C: If 语句: else(有子节点)	遍历子节点
A: If 语句: else: return: getSymbal(a)	B: If 语句: else: return: getSymbal(a)	C: If 语句: else: return: getSymbal(a)	If 语句: else: return: getSymbal(a) 合并到结果集

根据上述语法树分析结果我们可知, 最后的合并结果以及对比 git 工具的分支合并的结果如表 6 所示.

表 6　分支合并结果对比

使用抽象语法树的合并结果

```
class test
{     int f;
      int num1 =0;
      float num2;
      int getf()
{
            int a = getAValue();
            int b= getBValue();

if(a!=NumberBase.Adefault)
            <<<<<<<<<<<<<<<
            A:    return a + b;
            B: return a - b;
            C:    return a;
            >>>>>>>>>>>>>>>>
            else{   return
getSymbal(a) ; }
    }
}
```

使用版本控制工具 git 的合并结果

```
class test
<<<<<<< A
{    int f;
     int num1;
     float num2;

     int getf()
=======
{    int f; float num2;
     int num1 =0 ;
      int getf()
>>>>>>>B
      {   int a = getAValue();
          int b= getBValue();

if(a!=NumberBase.Adefault)
<<<<<<<A
            return a + b;
            else{   return
getSymbal(a) ; }
      }
}
=======
            return a - b;
            else{   return
getSymbal(a) ; }
      }
}
>>>>>>> B
```

3 工具实现与实验

由以上的实例分析我们可以看到, 基于抽象语法树的版本合并算法是通过树节点的比较然后根据相应的三路合并规则决定分支合并的结果, 其中的技术点集中在对代码进行词法分析, 语法分析构造对应的抽象语法树, 对抽象语法树的节点进行遍历, 抽象语法树节点的搜索, 树节点类型比较, 是否存在树子节点. 上文中提到 ANTLR 项目提供了一个通过语法描述来自动构造自定义语言的词法分析器, 语法分析器和树分析器的框架. ANTLR 还可以根据输入自动生成抽象语法树并可视化的显示出来, 因此本文采用 ANTLR 项目构造算法的实现.

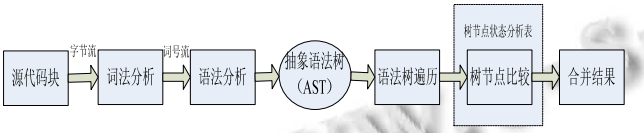

图 10 基于抽象语法树的分支合并工具实现

在实际工作中, 特别是由于产品的分布式研发, 研发团队分布在不同的地域或组织机构, 在开发过程中的沟通不如集中式开发及时, 代码的版本控制遇到更大的难度. 分布式版本控制工具 Git 等分支创建的随意性, 使代码分支合并变成了非常繁重的任务. 同时一旦分支合并操作中造成了代码冲突, 可以解决该冲突的人必须对代码的原始版本, 多个分支上衍生版本都有着清楚的了解, 因此只有这样才能决定哪些代码是应该出现在最终留下的版本中.

因为本文作者所在的是具有分布式研发背景的开发团队, 因此对团队中在一个周期内(一周)使用 Git 工具进行分支合并中出现的冲突进行了一个简单统计, 并且审查这些冲突中哪些可以使用本文中基于抽象语法树的分支合并方法避免冲突发生, 给出了对比统计结果.

表 7 对比统计结果

功能模块	使用 Git 进行分支合并冲突的代码块	用本文算法进行分支合并冲突代码块
framework	51	36
governance	23	14
ide	69	48
kernal	56	37
mobile	15	11
mxframework	74	32
report	32	21
taskdispatch	20	13

4 结语

随着软件项目面对着越来越庞大的功能需求, 分布式研发已经发展成为一种趋势, 更多的国际化公司和开发团队开始跨地域的, 更严格的功能划分, 这些客观上增加了软件开发环境中并发修改的活动, 虽然大多数软件开发开始使用各种各样的版本管理工具完成版本控制. 但解决传统以一行一行的代码为单位的三路合并的冲突问题变为非常繁重的任务, 本文中提出基于抽象语法树进行分支的三路合并, 通过对源码进行分析, 生成对应的抽象语法树, 遍历抽象语法树进行多分支节点比较, 最终决定哪些节点应该被留在合并文件中, 保证了合并的语法正确性, 准确识别出源码中可以自动合并不产生歧义的代码块, 减少分支合并操作中产生的冲突问题. 从作者所在项目开发过程中对分支操作的统计结果的数字比较我们可以看出本文的算法确实能更好的进行分支合并, 但对于多分支交叉引用的情况, 以及文件重命名, 以及语义层面的参数名称替换等方面, 本文算法并没有涉及, 因此在后续工作中, 在版本控制分支合并更广阔的研究方向上进行深入的研究.

参考文献

1 高传平,谈利群,宫云战.基于抽象语法树的代码静态自动测试方法研究.北京化工大学学报(自然科学版),2007,S1: 24–29.

2 张丽萍,刘东升.一种基于 AST 的代码抄袭检测方法.计算机应用研究,2011,12:4616–4621.

3 刘楠,韩丽芳,等.一种改进的基于抽象语法树的软件源代码比对算法.信息网络安全,2014,1:38–43.

4 张玉州,王一宾,江克勤.抽象语法树在属性计算中的应用.安庆师范学院学报(自然科学版),2008,4: 84–89.

5 廖兴,尹俊文,蔡放.基于 Java 语言的抽象语法树的创建与遍历.长沙大学学报,2004,4:50–54.

6 于冬琦,彭鑫,赵文耘.使用抽象语法树和静态分析的克隆代码自动重构方法.小型微型计算机系统,2009,9:1752–1761.

7 Wang QQ, Rong LL, Kai Y. A pushouts based knowledge merging method: Knowledge reorganization in emergency decision-making support. Wireless Communications, Networking and Mobile Computing(WiCOM'08). 2008. 1–4.

8 http://git-scm.com/docs/git-merge

9 http://en.wikipedia.org/wiki/Merge_(revision_control)

10 James J. Hunt Extensible language-aware merging. ICSM'02. 2002

11 Hayase Y, Matsushita M, Inoue K. Revision control system using delta script of syntax tree. International Workshop on Software Configuration Management. 2005. 133–149.

12 Appel S, Liebeg J, Brandl B. Semistructured merge: Rethinking merge in revision control systems. 19th ACM SIGSOFT Symposium and the 13th European Conference on the Foundations of Software Engineering. 2011. 190–200.

13 杨君.一种新的多版本增创算法.计算机学报,2008, 4(31):702–707.

14 刘峰.针对有向无环图结构的多版本分布模式优化.计算机工程,2011.11(37):74–77.

15 www.antlr.org.

16 高灿,侯秀萍,孙士明.基于抽象语法树的修改影响分析方法.长春工业大学学报(自然科学版),2012,4(33):387–391.

17 http://www.linuxcommand.org/man_pages/diff31.html.

18 Munson JP. A Flexible Object Merging Framework. CSCW 94-10/94, ACM 0-89791-689-1/94/0010. 1994.