计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(5):57−63 [doi: 10.15888/j.cnki.csa.006893]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

代码依赖可视化系统①

刘　旭

(SAP 中国研究院 商务智能部, 上海 201203)
通讯作者: 刘　旭, E-mail: liuxuhere@hotmail.com

摘　要: 分析了代码依赖可视化在智能化软件开发中的作用, 在总结代码依赖特点和信息可视化系统一般流程的基础上, 提出了代码依赖可视化系统的设计. 该系统使用力导向节点连接图和层次边聚合图作为可视化形式, 基于对两种可视化形式布局特点的深入分析, 分别针对力导向节点连接图和层次边聚合图创造了过滤子节点和聚合叶节点的交互设计. 在使用多种软件技术实现了代码依赖可视化原型系统 dpViz 之后, 将系统试用于企业软件开发中, 实验结果表明该可视化原型系统可有效增进代码分析效率.

关键词: 代码依赖; 软件可视化; 信息可视化; 力导向节点连接图; 层次边聚合图

引用格式: 刘旭.代码依赖可视化系统.计算机系统应用,2019,28(5):57–63. http://www.c-s-a.org.cn/1003-3254/6893.html

Code Dependency Visualization System

LIU Xu

(Department of Business Intelligence, SAP Labs China, Shanghai 201203, China)

Abstract: The paper analyzes the role of code dependency visualization in intelligent software development. On the basis of summarizing the characteristics of code dependency and the general process of information visualization system, the design of code dependency visualization system is proposed. The system uses force-directed node-link graph and hierarchical edge bundles as visualization types. Based on in-depth analyses of the layout characteristics of the two visualization types, the interaction design of filtering children nodes is created for force-directed node-link graph, and the interaction design of aggregating leaf nodes is created for hierarchical edge bundles. After implementing the code dependency visualization prototype system dpViz using a variety of software technologies, the system is tested in enterprise software development. The experimental results show that the visualization prototype system can effectively improve the efficiency of code analysis.

Key words: code dependency; software visualization; information visualization; force-directed node-link graph; hierarchical edge bundles

代码依赖与软件架构紧密相关, 代码各模块的相互依赖直接反映了高内聚及低耦合的设计目标. 在软件开发过程中和软件维护阶段, 对于代码依赖的分析有助于降低不必要和不合理的依赖, 适时重构软件以维护软件架构的稳定性, 减少程序错误, 提高开发效率和降低维护成本[1]. 然而, 随着现代软件代码量的膨胀, 代码间的依赖关系日趋复杂, 传统的依靠开发人员经验进行代码依赖检查的方式效率较低, 智能化的软件开发尝试在软件开发过程中引入数据驱动的思想以改善开发体验[2], 提高开发效率.

软件架构的设计, 实现和维护最终依靠架构师, 开发和测试人员完成, 而视觉是人类接受信息最主要的

① 收稿时间: 2018-11-18; 修改时间: 2018-12-10; 采用时间: 2018-12-20; csa 在线出版时间: 2019-05-01

© 中国科学院软件研究所　http://www.c-s-a.org.cn

渠道. 软件可视化利用信息可视化和可视分析的方法进行软件数据展示和分析, 发掘数据的非结构化信息. 利用可视化方式对代码依赖数据进行分析, 有助于技术人员更快地发现依赖关系的变化, 还可发现关系中蕴含的模式[3]. 针对代码依赖设计并实现功能完善的可视化系统, 重点是需要分析代码依赖数据的特点, 选取既能够降低视觉复杂度, 又便于进行交互的可视化形式. 目前相关的工作多集中在对代码依赖数据的提取和分析, 对可视化部分的研究较少, 已有的数据可视化分析强调统计结果, 多使用传统的散点图, 折现图等统计图表. 本文的重点在于如何针对依赖数据本身使用不同的现代可视化形式, 并改善其交互效果.

1　代码依赖及软件可视化

在现代软件的开发中, 无论使用过程式, 函数式还是面向对象的软件模块组织方式, 最终程序的执行必然包括一系列方法的调用过程, 这意味着代码之间的依赖关系广泛存在于软件源代码中. 典型的代码依赖一般存在于函数之间, 文件之间和对象之间. 在面向对象的开发方式中, 继承和方法调用是比较常见的对象之间产生依赖的原因. 代码依赖不可避免, 但在软件系统代码量越来越庞大的今天, 冗余的和破坏系统架构的代码依赖会增加代码理解和维护的难度. 开发, 测试和维护过程中对于代码依赖的自动化检查已经成为大型软件系统研发的常见手段, 最终目标是提高软件代码质量.

软件开发过程传统上主要基于开发人员的经验进行. 为了改进开发效率, 需要引入数据驱动的方式, 首要任务是充分利用开发过程中产生的各种数据, 包括代码依赖数据. 软件可视化试图在软件工程领域应用信息可视化与可视分析的发展成果, 即结合人类视觉和计算机结构化信息处理的优势[4], 使用人机交互的方式从数据中得出结论.

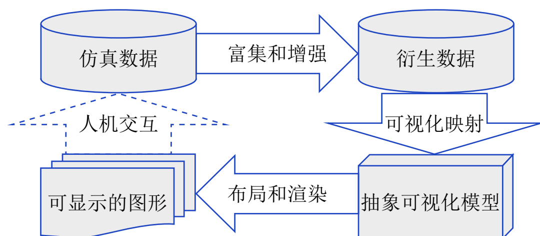

图 1　信息可视化流程

图 1 显示的是信息可视化的一般流程[5]. 可视化的最终实现目标是可显示, 通常还带有交互性的图形. 在数据的富集和增强阶段主要对数据进行标准化, 聚合, 过滤, 去除不需要进行可视化的维度. 可视化映射的重点是视觉编码, 也就是如何从数据映射到位置, 颜色等属性. 视觉编码和布局算法往往和选取的具体可视化形式有关. 在数据分析已经完成的情况下, 可视化被作为数据挖掘的结果呈现方式, 但在可视分析逐渐被视为大数据分析的重要方法的情况下[6], 高度交互性的可视化不仅是分析的结果, 也是分析的手段, 可以用于数据的再次获取和分析. 软件可视化的关键在于使用适当的隐喻和交互设计, 帮助技术人员进行分析推理, 发掘隐藏在软件数据中的模式, 从而改善开发过程[7].

2　原型系统 dpViz 的设计

随着智能化的发展和数据量的膨胀, 在企业软件开发中越来越强调敏捷[8], 软件研发的迭代周期越来越短. 具有高度适应性和完善开发框架支持的 B/S 模式逐渐成为信息系统应用的主流, 对于 JavaScript, Python 等脚本语言的依赖与日俱增[9]. JavaScript 是 2017 年 GitHub 上最为流行的程序设计语言, 广泛应用于前端和后端的企业软件开发, 拥有庞大的代码基础. JavaScript 早期版本不具有模块 (module) 的实现, ES6 虽然引入了内置的模块实现, 但此标准出现较晚, 目前大量的 jQuery 等 JavaScript 软件项目多使用 RequireJS 等第三方模块加载器定义和管理模块[10], 将每个文件定义为一个模块, 在模块的开始部分定义模块之间的依赖关系.

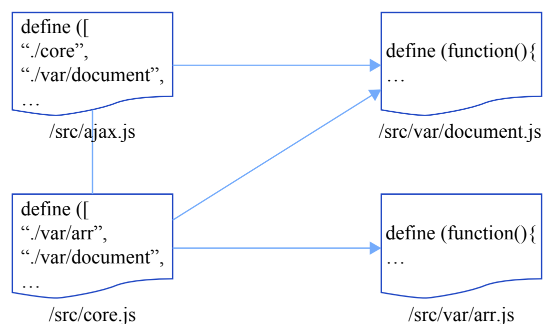

图 2　JavaScript 模块之间的依赖

图 2 是一个使用 RequireJS 定义的 JavaScipt 模块之间的代码依赖示意图, 代码示例取自 jQuery. 可以看到有的模块同时依赖于几个模块, 也有的模块同时被

© 中国科学院软件研究所　http://www.c-s-a.org.cn

其他几个模块依赖. 模块之间的依赖从结构上说是一个有向图[11], 但是在企业软件开发实践中, 开发人员最关心的是如何快速定位依赖关系存在于哪两个模块之间, 具体的依赖方向一般结合项目架构和开发经验判断, 也可以由软件开发者在修改代码文件时查看, 使用无向图表示的代码依赖已经可以满足大部分检查代码依赖的需要.

本文试图实现的原型系统 dpViz 以 JavaScript 为目标语言, 以 RequireJS 定义的模块为代码依赖的单元. 目标是以无向图的方式可视化各单元之间的依赖关系, 可视化形式需要带有交互性以便于进一步分析. 原型系统 dpViz 针对代码的静态分析结果, 和一般的信息可视化系统类似, 同样遵循数据获取, 数据结构化处理, 可视化布局和渲染等基本步骤. 图 3 是使用 SAP PowerDesigner 绘制的 dpViz 组件图.

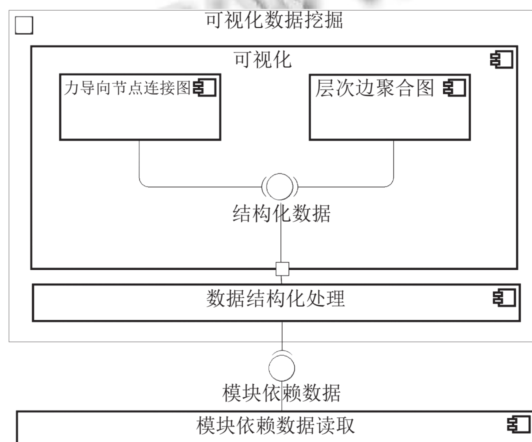

图 3 dpViz 组件图

从组件图可以看到, dpViz 系统中各模块之间的依赖较少, 数据传递比较清晰, 在获取模块依赖的原始数据之后, 数据的结构化和可视化都可以归为可视化数据挖掘领域, 可视化形式的确定是设计重点. 开发人员最关注的代码模块依赖信息无疑是哪些模块之间有联系, 模块位于何处, 这些联系有什么特点. 由于理论上任何两个模块之间都可能有依赖, 模块间的依赖关系具有典型的网络数据特点, 同时, 模块在文件系统中位于不同的路径, 路径层次与功能模块有关, 模块的层次关系也需要被显示. 在信息可视化系统的设计中, 经常组合使用多种不同的可视化形式以满足用户需要. dpViz 的可视化形式选取了在信息可视化领域较为成熟的力导向 (Force-Directed) 节点连接图[12]和层次边聚合图 (Hierarchical Edge Bundles)[13]. 这两种图形对于网络关系的可视化布局有不同的优化方式.

3 原型系统 dpViz 的可视化布局

网络数据可以借鉴图论中常用的邻接矩阵等方式进行可视化表示, 但是节点连接图可能是最为重要和常用的[14], 很多布局设计也参考节点连接图, 这是因为视觉系统对于位置这一视觉编码最为敏感. 节点连接图使用较小的图形表示每个节点, 节点之间的关系用连接线表示. 代表节点的图形可以相同, 也可以分别有不同的颜色或形状以表示更多维度的信息. 节点连接图的缺点是当节点数量和连接线变多时, 连接线和节点很容易互相交叠在一起, 使图形变得难以分辨. 设计较好的布局算法可以减少这种情况.

力导向布局算法可能是最为常用的节点连接图布局算法, 其基本思想是将网络数据中的节点对应为现实中的带同种电荷的小球, 节点之间的连线对应于弹簧, 而画图的区域对应于现实中的平面. 当给定小球的初始状态和弹簧长度之后, 根据胡克定律, 以弹簧连接的小球之间存在互相吸引或排斥的弹力, 而各个带同种电荷的小球之间存在库仑斥力, 因此这些小球会在这些力的作用下进行运动, 一些小球会靠近, 而一些小球会分开, 这个过程会不断重复下去. 在平面存在摩擦力的情况下, 力导向算法中的小球最后会逐渐趋于一个平衡位置, 平衡位置的小球和弹簧即构成获得的布局.

在算法的具体实现过程中, 一般会在物理模型的基础上进行改进[15]. 由于结点距离较远时, 由胡克定律所确定的弹簧引力太强, 可以使用对数法则 $c1\times\log(d/c2)$ 来确定弹簧的引力, 其中 d 是结点之间的距离, $c1$ 与 $c2$ 都是常数. 在计算库仑斥力时, 由于库仑力与距离的平方成反比, 故可以使用 $c3/d^2$ 计算, 其中 d 为结点间距离, $c3$ 为常数. 算法对结点位移的计算也可以简化, 并且设置一定的迭代次数来完成布局. 使用伪代码表示的算法如下:

```
randomPlace(notes); //随机给定初始节点位置
for(int i = 0; i < k; i++) { //循环以达到理想效果
    calculateEachNoteForce(notes); //计算每个节点的受力, 记为 eachNoteForce
    moveAll(c4, notes); //对所有结点移动 c4 * eachNoteForce
}
```


© 中国科学院软件研究所 http://www.c-s-a.org.cn

算法中的 $c1, c2, c3, c4$ 都是常数, 一个常用的经验值是 $c1=2$, $c2=1$, $c3= 1$, $c4= 0.1$. 由于每两个结点之间的斥力都需要被考虑, 计算结点之间的斥力造成的位移需要的时间为 $O(n^2)$, 而计算边的引力给结点带来的位移需要的时间为 $O(e)$, 如果此算法共进行 k 次迭代计算, 那么需要的时间为 $O(k\times(n^2+e))$, 其中 n 为结点数, e 为边数. 图 4 是使用 SAP Lumira 的布局算法生成的典型力导向布局节点连接图. 可以看到, 如果布局时给定的弹簧长度都是一致的, 最后获得的布局结果中, 有连接的节点之间的连线长度会趋于均匀, 而电荷之间斥力的存在会使无连接线的节点之间趋于分散, 这样节点的分布也会比较均匀, 同时使整个图形呈现对称性. 这种对称性的特点是其他的布局方式难以保证的.

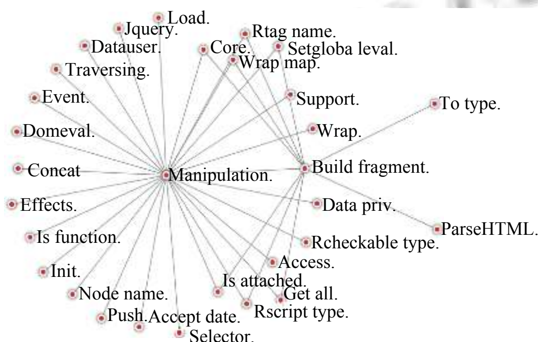

图 4　典型的力导向节点连接图

力导向算法基于物理模型, 物理特性保证了迭代结果收敛, 所以此算法的行为容易预测和理解, 代码的实现也较容易, 并且可以方便地扩展到三维空间. 交互性也是力导向算法的明显优点. 在布局的生成过程中, 用户可以看到结点集是怎样从混乱的初始状态展开成为比较具有可读性的布局的, 在绘制完成之后, 还可以拖动一个或多个结点偏离平衡位置, 看结点如何返回原位. 力导向布局的这个特性让它很适合用于绘制动态生成的数据的节点连接图. 但是, 力导向算法在连接线较多的情况下, 线和节点容易交叉, 影响视觉效果. 布局过程将改变图中节点的位置, 并且节点的位置在布局完成前不可预知, 这使得力导向算法难以用于节点位置相对固定的场合. 力导向算法的时间代价也较高. 如果以 n 代表节点的数目, 一般情况下, 力导向算法需要 n 次迭代才能得到较稳定的结果, 即使不考虑边的引力计算, 力导向算法需要的时间复杂度仍为 $O(n^3)$, 在结点较多的情况下, 这样的时间性能并不理想.

为了规避此布局的缺点, 可以在实际使用的节点连接图中加入过滤功能, 减少显示的节点和连接线的数量. 在选中某个节点之后, 用户一般只会对于跟此节点关系密切的节点感兴趣, 例如在代码依赖关系中, 开发人员一般最关心的只是跟自己开发的那个模块有关系的模块. 一种常用的过滤方法是对于某个选中节点, 仅显示该节点本身, 子节点和二级子节点 (子节点的子节点). 图 5(a) 是一个力导向布局的节点连接图 (局部), 其中高亮的节点仅有两个子节点, 但节点个数较多, 布局密集, 视觉效果不佳. 图 5(b) 是过滤后的结果 (局部), 很容易从中看出依赖于高亮节点的子节点和二级子节点.

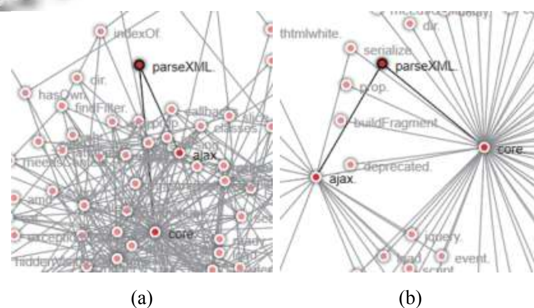

图 5　节点连接图的过滤

层次边聚合图 (Hierarchical Edge Bundles) 在节点连接图的基础上尝试用另一种方式来降低连接线的视觉复杂度, 即聚合有一定关系的连接线成为线束, 避免过多的连接线交叉[16]. 层次边聚合图在环形区域中用扇环显示节点, 不同层次的扇环代表节点之间的层次关系, 节点之间的相对位置是固定的. 根节点绘制在最外层 (根节点必然是圆环, 故往往省略根节点), 需要绘制连接线的节点都排布为叶节点, 这样连接线都将绘制在环形区域的边缘. 在扇环内部用样条绘制连接线表示叶节点之间的关系, 样条曲线的控制点位置跟父节点的位置有关. 这样绘制的样条曲线将根据节点位置的不同分别聚合成为若干线束[17]. 具有同一祖先的叶节点, 其连接线会比较容易聚拢在一起, 而关系较远的节点, 对应的连接线会比较分散. 线束的布局可以从一定程度上代表节点的层次关系, 扇环的大小和颜色和连接线的颜色可以显示更多维度的信息. 图 6(a) 是初始的节点连接图, 图 6(b) 是转换后的层次边聚合图, 可以看到节点层次关系通过类似于 Treemap (树图) 的内隐方式表达, 减少了连接线的数量[18].

© 中国科学院软件研究所　http://www.c-s-a.org.cn

(a) (b)

图 6　生成层次边聚合图

从设计上来看, 层次边聚合图针对的还是网络关系的数据, 其思路是将节点连接图中的网络关系从语义上区分为两种不同性质的联系: 节点之间普遍存在的层次连接和仅存在于叶节点之间的网络连接, 将层次连接用扇环之间的层次关系表示, 而叶节点之间的网络连接用聚合线束表示. 一般的网络关系可以看作是这种网络关系的特例, 只需把所有节点看做是一个虚拟的根节点的叶节点, 也可以使用这种布局方式绘制.

在 JavaScript 项目的源代码模块组织结构中, 代码模块往往放置于不同的文件夹下, 不同的文件夹代表了软件架构中不同的功能模块. 在这种情况下, 代码模块之间具有网络关系, 而模块与其所处的各层文件夹之间具有层次关系, 各层文件夹可以被绘制为父节点, 用父节点的颜色表示子节点的数量. 在节点个数和节点之间连接线很多的情况下, 可以考虑聚合叶节点的连接到最末一级或更高层次的父节点, 将选定的父节点作为叶节点显示, 用连接线的颜色深度反映聚合的连接数量, 这样可以进一步减少连接线的数量. 图 7(a) 和图 7(b) 显示了聚合前后层次边聚合图的变化, 可以看到父节点之间的关系更加清晰.

(a) (b)

图 7　聚合叶节点

4　原型系统 dpViz 的实现

系统的实现首先需要考虑到模块依赖数据的获取和处理. 在基于数据的应用系统中, 原始数据往往都是不规范的, 会有很多空值, 错误值, 数据的规范化需要占用大量的系统资源[19]. 但是由于 dpViz 系统需要的代码依赖数据来自于代码本身, 可以通过程序化的方式获取, 这样获取的原始数据会比较容易规范化. dpViz 使用的方式是使用 Java 语言读取选定路径下的 JavaScript 文件, 从 JavaScript 的 define 中解析出其依赖的模块. 在获取依赖关系之后, 可以进行一些基本的数据处理, 例如排除重复的依赖项, 将路径中的"./"和"../"进行适当替换. 处理完成之后获得的基本对应关系可以使用 Java Swing 中的 JTable 控件来展示为图 8 所示的二维表格 (以 jQuery 代码为例), 便于用户浏览和更改数据.

dpviz

Code Dependency　Visualization　Help

File	Target File
src\ajax\jsonp	src\core
src\ajax\jsonp	src\var\isFunction
src\ajax\jsonp	src\ajax\var\nonce
src\ajax\jsonp	src\ajax\var\rquery
src\ajax\jsonp	src\ajax
src\ajax\load	src\core
src\ajax\load	src\core\stripAndCollapse

图 8　用表格浏览依赖关系

数据结构化处理和可视化部分是 dpViz 系统的核心功能, dpViz 这部分的实现使用 JavaScript 结合 SVG 完成. JavaScript 的执行性能已经可以满足大型企业应用的需要[20], SVG 作为 Web 矢量图形的标准, 以其跨平台特性在可视化开发领域受到广泛应用[21], 为底层图形的绘制提供了完善的支持[22]. 对于选定可视化形式布局的实现是系统的关键. 使用 SVG 进行开发可以使用的第三方库有很多, 其中的 D3 作为数据驱动方式的可视化库, 已经实现了很多数据处理和布局功能, 合理利用 D3 可以明显提高可视化布局的开发效率.

在 dpViz 可视化部分的具体实现中, 先使用 Java 将模块依赖数据存储为 JavaScript 文件, 写入文件系统, 然后使用 JavaScript 在 Web 中载入这些数据文件, 将二维数组方式存储的数据结构化为具有网络结构的 JavaScript 数据对象, 再根据数据对象进行布局并绘制. 对于力导向节点连接图的布局, 在 d3.layout.force 中有完整实现, 可以参考其算法并加以改进. 层次边聚合图分为扇环和连接线两部分. 扇环的布局可以参考 d3.layout.partition 函数实现. 扇环本身不是

© 中国科学院软件研究所 http://www.c-s-a.org.cn

SVG 原生支持的基本形状, 需要使用 path 绘制, 具体过程可借助 d3.svg.arc 完成. 层次边聚合图的连接线可在扇环的布局确定之后, 使用 d3_svg_lineBasis 函数完成.

可视化的布局和绘制完成之后, 还需要实现交互性功能以便于可视分析. 由于在实际进行数据分析的时候节点往往较为密集, 高亮鼠标盘旋或选中的某些图形元素是交互性的基本要求. 节点较多时, 文字标签往往只能缩略, 高亮状态下需要显示完整标签. 对于力导向节点连接图, 需要允许用户拖放各个节点改变布局效果, 从而可以通过不同的布局方式分析数据特点, 还需要实现仅显示选中节点的子节点和二级子节点的过滤功能, 便于针对特定节点进行进一步分析. 层次边聚合图在可视分析中较实用的是调整连接线的聚合程度, 鼠标拖放的旋转和叶节点的聚合效果. 这些都可以通过数据的过滤和布局的更新实现.

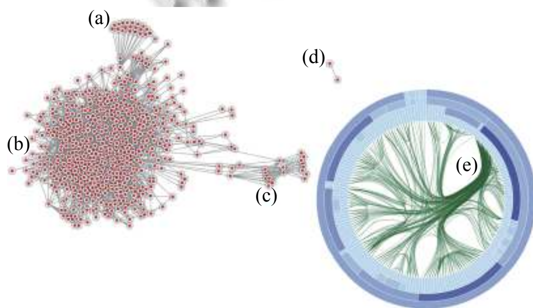

图 9　使用 dpViz 对某系统模块可视分析

dpViz 已经作为企业级软件代码依赖的可视分析工具试用, 其分析结果对软件开发工作显示出实用价值. 图 9 是将 dpViz 用于分析某商务智能系统模块过程中所生成的力导向节点连接图和层次边聚合图 (为保护知识产权, 图中隐藏了文本标签). 从图中容易看出, 图 9(a) 和图 9(c) 的部分是模块化较好的部分, 这两个部分节点较密集, 但与其他部分依赖关系较少. 图 9(b) 的部分节点多而密集, 连接线也很多, 模块化程度较差. 图 9(d) 是两个与其他部分无关联的孤立的文件, 很可能是系统试验性开发中留下的无用文件, 应该清除, 通过进一步检查源程序证明了这一点. 图 9(e) 的部分与其他模块依赖较多, 通过提高该部分的代码性能和可读性够有效改善整个系统的代码质量. 在可视分析过程中, 可以发现力导向节点连接图和层次边聚合图显示出互补性. 力导向节点连接图交互性较好, 允许用户拖放改变布局, 但在节点和连接线很多的情况下布局较慢, 视觉复杂度高. 层次边聚合图布局较快, 对连接线较多的情况处理很好, 但在交互性分析方面表现较差.

5　总结与展望

现代软件代码仍然主要由开发人员编制, 而人倾向于反复在同一个地方犯错, 静态代码分析与检测一直是消除软件错误的重要方法[23]. 可视化以其高度的人机交互性, 逐渐从海量数据的分析结果进化为分析手段. 作为静态软件代码可视化的一个重要组成部分, 代码依赖关系的可视化可以极大提高开发人员对于软件系统的理解, 提高软件开发, 测试和软件再工程的效率. dpViz 作为代码依赖可视化的尝试, 尽管使用的可视化形式还比较有限, 支持的程序设计语言种类也较单一, 但完整地设计并实现了代码依赖可视化的各模块, 使用了力导向节点连接图和层次边聚合图这两种现代可视化形式, 并创造了节点过滤和叶节点聚合功能, 增加了图形的交互性, 系统的试用验证了企业软件开发效率的提升.

对于 dpViz 的改进, 主要的方向可以是从代码依赖特征入手, 创造和改进更有效的可视化形式和交互方式. 从软件度量的角度看, 代码模块包含的信息很多, 针对大量高维网络数据节点设计有效可视化方式一直是信息可视化的难点. 对于不同的代码模块类型和不同的依赖方式, 以及循环依赖, 间接依赖等特殊依赖关系的可视化也是软件开发过程中非常实用的软件可视化功能. 软件生命周期是一个时间上动态的过程, 通过软件仓库挖掘可以得到代码依赖关系不断变化和发展的数据, 如何使用可视化工具分析这些数据的特点和趋势也是有待解决的问题[24]. 随着软件开发社区的发展, 代码依赖的范围并不局限于单个工程项目内部, 跨项目的代码依赖可视化可能需要与单个项目不同的实现方法和目标. 希望 dpViz 对于代码依赖可视化的努力能够为进一步研究和系统实现提供思路.

参考文献

1 Mohammed M, Fawcett JW. Package dependency visualization: Exploration and rule generation. The 23rd International Conference on Distributed Multimedia Systems, Visual Languages and Sentient Systems. Pittsburgh, PA,

© 中国科学院软件研究所 http://www.c-s-a.org.cn

USA. 2017. 8–13.
2 谢冰, 魏峻, 彭鑫, 等. 数据驱动的软件智能化开发方法与技术专题前言. 软件学报, 2018, 29(8): 2177–2179. [doi: 10.13328/j.cnki.jos.005534]
3 Kristensson PO, Lam CL. Aiding programmers using lightweight integrated code visualization. Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools. New York, NY, USA. 2015. 17–24.
4 Van Wijk JJ. The value of visualization. Proceedings of IEEE Visualization, 2005. Minneapolis, MN, USA. 2005. 79–86.
5 Card SK, Mackinlay J. Readings in Information Visualization: Using Vision to Think. San Francisco, USA: Morgan Kaufmann, 1999.
6 邱均平, 余厚强. 从 VAST 会议解读可视分析学新进展. 数据分析与知识发现, 2014, 30(10): 14–24. [doi: 10.11925/infotech.1003-3513.2014.10.04]
7 刘旭. 代码重复检测结果可视化设计与实现. 西华大学学报 (自然科学版), 2017, 36(6): 13–22. [doi: 10.3969/j.issn.1673-159X.2017.06.003]
8 Paredes J, Anslow C, Maurer F. Information visualization for agile software development. 2014 Second IEEE Working Conference on Software Visualization. Victoria, BC, Canada. 2014. 157.
9 刘旭. 乐器教育 MOOC 设计探索. 数字教育, 2018, 4(4): 17–22. [doi: 10.3969/j.issn.2096-0069.2018.04.003]
10 Franko G. Instant Dependency Management with RequireJS How-to. Birmingham: Packt Publishing Ltd, 2013.
11 Decan A, Mens T, Grosjean P. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering, 2018, 24(1): 381–416.
12 Holten D, Van Wijk JJ. Force–directed edge bundling for graph visualization. Computer Graphics Forum, 2009, 28(3): 983–990. [doi: 10.1111/cgf.2009.28.issue-3]
13 Holten D. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. IEEE Transactions on Visualization and Computer Graphics, 2006, 12(5): 741–748. [doi: 10.1109/TVCG.2006.147]
14 陈为, 张嵩, 鲁爱东. 数据可视化的基本原理与方法. 北京. 科学出版社, 2013.
15 Tamassia R. Handbook of Graph Grawing and Visualization. Boca Raton: CRC Press, 2013.
16 刘旭. 基于 D3 的层次边聚合图设计与实现. 西华大学学报 (自然科学版), 2017, 36(4): 27–33, 50. [doi: 10.3969/j.issn.1673-159X.2017.04.005]
17 Holten D, Cornelissen B, Van Wijk JJ. Trace visualization using hierarchical edge bundles and massive sequence views. 20074th IEEE International Workshop on Visualizing Software for Understanding and Analysis. Banff, Ont., Canada. 2007. 47–54.
18 刘旭. 基于深度优先搜索的正方化树图布局算法. 计算机系统应用, 2017, 26(5): 105–112.
19 Keck M, Kammer D, Gründer T, *et al*. Towards glyph-based visualizations for big data clustering. Proceedings of the 10th International Symposium on Visual Information Communication and Interaction. New York, NY, USA. 2017. 129–136.
20 刘旭. Chrome V8 引擎中的 JavaScript 数组实现分析与性能优化. 计算机与现代化, 2014, (10): 66–70. [doi: 10.3969/j.issn.1006-2475.2014.10.016]
21 Mei HH, Ma YX, Wei YT, *et al*. The design space of construction tools for information visualization: A survey. Journal of Visual Languages & Computing, 2017, 44: 120–132.
22 刘旭. 双向文本元素在 SVG 中的显示技术. 计算机系统应用, 2017, 26(4): 246–251. [doi: 10.15888/j.cnki.csa.005698]
23 Bardas AG. Static code analysis. Journal of Information Systems and Operations Management, 2010, 4(2): 99–107.
24 Gharehyazie M, Ray B, Keshani M, *et al*. Cross-project code clones in GitHub. Empirical Software Engineering, 2018: 1–36. [doi: 10.1007/s10664-018-9648-z]

© 中国科学院软件研究所 http://www.c-s-a.org.cn