**开题报告**

**论文题目：** Unity3D内存优化工具的设计与实现

**学 号：** 21632059

**姓 名：** 舒林

**专 业：** 控制科学与控制工程

**指导老师：** 贺诗波研究员 陈积明教授

**日 期：** 2017-12-07

目录

[1 研究背景及意义 3](#_Toc508285924)

[1.1游戏行业发展迅猛 3](#_Toc508285925)

[1.2 Unity3D游戏引擎 3](#_Toc508285926)

[1.3 Unity Editor扩展工具 5](#_Toc508285927)

[1.4 游戏内存问题 6](#_Toc508285928)

[2 相关研究及研究现状 7](#_Toc508285929)

[2.1 Unity technologies 推出的Memory Profiler 7](#_Toc508285930)

[2.2 游戏优化公司的内存分析 9](#_Toc508285931)

[2.3内存泄露检测方法研究现状 9](#_Toc508285932)

[3主要研究内容 11](#_Toc508285933)

[3.1内存数据获取（已完成） 12](#_Toc508285934)

[3.2.1内存可视化 13](#_Toc508285935)

[3.2.2两个内存快照对比 14](#_Toc508285936)

[3.2.3 多个内存快照对比 15](#_Toc508285937)

[3.3 Mono内存泄露分析（下一步） 15](#_Toc508285938)

[3.3.1 Mono内存增长原因 15](#_Toc508285939)

[3.3.2特征选择 16](#_Toc508285940)

[3.3.2 数据集获取 17](#_Toc508285941)

[3.3.3 学习方法选择与方法效果评估 18](#_Toc508285942)

[3.3.4 应用内存泄露检测方法 18](#_Toc508285943)

[4研究工作计划安排 18](#_Toc508285944)

[5预期成果 18](#_Toc508285945)

[参考文献 19](#_Toc508285946)

# 1 研究背景及意义

## 1.1游戏行业发展迅猛

21世纪以来，随着硬件技术的发展，电脑、手机、平板等智能终端功能日益强大，游戏行业迅猛发展，游戏正成为全球最受欢迎的娱乐消遣方式。全球游戏市场情报公司Newzoon发布的《2017全球游戏市场报告》显示，截至2017年，全球约有22亿游戏玩家，其中47%的玩家会在游戏上消费，2017年全球游戏玩家话费预计为1089亿美元，数字收入将达到944亿美元，占市场的87%。iphone的第一部智能手机发布十年后，移动游戏连续两年成为最受欢迎的市场，智能手机和平板游戏收入同比增长19%达到461亿美元，占全球游戏市场收入的42%。

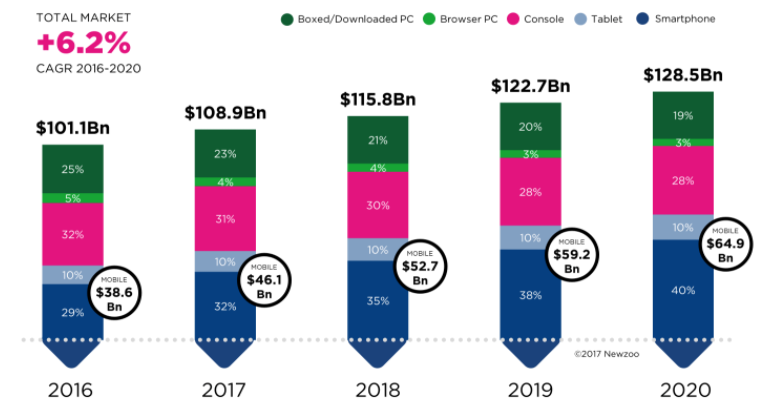


图1.1 2016-2020全球游戏市场规模预测

## 1.2 Unity3D游戏引擎

游戏引擎是指一些已编写好的可编辑游戏系统或者一些交互式实时图像应用程序的核心组件。这些系统为游戏设计者提供各种编写游戏所需的各种工具，其目的在于让游戏设计者能容易和快速地做出游戏程序而不用由零开始。游戏引擎包含以下系统：渲染引擎（即“渲染器”，含二维图像引擎和三维图像引擎）、物理引擎、碰撞检测系统、音效、脚本引擎、电脑动画、人工智能、网络引擎以及场景管理[[[1]](#endnote-1)]。Unity3D是由Unity Technologies开发的一个让开发者轻松创建诸如三维视频游戏、建筑可视化、实时三维动画等类型互动内容的多平台的综合型游戏开发工具，是一个全面整合的专业游戏引擎[[[2]](#endnote-2)]。在最受欢迎的1000款视频游戏中，Unity3D制作的游戏占34%[[[3]](#endnote-3)]。许多风靡全球的游戏都是由Unity3D制作，如王者荣耀，纪念碑谷，炉石传说等等。

典型的Unity 包含多个游戏场景（关卡），每个场景又是由多个游戏对象构成。每个游戏对象包括网格，动画，音效，粒子特效，渲染，逻辑代码等多个组件构成[[[4]](#endnote-4)]。逻辑代码控制着游戏对象在外部事件的触发下按照设定好的逻辑做出相应变化。Unity3D引擎的编辑界面如1.2图所示，左边的Hierarchy显示的是在这个场景下的游戏对象，选中游戏对象在最右边的Inspector框内能显示其游戏组件的详细信息，并能直接添加删除组件。中间的窗口为正在编辑的场景场景，场景中的资源可以由Assets窗口中直接拖到场景中。

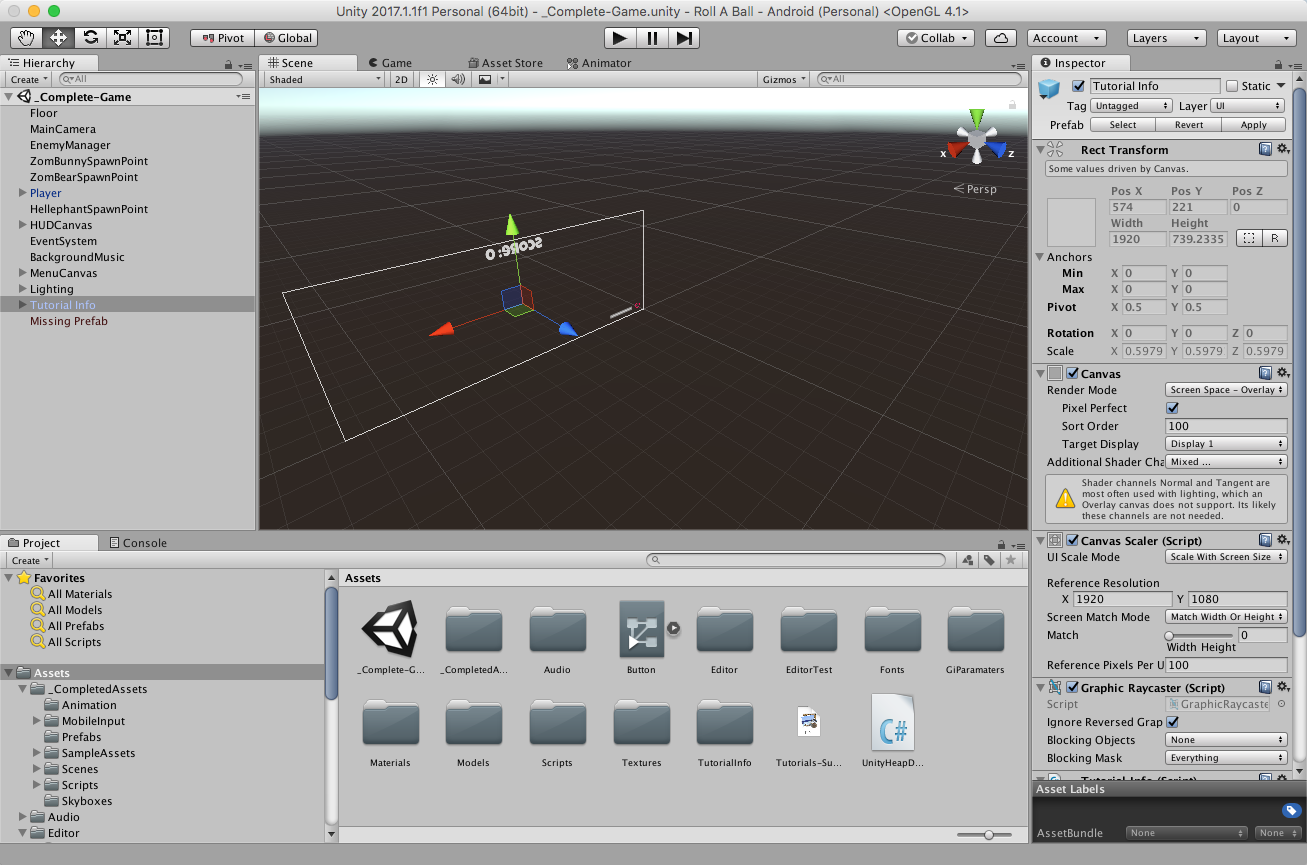


图1.2 Unity3D编辑界面图

## 1.3 Unity Editor扩展工具

随着游戏的发展，游戏引擎也需要不断提升来满足游戏的需求[[[5]](#endnote-5)]，UnityEditor扩展工具能让使用者自己编写脚本来扩展Unity引擎的功能。本课题的目的是做一款Unity3D的内存分析工具，如图1.2所示，Unity编辑界面的左下角Assets文件夹下有一个Editor文件夹，该文件夹中的脚本主要用来扩展Unity编辑器的功能方便开发。这些脚本将不会打包到最终发布的游戏中。扩展工具的实现分为三部分：1.创建继承EditorWindow的脚本；2.编写触发显示扩展窗口的代码；3.实现工具所需要的GUI[[[6]](#endnote-6)]。一个简单的扩展工具的实现如下[[[7]](#endnote-7)]：

class MyWindow : EditorWindow {

[MenuItem ("Window/MyWindow ")]

public static void ShowWindow () {

EditorWindow.GetWindow(typeof(MyWindow));

}

void OnGUI () {

// The actual window code goes here

}

}

MyWindow类继承了EditorWindow类，MenuItem这条语句说明在Window窗口下的MyWindow能出发该工具的运行。OnGUI里面是工具窗口的UI部分。点击MyWindow将出现编写的窗口界面。

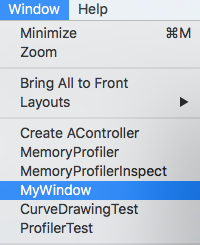
 

图1.3 Unity3D Editor扩展工具示例

## 1.4 游戏内存问题

游戏内存问题主要包括内存占用过大和内存泄露两个方面[[[8]](#endnote-8)]，会引发诸如闪退，卡顿，自动重启，耗能[[[9]](#endnote-9)]，游戏玩家流失等问题。游戏特别是视频游戏，占用内存最大的部分是美术资源部分，这部分的资源的格式和大小能极大的影响游戏的内存大小，游戏内存过大会导致很多中低端机型无法安装或者运行该游戏。内存泄露市值由于疏忽或者错误造成程序未能释放已经不再使用的内存，导致在释放该段内存之前就失去了对该段内存的控制，造成了内存的浪费。Unity游戏程序主要由代码和美术资源两部分组成，Unity下的内存泄漏也主要分为代码测的泄露和资源测的泄露。如图是unity运行时的内存组成。之前介绍过Unity游戏由一个一个的场景（关卡）构成，每个场景由一个个游戏对象组成，游戏对象是由组件构成的，这些组件包括C#代码，音效，动画等各种美术资源，另外还包括Unity提供的物理引擎，渲染系统等。这些资源对应到游戏运行时的内存依次为mono堆，native堆和代码库。Unity引擎的底层实现是由C++代码实现的，我们称之为Native，游戏逻辑中的代码是由C#实现的，为了实现这两者之间的交互，Unity采用了Mono运行时，由C#代码申请的内存分配是通过Mono虚拟机分配在mono堆上的，由美术资源加载分配的内存分配在Native堆上。在Unity环境下，Mono内存堆的占用是只会增加不会减少的。具体来说，可以将Mono堆理解成一个内存池，每次Mono内存的申请都会在池内进行分配，释放的时候也是归还给池，而不会归还给系统。如果某次分配发现内存池不够了，则会对池进行扩建。

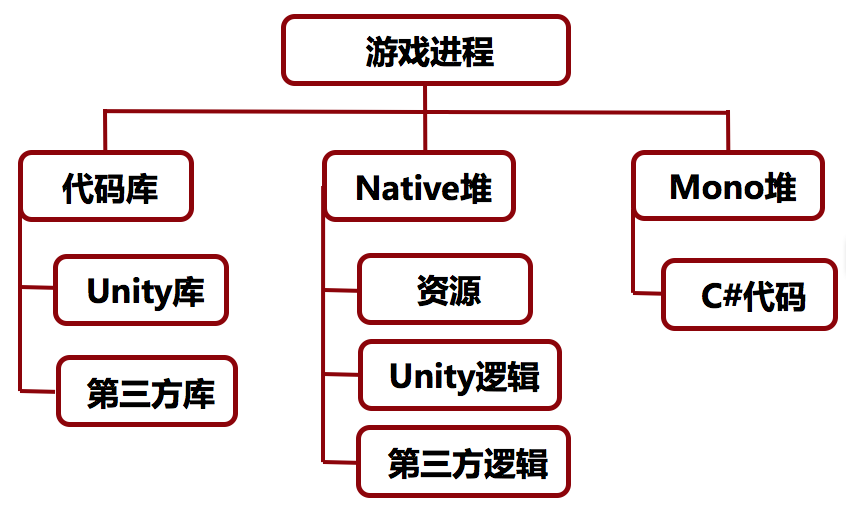


图1.4 Unity3D运行时内存组成

# 2 相关研究及研究现状

目前有关游戏内存优化的研究比较多，由于手机的内存对于电脑更为有限，针对手游的内存优化更为重要和迫切，手游的模型优化，纹理格式优化[[[10]](#endnote-10)]，渲染方式优化[[[11]](#endnote-11)]，这些研究都是在已知内存问题出在哪的情况下去优化，但是针对如何去找游戏内存的问题的研究比较少， 内存问题的查找只是出现在游戏优化公司或者个人整理的博客上，需要游戏开发人员借助工具凭经验去发现， Unity technologies公司到目前为止推出了两款内存分析工具，游戏优化公司的优化往往需要将游戏打包，优化过程不直观且比较耗时，另外，处于一些安全因素的考虑，如果自己有合适的内存分析工具，游戏开发人员也不愿将自己的游戏安装包上传给其他公司去分析。

## 2.1 Unity technologies 推出的Memory Profiler

在Unity5.3版本之前Unity提供了比较功能全面的Proflier工具如图，但是内存分析存在很多的不足之处，内存引用条不准确，形同虚设，但是这对内存优化来说至关重要。没有Mono内存对象的统计，会错误地把Editor的内存也统计进来。

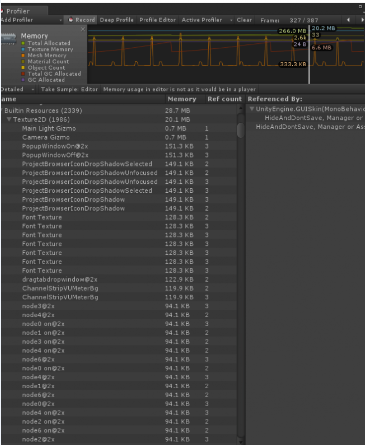


图2.1 Unity自带的内存分析工具

由于以上的不足，Unity technologies公司推出了新版本的只针对内存的分析工具，如图所示，该工具解决了之前工具的不足，图形化的结构能看出每种资源的大小占比情况，具有详细的引用链。但是也存在一些问题：由于新版的MemoryProfiler通过Treemap的形式展现数据，在操作时很容易因为数据量太大而定位到单个对象。快照对比是内存分析中常用的手段之一，对于游戏来说，在进入关卡和跳出关卡的时候各做一次内存快照，通过快照的对比很容易找出该卸载而没有卸载的资源，也就是找出了泄露的内存。再者，无法通过搜索定位到想要的资源，这是一个快速定位到目标资源的方法。另外没有对mono内存的分析。

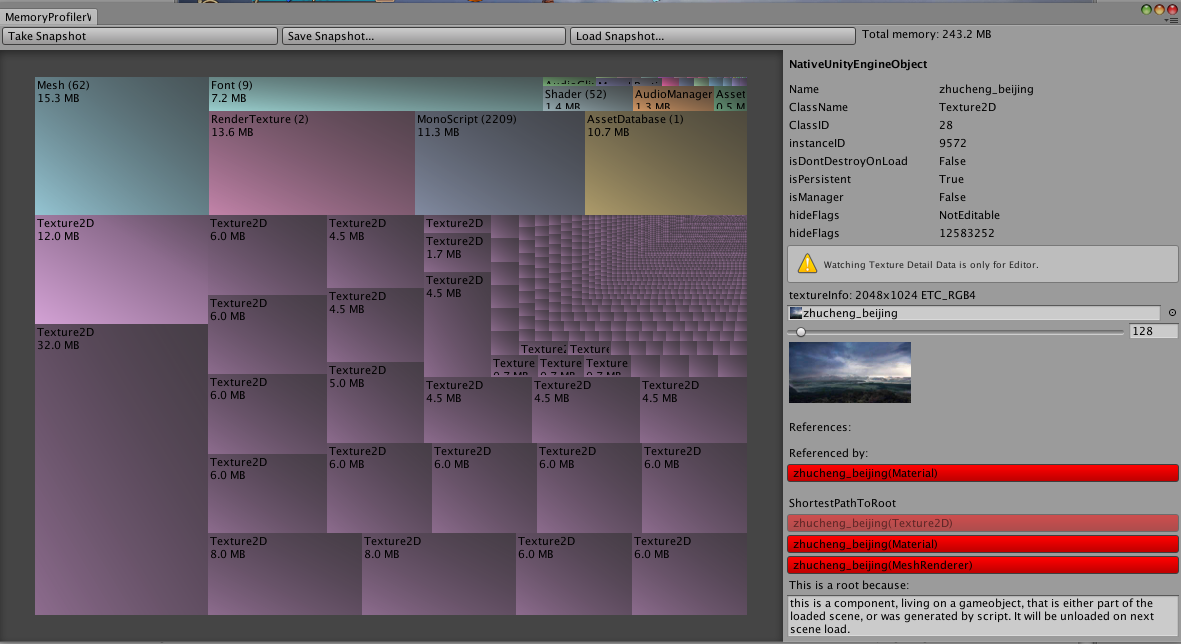


图2.2 Unity5.3版本后的内存分析工具

## 2.2 游戏优化公司的内存分析

现在国内的一些著名的游戏公司开发了自己的游戏内存分析工具，如腾讯的WeTest，另外还有专门的测试平台如UWA，但是这些测试平台针对的是游戏安装包，你需要将安装包上传到他们的网站，通过一段时间的运行，他们会给出比较全面的游戏测评结果，还有一种就是你安装他的客户端测评软件，需要对手机进行root，然后hook进你要测试的安装在手机上的游戏程序，通过让你的游戏运行一段时间，他会给出一个在游戏运行时间内内存的分析结果，这个结果往往是文本结果。 以上的内存分析的工具的最大的缺陷在于，必须将游戏打包之后，安装或者上传游戏安装包，才能分析内存的不足。这样做的坏处，一是打包一个游戏往往需要很长时间，对于大型的3D游戏来说更加严重，二是游戏开发人员希望在编辑的时候就定位到内存问题，直击问题内存，这样修改起来快速又方便，还省去了打包环节，大大节省了游戏开发时间。

本文在充分考虑当前Unity3D内存分析面临的困难的前提下，从内存可视化，内存快照对比,Mono内存泄露分析，内存分析实例应用这四个方面入手，为Unity设计和实现了一款内存分析工具，来为游戏开发人员迅速定位内存问题，缩短游戏开发周期。

## 2.3内存泄露检测方法研究现状

关于C#内存泄露的学术论文极少，但是c#和java一样都是具有垃圾回收功能的语言，很多java内存泄露的检测方法一样可以运用到c#上。对于Java内存泄露的研究很多，可以分为在线方法，离线方法和混合方法三类。

在线方法主动监控并与运行的虚拟机进行交互，以检测泄露对象，在线方法的有点是可以访问运行时的信息，如跟踪内存分配，监测对象是否能被会回收等，在线方法的缺点是资源消耗大（时间，内存），另外一些在线方法需要修改虚拟机或者垃圾回收器。在线检测方法又可以分为Staleness detection 和 Growth analysis 两类，前者的原理：如果一个对象在内存中存在了很久但是没有被使用，那么这个对象很可能泄露了。后者通过检测类的实例的无边界增长导致内存的增加，来判定该类为内存泄露。关于Staleness detection的研究，Bond等提出了Bell（Bit Encoding Leak Detection）[[[12]](#endnote-12)]，需要改变垃圾回收器来使每个对象的头部带有对象存活信息的比特位，每次回收时更新该位置信息，这样我们得到了各个对象的staleness信息从而得到可能的泄露内存。Bell方法被Tang等人用在LeakSurvivor[[[13]](#endnote-13)]中来检测泄露然后将可能的泄露从内存中移出到磁盘上，LeakSurvivor在Jikes RVM中作为垃圾收集齐的一部分而实现的。Rayside和Mendel提出了另一种利用staleness的内存泄露检测方法[[[14]](#endnote-14)]：Object ownership profiling，该方法记录每个对象的唯一标识符，大小，创建时间，收集时间，每个方法调用或字段访问的源和目标，通过绘制可到达的和主动的（在跟踪中主动引用的）对象的分配空间图来分析收集的跟踪。从图中看出对象是否是staleness。Xu等人提出了一种container profiling的方法来对java进行精确的内存泄露检测[[[15]](#endnote-15)][[[16]](#endnote-16)]。该方法假设java大部分的内存泄露发生在container中，因此该方法通过对集合类staleness情况的监测来检测内存泄露。关于Growth analysis 类方法的研究，Jump和McKinley提出了一种动态内存泄露检测方法，该方法根据内存信息画出Type Points-From Graph(TPFG)图，通过图中节点的增长来确定内存泄露对象[[[17]](#endnote-17)]。通过统计学方法来检测内存泄露是现在的主题，由Sor等人首次提出[[[18]](#endnote-18)][[[19]](#endnote-19)]。该方法着眼于对象增长的检测，尽管该方法搜寻创建后在很长一段时间内没有被释放的对象，但该方法不考虑对象的访问时间，因此它是一个Growth analysis而不是一个staleness检测方法。该方法又被进一步发展和研究[[[20]](#endnote-20)][[[21]](#endnote-21)]，Vladimir提出了在统计学习的基础上用机器学习的方法来提升内存检测正确率的方法[[[22]](#endnote-22)]，Andrzejak等人将该方法应用到C/C++上[[[23]](#endnote-23)]。

离线方法旨在帮助程序员通过分析内存转存信息或者一些其他数据的分析和可视化来检测内存泄露，这些方法消耗低不需要改变运行系统。Eclipse 的MAT 是一个分析Heap dump的工具，该工具分析各个对象间的引用关系大小极大地帮助人工检测内存泄露。.net的内存分析工具memProfiler通过可视化内存信息来帮助人工检测内存泄露。Reiss提出了一个可视化的内存检测方法[[[24]](#endnote-24)]。另一个分析工具LeakBot[[[25]](#endnote-25)]着眼于部分的heap dump分析，该方法在数据结构体的层次观测泄露而不是观察单个的对象。Maxwell等提出了一个通过应用图像挖掘算法在heap dumps中寻找内存泄露的方法[[[26]](#endnote-26)]。该方法将heap dump转成dominator树，然后挖掘这棵树找到重复的子树，该子树即为可能的内存泄露点。另外静态分析方法通过分析代码结构找出可能泄露代码[[[27]](#endnote-27)][[[28]](#endnote-28)]。

混合方法同时结合了在线和离线分析来实现最好的结果，但是往往需要和用户交互来帮助分析中间结果或者为接下来的步骤准备数据，如Xu等人在LeakChaser中提出的three-step iterative profiling 方法[[[29]](#endnote-29)]。

# 3主要研究内容

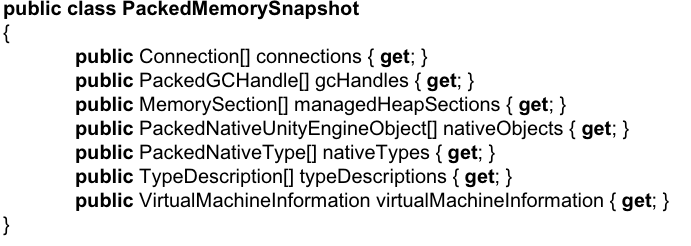
本文针对unity内存的特点，分别对native堆内存和托管堆内存进行内存优化，对于native堆，由于主要是美术资源，采用快照对比的方法能很好的找出内存泄露，针对unity现有的内存分析工具的问题，本文采用了列表的内存可视化方式，并且加入了快照对比功能以及搜索过滤功能，通过相同场景间的比较能快速找出增加的native资源，通过不同场景间的比较能找出内存常驻资源，有助于找出不该常驻的内存资源。对于托管堆内存的优化，本文借鉴了Vladimir提出的机器学习的方法，将内存泄露处理成一个分类问题，并在特征向量的方面采用了其gencount的概念，在数据集方面，原始数据分为两部分，一部分为杭州卓亨游戏公司unity游戏发生内存泄露的原始数据，另一部分为在没有内存泄露的游戏代码中插入已知的c#泄露代码，采用随机森林,支持向量机决策树等方法进行模型训练并比较这些方法分类的正确率。

## 3.1内存数据获取（已完成）

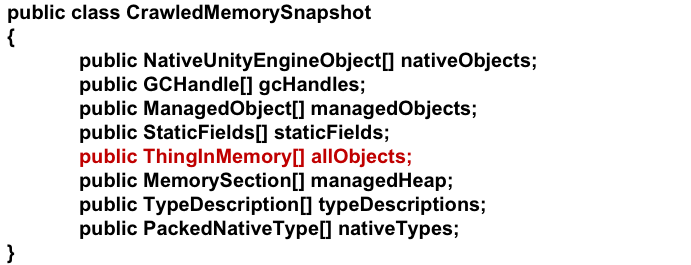
要可视化内存信息，首先要获取内存数据，Unity提供了一个API能获取内存快照信息：

UnityEditor.MemoryProfiler.MemorySnapshot.RequestNewSnapshot();

通过调用这个函数，可以向运行着的Unity程序申请一个新的内存快照。如果是运行于编辑器内的程序，会返回资源内存的信息（native 内存），如果是运行基于il2cpp的应用，则会获取所有的内存信息（Native，mono，others），通过这个接口，获取的信息会保存在如下图所示的对象里：



通过对上面对象进行解压缩处理，得到完成的引用信息链，从而的到内存的完整引用信息，保存在如图所示的类中，此时，所有的内存信息都能在红色标识的变量中获得。

3.2 native内存优化（已完成）

### 3.2.1内存可视化

考虑到最新的unity内存分析工具在场景资源很多时很难定位到小资源，在可视化的时候采用了TableList双列表的形式来展示资源。为了迅速定位到想要的资源，设计了一个搜索框，为了过滤掉尺寸较小的内存，只展现出大于某个尺寸的内存资源，设置了资源大小过滤框，另外对于某一类的资源，如果超过了建议值，那么将其高亮，根据以上的设计准则，实现的内存可视化如下图所示：

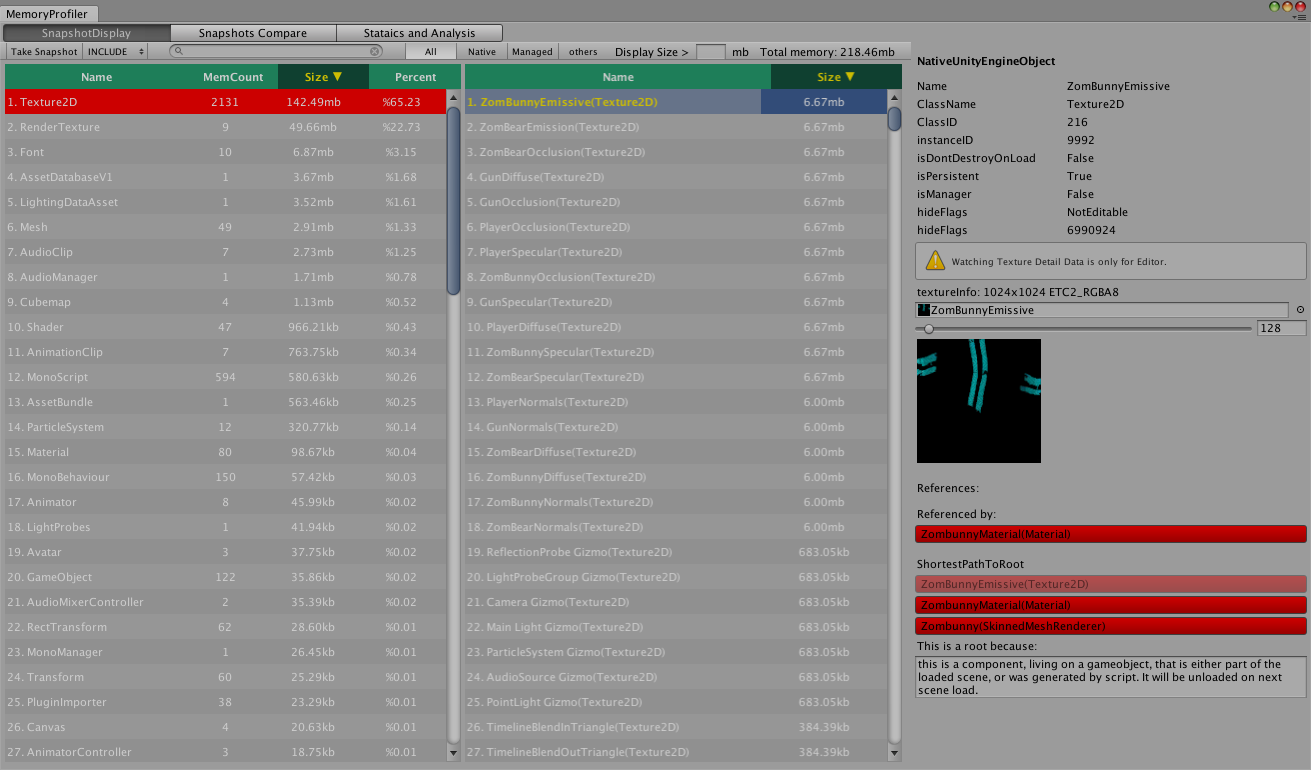


图3.1 内存资源展示图

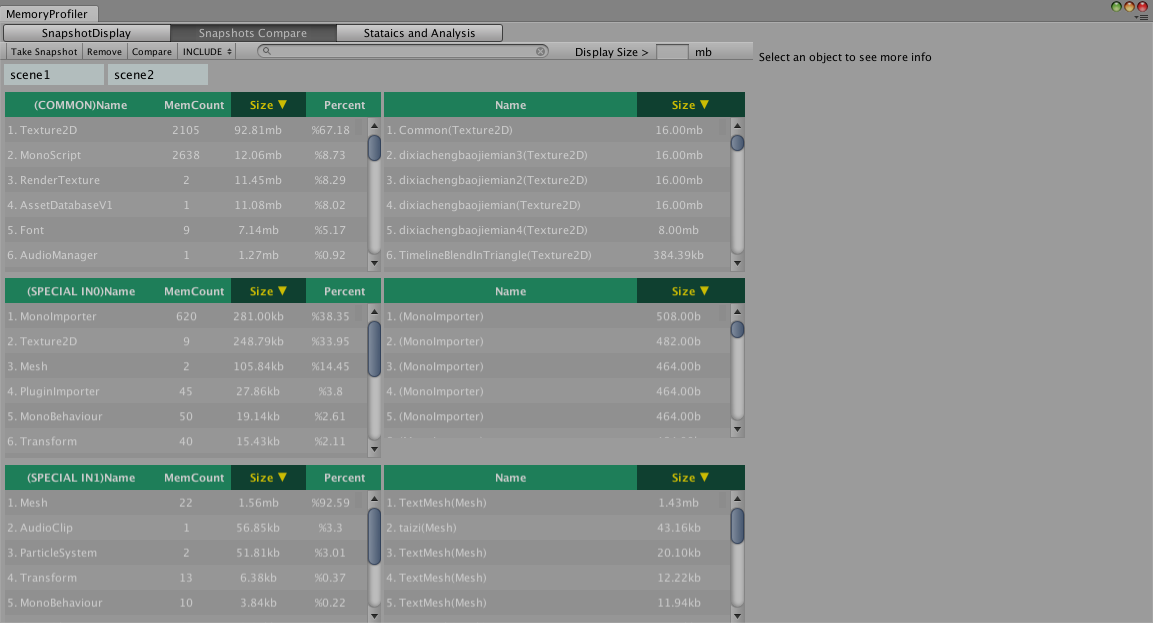
从图中可以看出，SnashotDisplay是内存展示的内容，下面的take Snapshot按钮按下之后会获取此时的内存快照信息组，如Textrue2，Shader，Script等各种组别，用双列表的形式展示在左下角，左边的是大组的名字，选中后，双列表的右边列表则会展示出该大组下的具体内存内容，当选中右边列表中想查看的资源，在整个编辑视图的右边展示出该内存的具体内容，包括名字，引用等信息。Take snapshot旁边是一个下拉菜单，可以选择include和exlude，这个下拉菜单是和右边的搜索框配合起来使用的，include，表示只展示包含搜索框内容的内存内容，exclude表示只展示不包含搜索框内容的内存内容。设置这个下拉菜单的意义在于，当需要定位到每个具体的资源的时候选择include，美术在命名美术资源时，对于同一个场景中的美术资源，会包含这个场景的名字，exclude出包含这个场景名字的资源，剩下的资源很可能不应该出现在该场景中，这样能很快的查看是否存在资源内存泄露。Display size >后面的框，能过滤掉尺寸小于该输入值的内存，这样能更快的查看大尺寸的资源。在TableList列表的左边，Texture2D被高亮成红色，说明Texture2D占用内存的大小超过了建议值。关于建议值，并没有一定要是多少，需要根据游戏的受众和游戏的具体题材来调整，不过，游戏测试平台UWA经过多年的对游戏的优化，给出了主要资源的建议值，游戏的各项内存尺寸最好在建议值内。游戏开发人员根据自己的项目的具体情况，对比建议值就能判断自己的内存占用是否合理，UWA的主要的内存占用如下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 纹理资源 | 网格资源 | 动画片段 | 音频片段 | Mono内存 | 其他 |
| 50M | 20M | 15M | 15M | 40M | 10M |

表3.1 UWA内存建议值

### 3.2.2两个内存快照对比

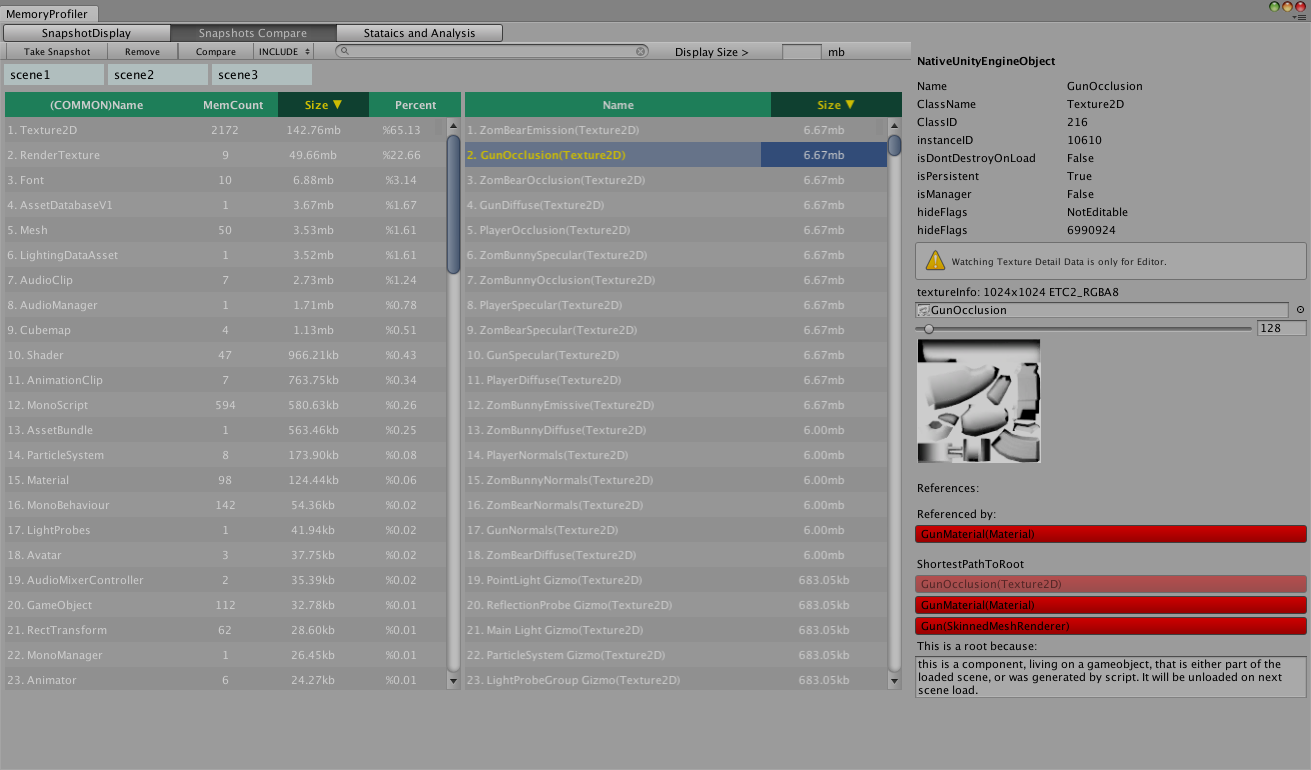
两个内存快照对比，是内存对比最常用的方式，在之前我们已经完成了内存快照的新建、选择和删除，在此基础上，选择需要对比的两个快照进行内存对比，这样得到两个快照中的相同部分和不同部分，当我们对比关卡进入前和关卡退出后的内存变化，很容易找出资源内存泄露。实现后的窗口图如下图所示：



COMMON指的是两个快照相同的部分，SPECIAL IN0指先选中的快照中不同的部分，SPECIAL IN1指后选中的快照中不同的部分。

### 3.2.3 多个内存快照对比

两个内存快照对比主要是为了找出两者之间的不同点，多个内存之间的对比只展示他们相同的内存，这部分能存可视为常驻内存，对于这部分内存，游戏开发者可以观察是否有不该常驻的内存存在。



## 3.3 Mono内存泄露分析（下一步）

前述工作主要着重解决资源内存占用过大和资源内存泄露的问题，本章主要解决内存不合理的另一个方面：Mono内存泄露，采用机器学习的方法，从特征向量的选择，数据集的来源，和学习算法来介绍。

### 3.3.1 Mono内存增长原因

mono内存的分配主要分为以下三步：

1.首先，unity检测是否有足够的闲置内存单元用来储存数据，如果有，则分配对应的内存单元；

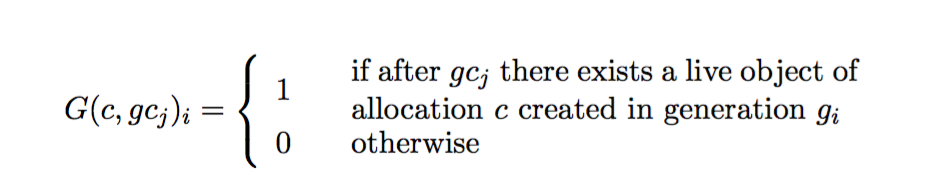
2.如果没有足够的存储单元，unity会出发垃圾回收来释放不再被使用的堆内存。

3.如果垃圾回收后并没有足够的内存单元，则unity会扩展堆内存的大小。

4.如果之前分配的堆内存一直不被回收的话，mono堆会一直增大从而造成游戏性能的降低甚至闪退。

## 3.3.2特征选择

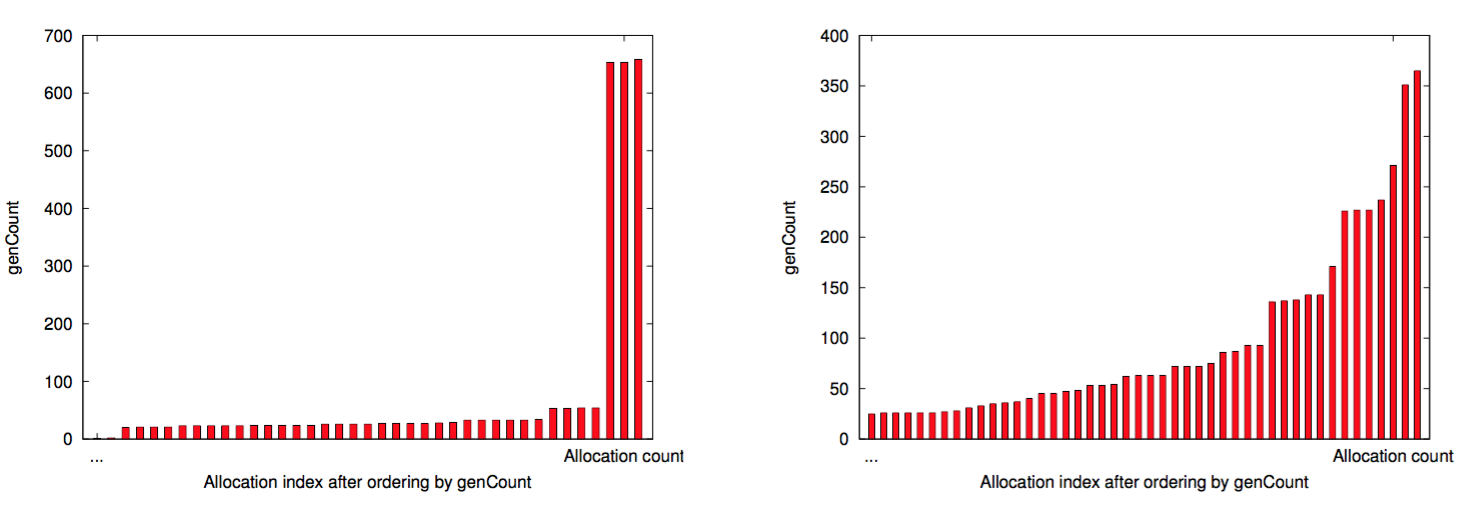
文章引用了Vladimir论文中的这个概念，该特征的定义过程如下，定义程序运行中的垃圾回收为。如果一个对象在垃圾回收和中间创建，那么就说该对象属于代。定义C为程序中的所有内存分配点的集合，定义定义域为{0，1}的函数：



其中i属于{0,1,…,j}，那么定义如下：



该特征是检测内存泄露的重要指标，Vladimir论文首先运用了统计学习方法，然后用机器学习方法提高正确率，统计学习方法的核心思想就是发生内存泄露的分配站点的值明显高于没有发生内存泄露的分配站点的值。将分配站点的值进行排序，通过比较相邻站点的是否有明显差异来判断泄露是否发生。下图，左边的图间有明显的分隔边界，可以判断边界右边的分配站点发生了内存泄露，而右边的图并没有明显的边界，因此没有足够证据判断其为泄露。



在特征变量的选择方面本文引用了Vladimir论文的特征,另外考虑了如下特征：

: 该特征为一个逻辑变量，为0时树状图没有明显的分隔线或者有明显的分隔线但是该站点在分隔线左边。为1时表明有明显的分隔线且该站点在分隔线的右边。

: Vladimir论文的特征中并没有考虑时间，而一个变量在分配之后很长的一段时间内都没有被回收，是判断内存泄露的重要根据，据此本文在的基础上建立了另一个特征。定义为，数学表达式如下：



上式不仅仅是计算到最后一次垃圾回收，他将之前代数的垃圾回收的值相加，这样当一个对象被创建后，时间越长其值越大。

: 考虑到最后一次GC后每个站点所存的对象的数量对是否发生泄漏有影响。因此加入这个特征，来表征最后一次GC后每个站点的对象数量。

因此特征向量表示如下：



### 3.3.2 数据集获取

研究内容与现状中所提及的内存泄露的数据集可分为两种，一种是现有的已知泄露的数据集，另一种是自己在无泄露代码中插入泄露代码的内存泄露数据集。本文能获取的一部分数据为杭州卓亨游戏公司开发的几款游戏中的内存泄露数据，因为泄露数据有限，需要在开源的没有内存泄露的游戏代码中插入已知的泄露代码来增加数据集的规模。

### 3.3.3 学习方法选择与方法效果评估

由于数据集不大，因此选用支持向量机，随机森林，决策树方法，对数据集进行学习，并对学习效果进行验证，给出各方法的学习效果比较，并选择较好的学习方法作为unity分析工具的学习方法。

## 3.3.4 应用内存泄露检测方法

经过上述步骤学习到mono内存泄露检测方法的模型，将该模型在内存检测工具中实现。

# 4研究工作计划安排

2018.3-2018.6 数据集的获取与处理；

2018.7-2018.11 训练与评估学习模型；

2018.12-2019.3 整理文档以及实验数据，撰写毕业论文，准备毕业设计结题答辩。

# 5预期成果

毕业论文侧重内存分析工具的设计与实现，着重针对资源内存占用过大，资源内存泄露和mono内存泄露问题。预计撰写并发表1篇以上高质量的文章。同时希望积累对游戏内存的优化、Unity GUI的实现和大型的C#工程的实现的经验。

# 参考文献

1. [] Messaoudi F, Simon G, Ksentini A. Dissecting games engines: The case of Unity3D[C]// International Workshop on Network and Systems Support for Games. IEEE, 2016:1-6. [↑](#endnote-ref-1)
2. [] B. Cowan and B. Kapralos, “A survey of frameworks and game engines for serious game development,” in IEEE Int. Conf. on Advanced Learning Technologies (ICALT), 2014.  [↑](#endnote-ref-2)
3. [] https://unity3d.com/cn/public-relations [↑](#endnote-ref-3)
4. [] Xie J. Research on key technologies base Unity3D game engine[C]// International Conference on Computer Science & Education. IEEE, 2012:695-699. [↑](#endnote-ref-4)
5. [] Anderson E F, Engel S, Comninos P, et al. The case for research in game engine architecture[C]// Conference on Future Play: Research, Play, Share, Future Play 2008, Toronto, Ontario, Canada, November. DBLP, 2008:228-231. [↑](#endnote-ref-5)
6. [] Tadres A. Extending Unity with Editor Scripting[M]. Packt Publishing, 2015. [↑](#endnote-ref-6)
7. [] https://docs.unity3d.com/Manual/editor-EditorWindows.html [↑](#endnote-ref-7)
8. [] 刘克, 李建方. 基于Unity3D的内存优化的研究[J]. 中国传媒大学学报(自然科学版)自然科学版, 2016, 23(5):56-61. [↑](#endnote-ref-8)
9. [] Ma X, Deng Z, Dong M, et al. Characterizing the Performance and Power Consumption of 3D Mobile Games[J]. Computer, 2013, 46(4):76-82. [↑](#endnote-ref-9)
10. [] Jiang J, Kuang Y, Shen H. Research on the 3D Game Scene Optimization of Mobile Phone Based on the Unity 3D Engine[C]// International Conference on Computational and Information Sciences. IEEE, 2011:875-877. [↑](#endnote-ref-10)
11. [] M. V. de Hoef and B. Zalmstra, “Comparison of multiple rendering techniques,” University of Utrecht, Tech. Rep., 2010. [↑](#endnote-ref-11)
12. [] Bond M D, Mckinley K S. Bell:bit-encoding online memory leak detection[J]. Acm Sigops Operating Systems Review, 2006, 40(5):61-72. [↑](#endnote-ref-12)
13. [] Tang Y, Tang Y, Gao Q, et al. LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages.[C]// Usenix Technical Conference, Boston, Ma, Usa, June 22-27, 2008. Proceedings. DBLP, 2008:307-320. [↑](#endnote-ref-13)
14. [] Rayside D, Mendel L. Object ownership profiling: a technique for finding and fixing memory leaks[C]// Ieee/acm International Conference on Automated Software Engineering. ACM, 2007:194-203. [↑](#endnote-ref-14)
15. [] Xu G, Rountev A. Precise memory leak detection for java software using container profiling[C]// ACM/IEEE, International Conference on Software Engineering. IEEE, 2008:151-160. [↑](#endnote-ref-15)
16. [] Xu G, Rountev A. Precise memory leak detection for java software using container profiling[J]. Acm Transactions on Software Engineering & Methodology, 2013, 22(3):1-28. [↑](#endnote-ref-16)
17. [] Jump M, Mckinley K S. Cork:dynamic memory leak detection for garbage-collected languages[J]. Acm Sigplan Notices, 2007, 42(1):31-38. [↑](#endnote-ref-17)
18. [] Šor V, Salnikov-Tarnovski N, Srirama S N. Automated Statistical Approach for Memory Leak Detection: Case Studies[C]// Th Confederated International Conference on on the Move To Meaningful Internet Systems. Springer-Verlag, 2011:635-642. [↑](#endnote-ref-18)
19. [] Vladimir Šor, Srirama S N. Evaluation of embeddable graph manipulation libraries in memory constrained environments[C]// ACM Research in Applied Computation Symposium. ACM, 2012:269-275. [↑](#endnote-ref-19)
20. [] Or V, Ou P, Treier T, et al. Improving Statistical Approach for Memory Leak Detection Using Machine Learning[J]. 2013, 8144:544-547. [↑](#endnote-ref-20)
21. [] Šor, Vladimir, Srirama, Satish Narayana, Salnikov‐Tarnovski, Nikita. Memory leak detection in Plumbr[J]. Software Practice & Experience, 2015, 45(10):1307-1330. [↑](#endnote-ref-21)
22. [] Šor, Vladimir. Statistical approach for memory leak detection in Java applications[C]// Mid-Year Meeting of the Chemical Research Society of India. 2014. [↑](#endnote-ref-22)
23. [] Andrzejak A, Eichler F, Ghanavati M. Detection of Memory Leaks in C/C++ Code via Machine Learning[C]// IEEE International Symposium on Software Reliability Engineering Workshops. IEEE Computer Society, 2017:252-258. [↑](#endnote-ref-23)
24. [] Reiss S P. Visualizing the Java heap to detect memory problems[C]// IEEE International Workshop on Visualizing Software for Understanding and Analysis. IEEE, 2009:73-80. [↑](#endnote-ref-24)
25. [] Mitchell N, Sevitsky G. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications[C]// ECOOP 2003 - Object-Oriented Programming, European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings. DBLP, 2003:351-377. [↑](#endnote-ref-25)
26. [] Maxwell E K, Back G, Ramakrishnan N. Diagnosing memory leaks using graph mining on heap dumps[C]// ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, Dc, Usa, July. DBLP, 2010:115-124. [↑](#endnote-ref-26)
27. [] Distefano D, Filipović I. Memory Leaks Detection in Java by Bi-abductive Inference[C]// International Conference on Fundamental Approaches To Software Engineering. Springer-Verlag, 2010:278-292. [↑](#endnote-ref-27)
28. [] Yan D, Xu G, Yang S, et al. LeakChecker: Practical Static Memory Leak Detection for Managed Languages[C]// Ieee/acm International Symposium on Code Generation and Optimization. ACM, 2014:87. [↑](#endnote-ref-28)
29. [] Xu G, Bond M D, Qin F, et al. LeakChaser: helping programmers narrow down causes of memory leaks[J]. Acm Sigplan Notices, 2011, 46(6):270-282. [↑](#endnote-ref-29)