Софийски университет “Св. Климент Охридски”

Факултет по математика и информатика

Катедра Информационни и комуникационни технологии

Дипломна работа

JProfiler – инструмент за оптимизиране на Java приложения

Автор:	Иво Митков Карабашев
Факултетен номер:	42625
Специалност:	Информатика
Специализация:	Информационни и комуникационни технологии
Научен ръководител:	Сергей Върбанов

София 2005

I. Увод

Езика за програмиране Java е лесен за научаване и използване, затова през последните няколко години добива все по-голяма популярност. Вече има доста сървърни и потребителски приложения, дори мултимедия и игри написани на Java. Виртуални машини се вграждат в превозни средства, потребителски уреди и мобилни устройства. Но преносимостта и лесната употреба си имат своята цена, софтуера на Java изисква повече ресурси - процесорна мощ и памет. Получава се парадокс, тъй като тези ресурси са ограничени във вградените системи. Разработчиците често се сблъскват с три типа проблеми:

- *Управление на паметта*. Garbage collector е оптимизиран за приложения с относително малък размер на използваната памет. Като резултат, големите приложения могат да страдат от чести, продължителни и непредвидими забавяния по време на garbage collection.
- *Неефективни методи*. Методите написани за гъвкавост и универсалност могат да предизвикат значителни проблеми с производителността, когато се използват многократно. Например, разработчиците често откриват, че необичайно много време се прекарва в определени методи от пакета `java.util.String`.
- *Синхронизирани методи*. Синхронизираните методи налагат взаимно изключващ се достъп до защитените обекти. Надпреварата породена от синхронизиран метод може да намали производителността значително.

Без подходящо помощно средство е трудно да се анализират програмите за проблеми с производителността. JProfiler - целта на тази дипломна работа, е ориентиран към решаването на първите два проблема. За в бъдеще той може да бъде разширен към адресирането и на третия проблем. JProfiler е предназначен не само за използване от професионалисти. Начинаещи програмисти също могат да се възползват от

функциите, който той предлага, тъй като понякога не е толкова очевидно, къде точно може да се корени проблема за ниската производителност. JProfiler притежава инструменти за визуализация и статистически анализ, който помагат за бързо извличане на резултатите от профилирането и по-пълно разбиране на породения огромен обем информация.

Разработването на JProfiler се ръководеше от няколко основни изисквания, залегнали още в началната фаза на анализ и дизайн:

Изискването за функционалност включва:

- способността да проследява и записва използваното време за всички методи изпълнени във виртуалната машина;
- способността да проследява и записва изразходваната памет за всички създадени обекти от виртуалната машина;
- способността да възстановява пътя на извикване на методите в дървовидна структура, с извикващите методи родители на извикваните;

Изискването за производителност включва:

- изпълнение, което не изкривява значително профила на производителността на приложението, което се проследява;
- контрол от страна на потребителя върху генерираната информация, т.е. потребителя трябва да може да изключва проследяването на отделни модули (памет, процесорно време) и скриването на информацията от чужди пакети, ако се интересува само от собствения си код;

Изискването за удобство включва:

- лесен за използване интерфейс при изследване на получените резултати;

Изискването за универсалност включва:

- използване на интерфейс, който да се поддържа от максимален брой виртуални машини, като най-подходящия избор се оказа Java Virtual Machine Profiler Interface (JVMPI);

Конвенции

Следните конвенции са използвани в текста.

Типографски конвенции:

`машинописен`	• Изходен код на Java и примери на съдържание на файлове • Команди и аргументи
курсив	Имена на пътища и файлове
<стойност>	Стойност, която трябва да се замени Например, <profiler_dir> е директорията, където е инсталиран JProfiler

II. Препоръки за оптимизация без помощта на допълнителни инструменти и ползата, която може да се очаква от всяка от тях

II.1. Оптимизиране с цел бързодействие

а) структура

Архитектурата и алгоритмите на програмите са много по-важни отколкото каквато и да е оптимизация от ниско ниво. Лошата архитектура и алгоритми могат да направят всяка система бавна. Предварителната оптимизация често води до грешки, но ако не се вземе в предвид производителността още от самото началото, програмата може да се окаже безполезна.

- да се настройва от най-високото към по-ниското ниво;
- да се оптимизира общия код или този, който се изпълнява многократно;

- да се проверява дали системата не губи време, дори когато няма вход;

b) обектно ориентирано програмиране

- къса верига от супер класове - цената при създаване на обект е свързана с размера на веригата от супер класове. Тъй като всеки супер клас трябва да присъства преди подкласа да може да бъде създаден, се получава натоварен мрежов трафик при аплети и RMI програми. От друга страна, късата верига противоречи на принципа на ООП.
- предпочитание на променливите от стека пред променливите на класа - променливите на класа са достъпни чрез по-заобиколен път, който изисква повече време.
- сливане на класовете - дали си заслужава зависи от това как ще се направи. Един голям клас може да съдържа много код, който рядко се използва, но все пак трябва да се зареди и инициализира само един клас.
- `final` методи - тъй като методите, който не са `final` се свързват по време на изпълнение, ако метода няма да се пренаписва, е хубаво да се декларира като `final`, или още по-добре целия клас.
- по-малко създадени обекти - обектите трябва да се използват многократно, но тъй като синхронизираното извличане на обект от масив е също толкова бавно колкото създаването на нов, ако обекта е един, нека той е статичен с преинициализираща функция.

c) аритметика

- съставни оператори - `x += 4` е по-бързо от `x = x + 4`. Един добър компилатор би трябвало да ги замества автоматично.

- побитово изместване - побитовото изместване е по-бързо от умножението със степени на двойката. Отново компилатора би трябвало да се грижи за това.
- операциите с `int` са по-бързи от операции с `byte` и `short` - повечето процесори са оптимизирани за работа с 32-битови стойности.
- операциите с `float` са доста по-бавни от операции с `int` - а операциите с `double` са малко по-бавни и от `float`.
- достъп до паметта - от най-бърз към най-бавен: локални променливи, променливите от супер супер класа, променливите от супер класа, променливите от класа, статичните променливи. Копирането на често използваните променливи от такива със бавен достъп в локални е препоръчително.

d) мрежи

- буфериране на мрежовия трафик - огромна полза, иначе трансфера е байт по байт.
- UDP вместо TCP - ако скоростта е по-важна от точността и гаранцията.

e) нишки

- разпределяне на задачите в отделни нишки - дори и да не се изпълнява програмата на многопроцесорна машина, така се използва напълно потенциала на процесора. Естествено не бива да се прекалява, за да не се губи повече време за превключване между нишките, отколкото за полезна работа.
- използване на приоритети - нишки, чиито резултати са необходими възможно най-бързо, е добре да са с по висок приоритет, за да не се забавят другите части на програмата в изчакване на тези резултати. От друга страна нишките, чиито резултат не е толкова важен в момента, да са с по-нисък

приоритет. Например, `paint` или запис във файл, който ще се разглежда на по-късен етап от потребителя.

- `notify` вместо `notifyAll` – `notifyAll` е доста по-времеотнемащ.
- синхронизация само при нужда - синхронизацията е необходима, за да работи програмата коректно, но си има своята цена и трябва да се избягва ако е възможно. Може да има полза да се пренапишат някой класове от библиотеките на Java, ако няма нужда от синхронизация. Синхронизацията може да причини по-бавно изпълнение на метод с JIT отколкото без.

f) цикли

- Броенето надолу е често по-бързо от броенето нагоре - "преход при нула" има по-малко вътрешни инструкции, съответно се изпълнява за по-кратко време от "преход при различно от нещо". Друга идея, която може да помогне в някои случаи е да не се прави проверка, а да се използва изключение, за да се излезе от цикъла, но трябва да се внимава, тъй като изключенията са скъпоструващи.
- синхронизацията извън циклите - входно-изходните класове са синхронизирани, така че е по-добре да се правят такива операции наведнъж, вместо в цикъл, защото всяка итерация ще губи време за отключване и заключване.
- променливите на класа са по-бързи от достъп до елемент на масив - тъй като достъпа до масива изисква проверка за границите.

g) стрингове

- намаляване на броя на стринговете - в стандартните приложения стринговете заемат най-много памет, което означава, че garbage

collector-а ще бъде затруднен и програмата ще се изпълнява тромаво, освен това операциите със стрингове са неефективни.

- избягване на манипулациите върху стрингове - дори най-елементарните операции със стрингове са доста времеотнемащи. Слепването на стрингове е може би най-бавната от тях. Вместо оператора + е по-добре да се използва `StringBuffer`, тъй като при всяко слепване се създава нов обект `String`, в който се копират началните два, като ползата може да надхвърли 300%.

h) графика

- бавни шрифтове - екзотичните шрифтове изискват време за зареждане и инициализиране, в сравнение със стандартно вградените. Съответно колкото по-малко шрифтове се използват, толкова по-добре.
- кратък метод `paint` - извиква се доста често. Спестява се повече време за същинска обработка на информацията.
- изрязване в метода `paint` - намалява броя на извършваните операции. Вместо да се обработват всички редове от документ, се отпечатват само видимите.
- използване на компресия на изображенията - дори най-елементарната компресия намалява значително обема на предаваните данни, което компенсира допълнителното време за декомпресия. Да не говорим за спестеното място в паметта и носителите.
- двоен буфер за по-плавна графика - буферирането понякога предлага много по-качествено изображение отколкото двойно по-бърз процесор.

i) изключения и обработка на грешки

- използване на системата да извършва проверките за грешка - тъй като системата така или иначе извършва проверки за грешка, в

някой случаи е ползотворно да не се проверяват и в програмата. Например в цикъл за обработка на масив, пропускането на проверката за достигане до крайния елемент на масива, а вместо това прихващане на `ArrayIndexOfBoundException`. Но това важи само за големи масиви, иначе си получава обратен ефект.

II.2. Оптимизиране с цел размер

- да не се преоткриват API класовете - да се използват или наследяват класовете от Java API, когато е възможно. Sun са написали тези класове, за да не се налага програмиста да ги пише. Освен това може да се допусне грешка.
- използване на наследяването - колкото повече методи се наследяват, толкова по-малко код трябва да се пише.
- включване на оптимизатора на компилатора - използването на `javac -O`, води до вмъкването на малките функции и премахването на номерата на редовете. Освен, ако не се използват много такива функции, премахването на номерата на редовете ще доминира и компилирания код ще бъде по-малък.
- отделяне на общия код - код, който е един и същ на различни места да се постави в отделен метод (обратното на *inlining* (вмъкването) на код за скорост)
- избягване инициализирането на големи масиви - въпреки, че инициализацията на масив е единична операция в Java, генерирания байткод вмъква елементите един по един. Ако масива е голям, това изисква много байткод.
- по-кратки имена - имената на видимите единици (т.е. имена на клас, метод или променлива) са изредени в клас файла, така че използването на кратки имена спестява място. Обфускаторите на код пренаписват клас файловете да използват по-кратки (и неразбираеми имена).

- `static final` константите в интерфейс - константите дефинирани като `static final` се включват в клас файла, а също така и се вмъкват. Ако вместо това се дефинират в интерфейс, който се имплементира, се избягват допълнителното включване.
- слепване на стрингове - използването на + като оператор за слепване на стрингове се извършва чрез извикването на метод, ако операндите та са константи по време на компилирането. Премахването на цикъл, за да се елиминира слепването на стрингове може да спести място. Например, следния цикъл се компилира в 48 байта код:

```
for (int i = 0; i < 3; i++) {
  imgs[i] = getImage (getCodeBase(), "G" + i + ".gif");
}
```

Ако се премахне цикъла се елиминира слепването на стрингове по време на изпълнение и се намалява размера на байткода на 39 байта:

```
imgs[0] = getImage (getCodeBase(), "G0.gif");
imgs[1] = getImage (getCodeBase(), "G1.gif");
imgs[2] = getImage (getCodeBase(), "G2.gif");
```

III. Загуба на памет

III.1. Алгоритъм на работа на garbage collector

Изтъквано предимство на Java е вграденият garbage collector, който би трябвало да се справя с един от най-предизвикателните проблеми на програмирането - загубата на памет. Въпреки това в доста приложения се

наблюдава нарастване на неосвободената памет, което води лоша производителност и накрая до блокиране.

Първо нека да разгледаме как се осъществява управлението на паметта в Java и как работи garbage collector. Обектите се заделят в купа на паметта чрез оператора `new` и са достъпни чрез указатели. Може би най-подходящия начин да си представим паметта от купа е във формата на ориентиран граф, където обектите са възлите, а указателите между обектите са ребрата. Garbage collector вижда паметта по този начин.

Целта на garbage collector е да премахне от паметта обектите, който не са нужни повече. Това е трудно решим проблем - garbage collector не може да знае дали ни трябва даден обект, затова използва апроксимация и търси обекти, който не са достижими. Като използваме аналогията с ориентирания граф, той търси обекти, който не могат да бъдат достигнати по никой път започващ от корена. Корените - фиксирани места, който гарантирано съществуват винаги, са стартовите точки за garbage collector. В Java корените включват статичните полета на класовете и локалните променливи от стека. Всичко, което garbage collector не може да достигне от някой от корените се счита за боклук.

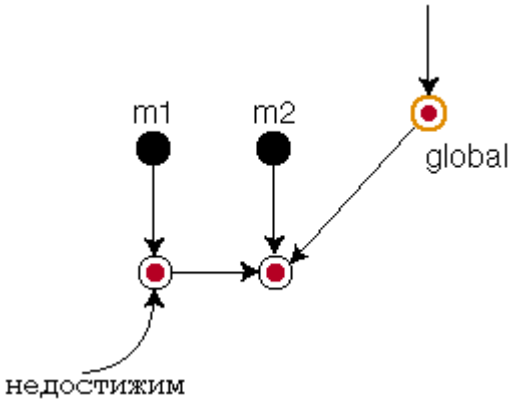
За да илюстрираме това, нека разгледаме следния пример:

```
public void useless() {
     MyObject m1 = new MyObject();
     MyObject m2 = new MyObject();
     m1.ref = m2;
     global.ref = m2;
     return;
}
```

Пример 1

Метода има два локални указателя в стека - `m1` и `m2`. Също така има една променлива извън обсега на метода наречена `global`. `m1` и `m2` са два

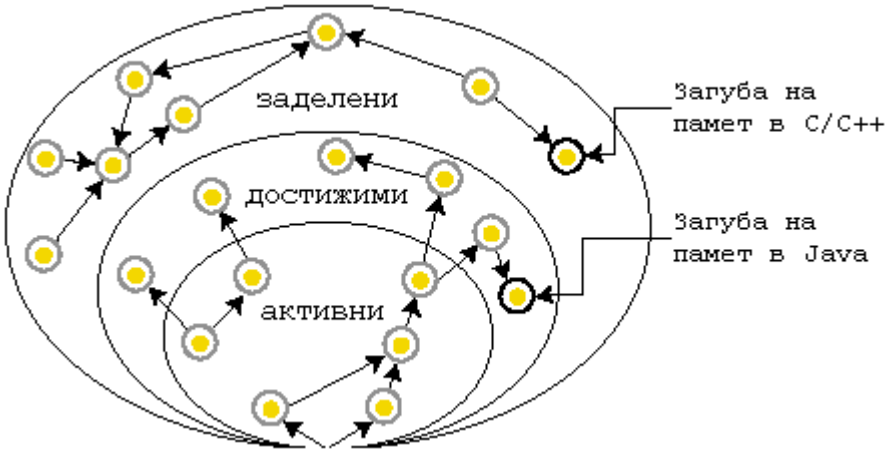
временни корена за garbage collector. Два обекта са създадени и два указателя или ребра са свързани с тези обекти (от локалните променливи в стека). Друг указател е добавен от `m1` към `m2` и още един указател от `global`. Когато се излезе от метода, `m1` и `m2` вече не са във стека, така че първия обект вече е недостижим. Тъй като garbage collector не може да достигне до този обект от някой път, по някое време ще бъде изчистен.



Фигура 1

Съществуват някои митове за garbage collector в Java. Един от тях е, че не може да се справи със циклите. Това не е така. Ако имаме три обекта - A, B и C, с указатели от A към B, от B към C, и от C към A, и това са единствените указатели към тези обекти, garbage collector ще ги изчисти. За разлика от някой други системи, който използват броячи на указателите (като COM на Microsoft) и които имат проблеми с циклите в графа от указатели на обектите.

След като обсъдихме как работи garbage collector, нека да видим какво означава да има загуба на памет в Java.



Фигура 2

От фигурата се вижда, че един обект може да бъде в три състояния:

- Заделени обекти са всички обекти, които са създадени, но не са премахнати от garbage collector;
- Достижими обекти са всички заделени обекти, които могат да бъдат достигнати от някой от корените;
- Активни обекти са достижими обекти, който се използват активно от програмата;

Garbage collector се грижи за обектите, които са заделени, но недостижими. За сравнение, тези обекти са загуба на памет в C. Инструменти като Rational's Purify и Numega's BoundsChecker са създадени да откриват такива проблеми в C.

В Java ситуацията е различна. Загуба на памет се получава вместо това от обекти, които са достижими, но неактивни. Въпреки, че има указател към обекта и има път от някой корен, обекта не е нужен на програмата и може да бъде изчистен, ако нямаше указател към него.

Така че за разлика от Java, в C когато даден обект е загубен, проблема не може да бъде оправен от програмата - тъй като вече няма указател към този обект. В Java обекта може да бъде достигнат, но кода, който управлява обекта може да не бъде достъпен. Например, указателя към ненужния обект може да бъде от `private` поле в класа, на който няма

изходен код. От друга страна, ако указателя е достъпен, е възможно действие, което програмата да предприеме, за да го премахне като по този начин направи обекта недостижим и приемлив за изчистване.

Друга разлика, ако се върнем към аналогията с представянето на купа на паметта като ориентиран граф от обекти и указатели е, че в C трябва да се грижим и за възлите и за ребрата. Ако оставим някое ребро да виси, като освободим обекта без да премахнем указателите, се получава висящ указател, което обикновено предизвиква небезизвестния в Windows - General Protection Fault. Обратно, ако се остави възела да виси, като се премахнат указателите, без да се освободи обекта, се получава загуба на памет. В Java може да се получи само второто, като се премахнат ребрата. Всъщност, има само контрол върху указателите, затова трябва да се управляват само ребрата. Ако не се премахнат указателите към обектите, garbage collector не може да ги изчисти. Трябва да се подпомогне като се управляват ребрата.

По принцип загубата на памет в Java е по-рядка в сравнение със C. В C е по-лесно да се получи като не се напише деструктор на класа или не се безпокоим да освободим паметта. В Java garbage collector върши доста от тази работа вместо програмиста. Неприятната част е, че въздействието върху загубата на памет, т.е. размера на тази загубена памет има склонността да бъде много по-голяма в Java.

Причината за това е, че когато има един обект, който не се използва вече, рядко това е единичен обект. Този обект ще има указатели към други обекти, които ще имат още указатели и така нататък, образуващи голям подграф от ненужни обекти, само защото един указател не е изчистен. Например контейнерите от AWT или Swing (панели или фреймове) включват други компоненти деца (бутони, текстови полета). Контейнерът може да достигне всички свои деца, защото има указатели към тях (за да може да ги разположи). В същото време всеки компонент има указател обратно към своя родител. По този начин има път от всеки обект към всеки друг. За да усложни проблема, UI обектите са често наследявани,

добавяйки допълнителни указатели и обекти в подграфа. Резултата е, че загубата на памет е не малка група компоненти, а доста голямо множество обекти.

След като има толкова много разлики при загубата на памет в C и Java е объркващо да се използва един и същ термин и за двете. Затова неизползваните обекти в Java се наричат "loiterers" - шляещи се, мотаещи се. Речниковото значение на loiterer е "забавящ дадена дейност с безсмислени и безполезни спирания и паузи" (което се случва когато garbage collector трябва да проверява все повече и повече обекти всеки път) и "оставащ в даден район без видима причина" (те не се използват, така че мястото им не и там). Друга причина за използване на различното название за този термин е, че Java виртуалната машина и много от библиотеките имат машинно зависим код написан на C и в този код може да има загуба на памет, водещо до объркване дали тази загуба е в Java или в C кода.

III.2. Речник на loiterers

Според Ethan Henry и Ed Lycklama (Ethan Henry and Ed Lycklama, 2000, How Do You Plug Java Memory Leaks?, Dr. Dobb's Journal February 2000), съществуват четири различни типа "шляещи се обекти":

a) Lapsed Listeners

Lapsed Listener - изостанал, пропуснат, невалиден слушател. Това е обект добавен към колекция, но никога не се премахва. Най-честия пример е слушател на събития, където обекта е добавен към списъка със слушатели, но не е премахнат, когато няма нужда вече от него. Въпреки, че той все още получава събития, вече не извършва полезна дейност. Допълнителен страничен ефект е, че колекцията от слушатели може да расте неограничено. Програмата се забавя тъй като събитията трябва да се препращат към все повече и повече слушатели.

Това е може би най-баналния проблем с паметта в Java - Swing и AWT са доста податливи към него, както и големите приложения. Например бъг #4177795 от Java Developer's Connection (*http://developer.java.sun.com/developer/bugParade/index.html*). В този случай, инстанциите на класа `javax.swing.JInternalFrame` стават loiterers, ако към тях се добави лента с меню. През сложна поредица от събития, се оказва, че хеш-таблицата, която проследява натисканията на клавишите регистрирани за бърз достъп, държи указател към менюто, което е свързано с вътрешния фрейм, попречвайки тези обекти да се изчистят, дори след като всички указатели от програмата се премахнат. Изненадващо лесно е да се получат такива проблеми.

Начин да се избегне тази грешка, е да се проверят дали всички добавяния и премахвания са по двойки. Хубава практика е двата метода да са близо в кода. Да се обръща внимание на жизнения цикъл - създаването на връзка от дълготраен към краткотраен обект, дава и на двата живот колкото на дълготрайния.

b) Lingerers

Lingerer - заседяващ се, маещ се. Обект, който застоява, след като програмата е свършила с използването му. Получава се обикновено, когато указател от дълготраен обект се използва за указване на временен обект и не се изчиства след това. Следващия път когато този указател се използва, ще бъде преустановен да сочи към друг обект, но междувременно, предишния остава да блуждае.

```
public class PrintService {
  static PrintService singleton;
  Printable target;

  public static PrintService getPrintService() {
    return singleton;
```

```
  }

  public void setTarget(Printable p){
    target = p;
  }

  public void doPrint() {
    // настройка
    // отпечатване на target
  }
}
```

Пример 2

Например в една типична услуга за отпечатване има поле `target`. Когато се извика `doPrint()`, се разпечатва обекта сочен от `target`. Но когато печатането свърши, указателя `target` не се нулира. Обекта, който се е отпечатал не може да бъде изчистен, защото има заседяващ се указател към него от услугата за отпечатване. Трябва да се нулират преходните указатели, след като се използват.

Една стратегия за справяне с lingerer е да се обособи управлението на състоянието в един обект вместо в няколко. Това прави промените на състоянието по-лесни и разбираеми. Друга стратегия е да се избягва преждевременното излизане от методите - трябва да се определят методи за инициализиране, обработване и изчистване. Ако методът излезе преди да има възможност да почисти - могат да останат указатели, който да държат обекти, който не са вече необходими.

c) Laggards

Laggard - тромав, несръчен, изостанал. Laggard се получава, когато обект променя състоянието си, но все още има указатели към данни от предишното си състояние. Това са обикновено функционални грешки, но се

откриват трудно, като отначало се проявяват като проблеми с паметта. Един от начините да се получат, е когато се промени жизнения цикъл на клас, например от множествен към единичен. Сега единичния клас променя състоянието си от време на време, обратно на създаването на нова инстанция, когато се изисква ново състояние.

То може да е например обект, който поддържа информация за файловете в директория, включваща статистика за най-големия, най-малкия и най-сложния файл (при някаква дефиниция за "сложен"). Когато се смени директорията, по някаква причина само указателите за най-големия и най-малкия файл се променят - указателя за най-сложния файл е изостанал, тъй като сочи все още към файл от предишната директория. Това разбира се е бъг, но неуловим. Ако се използва дебъгер за паметта, може да се види, че има повече файлови обекти отколкото файлове в директорията, заради допълнителния файл задържан от изостаналия указател.

Можем да се справим с laggards, ако обмислим внимателното стратегиите за кеширане. Необходимо ли е наистина кеширане или и допустимо да се изчисляват някои стойности динамично. Полезно е да се използва профайлър, за да се определи дали кеширането е подходящо. Друга възможност е да се капсулира прехода на състоянието в един метод.

d) Limbo

Limbo - танц от Източна Индия, при който танцьорите минават под пръчка, която се сваля все повече и повече. Обектите в лимбо състояние може да не са дълготрайни, но могат да заемат много памет, точно когато ни трябва. Лимбо се получава, когато обект от стека е обвързан от метод, която работи продължително. Проблема е, че garbage collector не може да направи "анализ на актуалността", за да разбере, че дадения обект не се използва до края на метода и трябва да се изчисти.

```
void method(File file) {
```

```
    // създава големия обект
    Big big = readIt();
    // this condenses it
    Small small = parseIt(big);
    // този ред е необходим, за да се освободи big преди
       края на метода
    // big = null;
    // този метод ще се изпълнява дълго време
    processIt(small);
  }
```

Пример 3

В примера метода прочита файл, прави разбор и обработва определени елементи. Това може да се случи например при работа с данни от XML файл. Първо се извиква метода `readIt()`, който прочита файла, което заема доста памет. После метода `parseIt()` минава и прави разбор като сгъстява информацията от големия обект в по-малък. От този момент няма повече нужда от големия обект и е хубаво да може да се използва паметта, която той използва. Но когато се извика `processIt()`, което може да отнеме много време, паметта от големия обект не може да се освободи, защото има указател от стека на метода, който е валиден до края на метода. Трябва да се подпомогне garbage collector като се нулира указателя към големия обект, както е показано в реда, който е коментиран.

Един начин за справяне с проблема е да се държи сметка за продължителни методи, където се заделя много памет, за да сме сигурни, че големите обекти не се държат чрез указатели от стека. Инструменти като профайлъри и дебъгери за памет могат да се използват, за да се открият такива методи и обекти. Изричното нулиране на указателите към огромни обекти, който са вече ненужни, може да доведе до доста положителна разлика. Блокирана нишка може също да бъде проблем, например когато

чака за входно-изходна операция.

IV. Java Virtual Machine Profiler Interface (JVMPI)

IV.1. Защо беше избран интерфейса JVMPI?

Използването на интерфейса JVMPI беше предопределено от следните му преимущества:

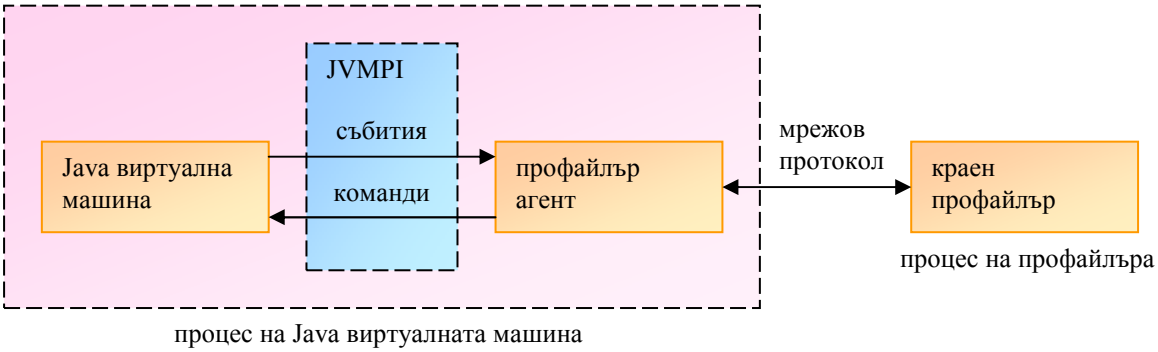
- изчерпателен - профайлърите базирани върху JVMPI могат да представят профили за използваното процесорно време, откриват мащабно заделяне на памет, намират ненужно задържане на обекти, посочват проблеми свързани с надпревара за монитори, разкриват *deadlocks* на нишките. Интерфейса JVMPI позволява на профайлърите да извличат тези типове информация чрез предоставянето на механизъм за проследяване на ключов набор от важни събития от работата на Java ВМ, да им назначават стойности и да приписват тези стойности на специфични контексти от изпълнението.
- универсален - профайлърите няма нужда да разчитат на специфична инструментация във виртуалната машина. Интерфейса JVMPI е резултатен и достатъчно мощен да отговори на нуждите на различни имплементации на профайлъри и виртуални машини. Поддържа разнообразие от техники на профилиране. Например инструментиране на кода и статистическо семплиране. Също така поддържа интерактивно профилиране с минимално натоварване. Профайлърите могат да взаимодействат с потребителя и изпълняват селективно профилиране или просто да записват резултатите във файл.
- преносим - интерфейса е проектиран да бъде абсолютно независим от Java виртуалната машина. Например, профилирането на купа на

паметта не зависи от алгоритмите за заделяне на памет и garbage collection използвани от виртуалната машина.

- ненатоварващ - когато профилирането е изключено, ВМ се натоварва само с тест и преход за всяко събитие проследявано от интерфейса JVMPI. Повечето събития се получават на места, където е допустима допълнителна проверка. Като резултат, ВМ може да се доставя с вградена поддръжка за профилиране. Измерванията в производителността могат да се извършват с минимално несъответствие между профилиращата и действителната работна среда.
- разширяем - JVMPI може да бъде лесно разширен, за да върви в крак с еволюцията на Java виртуалната машина. Могат да бъдат добавени нови събития и команди.

IV.2. Въведение

JVMPI е двупосочен интерфейса между Java виртуалната машина и профайлър агент като вътрешен процес. От една страна, виртуалната машина уведомява профайлър агента за разнообразни събития. Например, разпределението на купа на паметта, зареждане на клас, стартиране на нишка и т.н. От друга страна, профайлър агента осъществява контрол и изисква повече информация. Например, профайлър агента може да включва и изключва известяването за определени събития, базирано на нуждите на крайния профайлър. Пълното описание на интерфейса може да се намери на адрес *http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html*.



Фигура 3 - Архитектура на профайлър

Крайния профайлър може или може да не е в същия процес с профайлър агента. Той може да се намира в друг процес на същата машина или на отдалечена машина свързана чрез мрежа. Интерфейса JVMPI не специфицира стандартен мрежов протокол.

Инструмент за профилиране базиран на интерфейса JVMPI може да получава разнообразна информация като изобилно заделяне на памет, горещи места при използване на процесора, ненужно задържане на обекти, надпревара за монитори, за задълбочен анализ на производителността.

Интерфейса JVMPI поддържа частично профилиране, т.е. потребителя може да профилира избирателно за определен период от време и също така може да избере само определен тип профилираща информация.

В текущата версия на интерфейса JVMPI се поддържа само по един агент на виртуална машина.

а) Стартиране

Потребителя може специфицира името на профайлър агента чрез аргумент от командния ред към Java виртуалната машина. Например потребителя специфицира:

```
java -Xrun<име_на_профайлъра>:<опции> КласЗаПрофилиране
```

ВМ се опитва да открие библиотеката на профайлър агента наречена *име_на_профайлъра* в библиотечната директорията на Java:

- на Win32, това е [стойността на свойството `java.home`] `\bin\име_на_профайлъра.dll`
- на SPARC/Solaris, това е [стойността на свойството `java.home`] `/lib/sparc/име_на_профайлъра.so`

Ако библиотеката не е намерена там, ВМ продължава да я търси следвайки нормалния механизъм за търсене на динамична библиотека на дадената платформа:

- на Win32, ВМ претърсва текущата директория, системните директории на Windows и директориите от променливата на обкръжението `PATH`.
- на Solaris, ВМ претърсва директориите от `LD_LIBRARY_PATH`.

ВМ зарежда библиотеката на профайлър агента и търси началната точка:

```
jint JNICALL JVM_OnLoad(JavaVM *jvm, char *options,
void *reserved);
```

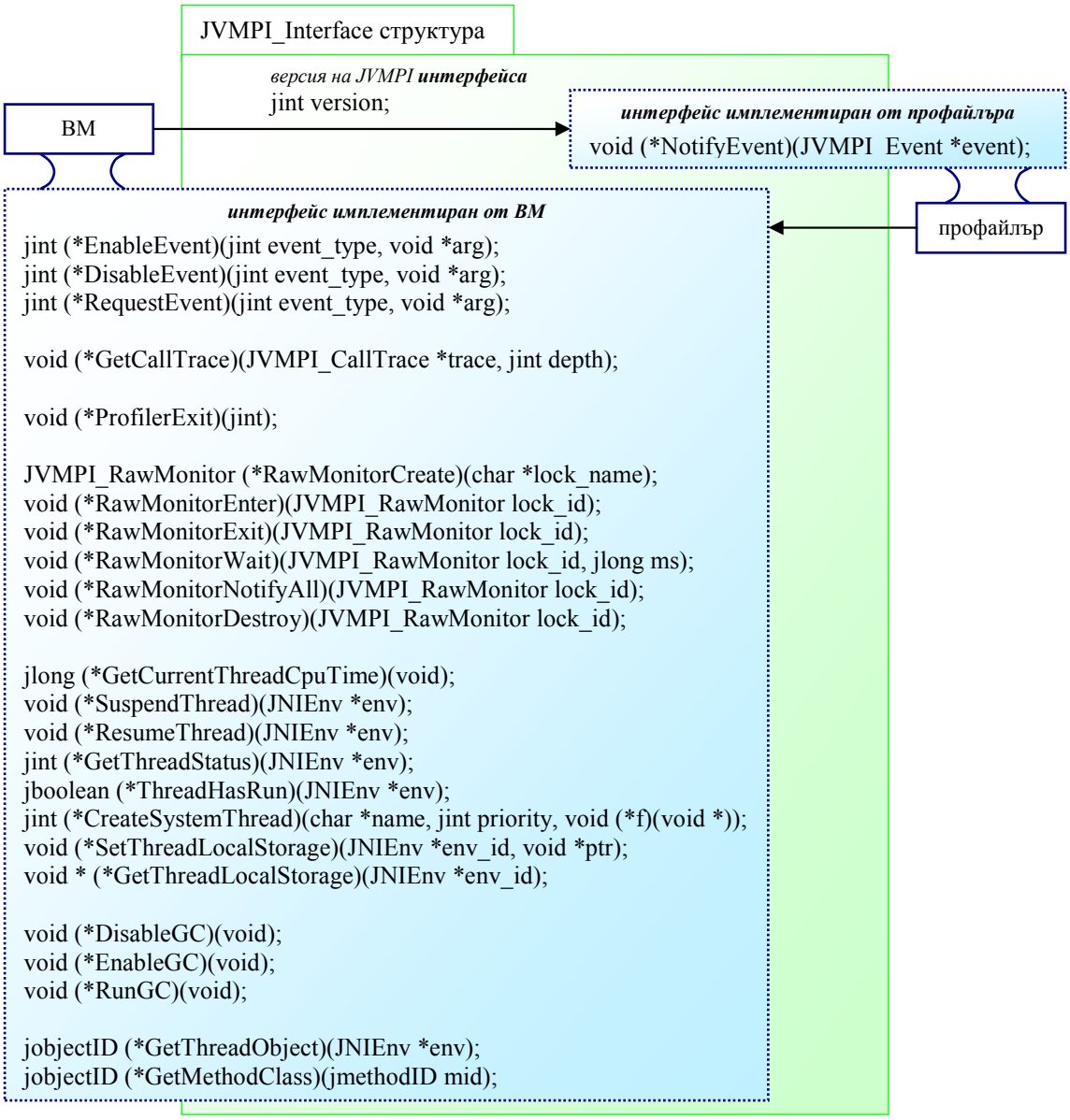
ВМ извиква функцията `JVM_OnLoad`, като и предава указател към инстанцията на `JavaVM` като първи аргумент и стринга `опции` като втори. Третия аргумент е резервиран (NULL).

При успех функцията `JVM_OnLoad` трябва да върне `JNI_OK`. Ако по някаква причина функцията `JVM_OnLoad` пропадне, тя трябва да върне `JNI_ERR`.

b) Интерфейс на функциите

Профайлър агента може да получи интерфейс към функциите чрез извикване на `GetEnv`.

Структурата `JVMPI_Interface` дефинира интерфейса към функциите между профайлър агента и ВМ:



Фигура 4 - Диаграма на структурата JVMPI_Interface

Функцията `GetEnv` връща указател към `JVMPI_Interface` чието поле `version` показва версия на JVMPI съвместима с номера на версията подадена като аргумент. Не е задължително двете версии да са еднакви.

В структурата `JVMPI_Interface` върната от `GetEnv` всички функции са установени освен `NotifyEvent`. Профайлър агента трябва да я установи преди да се върне от `JVM_OnLoad`.

c) Известяване за събития

ВМ изпраща събития чрез извикване на `NotifyEvent` със структура `JVMPI_Event` като аргумент. Поддържат се следните събития:

- влизане и излизане от метод
- заделяне, преместване и освобождаване на обект
- стартиране и свършване на GC
- начало и край на нишка
- зареждане и отзареждане на клас
- зареждане и отзареждане на компилиран метод
- влизане, излизане, чакане за монитор
- съдържание на мониторите
- съдържание на купа на паметта
- съдържание на обект

Структура `JVMPI_Event` съдържа типа на събитието, указател към `JNIEnv` на текущата нишка и друга специфична за събитието информация. Специфичната за събитието информация е представена като обединение на структурите на събитията.

d) Идентификатори в интерфейса JVMPI

Интерфейса JVMPI разграничава единиците от ВМ чрез различни видове идентификатори. Нишките, класовете, методите, обектите имат уникални идентификатори.

Всеки идентификатор има дефиниращо и заличаващо събитие. Дефиниращото събитие предоставя информация свързана с идентификатора. Например, дефиниращото събитие за идентификатор на нишка съдържа и името на нишката.

Идентификатора е валиден докато пристигне заличаващото събитие. То прави невалиден идентификатора, чиято стойност може де се използва за друга единица.

В Таблица 1 са дадени дефиниращите и заличаващите събития за различните типове идентификатори.

Ако дефиниращите събития са включени по време на инициализацията, се гарантира, че профайлър агента ще бъде уведомен за създаването на единицата чрез дефиниращо събитие, преди тази единица да се появи в други събития.

Ако дефиниращите събития са изключени, профайлър агента може да получи неизвестен идентификатор. В този случай профайлър агента може да изиска съответното събитие да бъде изпратено чрез функцията `RequestEvent`.

е) Комуникация между профайлър агента и крайния профайлър

Интерфейса JVMPI предлага механизъм от ниско ниво за комуникация между профайлър агента и ВМ. Целта е да се предложи максимална гъвкавост за профайлър агента да представи данните, който са необходими на крайния профайлър и също така да запази минимална работата извършвана от ВМ. Затова интерфейса JVMPI не специфицира мрежов протокол между профайлър агента и крайния профайлър. Вместо това производителите проектират техния собствен профайлър агент да отговаря на нуждите на техния краен профайлър.

Следните особености трябва да се имат предвид когато се проектира мрежовия протокол, за да позволи профайлър агент и крайния профайлър да се намират на различни машини.

- размер на указателите (т.е. 32 или 64 битови) - всички идентификатори от интерфейса JVMPI са от тип указател
- порядък на байтовете (little endian или big endian)
- порядък на битовете (най-старшия бит първи или най-младшия бит първи)
- кодиране на стринговете - интерфейса JVMPI използва UTF-8 кодиране документирано в спецификацията на ВМ

IV.3. Функции от интерфейса

a) CreateSystemThread

```
jint (*CreateSystemThread)(char *name, jint priority,
     void (*f)(void *));
```

Създава демон нишка във ВМ.

Аргументи:

`name` - име на нишката

`priority` - приоритет на нишката; стойностите могат да бъдат:

JVMPI_NORMAL_PRIORITY

JVMPI_MAXIMUM_PRIORITY

JVMPI_MINIMUM_PRIORITY

`f` - функция, която да се стартира от нишката

Резултат:

JNI_OK - успех

JNI_ERR - грешка

b) DisableEvent

```
jint (*DisableEvent)(jint event_type, void *arg);
```

Изключва уведомяването за определени събития. Освен типа на събитието могат да се подадат допълнителни аргументи специфични за дадения тип. Всички събития са изключени когато се стартира ВМ.

Функцията връща JVMPI_NOT_AVAILABLE, ако типа на събитието е JVMPI_EVENT_HEAP_DUMP, JVMPI_EVENT_MONITOR_DUMP или JVMPI_EVENT_OBJECT_DUMP.

Аргументи:

`event_type` - тип на събитието, JVMPI_EVENT_CLASS_LOAD т.н.

`arg` - специфична за дадения тип информация

Резултат:

JVMPI_SUCCESS - успешно изключване

JVMPI_FAIL - неуспешно изключване

JVMPI_NOT_AVAILABLE - не е възможно изключване на съответното събитие

c) DisableGC

```
void (*DisableGC)(void);
```

Изключва garbage collection, докато не се извика `EnabledGC`.

d) EnableEvent

```
jint (*EnableEvent)(jint event_type, void *arg);
```

Включва уведомяването за определени събития. Освен типа на събитието може да се подадат допълнителни аргументи специфични за дадения тип. Всички събития са изключени когато се стартира ВМ.

Функцията връща JVMPI_NOT_AVAILABLE, ако типа на събитието е JVMPI_EVENT_HEAP_DUMP, JVMPI_EVENT_MONITOR_DUMP или JVMPI_EVENT_OBJECT_DUMP.

Аргументи:

`event_type` - тип на събитието, JVMPI_EVENT_CLASS_LOAD т.н.

`arg` - специфична за дадения тип информация

Резултат:

JVMPI_SUCCESS - успешно включване

JVMPI_FAIL - неуспешно включване

JVMPI_NOT_AVAILABLE - не е възможно включване на съответното събитие

e) EnableGC

```
void (*EnableGC)(void);
```

Включва garbage collection.

f) GetCallTrace

```
void (*GetCallTrace)(JVMPI_CallTrace *trace, jint
     depth);
```

Получава текущия стек на извиквания на методи за дадената нишка. Нишката се определя от полето `env_id` в структурата `JVMPI_CallTrace`. В структурата трябва да има достатъчно памет за желаната дълбочина на стека. ВМ попълва буфера за фреймове и полето `num_frames`.

Аргументи:

`trace` - структурата, която трябва да бъде попълнена от ВМ

`depth` - дълбочина на стека на извиквания

g) GetCurrentThreadCpuTime

```
jlong (*GetCurrentThreadCpuTime)(void);
```

Получава акумулираното процесорно време консумирано от текущата нишка.

Резултат:

време в наносекунди

h) GetMethodClass

```
jobjectID (*GetMethodClass)(jmethodID mid);
```

Получава идентификатора на класа, който дефинира този метод.

Аргументи:

`mid` - идентификатор на метод

Резултат:

идентификатор на дефиниращия клас

i) GetThreadLocalStorage

```
void * (*GetThreadLocalStorage)(JNIEnv *env_id);
```

Получава стойността на локалния склад на нишката. Интерфейса JVMPI поддържа за агента локалния склад, който може да бъде използван за записване на профилираща информация за всяка отделна нишка.

Аргументи:

`env_id` - указател `JNIEnv` на нишката

Резултат:

стойността на локалния склад на нишката

j) GetThreadObject

```
jobjectID (*GetThreadObject)(JNIEnv *env);
```

Получава идентификатор на нишката от съответния указател `JNIEnv`.

Аргументи:

`env` - указател `JNIEnv` на нишката

Резултат:

идентификатор на нишката

k) GetThreadStatus

```
jint (*GetThreadStatus)(JNIEnv *env);
```

Получава статуса на нишка.

Функциите `SuspendThread` и `ResumeThread` нямат никакъв ефект върху статуса върнат от `GetThreadStatus`. Статуса на нишка спряна чрез JVMPI остава непроменен и се връща статуса преди спирането.

Аргументи:

`env` - указател `JNIEnv` на нишката

Резултат:

JVMPI_THREAD_RUNNABLE - нишката работи

JVMPI_THREAD_MONITOR_WAIT - нишката чака за монитор

JVMPI_THREAD_CONDVAR_WAIT - нишката чака на условна променлива

Ако нишката е спряна (от `java.lang.Thread.suspend`) или прекъсната в някой от горните състояния битовете JVMPI_THREAD_SUSPENDED или JVMPI_THREAD_INTERRUPTED са установени.

l) NotifyEvent

```
void (*NotifyEvent)(JVMPI_Event *event);
```

Извикан от ВМ, за да изпрати събитие на профайлър агента. Профайлър агента регистрира типовете събития, който го интересуват чрез `EnableEvent`, или изисква специфични събития чрез `RequestEvent`.

Когато събитието е включено от `EnableEvent`, нишката където се генерира е тази, която изпраща събитието. Когато събитието е изискана от `RequestEvent`, нишката където е поискано е тази, която изпраща събитието. Множество нишки могат да изпращат множество събития едновременно.

Ако специфичната за събитието информация съдържа `jobjectID`, тази функция се извиква с изключен GC. GC се включва когато функцията приключи.

Паметта заделена за структурата `JVMPI_Event` и друга специфична за събитието информация се освобождава от ВМ щом свърши функцията. Профайлър агента трябва да копира всяка информация, от която има нужда.

Аргументи:

`event` - JVMPI събитието изпратено от ВМ на профайлър агента

m) ProfilerExit

```
void (*ProfilerExit)(jint err_code);
```

Информира ВМ, че профайлъра иска да излезе с код на излизане `err_code`. Тази функция предизвиква ВМ да излезе с този код.

Аргументи:

`err_code` - код на излизане

n) RawMonitorCreate

```
JVMPI_RawMonitor (*RawMonitorCreate)(char *lock_name);
```

Създава суров монитор. Суровите монитори са подобни на Java мониторите. Разликата е, че суровите не са асоциирани с Java обекти. Профайлър агента не трябва да вика тази функция в спряна нишка, защото функцията може да извика системни функции като `malloc` и да блокира върху lock на системната библиотека. Ако името на суровия монитор започва със знак за подчертаване ('_'), тогава събитията за надпревара свързани с него не се изпращат към профайлър агента.

Аргументи:

`lock_name` - име на суровия монитор

Резултат:

суров монитор

o) RawMonitorDestroy

```
void (*RawMonitorDestroy)(JVMPI_RawMonitor lock_id);
```

Унищожава суров монитор и освобождава всички системни ресурси свързани с монитора.

Аргументи:

`lock_id` - идентификатор на суровия монитор, който трябва да бъде унищожен

p) RawMonitorEnter

```
void (*RawMonitorEnter)(JVMPI_RawMonitor lock_id);
```

Влиза в суров монитор.

Аргументи:

`lock_id` - идентификатор на суровия монитор, в който се влиза

q) RawMonitorExit

```
void (*RawMonitorExit)(JVMPI_RawMonitor lock_id);
```

Излиза от суров монитор.

Аргументи:

`lock_id` - идентификатор на суровия монитор, от който се излиза

r) RawMonitorNotifyAll

```
void (*RawMonitorNotifyAll)(JVMPI_RawMonitor lock_id);
```

Уведомява всички нишки, който чакат за този монитор.

Аргументи:

`lock_id` - идентификатор на суровия монитор, за който се уведомява

s) RawMonitorWait

```
void (*RawMonitorWait)(JVMPI_RawMonitor lock_id, jlong
     ms);
```

Чака върху даден суров монитор определен период от време. Ако периода е 0 нишката чака вечно.

Аргументи:

`lock_id` - идентификатор на суровия монитор, за който се чака

`ms` - време за изчакване (в милисекунди)

t) RequestEvent

```
jint (*RequestEvent)(jint event_type, void *arg);
```

Заявява определен тип събитие. Освен типа на събитието могат да се подадат допълнителни аргументи специфични за дадения тип.

Тази функция може да се извика, за да се заявят еднократни събития като JVMPI_EVENT_HEAP_DUMP, JVMPI_EVENT_MONITOR_DUMP и JVMPI_EVENT_OBJECT_DUMP. Известяването за тези събития не се контролира чрез функциите `EnableEvent` и `DisableEvent`.

В допълнение, тази функция може да се извика, за да се изискат дефиниращите събития за специфичен клас, нишка или обект. Това е полезно, когато профайлър агента трябва да разчете идентификатор на непознат клас, метод, нишка или обект, получен в събитие, чието дефиниращо събитие е било изключено преди това.

Профайлър агента може да получи информация за неизвестен идентификатор на клас чрез заявка на събитието JVMPI_EVENT_CLASS_LOAD като подаде като допълнителен аргумент идентификатора на класа.

Профайлър агента може да получи информация за неизвестен идентификатор на нишка чрез заявка на събитието JVMPI_EVENT_THREAD_START като подаде като допълнителен аргумент идентификатора на нишката.

Профайлър агента може да получи информация за неизвестен идентификатор на обект чрез заявка на събитието JVMPI_EVENT_OBJECT_ALLOC като подаде като допълнителен аргумент идентификатора на обекта.

Следователно профайлър агента може или да позволи горните три събития асинхронно чрез функцията `EnableEvent`, или да изисква тези събития синхронно чрез функцията `RequestEvent`. Заявеното събитие се изпраща в същата нишка, която е извикала `RequestEvent`, и се изпраща преди да свърши функцията.

Функцията `RequestEvent` не може да се използва, за да се заявяват други събития, който не са изброени по-горе.

Събитията изискани чрез `RequestEvent`, пристигат с включен бит JVMPI_REQUESTED_EVENT в техния `event_type`.

Аргументи:

`event_type` - тип на събитието, JVMPI_EVENT_CLASS_LOAD т.н.

`arg` - специфична за дадения тип информация.

Резултат:

JVMPI_SUCCESS - успешна заявка

JVMPI_FAIL - неуспешна заявка

JVMPI_NOT_AVAILABLE - не е възможна заявка на съответното събитие

u) ResumeThread

```
void (*ResumeThread)(JNIEnv *env);
```

Възобновява нишка.

Нишка поставена в режим на изчакване чрез метода `java.lang.Thread.suspend` не може да бъде възобновена чрез функцията от JVMPI `ResumeThread`.

Аргументи:

`env` - указател `JNIEnv` на нишката

v) RunGC

```
void (*RunGC)(void);
```

Изисква насилствено пълен garbage collection. Тази функция не трябва да се извиква, когато GC е изключен.

w) SetThreadLocalStorage

```
void (*SetThreadLocalStorage)(JNIEnv *env_id, void
     *ptr);
```

Вкарва стойност в локалния склад на нишката. Интерфейса JVMPI поддържа за агента локалния склад, който може да бъде използван за записване на профилираща информация за всяка отделна нишка.

Аргументи:

`env` - указател `JNIEnv` на нишката

`ptr` - стойността, която ще бъде вкарана в локалния склад на нишката

x) SuspendThread

```
void (*SuspendThread)(JNIEnv *env);
```

Поставя нишка в режим на изчакване. Системата влиза в режим на изчакване на нишката, след като функцията се извика.

Нишка поставена в режим на изчакване чрез функцията от JVMPI `SuspendThread` не може да бъде възобновена чрез метода `java.lang.Thread.resume`.

В имплементацията на 1.2, тази функция трябва да се извиква, когато GC е изключен. GC трябва да остане изключен докато всички нишки се възобновят.

Аргументи:

`env` - указател `JNIEnv` на нишката

у) ThreadHasRun

```
jboolean (*ThreadHasRun)(JNIEnv *env);
```

Установява дали нишката идентифицирана с подадения указател `JNIEnv` е консумирала процесорно време от последния път, когато нишката е била поставяна в режим на изчакване от `SuspendThread`. Тази функция трябва да се извиква, когато нишката е била възобновена от `ResumeThread` и после поставена в режим на изчакване от `SuspendThread`.

Аргументи:

`env` - указател `JNIEnv` на нишката

Резултат:

JNI_TRUE - нишката е имала възможност да работи

JNI_FALSE - нишката не е имала възможност да работи

IV.4. Събития от интерфейса

а) JVMPI_EVENT_CLASS_LOAD

Изпраща се, когато класа е зареден във ВМ, или когато профайлър агента изиска събитието чрез извикване на `RequestEvent`. Във втория случай, бита JVMPI_REQUESTED_EVENT е установен. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  char *class_name;
  char *source_name;
```

```
    jint num_interfaces;
    jint num_methods;
    JVMPI_Method *methods;
    jint num_static_fields;
    JVMPI_Field *statics;
    jint num_instance_fields;
    JVMPI_Field *instances;
    jobjectID class_id;
} class_load;
```

Съдържание:

`class_name` - име на заредения клас

`source_name` - име на изходния файл, където е дефиниран класа

`num_interfaces` - броя на интерфейсите имплементирани от класа

`methods` - методите дефинирани в класа

`num_static_fields` - броя на статичните полета дефинирани в класа

`statics` - статичните полета дефинирани в класа

`num_instance_fields` - броя на полетата на инстанция дефинирани в класа

`instances` - полетата на инстанция дефинирани в класа

`class_id` - идентификатор на клас

Забележка:

идентификаторите на класове са идентификатори на обектите на класовете и подлежат на промяна, когато пристигне JVMPI_EVENT_OBJECT_MOVE.

b) JVMPI_EVENT_CLASS_UNLOAD

Изпраща се, когато класа се отзарежда.

```
struct {
  jobjectID class_id;
} class_unload;
```

Съдържание:

`class_id` - идентификатор на клас

c) JVMPI_EVENT_GC_FINISH

Изпраща се, когато завърши GC. Профайлър агента може да освободи всички locks, взети по време на уведомяването за стартиране на GC за обслужване на събитията object free, object move и arena delete. Системата се връща обратно в многонишков режим след това събитие. Специфичната за събитието информация съдържа статистика за купа на паметта.

```
struct {
  jlong used_objects;
  jlong used_object_space;
  jlong total_object_space;
} gc_info;
```

Съдържание:

`used_objects` - броя на използваните обекти в купа на паметта

`used_object_space` - обем на пространството използвано от обектите (в байтове)

`total_object_space` - общ размер на пространството за обекти (в байтове)

d) JVMPI_EVENT_GC_START

Изпраща се, когато ще започне GC. Системата минава в режим на изчакване на нишките след това събитие. За да се избегне *deadlock*,

профайлър агента трябва да вземе всички locks който са необходими за обслужване на събитията object free, object move и arena delete в мениджъра на събития. Няма специфична за събитието информация.

e) JVMPI_EVENT_HEAP_DUMP

Изпраща се, при поискване чрез функцията `RequestEvent`. Профайлър агента може да специфицира нивото на изпратената информация като предаде структурата `JVMPI_HeapDumpArg` към `RequestEvent` като втори параметър, с полето `heap_dump_level` установено на желаното ниво.

Стойността на нивото може да бъде една от следните:

JVMPI_DUMP_LEVEL_0

JVMPI_DUMP_LEVEL_1

JVMPI_DUMP_LEVEL_2

Ако се предаде NULL, тогава нивото се установява на JVMPI_DUMP_LEVEL_2.

Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши. Специфичната за събитието информация съдържа изображение на всички активни обекти от купа на паметта на Java.

```
struct {
  int dump_level;
  char *begin;
  char *end;
  jint num_traces;
  JVMPI_CallTrace *traces;
} heap_dump;
```

Съдържание:

`dump_level` - нивото специфицирано в `RequestEvent`

`begin` - начало на съдържанието на купа на паметта

`end` - край на съдържанието на купа на паметта

`num_traces` - брой на стековите пътища на корените на GC, 0 за JVMPI_DUMP_LEVEL_0

`traces` - стековите пътища на корените на GC

Формата на съдържанието на купа на паметта между началото и края зависи от нивото на изисканата информация. Форматите са описани подробно в секцията JVMPI формати на съдържанията.

f) JVMPI_EVENT_JVM_INIT_DONE

Изпраща се от ВМ, когато свърши инициализацията. Сигурно е да се извиква `CreateSystemThread` само след уведомяването за това събитие. Няма специфична за събитието информация.

g) JVMPI_EVENT_JVM_SHUT_DOWN

Изпраща се от ВМ, когато се изключва. Типичния профайлър реагира като записва информацията за профилирането. Няма специфична за събитието информация.

h) JVMPI_EVENT_METHOD_ENTRY

Изпраща се, когато се влиза във метод. В сравнение с JVMPI_EVENT_METHOD_ENTRY2, това събитие не изпраща `jobjectID` на обекта-цел, от който се извиква метода.

```
struct {
  jmethodID method_id;
} method;
```

Съдържание:

`method_id` - метода, в който се влиза

i) JVMPI_EVENT_METHOD_ENTRY2

Изпраща се, когато се влиза във метод. Ако метода и метод на инстанция, `jobjectID` на обекта-цел се изпраща заедно със събитието. Ако метода е статичен, полето `obj_id` от събитието е NULL. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  jmethodID method_id;
  jobjectID obj_id;
} method_entry2;
```

Съдържание:

`method_id` - метода, в който се влиза

`obj_id` - обекта-цел, NULL за статичен метод

j) JVMPI_EVENT_METHOD_EXIT

Изпраща се, когато се излиза от метод. Излизането от метода може да бъде нормално или причинено от изключение.

```
struct {
  jmethodID method_id;
} method;
```

Съдържание:

`method_id` - метода, от който се излиза

k) JVMPI_EVENT_MONITOR_CONTENDED_ENTER

Изпраща се, когато нишка се опитва да влезе в Java монитор, който вече е взет от друга нишка. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  jobjectID object;
} monitor;
```

Съдържание:

`object` - идентификатор на обекта свързан с монитора

l) JVMPI_EVENT_MONITOR_CONTENDED_ENTERED

Изпраща се, когато нишка влезе в Java монитор, след като е чакала да бъде освободен от друга нишка. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  jobjectID object;
} monitor;
```

Съдържание:

`object` - идентификатор на обекта свързан с монитора

m) JVMPI_EVENT_MONITOR_CONTENDED_EXIT

Изпраща се, когато нишка излезе от Java монитор, и друга нишка чака да вземе същия монитор. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  jobjectID object;
} monitor;
```

Съдържание:

`object` - идентификатор на обекта свързан с монитора

n) JVMPI_EVENT_MONITOR_DUMP

Изпраща се, при поискване чрез функцията `RequestEvent`. Специфичната за събитието информация съдържа изображение на всички нишки и монитори от ВМ. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  char *begin;
  char *end;
  jint num_traces;
  JVMPI_CallTrace *traces;
  jint *threads_status;
} monitor_dump;
```

Съдържание:

`begin` - начало на буфера на съдържанието на монитор

`end` - край на буфера на съдържанието на монитор

`num_traces` - брой на стековите пътища на нишките

`traces` - стекови пътища на всички нишки

`thread_status` - статус на всички нишки

Формата на буфера на съдържанието на монитор е описан подробно в секцията JVMPI формати на съдържанията.

o) JVMPI_EVENT_MONITOR_WAIT

Изпраща се, когато нишка ще чака върху обект. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
```

```
  jobjectID object;
  jlong timeout;
} monitor_wait;
```

Съдържание:

`object` - идентификатор на обекта, върху който текущата нишка ще чака (NULL означава, че нишката е в `Thread.sleep`)

`timeout` - милисекунди, който ще чака нишката (0 означава, че ще чака вечно)

p) JVMPI_EVENT_MONITOR_WAITED

Изпраща се, когато нишка свърши чакането върху обект. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  jobjectID object;
  jlong timeout;
} monitor_wait;
```

Съдържание:

`object` - идентификатор на обекта, върху който текущата нишка е чакала (NULL означава, че нишката е в `Thread.sleep`)

`timeout` - милисекунди, който е чакала нишката

q) JVMPI_EVENT_OBJECT_ALLOC

Изпраща се, когато се задели обект, или когато профайлър агента изиска събитието чрез извикване на `RequestEvent`. Във втория случай, бита JVMPI_REQUESTED_EVENT е установен. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  jint arena_id;
  jobjectID class_id;
  jint is_array;
  jint size;
  jobjectID obj_id;
} obj_alloc;
```

Съдържание:

`arena_id` - арена, където е заделен

`class_id` - класа към който принадлежи този обект, или елементите на масива, ако `is_array` е JVMPI_CLASS

`is_array` - стойността може да бъде:

JVMPI_NORMAL_OBJECT обикновен обект

JVMPI_CLASS масив от обекти

JVMPI_BOOLEAN масив от booleans

JVMPI_BYTE масив от bytes

JVMPI_CHAR масив от chars

JVMPI_SHORT масив от shorts

JVMPI_INT масив от ints

JVMPI_LONG масив от longs

JVMPI_FLOAT масив от floats

JVMPI_DOUBLE масив от doubles

`size` - размер в байтове

`obj_id` - уникален идентификатор на обекта

r) JVMPI_EVENT_OBJECT_DUMP

Изпраща се, при поискване чрез функцията `RequestEvent`. `jobjectID` на обекта за който се изисква съдържание, трябва да бъде подаден като втори аргумент. Профайлър агента трябва да изисква това

събитие при изключен GC. Специфичната за събитието информация съдържа изображение на обекта.

```
struct {
  jint data_len;
  char *data;
} object_dump;
```

Съдържание:

`data_len` - дължина на буфера на съдържанието на обекта

`data` - начало на съдържанието на обекта

Формата на буфера на съдържанието на обект е описан подробно в секцията JVMPI формати на съдържанията.

s) JVMPI_EVENT_OBJECT_FREE

Изпраща се, когато се освободи обект. Това събитие се изпраща в режим на изчакване на нишките. Профайлъра не трябва да прави каквито и да са блокиращи извиквания като влизане в монитор или заделяне на памет от машинно зависимия куп на паметта (например чрез `malloc`). Това събитие винаги се изпраща между двойка събития JVMPI_EVENT_GC_START и JVMPI_EVENT_GC_FINISH. Профайлър агента трябва да вземе всички locks, необходими за обработването на това събитие в мениджъра на събития за JVMPI_EVENT_GC_START.

```
struct {
  jobjectID obj_id;
} obj_free;
```

Съдържание:

`obj_id` - обекта, който се освобождава

t) JVMPI_EVENT_OBJECT_MOVE

Изпраща се, когато се премести обект в купа на паметта. Това събитие се изпраща в режим на изчакване на нишките. Профайлъра не трябва да прави каквито и да са блокиращи извиквания като влизане в монитор или заделяне на памет от машинно зависимия куп на паметта (например чрез `malloc`). Това събитие винаги се изпраща между двойка събития JVMPI_EVENT_GC_START и JVMPI_EVENT_GC_FINISH. Профайлър агента трябва да вземе всички locks, необходими за обработването на това събитие в мениджъра на събития за JVMPI_EVENT_GC_START.

```
struct {
  jint arena_id;
  jobjectID obj_id;
  jint new_arena_id;
  jobjectID new_obj_id;
} obj_move;
```

Съдържание:

`arena_id` - текущата арена

`obj_id` - текущия идентификатор на обект

`new_arena_id` - нова арена

`new_obj_id` - нов идентификатор на обекта

u) JVMPI_EVENT_RAW_MONITOR_CONTENDED_ENTER

Изпраща се, когато нишка се опитва да влезе в суров монитор, който вече е взет от друга нишка.

```
struct {
  char *name;
```

```
  JVMPI_RawMonitor id;
} raw_monitor;
```

Съдържание:

`name` - име на суровия монитор

`object` - идентификатор на суровия монитор

v) JVMPI_EVENT_RAW_MONITOR_CONTENDED_ENTERED

Изпраща се, когато нишка влезе в суров монитор, след като е чакала да бъде освободен от друга нишка.

```
struct {
  char *name;
  JVMPI_RawMonitor id;
} raw_monitor;
```

Съдържание:

`name` - име на суровия монитор

`object` - идентификатор на суровия монитор

w) JVMPI_EVENT_RAW_MONITOR_CONTENDED_EXIT

Изпраща се, когато нишка излезе от суров монитор, и друга нишка чака да вземе същия монитор.

```
struct {
  char *name;
  JVMPI_RawMonitor id;
} raw_monitor;
```

Съдържание:

`name` - име на суровия монитор

`object` - идентификатор на суровия монитор

x) JVMPI_EVENT_THREAD_END

Изпраща се, когато приключи нишка във ВМ. Полето `env_id` от `JVMPI_Event` получен в това събитие е `JNIEnv` указател към интерфейса на нишката, която е приключила.

y) JVMPI_EVENT_THREAD_START

Изпраща се, когато стартира нишка във ВМ, или когато профайлър агента изиска събитието чрез извикване на `RequestEvent`. Във втория случай, бита JVMPI_REQUESTED_EVENT е установен. Това събитие се изпраща при изключен GC. GC се включва след като `NotifyEvent` завърши.

```
struct {
  char *thread_name;
  char *group_name;
  char *parent_name;
  jobjectID thread_id;
  JNIEnv *thread_env_id;
} thread_start;
```

Съдържание:

`thread_name` - име на нишката, която се стартира

`group_name` - група към, която принадлежи нишката

`parent_name` - име на родителя

`thread_id` - идентификатор на обекта на нишката

`thread_env_id` - указател `JNIEnv` на нишката

Нишките са свързани с указатели `JNIEnv` и идентификатор на обекта на нишката. Интерфейса JVMPI използва указателя `JNIEnv` като идентификатор на нишката.

IV.5. Формат на съдържанията в JVMPI

а) Формат на съдържанието на купа на паметта

Форматът на съдържанието на купа на паметта зависи от нивото на заявената информация.

JVMPI_DUMP_LEVEL_0:

Разпечатката се състои от поредица записи със следния формат:

`ty` - тип на обекта

`jobjectID` - обект

JVMPI_DUMP_LEVEL_1:

Формата на съдържанието е същия като този на JVMPI_DUMP_LEVEL_2, освен следните стойности, който са изключени от съдържанието: примитивните полета на инстанциите на обектите, примитивните статични полета на класовете и елементите на примитивните масиви.

JVMPI_DUMP_LEVEL_2:

Разпечатката се състои от поредица записи със следния формат, където всеки запис включва 8-битов тип на записа, последван от данни специфични за всеки тип запис.

Формата на съдържанието от ниво 2 е даден в Таблица 2.

b) Формат на съдържанието на обект

Буфера на съдържанието се състои от единичен запис, който включва 8-битов тип на записа, последван от данни специфични за всеки тип запис. Типа на записа може да бъде един от следните:

JVMPI_GC_CLASS_DUMP

JVMPI_GC_INSTANCE_DUMP

JVMPI_GC_OBJ_ARRAY_DUMP

JVMPI_GC_PRIM_ARRAY_DUMP

Формата на данните за всеки тип на запис е същия като описания по-горе в секцията с формата на съдържанието на купа на паметта. Нивото на информацията е същото като при JVMPI_DUMP_LEVEL_2, със включени следните стойности: примитивните полета на инстанциите на обектите, примитивните статични полета на класовете и елементите на примитивните масиви.

c) Формат на съдържанието на монитор

Буфера на съдържанието се състои от поредица записи, където всеки запис включва 8-битов тип на записа, последван от данни специфични за всеки тип запис.

Формата на съдържанието на монитор е даден в Таблица 3.

V. Профайлър агент

V.1. Въведение

Профайлър агента е динамично-свързана библиотека, която взаимодейства с JVMPI и изпраща профилираща информация или към файл или по мрежа в текстов или двоичен формат. Тази информация може да бъде използвана по-нататък от краен профайлър.

Има възможност да представи използваното процесорно време, статистика за използването на купа на паметта и надпреварата за монитори. В допълнение, може също така да отчете пълно съдържание на купа на паметта и състоянието на всички монитори и нишки в Java ВМ.

Профайлър агента може да бъде извикан чрез:

```
java -Xrunprof КласаЗаПрофилиране
```

В зависимост от вида на изисканото профилиране, профайлър агента инструктира виртуалната машина да му изпраща свързаните с това JVMPI събития и обработва данните от събитията в профилираща информация. Например, следната команда получава профил на използването на купа на паметта:

```
java -Xrunprof:heap=sites КласаЗаПрофилиране
```

Архитектурата на профайлър агента се съобразява с две основни цели. Първата цел е независимост – логиката е разделена от функциите на операционната система (четене и писане в мрежовия поток, получаване на време с висока точност и т.н.). Така чрез пренаписване само на малка част от кода, агента може да бъде пуснат на каквато и да е архитектура. Кодът е написан на чисто C, без използването на каквито и да е разширения и нестандартни конструкции. Практиката показва, че това решение е доста удачно. В момента агента работи без проблеми на следните процесорни архитектури – x86, ARM, MIPS, PowerPC, SH и операционните системи – Windows, Windows CE, Linux, Solaris, QNX, VxWorks. Втората цел е да се балансира използването на ресурсите. Агента събира и обработва информацията от виртуалната машина и праща само част от нея на крайния профайлър. По този начин се намалява мрежовия трафик, а използването на паметта и процесора се поделя между двата процеса,

което позволява стартирането на по-мащабни приложения при положение, че се намират на различни машини.

Пълния списък с опциите, който могат да бъдат подадени на профайлър агента е даден в Таблица 4.

използване на PROF: `-Xrunprof[:help]|[`*`опция=стойност`*`, ...]`

Пример: `java -Xrunprof:cpu=samples,file=log.txt,depth=3` *`НякакъвКлас`*

По подразбиране, информацията за използването на купа на паметта (местоположения на обекти и съдържание) се записва в java.prof.txt (в ASCII формат).

В описанията на форматите са използвани следните съкращения за улеснение при представянето на различните размери и типове:

- *u1*: 1 байт
- *u2*: 2 байта
- *u4*: 4 байта
- *u8*: 8 байта
- *id*: идентификатор (обикновено 4 байта)

Информацията изпращана от профайлър агента се състои от първоначален запис, следван от поредица от записи свързани с дадени събития по време на изпълнението на профилирания процес. Първоначалния запис се използва за установяване на версията на профайлър агента, за да може крайния профайлър да работи с различни версии на агенти (дори и на други производители), тъй като те поддържат различни набори команди и събития. Използва се също така и за определяне на размера на идентификаторите, както и точното време на започване на процеса.

Формата на първоначалния запис е даден в Таблица 5.

Всички останали записи съдържат в себе си информация за типа на събитието, което ги е породило (във вид на етикет), времева мярка от началото на процеса и броя байтове в тялото на записа, чиито формат зависи от типа на събитието.

Формата на останалите записи е даден в Таблица 6.

V.2. Събития

Поддържаните събития до момента от агента към крайния профайлър са петнадесет. Формата на всеки запис е разгледан поотделно.

a) PROF_UTF8 име кодирано в UTF8 формат

Използва се за представяне на всички стрингове от профилирания процес. Това са имената на нишките, класовете, методите, полетата, мониторите, изходните файлове и т.н. Кодирането е по UTF8 и позволява представянето на всички възможни букви от различните азбуки.

Формата на записа е даден в Таблица 7.

b) PROF_FRAME Java стекова рамка

Отговаря на ред от кода в изходния файл. Използва се за представяне на Java стеков път.

Формата на записа е даден в Таблица 8.

c) PROF_TRACE Java стеков път

Представя Java стеков път. Съставен е от рамки, който отговарят на ред от кода в изходния файл. Използва се за посочване на точното място на дадено събитие - зареждане на клас, стартиране на нишка, заделяне на памет, използване на процесора.

Формата на записа е даден в Таблица 9.

d) PROF_LOAD_CLASS новозареден клас

Изпраща се, когато класа е зареден във ВМ. Използва се чрез серийния номер в Java стековите рамки и набора от заделени обекти в купа на паметта и чрез идентификатора на класа-обект в съдържанието на купа на паметта.

Формата на записа е даден в Таблица 10.

e) PROF_UNLOAD_CLASS отзареждан клас

Изпраща се, когато класа се отзарежда.

Формата на записа е даден в Таблица 11.

f) PROF_START_THREAD новостартирана нишка

Изпраща се, когато стартира нишка във ВМ. Използва се при представяне на Java стеков път.

Формата на записа е даден в Таблица 12.

g) PROF_END_THREAD завършваща нишка

Изпраща се, когато приключи нишка във ВМ.

Формата на записа е даден в Таблица 13.

h) PROF_CONTROL_SETTINGS настройки на ключове от тип включено / изключено

Използва се за проверка на текущото състояние на модулите за проследяване и дълбочината на стековия път.

Формата на записа е даден в Таблица 14.

i) PROF_ALLOC_SITES набор от заделени обекти в купа на паметта

Необходим за определяне на използването на заетата памет. Информацията съдържа класа на обекта, мястото на заделяне, броя байтове и броя инстанции за периода (обикновено около 2 секунди).

Поотделно са дадени броя на активните и броя на общо заделените. Броя на общо заделените е необходим, тъй като множество излишни временни обекти натоварва garbage collector-а и фрагментира използваната памет. Могат да се изпращат само разликите между периодите, за по-добра производителност.

Формата на записа е даден в Таблица 15.

j) PROF_CPU_SAMPLES набор от стекови проби на работещите нишки

Необходим за определяне на "горещите места" при изпълнение на програмата, т.е. местата, които използват най-интензивно процесора. Използвания метод е семплиране - през определен период се записва докъде е стигнало изпълнението и накрая се присвоява тежест на всяка такава проба в зависимост от това колко пъти се среща. Така метод, който се изпълнява по-бавно или по-често, ще попадне в пробите с по-голяма тежест.

Формата на записа е даден в Таблица 16.

k) PROF_GRAPH_INFO информация за виртуалната машина

Съдържа ограничена информация за използването на купа на паметта, броя заредени класове, активността на нишките и GC. Подходящ за представяне в графичен вид на натовареността на виртуалната машина.

Формата на записа е даден в Таблица 17.

l) PROF_HEAP_SUMMARY резюме на купа на паметта

Съдържа ограничена информация за използването на купа на паметта - количеството активни и заделени байтове и инстанции.

Формата на записа е даден в Таблица 18.

m) PROF_HEAP_DUMP съдържание на купа на паметта

Съдържа подробна информация за използването на купа на паметта - всички класове със пула от константи и статични полета, инстанциите и техните полетата, както и стойностите им. Използва се, за да се открие защо определени инстанции не се освобождават от garbage collector, а именно защото има указател от някъде към тях.

Формата на записа е даден в Таблица 19.

n) PROF_MONITOR_DUMP статус на всички нишки и съдържание на мониторите

Съдържа подробна информация за състоянието на всички нишки (изпълняващи се, в режим на изчакване) както и всички използвани монитори. Може да се използва за откриване на евентуален deadlock.

Формата на записа е даден в Таблица 20.

o) PROF_TRIGGER_EVENT влизане или излизане от метод

Изпраща се, когато се изпълни дадено условие (влизане или излизане от определен метод на даден клас). Използва се за прецизно измерване в областта на даден метод.

Формата на записа е даден в Таблица 21.

V.3. Команди

Командите представляват еднобайтови етикети, евентуално съпроводени с няколко параметъра. Повечето отговарят на съответните команди достъпни от крайния профайлър, но има и някои, които се използват вътрешно от ядрото.

- ***PROF_CMD_GC*** предизвиква GC

- ***PROF_CMD_DUMP_HEAP*** получава съдържанието на купа на паметта
- ***PROF_CMD_ALLOC_SITES*** получава заделените обекти

Списъка на параметрите е даден в Таблица 22.

- ***PROF_CMD_HEAP_SUMMARY*** получава резюме на купа на паметта
- ***PROF_CMD_EXIT*** предизвиква преждевременно излизане
- ***PROF_CMD_DUMP_TRACES*** получава всички новополучени стекови пътища
- ***PROF_CMD_CPU_SAMPLES*** получава набор от стекови проби на работещите нишки

Списъка на параметрите е даден в Таблица 23.

- ***PROF_CMD_CONTROL*** сменя настройките

Списъка на параметрите е даден в Таблица 24.

- ***PROF_CMD_SUSPEND_ALL*** поставя всички нишки в режим на изчакване
- ***PROF_CMD_RESUME_ALL*** възобновява всички нишки
- ***PROF_CMD_GRAPH_INFO*** получава информация за виртуалната машина
- ***PROF_CMD_DUMP_MONITORS*** получава съдържанието на мониторите
- ***PROF_CMD_DUMP_ON_EXIT*** предизвиква изпращане на съдържанията на излизане
- ***PROF_CMD_SET_TRIGGER*** установява спусък

Списъка на параметрите е даден в Таблица 25.

VI. Краен профайлър

Класовете на крайния профайлър са разделени в 6 основни пакета, като визуалната част е отделена в собствен пакет с два подпакета. В приложението са добавени UML диаграми на класовете, чрез които може да се добие по-лесно представа за техните взаимовръзки (наследявания, асоциации), операции и атрибути.

VI.1. пакет `com.prosyst.jpd.profiler`

Тук се намират основните класове, чрез които се създава и управлява входно-изходния поток между профайлър агента и крайния профайлър.

За изграждането на комуникацията между крайния профайлър и профайлър агента в различните режими - `Launch`, `Listen` и `Attach` се грижи `ProfilerServer`. Тук се създават входния и изходния поток. Заглавния комуникационен запис се прочита от `BinHeader`, за да се провери производителя и версията на профайлър агента, както и да се определи дължината на идентификаторите и времето на стартиране. JProfiler работи не само с разработения специално за него агент, но и с вградените в повечето ВМ агенти. Естествено не всички от функциите са достъпни с тези агенти и `BinHeader` определя в зависимост от версията, кои ще могат да се използват.

Интерфейса с необходимите методи за комуникация с входния поток - размер на идентификаторите, пропускане на определен брой байтове и естествено прочитане на байт, дума, двойна дума, някой от основните примитивни типове, стринг, идентификатор и масив от байтове и идентификатори се намира в `BinInput`, а съответно `BinOutput` се използва за изпращане на необходимите команди към изходния поток. `BinInputImpl` е наследника на `ProfilerInput`, който имплементира интерфейсите `BinInput` и `ProfilerListener`. Грижи се да чете

непрекъснато записите от входния поток в двоичен формат и да изпраща съответните събития при настъпването им към регистрирания слушател. `ProfilerInput` е базов абстрактен клас представящ входния поток, който може да бъде разширен до текстов - `ProfilerInputAscii`, двоичен - `BinInputImpl` или просто игнориращ - `ProfilerInputNone`. `ProfilerInputAscii` и `ProfilerInputNone` се използват предимно при разработка. Първия извежда всички данни от входния поток в текстов режим на стандартния изход, а втория игнорира всички данни от входния поток. Интерфейса `ProfilerListener` служи за описание на всички възможни събития, които могат да възникнат - стартиране / спиране на профилирането, грешка във входния поток, набор от заделени обекти в купа на паметта, набор от стекови проби на работещите нишки, информация за виртуалната машина, съдържание на купа на паметта, статус на всички нишки и съдържание на мониторите. Интерфейса се имплементира от модулите, които искат да бъдат уведомявани за тези събития.

VI.2. пакет `com.prosyst.jpd.profiler.binbody`

Класовете от този пакет представлява отделен тип двоичен запис със собствен метод за прочитане от потока и добавяне в склада на ядрото. Това кой клас от пакета `binbody` трябва да прочете съответния запис се разпределя в зависимост от типа от `BinRecord`. За по-добра производителност в по-дългите или често използвани записи се използва `BinRecordReader`, който буферира входния поток и прочита целия запис наведнъж. Базовия абстрактен клас представящ обобщен двоичен запис е `BinBody`. Всички останали записи го наследяват. Набора от заделени обекти в купа на паметта се прочитат от `BinBody_AllocSites`, а набора от стекови проби на работещите нишки - от `BinBody_CpuSamples`. Те използват `BinRecordReader` за буфериране на входния поток и по-добра производителност. Изпращат съответното събитие към регистрирания слушател. `BinRecordReader` се използва и за прочитането на стекова

рамка и стеков път от `BinBody_Frame` и `BinBody_Trace`. Събитията за начало и край на нишка се прочитат от `BinBody_StartThread` и `BinBody_EndThread`. Използват се, за да се инициализира в ядрото и да се засече, кога е стартирала и кога точно е спряла нишката, колко време е работила и т.н. При край не бива да се премахва от ядрото, защото може да се използва от останалите елементи чрез стековия път. Съответно събитията за зареждане и отзареждане на клас се управляват от `BinBody_LoadClass` и `BinBody_UnloadClass`. Класа трябва да се добави в ядрото с два различни идентификатора, тъй като единия се ползва за указател от купа на паметта, а другия от останалите елементи. При отзареждане не бива да се премахва от ядрото, защото се използва от останалите елементи чрез стековия път. Информацията за виртуалната машина генерира през интервал от една секунда - използване на купа на паметта, броя заредени класове, активността на нишките и GC се прочита от `BinBody_GraphInfo`. Друг основен елемент са стринговете кодирани в `UTF8` формат, които се управляват от `BinBody_Utf8`.

`BinBody_HeapDump` се грижи за подробната информация за използването на купа на паметта - всички класове със пула от константи и статични полета, инстанциите и техните полетата. За по-добра производителност, ако профайлър агента позволява, стойностите на примитивните типове се пропускат, тъй като не са необходими. Разпределя кой клас от пакета `heapdump` трябва да прочете съответния подзапис в зависимост от типа. След прочитането анализира зависимостите между класовете и обектите и изчислява броя на инстанциите. Изпраща съответното събитие към регистрирания слушател. `BinBody_HeapSummary` и `BinBody_ControlSettings` се използват сравнително рядко. Първия прочита информация за използването на купа на паметта - количеството активни и заделени байтове и инстанции, а втория текущото състояние на профилиращите модули, кои са включени и кои изключени, и дълбочината на стековия път.

VI.3. пакет `com.prosyst.jpd.profiler.gui`

Класовете, които са отговорни за управлението на визуалната част са отделени в този пакет и неговите подпакети. Основния клас, който се грижи за управлението на действията на потребителя и представянето на информацията е `ActionManager`. Той имплементира интерфейса `ActionCodes`, който съдържа кодовете на менютата и бутоните. Другия не по-малко важен клас е `VisualManager`. Той пък създава и управлява състоянията на менюто, лентата с инструментите, прозорците, панелите и листчетата. Зарежда и съхранява използваните иконки. JProfiler е замислен като интернационално приложение, затова всички стрингове не са фиксирани в изходния код, а се четат от файл със съответния език. `ResourceManager` определя текущите регионални настройки и зарежда необходимия файл. Също така се грижи за запазването на настройките между две сесии. Съдържа методи за четене и запис на основните примитивни типове.

Панела за използването на паметта се управлява от `AllocSamplerGuiImpl`. Съдържа дърво, таблица, два филтъра и две превключващи се опции. Обработва различните събития като получаване на набор от заделени обекти в купа на паметта, показване на всички местоположения, стартиране, продължаване и край на маркиране. Също така опреснява динамично и филтрира дървото и таблицата, чете и записва във файл. Тъй като поддържането на информация с такъв обем (дори за средно големи приложения таблица със стотици хиляди редове и дърво с 2-3 пъти повече възли) е невъзможно, в таблицата се показват само първите няколко по типа на сортиране редове, а дървото при ръчно поискване. Когато профилирането свърши или спре за момент се показва цялата информация. Дори тогава дървото се изгражда динамично, т.е., в началото се създават само първите две нива, а когато даден възел се разгъне се добавят и неговите внуци, ако се продължи с разгъването още едно ниво по-навътре и т.н. Алтернативно представянето на заделянето на памет

може да се представи чрез `AllocSamplerImpl`. Информацията се извежда на стандартния изход. Местоположенията на обектите и в двата случая се съхраняват в ядрото през `AllocSiteSorter`.

Панела представящ натоварването на процесора се контролира от `CpuSamplerGuiImpl`. Съдържа дърво, таблица и филтъра. Обработва различните събития като получаване на набор от стекови проби, показване на всички стекови проби, стартиране, продължаване и край на маркиране. Също така опреснява динамично и филтрира дървото и таблицата, чете и записва във файл. Тук дървото не е толкова голямо, а и информацията се получава при спиране на профилирането затова тя се представя статично. Алтернативното конзолно представяне на натоварването на процесора върху стандартния изход е чрез `CpuSamplerImpl`. Стековите проби пак и в двата случая се съхраняват в ядрото през `CpuSampleSorter`.

Тъй като възлите от дървото трябва да разширяват `TreeNode`, за да се спести памет, данните не се съдържат директно в тях, а има указател към елемента от ядрото, като възела само определя формата на представянето на информацията, съответно за рамка от стековия път на заделянето на памет - `AllocSiteNode` и `CpuSampleNode` за рамка от стековия път на натоварването на процесора.

`HeapDumpPanel` представя панела със съдържанието на купа на паметта. Включва таблица на класовете и филтър, списъци с инстанциите и инициализираните статични полета сочещи към обекти, дърво на супер класовете, дърво на указателите и списък с елементите на стековия път на мястото на заделяне на обекта. Управлява връзките между компонентите. И тук дървото на указателите е динамично. Възлите от дървото на указателите в купа на паметта са от тип `FieldNode`. Те капсулират данните за указващия или указвания обект от дадено поле и формата на представянето му.

Компонента, който управлява изобразяването на графиката със статистиките за виртуалната машина е `VMInfoComponent`. Обработва различните събития като добавяне на нова точка, смяна на типа на

графиката, промяна на мащаба, скролиране. Точките от графиката със статистиките за виртуалната машина се пазят във `VMPoint`, това са данните за различните параметри в даден момент. `VMInfoComponent` се намира в контейнера `VMInfoPanel`, който съдържа и двата спускащи се списъка за типа на графиката и мащаба.

Когато от друг панел е необходимо да се разгледа изходния код на профилираното приложение, за да се открие причината за дадено събитие, `SourcePanel` предоставя тази възможност. Различните елементи - запазени думи, идентификатори и др., са оцветени в различни цветове, за да улеснят четенето.

VI.4. пакет `com.prosyst.jpd.profiler.gui.dialogs`

Всички диалози, които се използват в приложението, се намират в този пакет.

Чрез `StartDialog` се настройват опциите за режима `Launch` - пътищата необходими да се намерят Java класовете и изходните файлове, опциите на Java виртуалната машина и командните параметри на приложението. Освен това като листчета за улеснения са добавени и `OptionPanel` и `VMUsePanel`. `OptionPanel` се вгражда в диалозите за избор на режим и позволява да се изключват някои от модулите за профилиране, а за режима `Launch` и да се променят параметрите подавани на профайлър агента както и типа на локално стартиране, а `VMUsePanel` позволява избор между различни виртуални машини, на които ще се изпълнява приложението, без промяна на системните настройки, като само се избере съответната от списъка. Търси, добавя, премахва и проверява версиите на виртуалните машини, за да може да се адаптира към конкретните им особености. Вместо пътищата да се пишат на ръка и да се внимава за правилния синтаксис, чрез `PathDialog` лесно могат да се добавят, изтриват, редактират и избират директно от файловото дърво необходимите директории или архиви. Опциите за режима `Listen` или

Attach се настройват чрез ListenDialog. Това са порта, на който ще се осъществи комуникацията, пътя до изходните файлове и дали даден профилиращ модул да бъде включен или изключен. Начина, по който ще бъдат представени резултатите, в зависимост от предпочитанията и конкретните нужди на потребителя се определя от PreferencesDialog. Понякога е много трудно да се намери някой елемент в голяма таблица, дърво или списък, но с помощта на FindDialog това не е проблем. Всички състояния на избраната нишка, кога и колко точно е работила или изчаквала за съответния монитор се показват в ThreadInfoDialog.

VI.5. пакет com.prosyst.jpd.profiler.gui.help

Класовете от този пакет се грижат за предоставянето на помощна информация. Всички основни компоненти имат такова контекстно описание. То е достъпно, когато съответния елемент, за който е нужно съдействие, е на фокус и се натисне клавиша F1 или се използва съответното меню. HelpManager е отговорен за създаването и стартирането на сървъра за помощ, показването на необходимата страница и след приключване затваряне на сървъра, а HelpResource извлича адреса на страницата за съответния език.

VI.6. пакет com.prosyst.jpd.profiler.heapdump

Подобно на класовете от пакет binbody, класовете в този пакет също представляват отделен тип двоичен подзапис на текущото съдържание на купа на паметта със собствен метод за прочитане от потока и добавяне в склада на ядрото. Базовия абстрактен клас представящ обобщен подзапис на съдържанието на купа на паметта е HeapDump. Описанието на клас се прочита от HeapDump_ClassDump, описанието на обект, който не е масив - от HeapDump_InstanceDump, описанието на масив от обекти - от HeapDump_ObjArrayDump, описанието на масив от

примитивен тип - от `HeapDump_PrimArrayDump`, глобален указател към обект през JNI - от `HeapDump_RootJniGlobal`, а локален указател към обект през JNI - от `HeapDump_RootJniLocal`. Указателите към локални променливи съответно от Java стекова рамка - `HeapDump_RootJavaFrame`, а от машинно зависим код - `HeapDump_RootNativeStack`. Останалите подзаписи `HeapDump_RootMonitorUsed`, `HeapDump_RootStickyClass`, `HeapDump_RootThreadBlock`, `HeapDump_RootThreadObj`, `HeapDump_RootUnknown` не се използват в момента.

VI.7. пакет `com.prosyst.jpd.profiler.info`

Всеки елемент от склада на ядрото се представя със свой собствен клас от този пакет. Съответно - нишка от `ThreadInfo`, стеков път от `TraceInfo`, стекова рамка `FrameInfo` и описание на клас от `ClassDumpInfo`. За по-бързо визуализиране, сортиране, филтриране и цялостно управление заделените обекти в купа на паметта са структурирани първо по класове чрез `AllocSiteGroup`, а след това по редове от стековия път в `AllocSiteInfo`. За стековите проби това не е необходимо, а се използва директно `CpuSampleInfo`. Възлите от дървото на паметта, представящи елементите създадени на един и същ ред от стеков път са от тип `AllocSiteData`, защото е възможно на това място в изходния код да има обекти от различни класове. За сортиране на набора от заделени обекти в купа на паметта и стековите проби по определени критерии се използват `AllocSiteSorter` и `CpuSampleSorter`. Точките от графиката с информацията за виртуалната машина - използване на купа на паметта, броя заредени класове, активността на нишките и GC се пазят в `GraphInfo`. Такъв елемент се генерира на всяка секунда. `InstDumpInfo` е базовия клас за обектите от купа на паметта, който се наследява от `PrimArrayDumpInfo` и `ObjArrayDumpInfo`, съответно за масивите от примитивни елементи и масивите от обекти, а `ObjectDumpInfo` за

обектите, който не са масиви. Указателите към локални променливи от Java стекова рамка и от машинно зависим код се пазят в `JavaFrameRoot` и `NativeStackRoot`. `Misc` предоставя няколко общи метода и структури използвани от елементите на ядрото.

VI.8. пакет `com.prosyst.jpd.profiler.storage`

Цялата информация събирана от профайлъра се индексира в склада на ядрото за по-бърз достъп, когато е необходима. За по-голямо бързодействие беше разработена хеш-таблицата `HashIntObjNS`, чиито ключове са директно цели числа `int`, тъй като стандартната хеш-таблица от `java.util.Hashtable` изисква `Integer` за ключ, което води не само до излишно създаване на обекти от този тип, но и на вътрешни класове `Entry` от `Hashtable`, а освен това е и с усложнено управление. `HashIntObjNS` се отличава и с доста по-малък разход на памет. Различните типове елементи на ядрото - имена, нишки, стекови пътища, стекови рамки, класове и обекти се добавят в самостоятелни хеш-таблици. Интерфейса с необходимите методи за работа с хеш-таблица от дадена подобласт на ядрото - търсене по ключ, добавяне и премахване на елемент, изчистване, размер, вземане на масив от елементите или ключовете е `ObjectStorageInt`, а `ObjectStorageIntDefaultImpl` е неговата имплементацията. `ProfilerNamespace` е базов абстрактен клас представящ склада на ядрото, а `ProfilerNamespaceDefaultImpl` е неговата имплементация. Позволява достъп до отделните подобласти на ядрото - имена, нишки, стекови пътища, стекови рамки, класове и обекти. Търсенето се осъществява по идентификатор. При инициализиране отделните подобласти имат различна големина, базирана на предполагаемия обем елементи, който ще бъдат добавени, за едно средностатистическо приложение. Също така служи за четене и записване на ядрото във файл.

VII. Потребителска документация

VII.1. Стартиране

JProfiler е създаден да задоволи нуждите на потребителя по отношение на изпълнението на приложението и особеностите свързани с производителността. Някои приложения се стартират на локалната машина, но някои работят отдалечено. Не винаги е възможно да се пусне профайлъра на същата машина, на която е и профилираната програма. Например малко устройство с ограничена памет. Освен това работата на една и съща машина изкривява резултатите. Затова JProfiler предлага три режима на стартиране - Launch, Attach и Listen. Последните два са предназначени за профилиране на приложения на отдалечени машини.

При режима Launch профайлъра стартира приложението на същата машина и се свързва с нея или се използва просто като среда за стартиране. При режима Listen се чака отдалечено приложение да се стартира и свърже с профайлъра. А при Attach се свързва с предварително стартирано приложение.

Стартирането в различните режими изисква настройки, които се въвеждат с помощта на диалози.

а) Диалог Launch

Бутоните *Manage Settings* позволяват да се записват и зареждат настройките необходими за стартиране. Те се съхраняват в `ini` файлове, като последния използван автоматично се зарежда при следващо стартиране. За запис се използва бутона *Save As...*, за зареждане бутона *Load*, а бутона *New* изчиства всичката въведената информация.

Разделен е на три секции - *Program, Options* и *Virtual Machines*.

Секцията ***Program*** съдържа пътищата необходими да се намерят Java класовете и изходните файлове, опциите на Java виртуалната машина и командните параметри на приложението.

Main class name or jar file to run - Пълното име на главния клас-файл на приложението, което трябва да се профилира. Могат да се стартират приложения и от jar файлове, стига да е настроен файла `Manifest.mf`. Бутона *Browse...* позволява по-лесно да се намери желания файл.

Execute program in directory - Съдържа пътя към желана работна директория. По подразбиране това е директорията, която съдържа пакета на главния клас файл.

Java custom options - Параметрите подавани на ВМ. Могат да бъдат например `-D`, която установява системно свойство или `-verbose`, за по-подробни системни съобщения.

Program command line arguments - Аргументите от командния ред необходими на приложението.

Class path - Съдържа пътищата към клас-файловете. Всички клас-файлове, който се използват (без тези от ВМ) и не са в работната директория, трябва да имат път до тях.

Source path - Съдържа пътищата към файловете с изходния код. Необходими са, за да се покаже дадено събитие (заделяне на памет, използване на процесорно време) в контекста на изходния код и да даде по ясна представа за причината да бъде породено.

Секцията ***Options*** определя системните настройки необходими за стартирането на Java процес в режим на профилиране, както и кой профилиращи модули да са изключени.

Disable Memory profiling - Изключва профилирането на паметта. Не се събират никакви резултати за консумацията на памет.

Disable CPU profiling - Изключва изчисляването на натоварването на процесора.

Auto start CPU profiling - Включва профилирането на процесора автоматично в началото на процеса. В противен случай трябва да се пусне ръчно, когато е необходимо, като се използва бутона *Start Mark Current State*.

Disable VM Info sampling - Изключва събирането на информация за ВМ - обем използвана памет, брой заделени обекти, брой заредени класове, брой активни нишки и активност на GC.

Command used for profiling - Съдържа командите подавани на ВМ, за да се стартира режим на профилиране. Ако се използва нестандартна ВМ или външен профайлър агент, който използва неконвенционални опции, тук те могат да се зададат. Препоръчва се, да се променят тези опции само от специалист. В краен случай има бутон *Reset*, с който могат да се върнат стойностите по подразбиране.

Радио бутоните *Local, Remote attach* и *Remote listen from* определят типа на локално стартиране:

Local - стартира и се свързва с ВМ локално.

Remote attach - стартира приложението в профилиращ режим с подходящите опции за отдалечен профайлър да се прикачи.

Remote listen from - стартира приложението в профилиращ режим и се опитва да се свърже с вече стартиран отдалечен профайлър на дадения адрес.

Profiler port - определя порта, на който ще се извършва комуникацията между крайния профайлър и профайлър агента.

Секцията ***Virtual Machines*** позволява избор между различни виртуални машини, на които ще се изпълнява приложението, без промяна на системните настройки, като само се избере съответната от списъка. Има три начина да се добавят виртуални машини:

- като се използва бутона *Search*, който отваря диалога за избор и се посочи директория или дори устройство и се остави JProfiler да претърси и открие всички виртуални машини, който поддържа;

- като се използва бутона *Search*, който отваря диалога за избор и се посочи изпълнимия файл на виртуалната машина;
- като се зададе пътя директно в текстовото поле най-отдолу и се натисне бутона *Add*;

Поддържат се само виртуални машини, който отговарят на спецификацията Java 2 и JVMPI. Не могат да се добавят други виртуални машини. При опит се извежда съобщение за грешка.

Поддържаните ВМ и проблемите с всяка от версиите са дадени в Таблица 26.

Търсенето може да бъде прекратено с бутона *Stop*, на който се сменя бутона *Search*. За да се премахне виртуална машина от списъка се използва бутона *Remove*. Бутона *Modify* се използва за промяна на пътя до виртуалната машина, като предварително е редактиран в текстовото поле.

b) Диалог Listen

Необходимата информация, която трябва да се попълни е порта, на който ще се осъществи комуникацията. Както и при Launch, може да се зададе път до изходните файлове и да се изключат някои от профилиращите модули.

Source path - Съдържа пътищата към файловете с изходния код. Необходими са, за да се покаже дадено събитие (заделяне на памет, използване на процесорно време) в контекста на изходния код и да даде по ясна представа за причината да бъде породено.

Опциите *Disable Memory profiling, Disable CPU profiling, Auto start CPU profiling, Disable VM Info sampling* изпълняват същите функции като тези при Launch.

c) Диалог Attach

Диалогът е същия като този при Listen, но трябва да се зададе и име на машината цел, където ще се закачи. Процесът, който ще се профилира трябва да е предварително стартиран в *Remote Attach* режим.

d) Диалог Preferences

Определя начина, по който ще бъдат представени резултатите.

StartUp

Настройва според предпочитанията на потребителя кой лист и диалог да се появяват в началото по подразбиране и дали да се записват позициите на прозореца и разделителите.

Sampler

CPU sampler precision показва резултатите от семплирането на процесора групирани по метод или с точност до ред в изходния файл.

Радио бутоните *Memory units* и *Time precision* контролират размерността на мерните единици, съответно за паметта в байтове, килобайтове и мегабайтове и за процесорното време с един, два, три или четири знака след запетаята.

Възможно е всички инстанции на даден клас да бъдат освободени. За да не се показват такива класове, се използва *Show 0 insts*.

Тъй като поддържането на таблица на обектите с толкова много редове (в повечето случаи стотици хиляди) е невъзможно, *Show top lines* определя броя на първите по типа на сортиране редове, който ще се показват. Когато профилирането свърши или спре за момент се показват всички редове.

Memory Columns и ***CPU Columns***

Настройват според предпочитанията на потребителя кои колони от таблиците да са видими, за да се пести място и да се игнорира ненужната информация.

VII.2. Управление

Управлението се осъществява с помощта на меню и лента с инструменти. В лентата с инструментите се намират само тези от командите, който се използват най-често. Командите са разделени в три групи - Program, Data и Controls.

Командите за управление са дадени в Таблица 27.

VII.3. Резултати

Резултатите от работата на профайлъра се показват в няколко панела - Memory, CPU, Heap, VMInfo, Source и Console.

а) Панел на паметта (Memory)

Обектите се показват заедно с информация за класа, към който принадлежат, размер в байтове, брой инстанции, нишка и метод, отговорни за заделянето. Панела е разделен на две части, който представят информацията по два различни начина - като дърво и като таблица. Двата изгледа са взаимно свързани, ако щракнем на ред от единия изглед, автоматично се отива на съответния ред от другия. Когато се щракне два пъти на възел от дървото или ред от таблицата се превключва на панела *Source* и се показва мястото от изходния код, отговорно за заделянето.

Размерността на единиците се контролира от настройките в *Preferences*. По подразбиране е в байтове. Може да бъде в кило или мега байтове.

Дърво на паметта - представя стековите пътища и техните рамки. Най-старшият възел е първата рамка от стековия път, най-младшият е последната. Подвъзлите са сортирани във низходящ ред по консумация на памет. Има четири типа възли:

Типовете възли са дадени в Таблица 28.

Всеки ред показва сумата на инстанциите и паметта на поддървото му. Подвъзлите представят разпределението на тази памет между методите, които се извикват от текущия метод. Възлите на нишките дават следната информация:

клас *метод* *ред* *байтове* *инстанции*
`demo.prosyst.profiler.RunMe. makeAllocations: 70 (80160 bytes, 10 insts)`

- *клас* - пълното име на класа съдържащо дадения ред.
- *метод* - метода част, от който е реда. Идентификатора `<init>` се използва вместо име на метод, ако заделянето е осъществено в конструктора на класа. Ако реда е част от статичен блок на класа, идентификатора е `<clinit>`. Метода също така може да бъде и машинно зависим.
- *ред* - номера на реда в изходния код. Позволява да се отиде на съответния ред с двойно щракване. Липсва при машинно зависимите методи.
- *байтове* - размер на паметта заделена на този ред и останалите подвъзли.
- *инстанции* - брой но инстанциите създадени на този ред и останалите подвъзли.

Таблица на паметта - информация за заделените класове и количеството заделена памет. Може да се използва таблицата за директно изследване на стековия път в дървото. Първо се сортира по признака, който ни интересува (например, по най-голям брой инстанции), след това с избиране на елемента се намира и разгъва съответният възел, представящ мястото, където е заделен.

Информацията за обектите от всеки клас е подредена в колони.

- *classname* - Името на класа с пакета.
- *bytes* - Броя байтове заделени за дадения клас или обект.

- *insts* - Броя създадени инстанции от класа.
- *diff insts* и *diff bytes* - Показват обикновено същата информация като колоните bytes и insts. Разликата се получава, когато се използва Begin Mark. Тогава стойностите се нулират и броенето започва отначало. Изчислението в тези колони спира, когато се натисне End Mark. Могат да се добавят статистики за други етапи от изпълнението чрез Continue Mark и End Mark отново. Тази функция е полезна, когато искаме да изчислим точното потребление на памет в даден период. Стойностите показват байтовете и инстанциите използвани само в тази част.
- *thread* - Нишката, в която е създаден обекта.
- *method* - Точното място на създаване на обекта в изходния код. Включва пълното име на класа, името на метода, където е създаден обекта и номер на реда в изходния код.

Четири елемента най-отдолу контролират изгледа на таблицата: *Class filters, Method filters, Show groups* и *Disable GC*.

Филтри

Текстовите полета на филтрите позволяват да се покажат само елементите, който ни интересуват. И двете полета разпознават малки и големи букви. Има включващи и изключващи филтри.

За да се включи показването на елементи, съдържащи даден стринг, напишете текста разделен със запетайки.

За да се изключат елементи, текста трябва да се предхожда от удивителен знак !.

Ето един пример за правилна заявка:

```
java., !.io., demo.
```

Резултата от този филтър за таблицата ще бъде, че ще се показват класовете от пакетите `java.` и `demo.`, но без пакета `.io.`.

Class filters - филтъра се прилага само за колоната с имената на класовете от таблицата.

Method filters - филтрира елементите от колоната на методите.

Show groups - Ако *Show groups* е включен, инстанциите на съответните класове са групиране по име на класа. Когато се разгъне възел могат да се видят детайлите за всеки обект.

Disable GC - Превключва между реалната консумация на памет и обема, който би бил зает, ако не нямаше garbage collector.

b) Панел на процесора (CPU)

През определен период се записва докъде е стигнало изпълнението на програмата и накрая се присвоява тежест на всяка такава проба в зависимост от това колко пъти се среща. Така метод, който се изпълнява по-бавно или по-често, ще попадне в пробите с по-голяма тежест.

Пробите се показват заедно с информация за нишката и метода, в който процесора е бил натоварен, време в проценти и милисекунди. Панела е разделен на две части, който представят информацията по два различни начина - като дърво и като таблица. Двата изгледа са взаимно свързани, ако щракнем на ред от единия изглед, автоматично се отива на съответния ред от другия. Когато се щракне два пъти на възел от дървото или ред от таблицата се превключва на панела *Source* и се показва мястото от изходния код, отговорно за натоварването.

Размерността на единиците се контролира от настройките в *Preferences*. По подразбиране е с два знака след запетайката. Може да бъде с един, три или четири.

Дърво на натоварването на процесора - представя стековите пътища и техните рамки. Най-старшият възел е първата рамка от стековия път, най-младшият е последната. Подвъзлите са сортирани във низходящ ред по натоварването. Има четири типа възли:

Типовете възли са дадени в Таблица 29.

Всеки ред показва сумата на процентите и милисекундите на поддървото му. Подвъзлите представят разпределението на това време между методите, които се извикват от текущия метод. Възлите на нишките имат следния формат:

клас *метод* *ред* *байтове* *инстанции*

```
demo.prosyst.profiler.RunMe. makeComputations: 78 (90.6 %,  19360 ms)
```

- *клас* - пълното име на класа съдържащо дадения ред.
- *метод* - метода част, от който е реда. Идентификатора `<init>` се използва вместо име на метод, ако натоварването е осъществено в конструктора на класа. Ако реда е част от статичен блок на класа, идентификатора е `<clinit>`. Метода също така може да бъде и машинно зависим.
- *ред* - номера на реда в изходния код. Позволява да се отиде на съответния ред с двойно щракване. Липсва при машинно зависимите методи.
- *проценти* - натоварването на процесора в проценти от общото време.
- *милисекунди* - натоварването на процесора в милисекунди.

Таблица на натоварването на процесора - информация за използваното процесорно време.

Информацията за пробите е подредена в колони.

- *thread* - Нишката, в която е извършвана операцията.
- *method* - точното място на изпълнение в изходния код. Информацията е организирана като в таблицата на паметта: пълното име на класа, името на метода, номер на реда в изходния код
- *time %* - Натоварването на процесора в проценти от общото време.
- *time ms* - Натоварването на процесора в милисекунди.

Филтри

Method filters - филтрира елементите от колоната на методите, по същия начин като филтрите в таблицата на паметта.

c) Панел на купа на паметта (Heap)

Показва текущото съдържание на купа на паметта. Може да се изпълнява, само ако приложението работи в момента. Разделен е на четири части - classes, class info, references и allocation stack trace.

Classes - таблица на заредените класове. Избрания ред определя съдържанието на останалите подпанели. Таблицата има три колони:

- *class loader* - показва пълното име на класа-зареждач, който е заредил съответния клас. Това може да бъде системния клас-зареждач. Ако е инстанция на потребителски клас-зареждач, с двойно щракване се отива на реда от таблицата, където се намира този клас. Името на инстанцията е следвано от идентификатор на ВМ.
- *classes* - пълното име на класа.
- *insts* - броя обекти от този клас, който са активни в момента.

Филтри

Class filters - филтрира елементите от колоната на класовете, по същия начин като филтрите в таблицата на паметта и таблицата на натоварването на процесора.

Class Info - горния десен подпанел има три листа Instances, Static fields и Super classes.

Instances - колоната insts от таблицата на класовете показва колко инстанции от този клас са текущо достижими. Листа *Instances* съдържа списъка на всички тези инстанции. Те са с името на класа, но се различават

по идентификатора от ВМ. Когато се избере инстанция, подпанела *References* долу вляво представя информация за нея.

Static Fields - съдържа всички статични полета на избрания клас, който са инициализирани. Определя съдържанието на подпанела *References*, когато се избере някое поле. Елементите имат следния формат:

```
име         клас                        идентификатор
address = java.lang.String[5] @ e0e80505
```

- *име* - името на статичното поле
- *клас* - името на класа, към който принадлежи инстанцията
- *идентификатор* - идентификатора на инстанцията от ВМ

Super Classes - дърво на супер-класовете, наследник на които е изследвания клас. При двойно щракване се отива на съответния супер-клас.

References - дърво на указателите. Съдържанието им се контролира от избора в листовете *Instances* или *Static fields* на подпанела *Class Info*. Също така зависи и от двата радио-бутона - *outgoing* и *incoming references*.

Outgoing references - Когато бъде избран, подпанела показва всички инстанции, който се указват от текущо изследваната. Тези инстанции са стойностите на полетата.

Формата на подвъзлите е следния:

```
име      клас                    идентификатор
out =  outgoing.TestLoop @ 5020120b
```

- *име* - името на полето на инстанцията
- *клас* - името на класа, към който принадлежи указваната инстанция

- *идентификатор* - идентификатора на указваната инстанция от ВМ

Incoming references - Показва полетата на инстанциите, който указват текущо изследваната.

Формата на подвъзлите е следния:

```
име      клас                    идентификатор
ti of  incoming.TestRef @ 1829120b
```

- *име* - името на полето на инстанцията, което сочи към текущия обект
- *клас* - името на класа, към който принадлежи полето
- *идентификатор* - идентификатора на указващата инстанция от ВМ

Типовете възли са дадени в Таблица 30.

Allocation Stack Trace - стеков път на заделянето на инстанцията избрана в листовете *Instances* или *Static fields* на подпанела *Class Info*. Когато се щракне два пъти на ред от подпанела *References*, *Allocation stack trace* показва стековия път на заделянето на избрания обект.

Всеки ред от стековия път съдържа пълното име на класа, метода и реда от изходния код. При двойно щракване се отива на съответното място в панела *Source*.

d) Панел на виртуалната машина (VMInfo)

Включва статистики за виртуалната машина. Линейни графики визуално представят прогреса във времето на различни характеристики на ВМ по време на профилирането. Графиките представят различни аспекти от производителността зависещи от филтрите в горния десен ъгъл.

филтър за мащаба - позволява да се контролира мащаба по хоризонталната ос. Чертаят се изминалото време през 10 секунди, а

видимия интервал е съответно 1, 2 или 5 минути. Пробите са на всяка секунда и се изобразяват върху мрежа от хоризонтални линии през 10% от максималната стойност и вертикални на всяка секунда.

филтър за изгледа - определя типа на информацията, която се показва. В горния ляв ъгъл има резюме на текущите стойности от графиката. Възможните настройки са Bytes, Instances, Threads, Classes и GC activity.

Bytes - зелена и червена графика. Зелената показва обема заделени байтове през цялото време на изпълнение на приложението. Последната стойност е след етикета *total allocated bytes*. Червената показва броя байтове, който са текущо заделени след garbage collection. Последната стойност е след етикета *live bytes*.

Instances - светло и тъмно синя графика. Светло синята представя общия брой създадени обекти, а тъмно синята броя активни инстанции, който в момента са в купа на паметта. Последните им стойности са съответно в *total allocated insts* и *live insts*.

Threads - розовата графика и на общия брой нишки. Виолетовата е само на брой активни нишки през периода

Classes - оранжева графика - общия брой заредени класове във ВМ.

GC Activity - жълта графика - активност на GC в проценти от общото време за периода.

е) Панел с изходния код (Source)

Потребителя може да открие тези части от изходния код, които отговарят на специфично показание на профайлъра. Ако щракнем два пъти на извикване на метод от стековия път на съответния възел от дърво или

ред от таблица или списък, се намира изходния файл, отваря се в панела и се позиционира на съответния ред. Пътищата до съответните файлове трябва да се зададат в *Source path* на диалозите Launch, Listen или Attach. Ако не може да бъде открит файла се извежда диалог, с помощта на който може да се намери и пътя към него евентуално да се добави за следващо ползване. Могат да се използват директно и `jar` файлове, както и изходния код на класовете от ВМ.

f) Панел на входно-изходните операции (Console)

Текстовия вход и изход от профилираното приложение или от профайлъра се извежда в Console. Това е изхода от `System.out` и `System.err` и входа към `System.in` потоците. Ако е стартиран в някой от двата отдалечени режима, конзолния вход и изход от профилираното приложение няма да се покаже в профайлъра, а на машината на която е стартирано приложението. Ако втори профайлър се използва като среда за стартирането на отдалеченото приложение, тогава ще се използва неговата конзола. Панела позволява търсене и копиране.

VIII. Конкурентни системи

Има две основни категории профайлъри - които използват интерфейса JVMPI и които използват статична или динамична модификация на байт кода. Тъй като предмета на тази дипломна работа се отнася към първата група, а и защото все още няма пълноценно реализиран профайлър от втората група, разчитащ изцяло на този метод, тях няма да ги разглеждаме. Трите най-разпространени профайлъра от първата група (в хронологичен ред на тяхното създаване) са - JProbe, OpimizeIt, JProfiler. И трите са комерсиални. Съществуват и доста безплатни и/или с отворен код варианти като например - Cougaar Memory Profiler, Extensible Java Profiler, JMemProf, JMP, JRat, но те са доста не

функционални. Освен това напоследък популярност добиват профайлърите като разширения на Eclipse - jMechanic, Eclipse Profiler. Последната версия на JProfiler също може да работи като разширение на Eclipse.

По същество всички профайлъри предлагат едни и същи възможности, обусловени от свойствата на интерфейса JVMPI. Разликата е предимно в представянето на огромната по обем информация, филтрирането и, и т.н.

VIII.1. JProbe (http://www.quest.com/jprobe/)

JProbe е съставен от няколко компонента - JProbe Profiler, JProbe Memory Debugger и JProbe Threadalyzer. Предоставя няколко уникални възможности, които липсват в другите профайлъри. Например така наречените "спусъци" или "вмъквания". JProbe може да се настрои (чрез диалози или директно от профилирания код) да започне профилирането в началото на даден метод и да свърши в края. Така резултатите ще съдържат само съответния метод и извиканите от него и потребителя се ориентира по-лесно. Друга положителна черта е записването и сравняването на отделни "снимки" на състоянието на процеса. След подобрения в кода така много лесно се открива ефекта от направените промени и дали те наистина са положителни. За съжаление някои от решенията не са особено удачни. Всеки изглед се представя в отделен прозорец и потребителя бързо се дезориентира, тъй като бива затрупан от множество прозорци. Информацията за използването на процесора се представя в граф, които не е подходящ за сложни стекови пътища, защото се получава плетеница от възли. Въпреки добрите идеи за улесняване на профилирането това е най-трудния за работа профайлър, а освен това и най-скъпия.

VIII.2. OpimizeIt (http://www.borland.com/optimizeit/)

OpimizeIt съдържа два инструмента - OptimizeIt Profiler и OptimizeIt Thread Debugger. По възможности и интерфейс JProfiler се доближава най-много до него. Комплекта е вече част от Borland JBuilder. Продукта е разработен доста професионално и въпреки, че липсват някои екзотични възможности от JProbe, има всички необходими функции, структурирани логически. Сравнително скъп е, а и факта, че отдавна не е добавяна нова функционалност, го правят все по-малко предпочитан.

VIII.3. JProfiler (http://www.ej-technologies.com/jprofiler/)

JProfiler (съвпадение на имената) е най-агресивно разработвания профайлър в момента. С много приятен, удобен и интуитивен интерфейс. За разлика от другите два продукта всички инструменти са интегрирани в едно приложение. По този начин, без да се налага да се рестартира, могат да се разглеждат различните аспекти на профилирания процес. Освен възможността за работа през JVMPI, се поддържа и разработваната като отворен код от същата фирма експериментална библиотека за работа с байт кода. Чрез умело използване на модификацията на байт код, след като се настрой JProfiler може да бъде най-бързия и лек профайлър, тъй като се избягват излишното уведомяване за събития от класове и методи, които не ни интересуват, а и от друга страна информацията е по-малко и по-лесно се асимилира. Въпреки липсата на някои разширени функции, лекотата на използване и атрактивната цена дават голямо предимство на JProfiler пред останалите.

Предимствата на нашия JProfiler са разширената поддръжка на операционни системи и виртуални машини и лесното стартиране в режим слушане и прикачване, подходящи за отдалечено профилиране.

IX. Насоки за развитие и усъвършенстване

Разработката на настоящата дипломна работа се разтегли твърде много във времето и настоящата версия се различава концепцуално. Докато предишната версия е насочена по-скоро към настолните системи, новата е ориентирана към малките устройства. Конкуренцията при настолните системи е доста ожесточена - три големи комерсиални продукта, няколко по-малки и неопределен брой безплатни, затова беше взето решение за препозициониране към друг сегмент, в който няма никаква конкуренция. Поддържат се по-екзотични платформи, операционни системи и виртуални машини, които най-често се използват във вградените устройства, като например процесори ARM, MIPS Little и Big Endian; операционни системи VxWorks, QNX; виртуални машини J9, Jeode. Но съответно, тъй като малките устройства разполагат с ограничени ресурси, сложния кеш, който преди беше балансиран между агента и крайния профайлър, трябваше да бъде преместен изцяло в крайния профайлър. Така се намалява използвана памет (основния проблем с предишната версия) както и натоварването на процесора при хеширане от агента, който се намира на малкото устройство, а се увеличава мрежовия трафик и крайния профайлър става по-тежък. Но при наличието на 100 Mbps и една съвременна конфигурация, в някои случаи това се оказва по-бързия вариант. Другите големи нововъведения бяха използването на по-бавния, но по-точен метод за профилиране на натоварването на процесора, чрез който се следят събитията за вход и изход от метод, възможността ако платформата позволява да се показва реалното използвано процесорно време и модула за проследяване на състоянието на нишките, чрез който се откриват ненужните изчаквания и съревнованието между нишките.

За в бъдеще е планирано да се добави и статична или динамична модификация на байт кода, което ще позволи JProfiler да работи и с други платформи, които нямат поддръжка на интерфейса JVMPI. Също така е

предвидена и основна реорганизация на профилирането за памет с оглед на по-опростена работа за потребителя.

X. Заключение

Настоящата дипломна работа си постави за цел да изследва теоретично проблемите с неефективното използване на ограничените ресурси при разработване на приложения написани на езика за програмиране Java и някои методи за тяхното предотвратяване, както и да реализира конкретна разработка подпомагаща откриването на тези проблеми. Първата от двете основни части, които бяха дефинирани в изложението, разгледа основни препоръки за оптимизация, описа и илюстрира със схеми типовете загуба на памет. Бяха разгледани технологиите, които предлага езика за програмиране Java и по-специално интерфейса JVMPI (Java Virtual Machine Profiler Interface), средата и платформите върху които работи, мястото и възможностите, които интерфейса предлага за профилиране. Във втората основна част беше описана реализацията на конкретния проект, като част от дипломната работа, както и съвети за работа на потребителя. Задачата имаше за цел да демонстрира теоретическото описание в изграждането на една реално работеща система.

Като цяло дипломната работа описва и анализира автоматизираното изследване на нерационално използване на ресурсите, като подкрепя теоретичното описание с конкретно реализиран проект, представляващ напълно функциониращо професионално Java приложение, което може да бъде използвано в ежедневната работа. Продукта може да бъде доизграден с добавянето на допълнителна функционалност, като например по-бавния, но по-прецизен метод за измерване на натоварването на процесора или дори нов модул за изследване на съревнованията между нишките.

XI. Използвана литература

- http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html
- http://developer.java.sun.com/developer/bugParade/index.html
- Ethan Henry и Ed Lycklama, 2000, How Do You Plug Java Memory Leaks?, Dr. Dobb's Journal February 2000
- Jack Shirasi, 2000, Java Performance Tuning, O'Reilly
- Joshua Bloch, 2001, Effective Java: Programming Language Guide, Addison Wesley
- http://www.quest.com/jprobe/
- http://www.borland.com/optimizeit/
- http://www.ej-technologies.com/jprofiler/

Приложение А (Таблици)

идентификатор	**тип на данните**	**дефиниращо събитие**	**заличаващо събитие**
идентификатор на нишка	JNIEnv *	начало на нишка	край на нишка
идентификатор на обект	jobjectID	заделяне на обект	освобождаване и преместване на обект
идентификатор на клас	jobjectID	зареждане на клас	отзареждане на клас
идентификатор на метод	jmethodID	дефиниращо зареждане на клас	дефиниращо отзареждане на клас

Таблица 1 - Дефиниращи и заличаващи събития за различните типове идентификатори

тип на записа	**данни на записа**	
JVMPI_GC_ROOT_UNKNOWN (неизвестен корен)	jobjectID	обект
JVMPI_GC_ROOT_JNI_GLOBAL (JNI глобален указател)	jobjectID jobject	обект JNI глобален указател

JVMPI_GC_ROOT_JNI_LOCAL (JNI локален указател)	jobjectID JNIEnv * u4	обект нишка номер на фрейма в стековия път (-1 ако е празен)
JVMPI_GC_ROOT_JAVA_FRAME (Java стеков фрейм)	jobjectID JNIEnv * u4	обект нишка номер на фрейма в стековия път (-1 ако е празен)
JVMPI_GC_ROOT_NATIVE_STACK (машинно зависим стек)	jobjectID JNIEnv *	обект нишка
JVMPI_GC_ROOT_STICKY_CLASS (системен клас)	jobjectID	обект-клас
JVMPI_GC_ROOT_THREAD_BLOCK (указател от блок на нишка)	jobjectID JNIEnv *	обект-нишка нишка
JVMPI_GC_ROOT_MONITOR_USED (монитор)	jobjectID	обект
JVMPI_GC_CLASS_DUMP (съдържание на обект-клас)	jobjectID jobjectID jobjectID jobjectID jobjectID jobjectID void * u4 [jobjectID]* u2 [u2, ty, vl]* [vl]*	клас супер-клас зареждач на класа подписи област на защита име на класа (обект тип String, може да бъде NULL) резервиран размер на инстанция в байтове интерфейси размер на пула с константи индекс в пула с константи, тип стойност стойности на статичните полета
JVMPI_GC_INSTANCE_DUMP (съдържание на обикновен обект)	jobjectID jobjectID u4 [vl]*	обект клас брой байтове, който следват стойности на полетата на инстанция
JVMPI_GC_OBJ_ARRAY_DUMP (съдържание на масив от обекти)	jobjectID u4 jobjectID [jobjectID]*	обект-масив брой елементи идентификатор на класа на елементите елементи
JVMPI_GC_PRIM_ARRAY_DUMP (съдържание на примитивен масив)	jobjectID u4 ty [vl]*	обект-масив брой елементи тип на елементите елементи

Таблица 2 - Формат на съдържанието от ниво 2

тип на записа	данни на записа	
JVMPI_MONITOR_JAVA (Java монитор)	jobjectID JNIEnv * u4 u4 [JNIEnv *]* u4 [JNIEnv *]*	идентификатор на обект нишка-собственик брой на влизанията брой на нишките, който чакат да влезнат нишки, който чакат да влезнат брой на нишките, който чакат да бъдат уведомени нишки, който чакат да бъдат уведомени
JVMPI_MONITOR_RAW (суров монитор)	char * JVMPI_RawMonitor JNIEnv * u4 u4 [JNIEnv *]* u4 [JNIEnv *]*	име на суровия монитор идентификатор на суровия монитор нишка-собственик брой на влизанията брой на нишките, който чакат да влезнат нишки, който чакат да влезнат брой на нишките, който чакат да бъдат уведомени нишки, който чакат да бъдат уведомени

Таблица 3 - Формат на съдържанието на монитор

Име на опцията и стойности	Описание	По подразбиране
heap=dump|sites|all	профил на купа на паметта	all
cpu=samples|times|old	използване на процесора	изключено
monitor=y|n	надпревара за монитори	n
format=a|b	текстов или двоичен	a
file=name	запис във файл	java.prof (.txt за ASCII)
net=host:port	изпраща данните по мрежа	запис във файл
depth=size	дълбочина на стековата рамка	4
cutoff=value	прецизност	0.0001
lineno=y|n	номера на редове в пътищата?	y
thread=y|n	нишки в пътищата?	n
doe=y|n	съдържание при излизане?	y

Таблица 4 - Списък с опциите на профайлър агента

тип	описание
[u1]*	"JAVA PROFILER AGENT 1.1" (завършва с 0)
u4	размер на идентификаторите. Идентификаторите се използват, за да представят UTF8 стрингове, обекти, стекови рамки, и т.н. Обикновено имат същия размер като тези на указателите на гостоприемника. Например, на Solaris и Win32, размера е 4.
u4	брой милисекунди от 0:00 GMT, 01.01.1970 (старша дума)
u4	брой милисекунди от 0:00 GMT, 01.01.1970 (младша дума)
[запис]*	поредица от записи

Таблица 5 - Формат на първоначалния запис от агента

тип	описание
u1	етикет обозначаващ типа на записа
u4	брой милисекунди от времевата мярка на записа (превърта се след около малко по-малко от час)
u4	брой байтове оставащи в записа (броя изключва етикета и самата дължина)
[u1]*	тяло на записа (поредица от байтове)

Таблица 6 - Формат на останалите записи от агента

тяло	забележка
id	идентификатор на име
[u1]*	UTF8 знаци (без завършваща нула)

Таблица 7 - Име кодирано в UTF8 формат

тяло	забележка
id	идентификатор на стековата рамка
id	идентификатор на име на метода
id	идентификатор на сигнатурата на метод
id	идентификатор на името на изходния файл
u4	серийния номер на класа
u4	номер на ред > 0: нормален -1: неизвестен -2: компилиран метод -3: машинно зависим метод

Таблица 8 - Java стекова рамка

тяло	забележка
u4	сериен номер на стековия път
u4	сериен номер на нишката
u4	брой рамки

Таблица 9 - Java стеков път

тяло	забележка
u4	сериен номер на класа (> 0)
id	идентификатор на класа-обект
u4	сериен номер на стековата рамка
id	идентификатор на името на класа

Таблица 10 - Новозареден клас

тяло	забележка
u4	сериен номер на класа

Таблица 11 - Отзареждан клас

тяло	забележка
u4	сериен номер на нишката (> 0)
id	идентификатор на обекта на нишката
u4	сериен номер на стековия път
id	идентификатор на име на нишката
id	идентификатор на име на групата на нишката
id	идентификатор на име на родителя на групата

Таблица 12 - Новостартирана нишка

тяло	забележка
u4	сериен номер на нишката

Таблица 13 - Завършваща нишка

тяло	забележка
u4	1: проследяване на заделянето на памет включено / изключено 2: проследяване на използването на процесора включено / изключено
u2	дълбочина на стековия път

Таблица 14 - Настройки на ключове от тип включено / изключено

тяло	забележка
u2	флагове 0x0001: разлики или пълен 0x0002: сортирани по всички заделени или активни 0x0004: дали да извика предварително GC
u4	прецизност (0.0 ~ 1.0)
u4	общо активни байтове
u4	общо активни инстанции
u8	общо заделени байтове
u8	общо заделени инстанции
u4	брой местоположения на обекти, който следват

[u1	е_ли_масив: 0: обикновен обект 2: масив от обекти 4: масив от boolean 5: масив от char 6: масив от float 7: масив от double 8: масив от byte 9: масив от short 10: масив от int 11: масив от long
u4	серийния номер на класа (може да бъде 0 по време на стартиране)
u4	сериен номер на стековия път
u4	брой активни байтове
u4	брой активни инстанции
u4	брой заделени байтове
u4]*	брой заделени инстанции

Таблица 15 - Набор от заделени обекти в купа на паметта

тяло	**забележка**
u4	общ брой проби
u4	брой стекови пътища
[u4	брой проби
u4]*	сериен номер на стековия път

Таблица 16 - Набор от стекови проби на работещите нишки

тяло	**забележка**
u4	продължителност на периода
u4	общо активни байтове
u4	общо активни инстанции
u8	общо заделени байтове
u8	общо заделени инстанции
u4	брой на всички нишки
u4	брой на активните нишки
u4	брой на всички заредени класове
u4	време на активност на GC

Таблица 17 - Информация за виртуалната машина

тяло	забележка
u4	общо активни байтове
u4	общо активни инстанции
u8	общо заделени байтове
u8	общо заделени инстанции

Таблица 18 - Резюме на купа на паметта

тяло	забележка
[под-записи на съдържанието на купа на паметта]*	
u1	тип на под-запис
PROF_GC_ROOT_UNKNOWN	неизвестен корен
id	идентификатор на обект
PROF_GC_ROOT_THREAD_OBJ	нишка-обект
id	идентификатор на обекта на нишката (може да бъде 0 за новоприсъединена нишка чрез JNI)
u4	последователен номер на нишката
u4	последователен номер на стековия път
PROF_GC_ROOT_JNI_GLOBAL	корен на JNI глобален указател
id	идентификатор на обект
id	JNI global ref ID
PROF_GC_ROOT_JNI_LOCAL	JNI локален указател
id	идентификатор на обект
u4	сериен номер на нишката
u4	номер на рамка в стековия път (-1 ако е празен)
PROF_GC_ROOT_JAVA_FRAME	Java стекова рамка
id	идентификатор на обект
u4	сериен номер на нишката
u4	номер на рамка в стековия път (-1 ако е празен)
PROF_GC_ROOT_NATIVE_STACK	машинно зависим стек
id	идентификатор на обект
u4	сериен номер на нишката
PROF_GC_ROOT_STICKY_CLASS	системен клас
id	идентификатор на обект
PROF_GC_ROOT_THREAD_BLOCK	указател от блок на нишка
id	идентификатор на обект
u4	сериен номер на нишката
PROF_GC_ROOT_MONITOR_USED	зает монитор
id	идентификатор на обект
PROF_GC_CLASS_DUMP	съдържание на обект-клас
id	идентификатор на обект-клас
u4	сериен номер на стековия път
id	идентификатор на обект на супер класа
id	идентификатор на обект на зареждащия клас
id	идентификатор на обект-подписвач
id	идентификатор на обект-защитник
id	резервиран
id	резервиран
u4	размер на инстанция (в байтове)

u2	размер на пула от константи
[u2,	индекс в пула от константи,
u1,	тип 2: обект 4: boolean 5: char 6: float 7: double 8: byte 9: short 10: int 11: long
vl]*	и стойност
u2	брой статични полета
[id,	име на статичното поле,
u1,	тип,
vl]*	и стойност
u2	брой полета на инстанцията (не включващ тези на супер-класа)
[id,	име на полето на инстанцията,
u1]*	тип
PROF_GC_INSTANCE_DUMP	съдържание на обикновен обект
id	идентификатор на обект
u4	сериен номер на стековия път
id	идентификатор на обект-клас
u4	брой байтове, който следват
[vl]*	стойности на полетата на инстанцията (на класа, следвани от тези на супер-класа, супер-класа на супер-класа и т.н.)
PROF_GC_OBJ_ARRAY_DUMP	съдържание на масив от обекти
id	идентификатор на масива от обекти
u4	сериен номер на стековия път
u4	брой елементи
id	идентификатор на класа на елементите
[id]*	елементи
PROF_GC_PRIM_ARRAY_DUMP	съдържание на масив от примитивни типове
id	идентификатор на масива
u4	сериен номер на стековия път
u4	брой елементи
u1	тип на елементите 4: масив от boolean 5: масив от char 6: масив от float 7: масив от double 8: масив от byte 9: масив от short 10: масив от int 11: масив от long
[u1]*	елементи

Таблица 19 - Съдържание на купа на паметта

тяло	**забележка**
u4	брой нишки
[u4	сериен номер на стековия път
u4]*	статус на нишката
u4	брой монитори
[u1	тип на монитора
JVMPI_MONITOR_JAVA	java монитор
u1	тип на класа на монитора
id	име на монитора
id	идентификатор на монитора
u4	нишка-собственик
u4	брой влизания
u4	брой нишки чакащи да влезнат
[u4]	нишки чакащи да влезнат
u4	брой нишки чакащи да бъдат уведомени
[u4]	нишки чакащи да бъдат уведомени
JVMPI_MONITOR_RAW	суров монитор
id	име на суровия монитор
id	идентификатор на суровия монитор
u4	нишка-собственик
u4	брой влизания
u4	брой нишки чакащи да влезнат
[u4]	нишки чакащи да влезнат
u4	брой нишки чакащи да бъдат уведомени
[u4]	нишки чакащи да бъдат уведомени

Таблица 20 - Статус на всички нишки и съдържание на мониторите

тяло	**забележка**
u1	тип (входен или изходен спусък)
PROF_TRIGGER_ENTRY	влизане в метод
id	име на класа
id	име на метода
PROF_TRIGGER_EXIT	излизане от метод
id	име на класа
id	име на метода

Таблица 21 - Влизане или излизане от метод

тяло	**забележка**
u2	флагове 0x0001: разлики или пълен 0x0002: сортирани по всички заделени или активни 0x0004: дали да извика предварително GC
u4	прецизност (0.0 ~ 1.0)

Таблица 22 - Параметри на командата за получаване на заделените обекти

тяло	забележка
u2	не се използва за момента
u4	прецизност (0.0 ~ 1.0)

Таблица 23 - Параметри на командата за получаване на набор от стекови проби на работещите нишки

тяло	забележка
u2	0x0001: включва проследяване на заделянето на памет 0x0002: изключва проследяване на заделянето на памет 0x0003: включва проследяване на използването на процесора id: идентификатор на обекта на нишката (NULL за всички) 0x0004: изключва проследяване на използването на процесора id: идентификатор на обекта на нишката (NULL за всички) 0x0005: изчиства информацията за използването на процесора 0x0006: изчиства информацията за заделянето на памет 0x0007: установява максимална дълбочина на стековия път в събираната информацията u2: дълбочина на стековия път

Таблица 24 - Параметри на командата за сменяне на настройките

тяло	забележка
u1	тип (входен или изходен спусък)
u1	действие при получаване на спусъка
u2	дължина на името на класа
[u1]	име на класа
u2	дължина на името на метода
[u1]*	име на метода

Таблица 25 - Параметри на командата за установяване на спусък

Windows

ВМ	Забележка
Sun 1.1	не се поддържа
Sun 1.2.2	няма проблеми
Sun 1.3.0	само classic режим
Sun 1.3.1	няма проблеми
Sun 1.4	няма проблеми
IBM 1.3.0	няма проблеми
J9 2.0	няма информация за процесора, не работи pause
Jeode 1.7	проблеми с GC

Solaris

ВМ	**Забележка**
Sun 1.1	не се поддържа
Sun 1.2.2 (reference implementation)	не работят pause и attach
Sun 1.2.2 (production release)	не се поддържа
Sun 1.3.0	не се поддържа
Sun 1.3.1	няма проблеми
Sun 1.4	няма проблеми

Linux

ВМ	**Забележка**
Sun 1.1	не се поддържа
Sun 1.2.2	проблеми с ВМ
Sun 1.3	проблеми с ВМ
Sun 1.4	проблеми с ВМ
J9	няма информация за процесора и за купа на паметта
Jeode 1.6	не се поддържа
Jeode 1.8	не се поддържа

Таблица 26 - Поддържани ВМ и проблеми с всяка от версиите

команда	**икона**	**описание**
Program		
Launch		Зависи от настройките в Launch Options и опционално може да стартира: • локално ВМ, за да изпълни приложението за профилиране. • приложението в режим remote attach, готово към него да се прикрепи друг профайлър. • приложението в режим remote listen from, като вече е стартиран друг профайлър, който слуша на дадения порт.
Listen		Отваря диалога Listen и стартира профайлъра в режим на чакане за отдалечена ВМ. Когато ВМ се стартира, тя се свързва към JProfiler.
Attach		Отваря диалога Attach и се закача за вече стартирана ВМ във съответния режим.
Start Program		Стартира профилирането с текущите настройки от последния използван режим – Launch, Attach или Listen.
Suspend Program		Прекратява временно профилирането.
Stop Program		Спира профилирането.
Data		
Find		Търси стринг в активния компонент. Може да се претърсват панелите Memory, CPU и Heap.
Find Next		Намира следващо повторение на търсения стринг използван при последното извикване на Find.
Import Data		Зарежда предварително записана профилираща информация от JProfiler. Отваря се диалог за избор на файл и се посочва някой, който е бил записан преди това чрез командата Export Data.
Export Data		Записва информацията от панелите Memory и CPU във файл, така че

		по-късно може да бъде зареден (например след оптимизация), за да се изследва влиянието на направените промени. Разширението е без значение.
Clear Data		Изчиства всички данни получени от последното профилиране.
Controls		
Begin Mark		Използва се в панела Memory за маркиране на текущото състояние като начално за колоните diff bytes и diff insts. В панела CPU стартира профилирането на използването на процесора.
Continue Mark		Ако маркирането е спряно, се продължава, като новите резултати се добавят към вече натрупаните.
End Mark		Маркира текущото състояние като крайно за колоните diff bytes и diff insts от панела Memory и спира профилирането на използването на процесора.
Run Garbage Collector		Извиква garbage collector на профилираната ВМ.
Show Source		Показва изходния код от избрания в момента елемент.
Refresh Tree		Обновява съдържанието на дървото на паметта по време на профилиране. Това става автоматично при pause и stop.
View Heap		Показва текущото съдържание на купа на паметта.

Таблица 27 - Команди за управление

възел	**икона**	**описание**
system		Корена на дървото на паметта.
нишка		Подвъзлите на корена са нишките на ВМ. Всяка нишка е начало на поддърво представящо нейните стекови пътища. Възлите на нишките дават информация за името на нишката, обема на паметта заделен от нея през жизнения и цикъл и броя на обектите.
заделяне на памет		Означават, че на съответния ред от изходния код е заделена памет. Могат да имат подвъзли, ако има извикване на друг метод. Всички листа на дървото са възли от този тип.
проследява пътя		Тези възли показват извиквания, който са на по-горно ниво в стековия път и самите те не заделят памет, а това става някъде по-навътре.

Таблица 28 - Типове възли в дървото на паметта

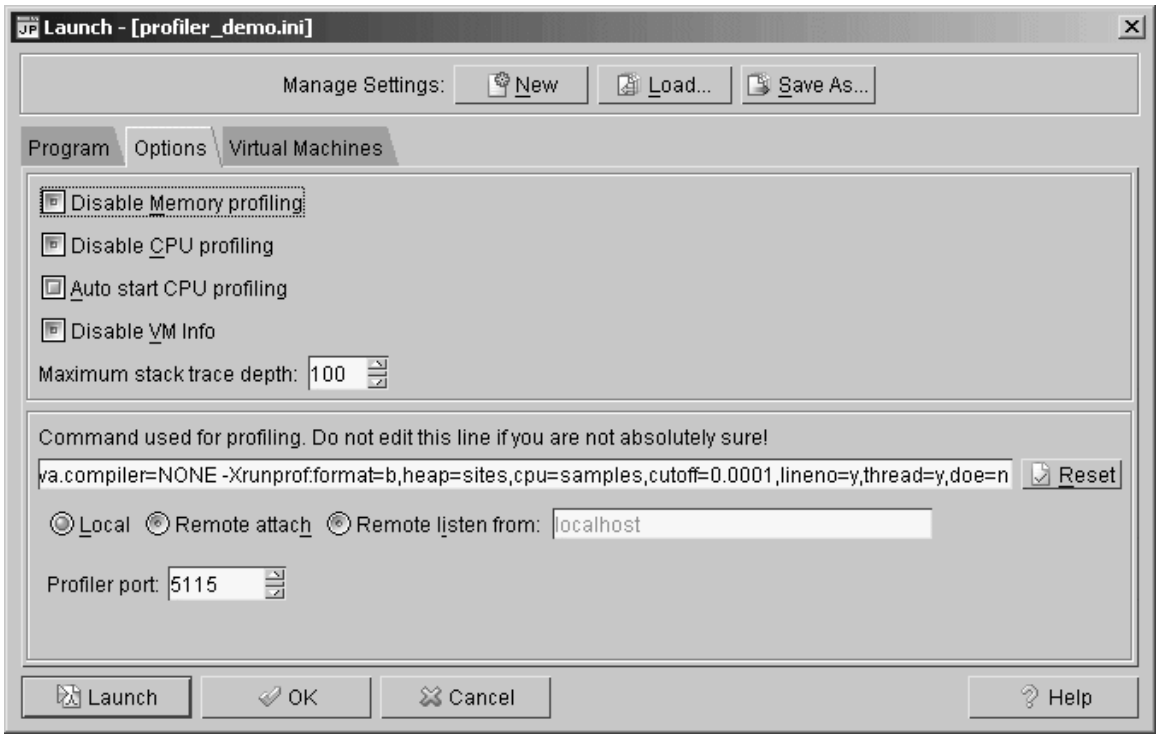
възел	икона	описание
system		Корена на дървото на паметта. В скоби е общото време, когато семплирането е било активно.
нишка		Подвъзлите на корена са нишките на ВМ. Всяка нишка е начало на поддърво представящо нейните стекови пътища. Възлите на нишките дават информация за името на нишката, време в проценти и милисекунди.
използване на процесора		Означават, че на съответния ред от изходния код е използван процесора. Могат да имат подвъзли, ако има извикване на друг метод. Всички листа на дървото са възли от този тип.
проследява пътя		Тези възли показват извиквания, който са на по-горно ниво в стековия път и самите те не използват процесора, а това става някъде по-навътре.

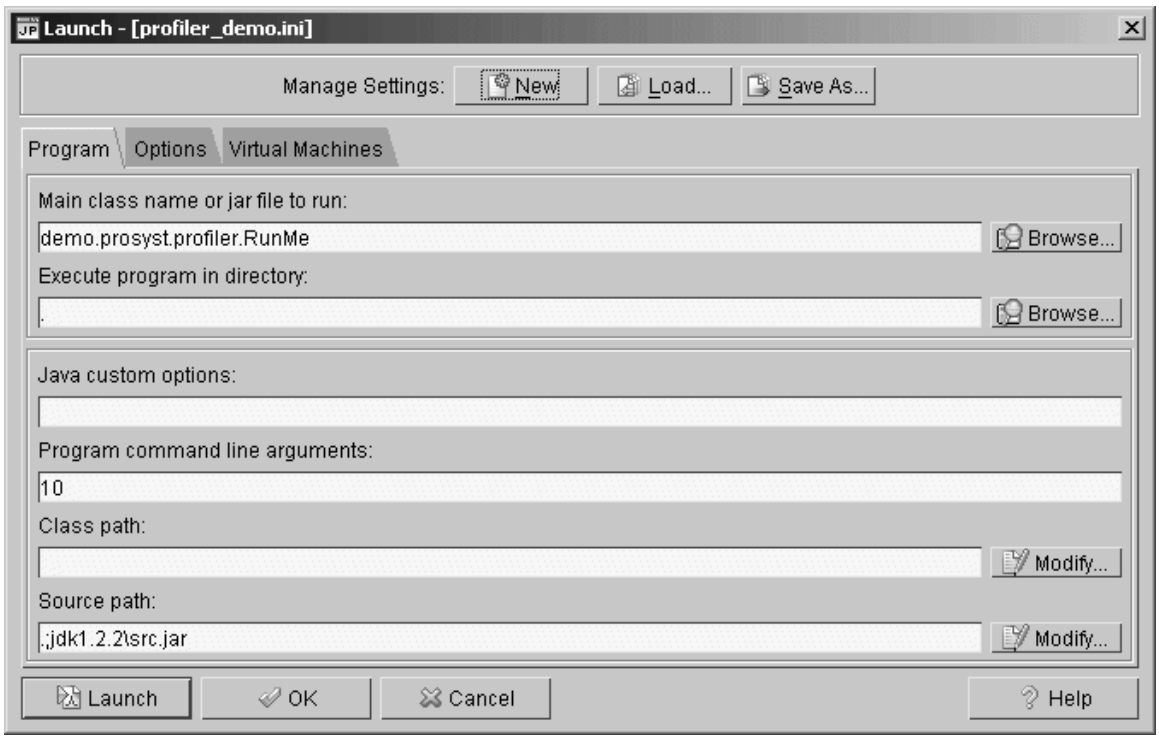
Таблица 29 - Типове възли в дървото на натоварването на процесора

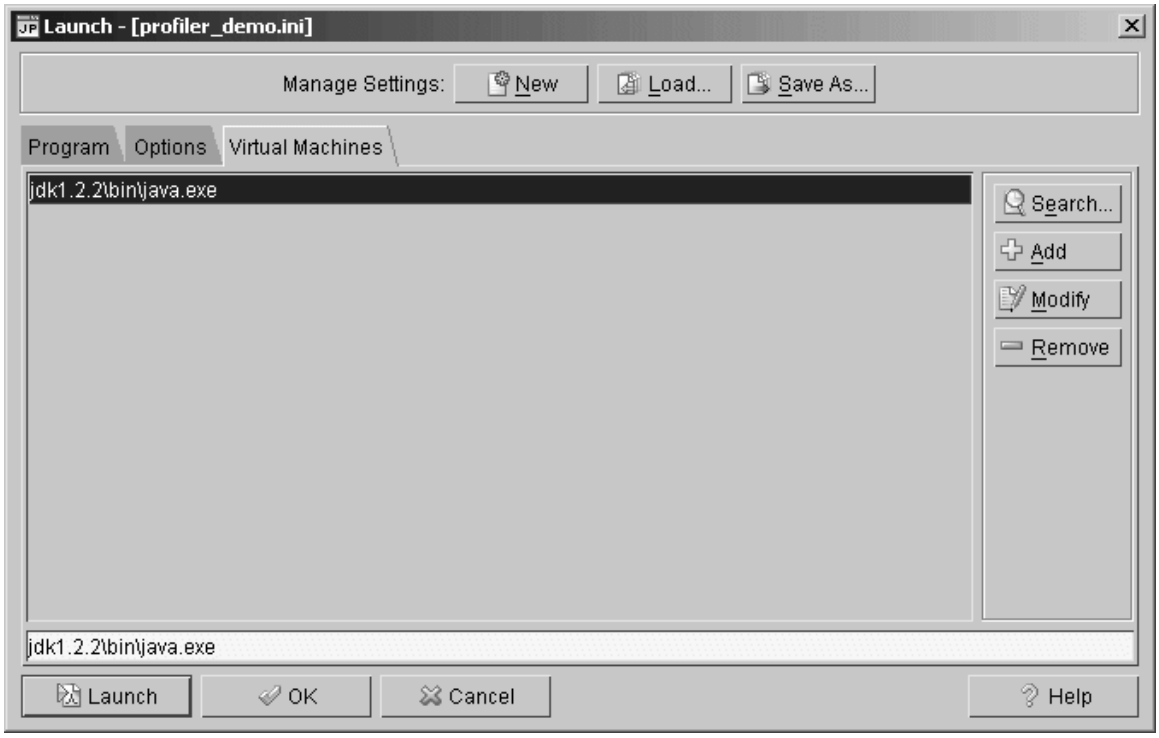
възел	икона	описание
обект		Корена на дървото. Представя инстанцията, разглеждана в момента.
Referencing/Referenced		Възел с подвъзли. Представя: • изглед outgoing references - обект с инициализирано нестатично поле; • изглед incoming references - обект, който е указан от: друг обект, статично поле, ВМ или машинно зависим код.
листо		Възел без подвъзли. Може да бъде: • изглед outgoing references - обект без инициализирани нестатични полета; • изглед incoming references - Java/Native stack of thread <thread_name>. Указван от ВМ или машинно зависим код.
цикъл		Указва възел, който е на по-горно ниво.
клас		Само при изгледа incoming references. Статично поле, което сочи към обекта.

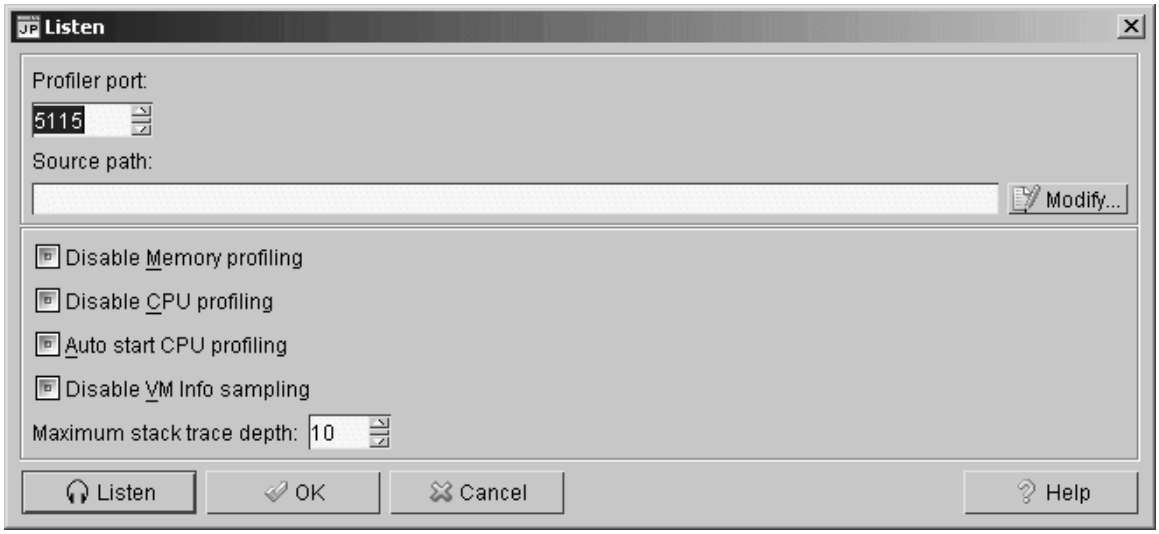
Таблица 30 - Типове възли в подпанела References

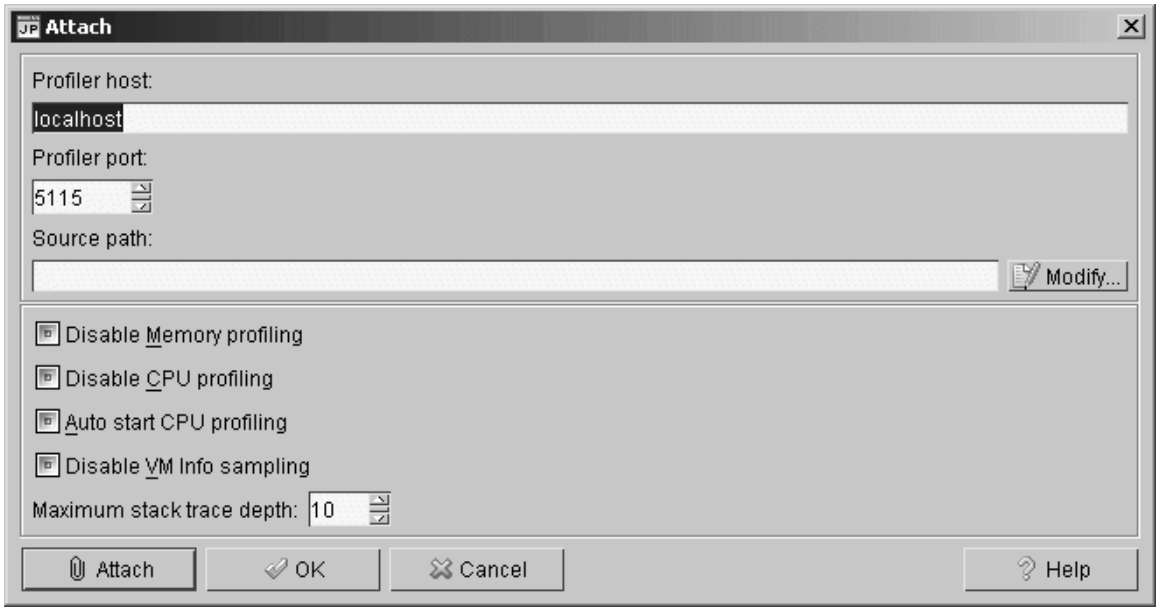
Приложение Б (разпечатки на екрани)

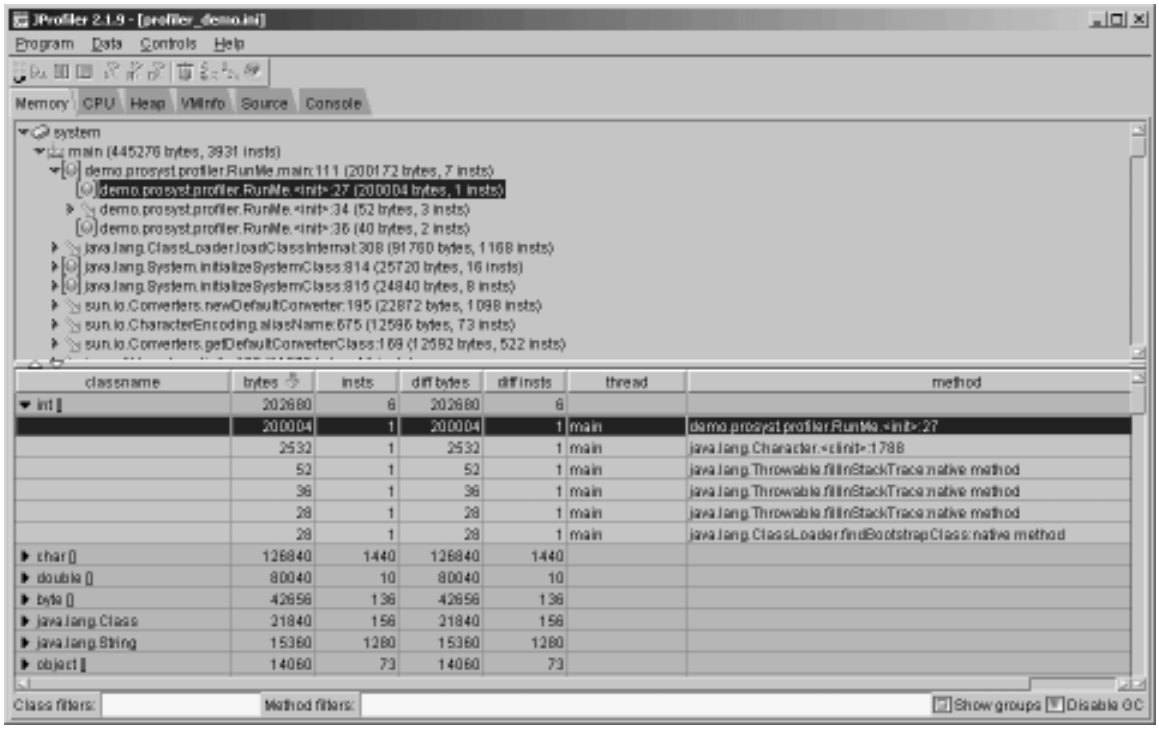


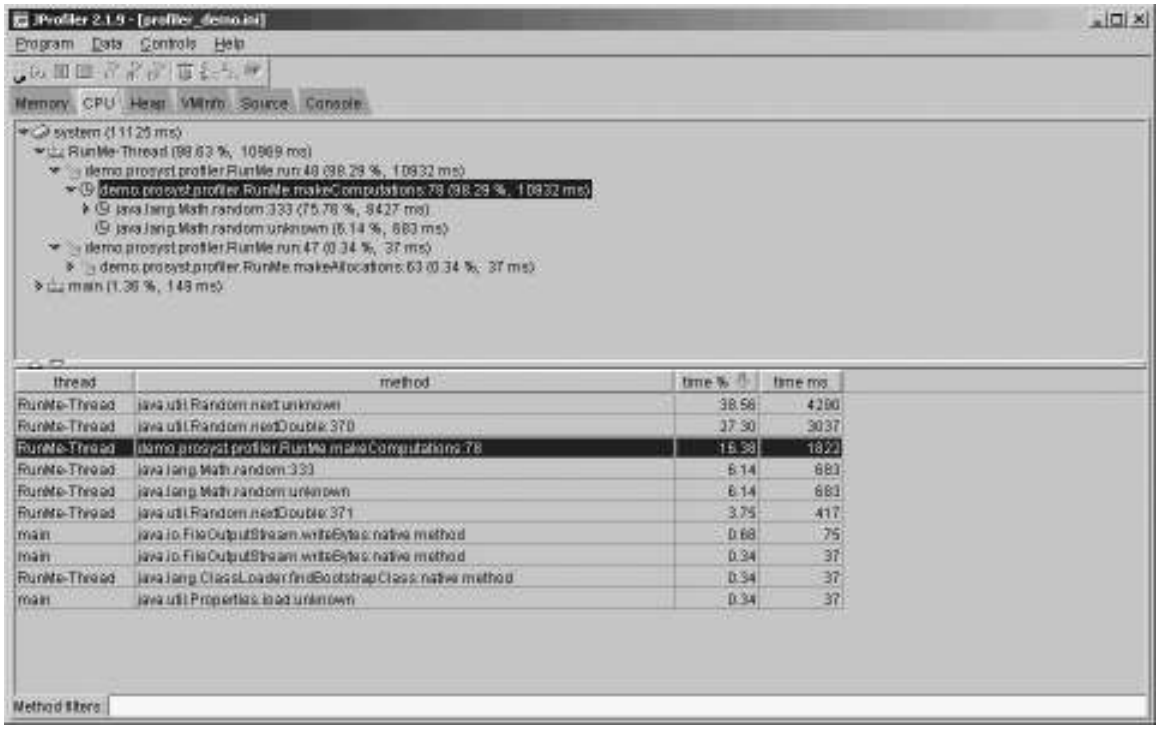


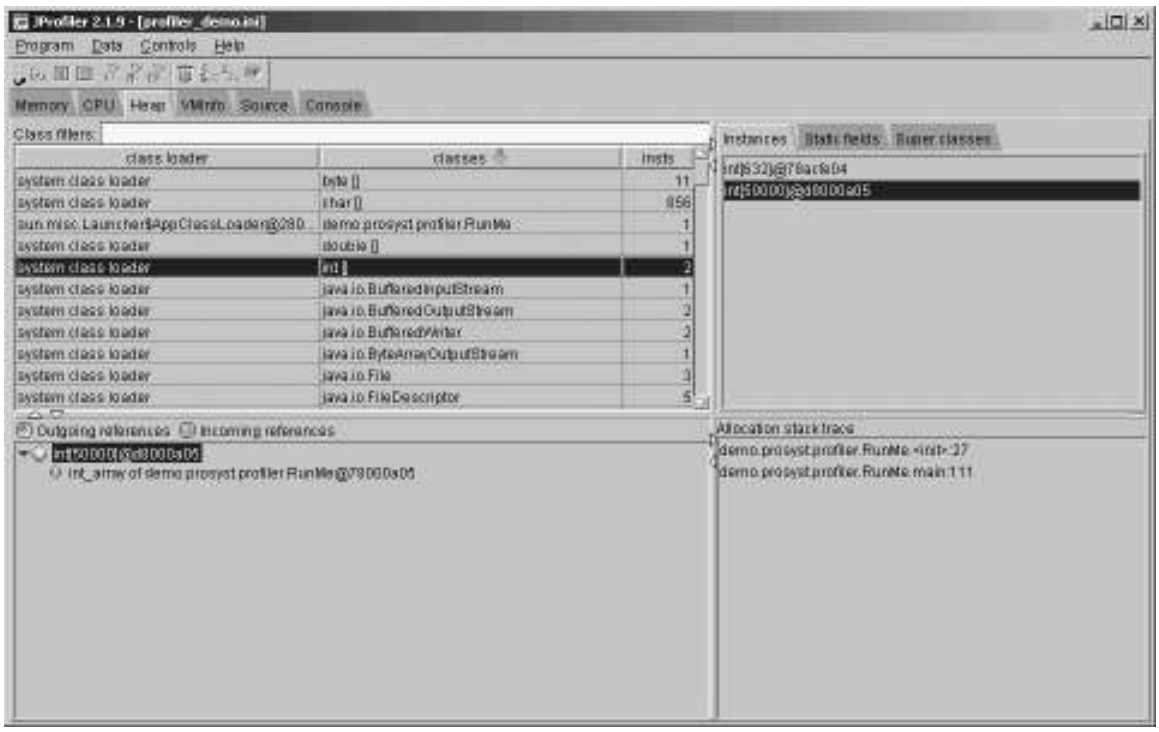

Launch - [profiler_demo.ini]
Manage Settings:
New
Load...
Save As...
Program
Options
Virtual Machines
jdk1.2.2\bin\java.exe
Search...
Add
Modify
Remove
jdk1.2.2\bin\java.exe
Launch
OK
Cancel
Help

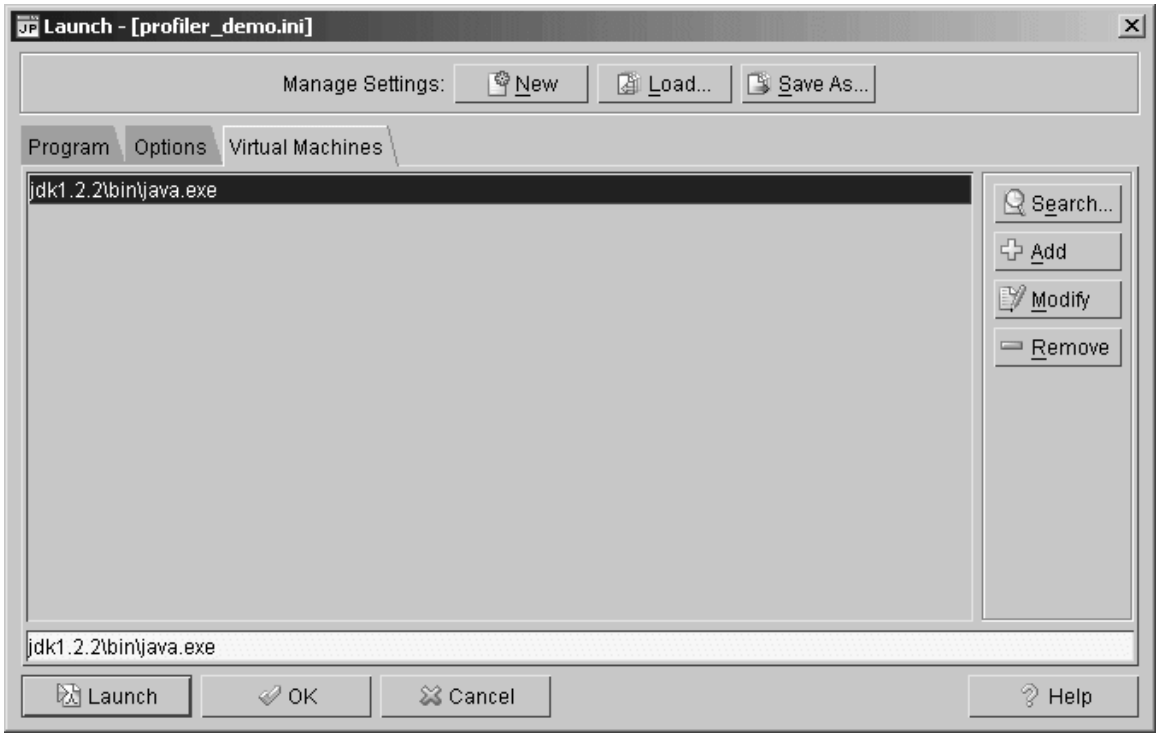

Listen
Profiler port:
5115
Source path:
Modify...
Disable Memory profiling
Disable CPU profiling
Auto start CPU profiling
Disable VM Info sampling
Maximum stack trace depth:
10
Listen
OK
Cancel
Help

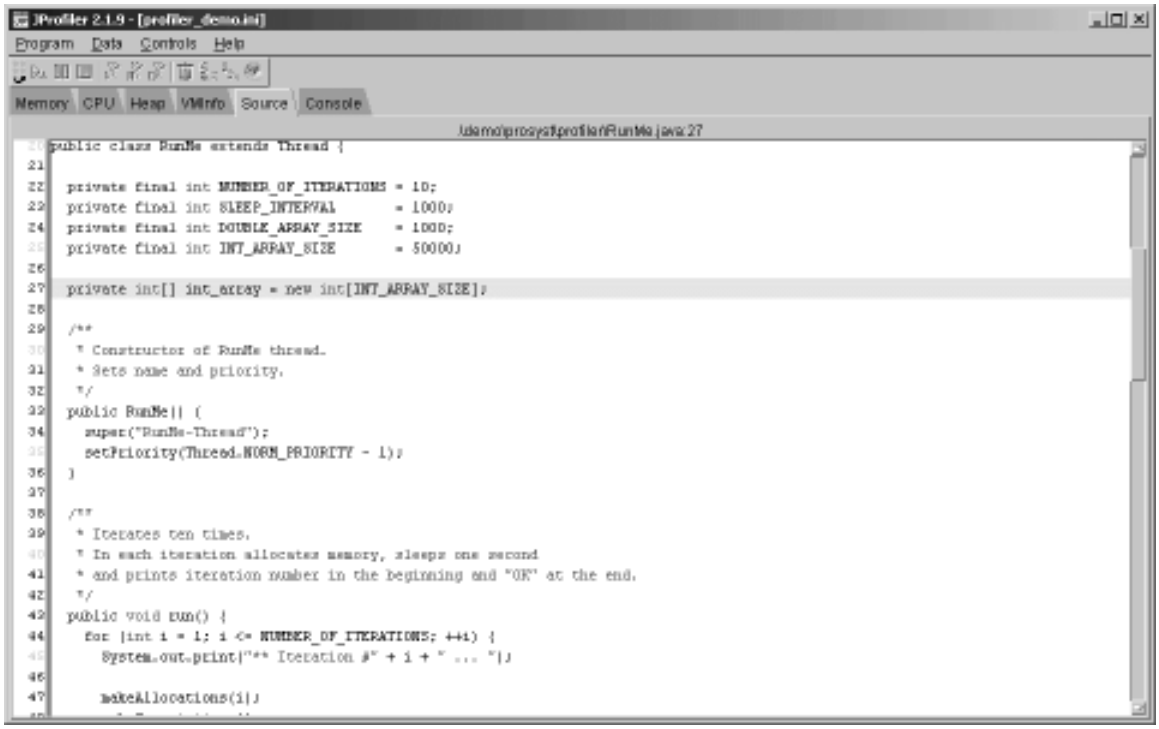

Attach
Profiler host:
localhost
Profiler port:
5115
Source path:
Modify...
Disable Memory profiling
Disable CPU profiling
Auto start CPU profiling
Disable VM Info sampling
Maximum stack trace depth:
10
Attach
OK
Cancel
Help

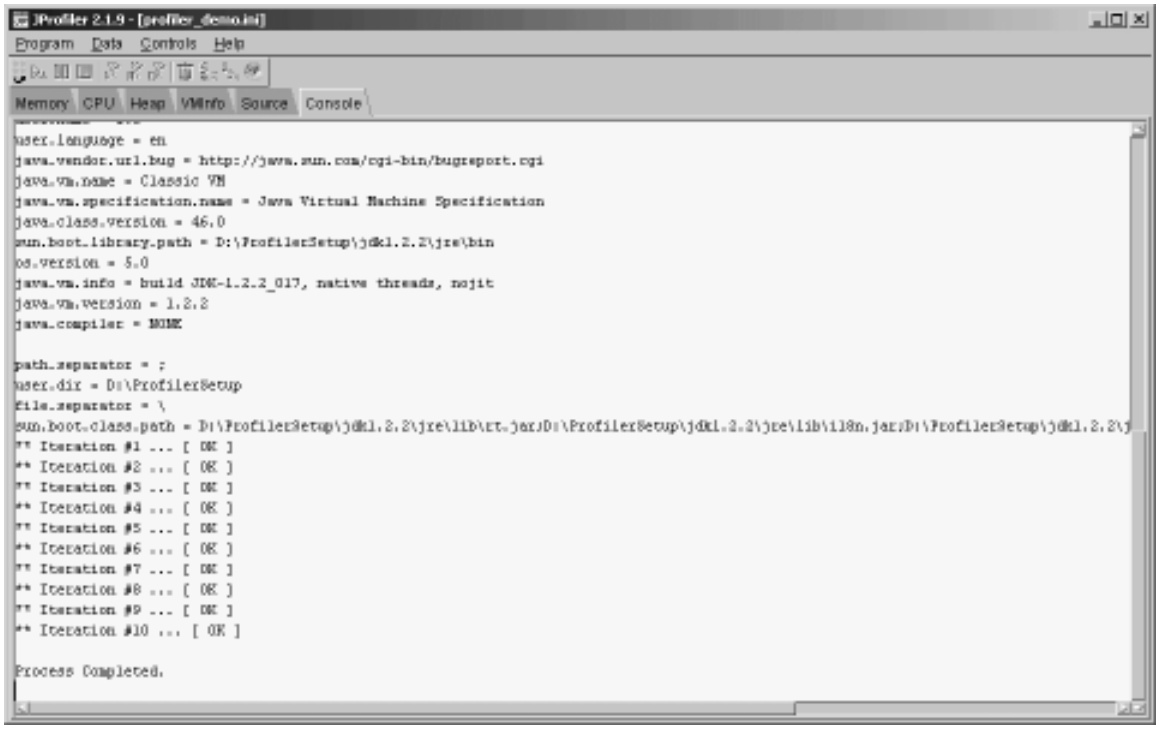

Program
Data
Controls
Help
Memory
CPU
Heap
VMInfo
Source
Console
system
main (445276 bytes, 3931 insts)
demo.prosyst.profiler.RunMe.main:111 (200172 bytes, 7 insts)
demo.prosyst.profiler.RunMe.<init>:27 (200004 bytes, 1 insts)
demo.prosyst.profiler.RunMe.<init>:34 (52 bytes, 3 insts)
demo.prosyst.profiler.RunMe.<init>:36 (40 bytes, 2 insts)
java.lang.ClassLoader.loadClassInternal:308 (91760 bytes, 1168 insts)
java.lang.System.initializeSystemClass:814 (25720 bytes, 16 insts)
java.lang.System.initializeSystemClass:816 (24840 bytes, 8 insts)
sun.io.Converters.newDefaultConverter:195 (22872 bytes, 1098 insts)
sun.io.CharacterEncoding.aliasName:675 (12596 bytes, 73 insts)
sun.io.Converters.getDefaultConverterClass:169 (12592 bytes, 522 insts)
classname
bytes
insts
diff bytes
diff insts
thread
method
int []
202680
6
202680
6
200004
1
200004
1
main
demo.prosyst.profiler.RunMe.<init>:27
2532
1
2532
1
main
java.lang.Character.<clinit>:1788
52
1
52
1
main
java.lang.Throwable.fillInStackTrace native method
36
1
36
1
main
java.lang.Throwable.fillInStackTrace native method
28
1
28
1
main
java.lang.Throwable.fillInStackTrace native method
28
1
28
1
main
java.lang.ClassLoader.findBootstrapClass native method
char []
126840
1440
126840
1440
double []
80040
10
80040
10
byte []
42656
136
42656
136
java.lang.Class
21840
156
21840
156
java.lang.String
15360
1280
15360
1280
object []
14060
73
14060
73
Class filters:
Method filters:
Show groups
Disable GC

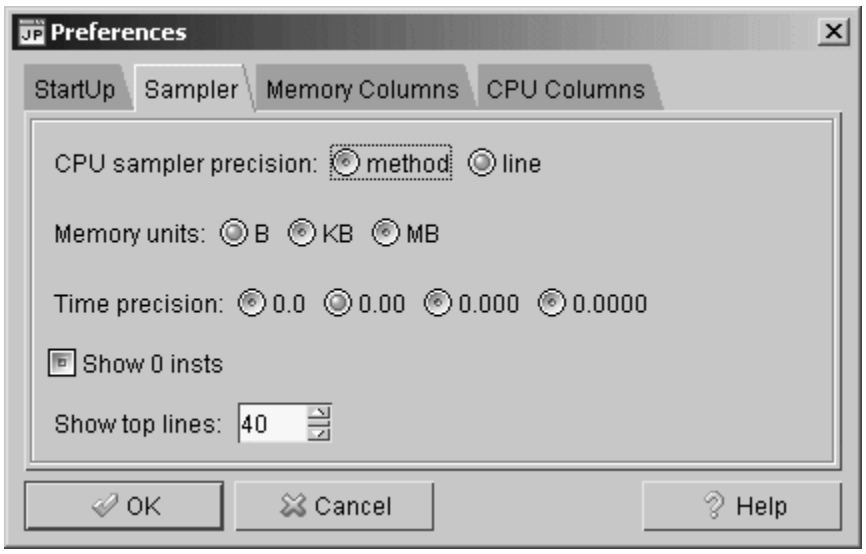

JProfiler 2.1.9 - [profiler_demo.jpi]
Program Data Controls Help
Memory CPU Heap VMInfo Source Console
system (1126 ms)
RunMe-Thread (98.63 %, 10969 ms)
demo.prosyst.profiler.RunMe.run 48 (98.29 %, 10932 ms)
demo.prosyst.profiler.RunMe.makeComputations 78 (98.29 %, 10932 ms)
java.lang.Math.random 333 (75.78 %, 8427 ms)
java.lang.Math.random unknown (6.14 %, 683 ms)
demo.prosyst.profiler.RunMe.run 47 (0.34 %, 37 ms)
demo.prosyst.profiler.RunMe.makeAllocations 63 (0.34 %, 37 ms)
main (1.36 %, 149 ms)
thread
method
time %
time ms
RunMe-Thread
java.util.Random.next unknown
38.56
4286
RunMe-Thread
java.util.Random.nextDouble 370
27.30
3037
RunMe-Thread
demo.prosyst.profiler.RunMe.makeComputations 78
16.38
1822
RunMe-Thread
java.lang.Math.random 333
6.14
683
RunMe-Thread
java.lang.Math.random unknown
6.14
683
RunMe-Thread
java.util.Random.nextDouble 371
3.75
417
main
java.io.FileOutputStream.writeBytes native method
0.68
75
main
java.io.FileOutputStream.writeBytes native method
0.34
37
RunMe-Thread
java.lang.ClassLoader.findBootstrapClass native method
0.34
37
main
java.util.Properties.load unknown
0.34
37
Method filters


JProfiler 2.1.9 - [profiler_demo.jpi]
Program Data Controls Help
Memory CPU Heap VMInfo Source Console
Class filters:
class loader
classes
insts
system class loader
byte []
11
system class loader
char []
856
sun.misc.Launcher$AppClassLoader@080...
demo.prosyst.profiler.RunMe
1
system class loader
double []
1
system class loader
int []
2
system class loader
java.io.BufferedInputStream
1
system class loader
java.io.BufferedOutputStream
2
system class loader
java.io.BufferedWriter
2
system class loader
java.io.ByteArrayOutputStream
1
system class loader
java.io.File
3
system class loader
java.io.FileDescriptor
5
Instances
Static fields
Super classes
int[32]@78ac8d04
int[50000]@a0000a05
Outgoing references
Incoming references
int[50000]@a0000a05
int_array of demo.prosyst.profiler.RunMe@78000a05
Allocation stack trace
demo.prosyst.profiler.RunMe.<init>:37
demo.prosyst.profiler.RunMe.main:111

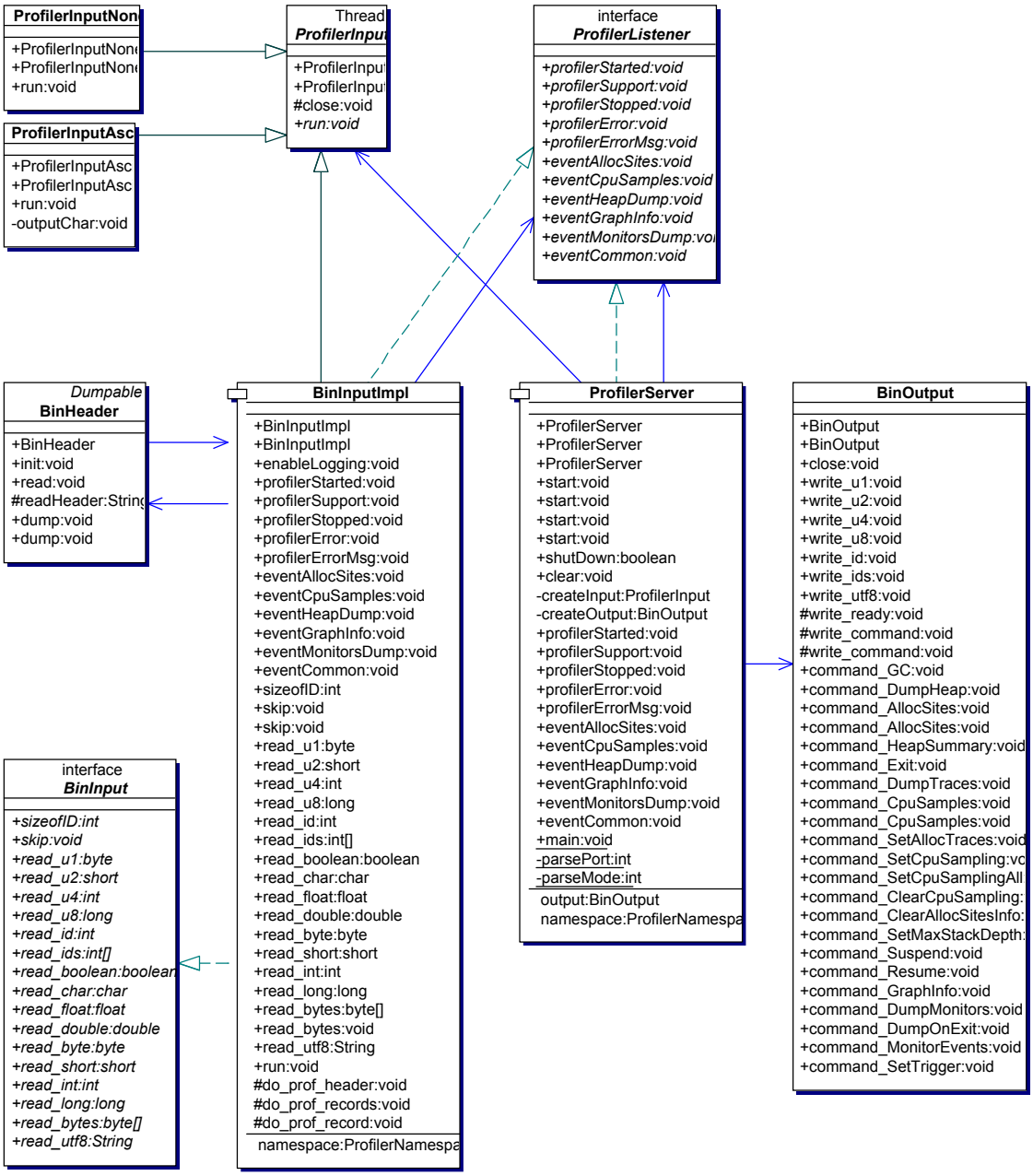

Launch - [profiler_demo.ini]
Manage Settings:
New
Load...
Save As...
Program
Options
Virtual Machines
jdk1.2.2\bin\java.exe
Search...
Add
Modify
Remove
jdk1.2.2\bin\java.exe
Launch
OK
Cancel
Help

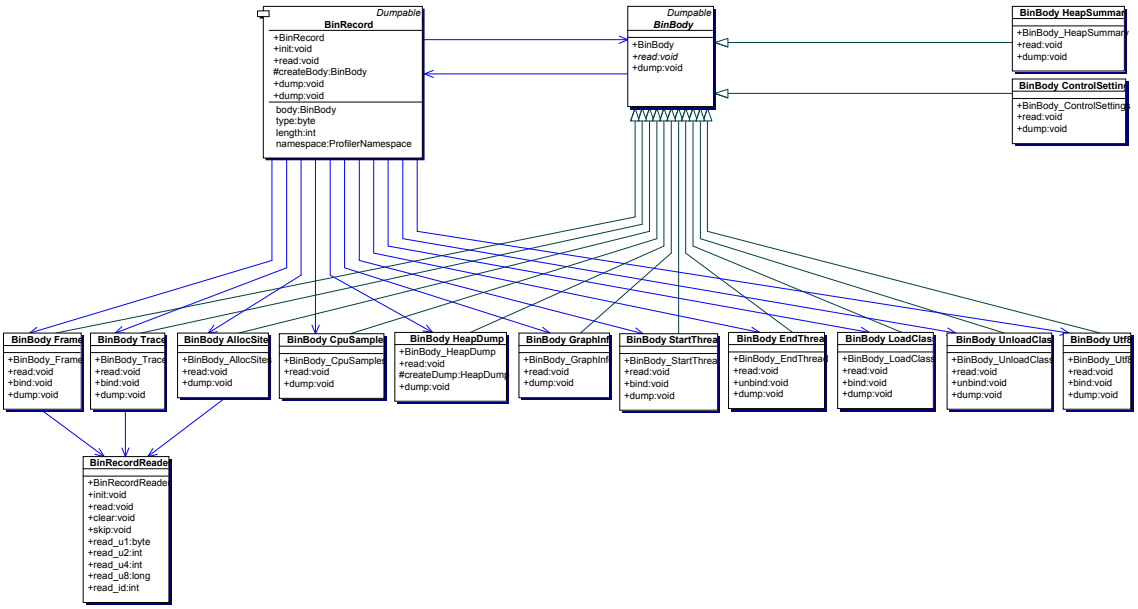

JProfiler 2.1.9 - [profiler_demo.ini]
Program
Data
Controls
Help
Memory
CPU
Heap
VMInfo
Source
Console
public class RunMe extends Thread {
private final int NUMBER_OF_ITERATIONS = 10;
private final int SLEEP_INTERVAL = 1000;
private final int DOUBLE_ARRAY_SIZE = 1000;
private final int INT_ARRAY_SIZE = 50000;
private int[] int_array = new int[INT_ARRAY_SIZE];
/**
* Constructor of RunMe thread.
* Sets name and priority.
*/
public RunMe() {
super("RunMe-Thread");
setPriority(Thread.NORM_PRIORITY - 1);
}
/**
* Iterates ten times.
* In each iteration allocates memory, sleeps one second
* and prints iteration number in the beginning and "OK" at the end.
*/
public void run() {
for (int i = 1; i <= NUMBER_OF_ITERATIONS; ++i) {
System.out.print("** Iteration #" + i + " ... ");
makeAllocations(i);

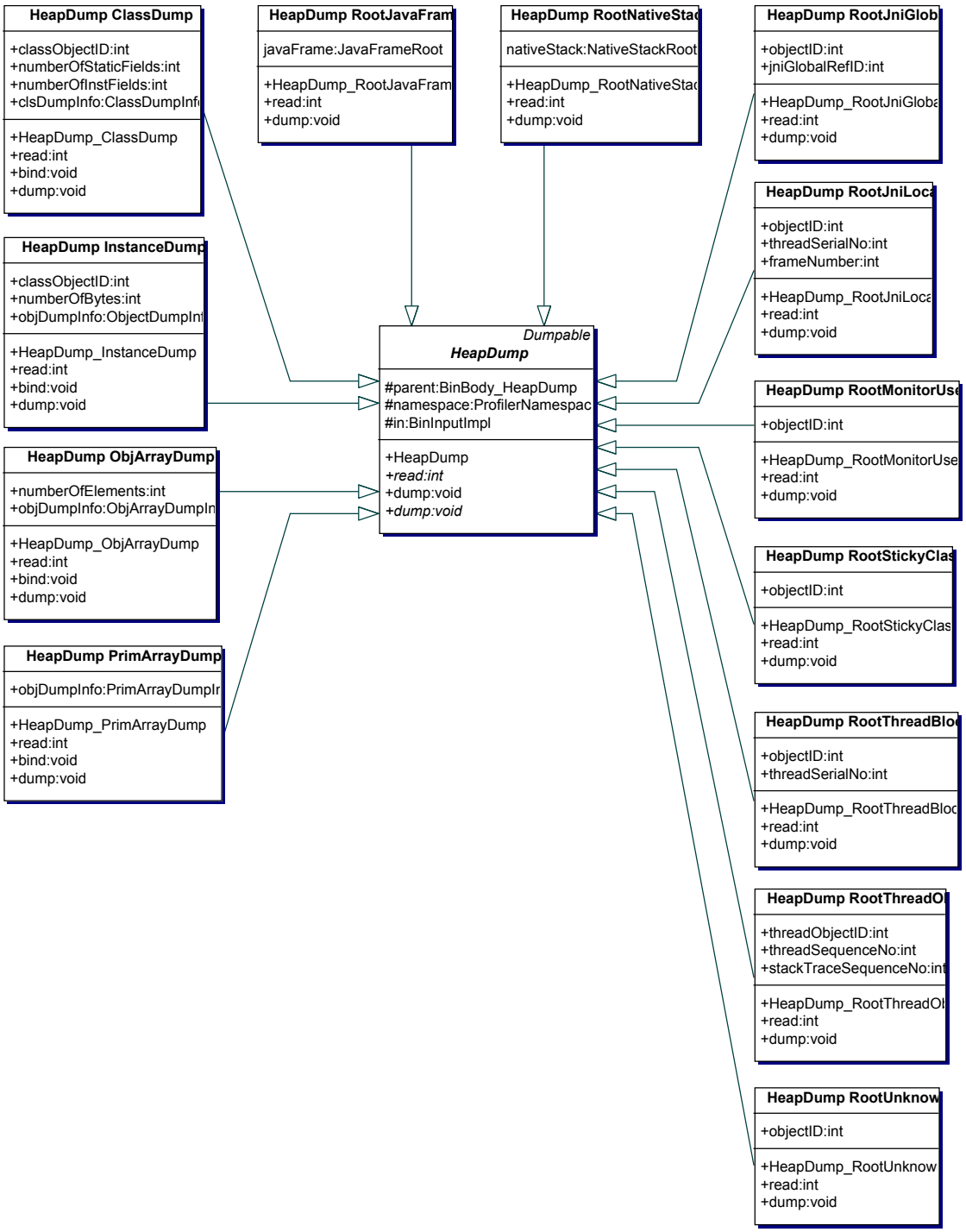

JProfiler 2.1.9 - [profiler_demo.ini]
Program Data Controls Help
Memory CPU Heap VMInfo Source Console
user.language = en
java.vendor.url.bug = http://java.sun.com/cgi-bin/bugreport.cgi
java.vm.name = Classic VM
java.vm.specification.name = Java Virtual Machine Specification
java.class.version = 46.0
sun.boot.library.path = D:\ProfilerSetup\jdk1.2.2\jre\bin
os.version = 5.0
java.vm.info = build JDK-1.2.2_017, native threads, nojit
java.vm.version = 1.2.2
java.compiler = NONE
path.separator = ;
user.dir = D:\ProfilerSetup
file.separator = \
** Iteration #1 ... [OK]
** Iteration #2 ... [OK]
** Iteration #3 ... [OK]
** Iteration #4 ... [OK]
** Iteration #5 ... [OK]
** Iteration #6 ... [OK]
** Iteration #7 ... [OK]
** Iteration #8 ... [OK]
** Iteration #9 ... [OK]
** Iteration #10 ... [OK]
Process Completed.

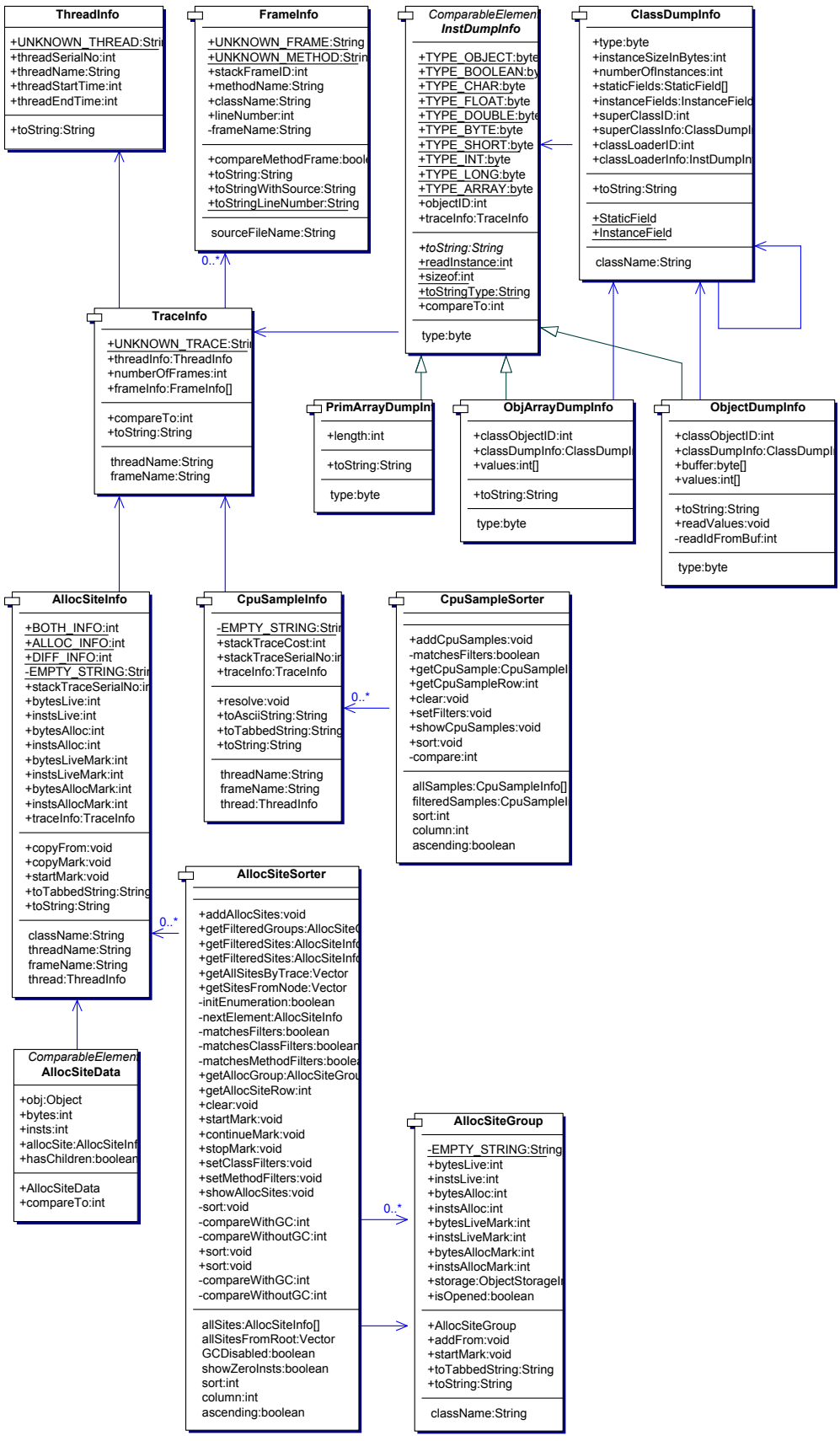

Preferences
StartUp
Sampler
Memory Columns
CPU Columns
CPU sampler precision: method line
Memory units: B KB MB
Time precision: 0.0 0.00 0.000 0.0000
Show 0 insts
Show top lines: 40
OK
Cancel
Help

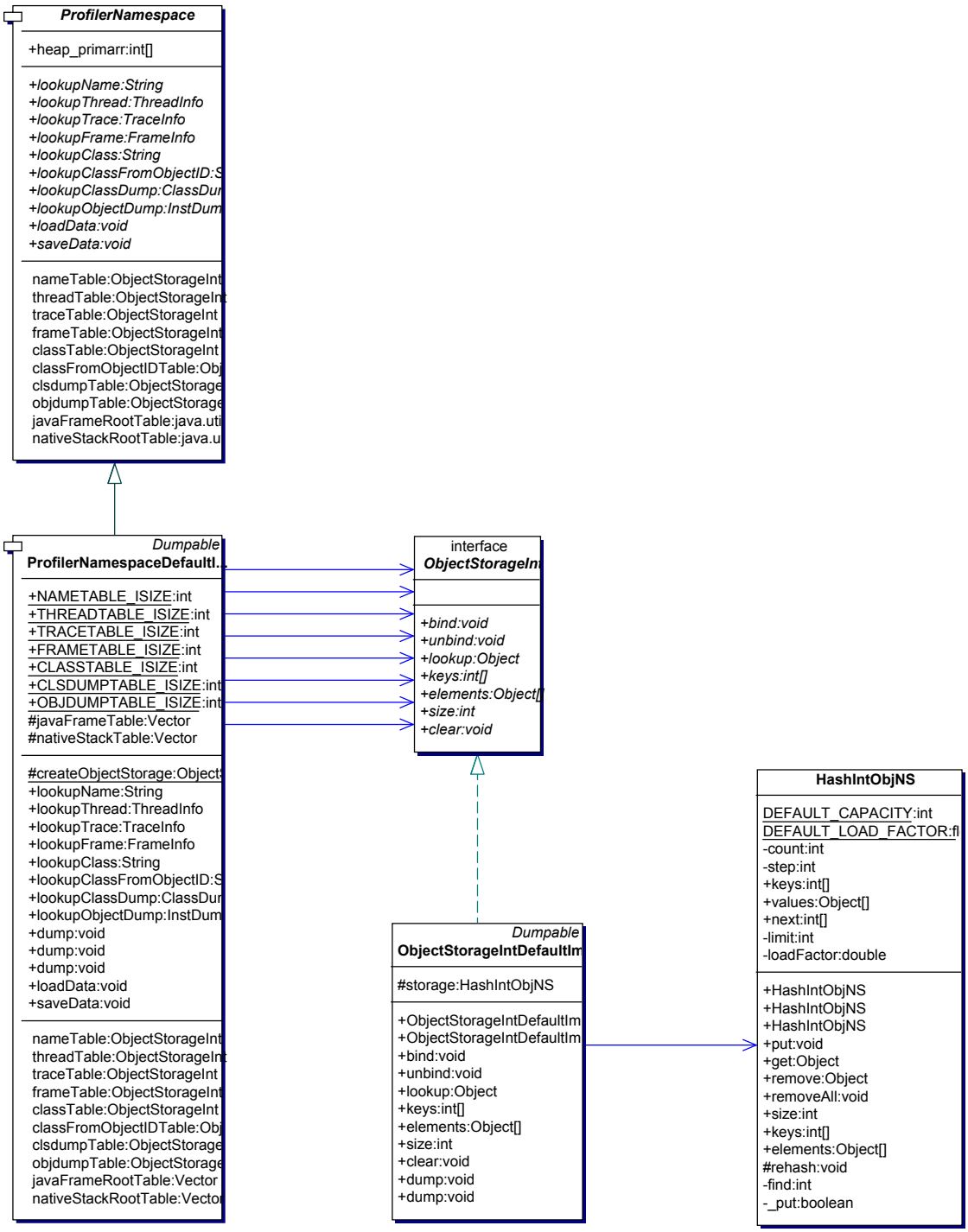
Приложение B (UML диаграми)

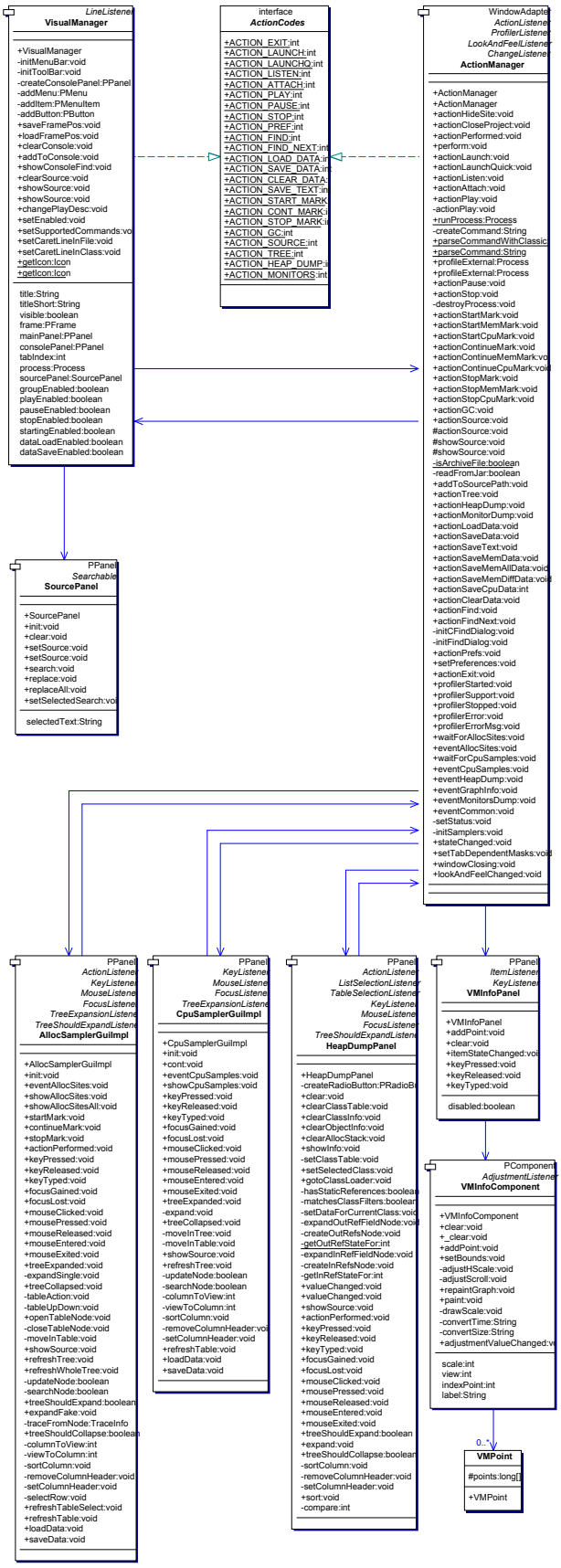


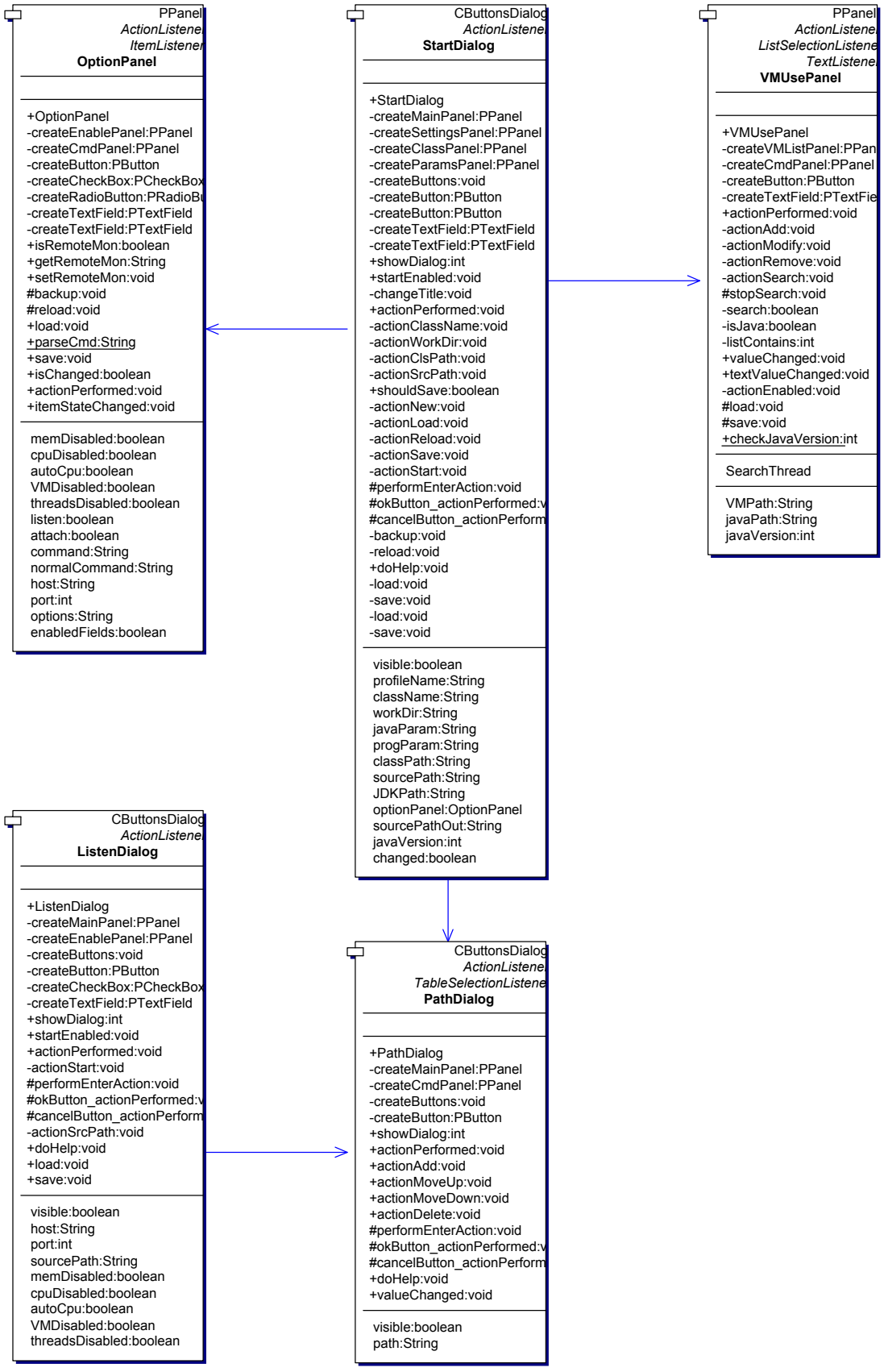

Dumpable
BinRecord
+BinRecord
+init:void
+read:void
#createBody:BinBody
+dump:void
+dump:void
body:BinBody
type:byte
length:int
namespace:ProfilerNamespace
Dumpable
BinBody
+BinBody
+read:void
+dump:void
BinBody HeapSummar
+BinBody_HeapSummary
+read:void
+dump:void
BinBody ControlSettin
+BinBody_ControlSettings
+read:void
+dump:void
BinBody Frame
+BinBody_Frame
+read:void
+bind:void
+dump:void
BinBody Trace
+BinBody_Trace
+read:void
+bind:void
+dump:void
BinBody AllocSite
+BinBody_AllocSite
+read:void
+dump:void
BinBody CpuSample
+BinBody_CpuSamples
+read:void
+dump:void
BinBody HeapDump
+BinBody_HeapDump
+read:void
#createDump:HeapDump
+dump:void
BinBody GraphInf
+BinBody_GraphInf
+read:void
+dump:void
BinBody StartThrea
+BinBody_StartThrea
+read:void
+bind:void
+dump:void
BinBody EndThrea
+BinBody_EndThrea
+read:void
+unbind:void
+dump:void
BinBody LoadClas
+BinBody_LoadClas
+read:void
+bind:void
+dump:void
BinBody UnloadClas
+BinBody_UnloadClas
+read:void
+unbind:void
+dump:void
BinBody Utf8
+BinBody_Utf8
+read:void
+bind:void
+dump:void
BinRecordReade
+BinRecordReade
+init:void
+read:void
+clear:void
+skip:void
+read_u1:byte
+read_u2:int
+read_u4:int
+read_u8:long
+read_id:int


HeapDump ClassDump
+classObjectID:int
+numberOfStaticFields:int
+numberOfInstFields:int
+clsDumpInfo:ClassDumpInf
+HeapDump_ClassDump
+read:int
+bind:void
+dump:void
HeapDump RootJavaFram
javaFrame:JavaFrameRoot
+HeapDump_RootJavaFram
+read:int
+dump:void
HeapDump RootNativeStac
nativeStack:NativeStackRoot
+HeapDump_RootNativeStac
+read:int
+dump:void
HeapDump RootJniGlob
+objectID:int
+jniGlobalRefID:int
+HeapDump_RootJniGloba
+read:int
+dump:void
HeapDump InstanceDump
+classObjectID:int
+numberOfBytes:int
+objDumpInfo:ObjectDumpInf
+HeapDump_InstanceDump
+read:int
+bind:void
+dump:void
Dumpable
HeapDump
#parent:BinBody_HeapDump
#namespace:ProfilerNamespac
#in:BinInputImpl
+HeapDump
+read:int
+dump:void
+dump:void
HeapDump RootJniLoca
+objectID:int
+threadSerialNo:int
+frameNumber:int
+HeapDump_RootJniLoca
+read:int
+dump:void
HeapDump RootMonitorUse
+objectID:int
+HeapDump_RootMonitorUse
+read:int
+dump:void
HeapDump ObjArrayDump
+numberOfElements:int
+objDumpInfo:ObjArrayDumpIn
+HeapDump_ObjArrayDump
+read:int
+bind:void
+dump:void
HeapDump RootStickyClas
+objectID:int
+HeapDump_RootStickyClas
+read:int
+dump:void
HeapDump PrimArrayDump
+objDumpInfo:PrimArrayDumpIr
+HeapDump_PrimArrayDump
+read:int
+bind:void
+dump:void
HeapDump RootThreadBloc
+objectID:int
+threadSerialNo:int
+HeapDump_RootThreadBloc
+read:int
+dump:void
HeapDump RootThreadOb
+threadObjectID:int
+threadSequenceNo:int
+stackTraceSequenceNo:int
+HeapDump_RootThreadOb
+read:int
+dump:void
HeapDump RootUnknow
+objectID:int
+HeapDump_RootUnknow
+read:int
+dump:void


ThreadInfo
+UNKNOWN_THREAD:Stri
+threadSerialNo:int
+threadName:String
+threadStartTime:int
+threadEndTime:int
+toString:String
FrameInfo
+UNKNOWN_FRAME:String
+UNKNOWN_METHOD:Strin
+stackFrameID:int
+methodName:String
+className:String
+lineNumber:int
-frameName:String
+compareMethodFrame:bool
+toString:String
+toStringWithSource:String
+toStringLineNumber:String
sourceFileName:String
0..*
ComparableElemen
InstDumpInfo
+TYPE_OBJECT:byte
+TYPE_BOOLEAN:by
+TYPE_CHAR:byte
+TYPE_FLOAT:byte
+TYPE_DOUBLE:byte
+TYPE_BYTE:byte
+TYPE_SHORT:byte
+TYPE_INT:byte
+TYPE_LONG:byte
+TYPE_ARRAY:byte
+objectID:int
+traceInfo:TraceInfo
+toString:String
+readInstance:int
+sizeof:int
+toStringType:String
+compareTo:int
type:byte
ClassDumpInfo
+type:byte
+instanceSizeInBytes:int
+numberOfInstances:int
+staticFields:StaticField[]
+instanceFields:InstanceField
+superClassID:int
+superClassInfo:ClassDumpI
+classLoaderID:int
+classLoaderInfo:InstDumpIn
+toString:String
+StaticField
+InstanceField
className:String
TraceInfo
+UNKNOWN_TRACE:Stri
+threadInfo:ThreadInfo
+numberOfFrames:int
+frameInfo:FrameInfo[]
+compareTo:int
+toString:String
threadName:String
frameName:String
PrimArrayDumpIn
+length:int
+toString:String
type:byte
ObjArrayDumpInfo
+classObjectID:int
+classDumpInfo:ClassDumpI
+values:int[]
+toString:String
type:byte
ObjectDumpInfo
+classObjectID:int
+classDumpInfo:ClassDumpI
+buffer:byte[]
+values:int[]
+toString:String
+readValues:void
-readIdFromBuf:int
type:byte
AllocSiteInfo
+BOTH_INFO:int
+ALLOC_INFO:int
+DIFF_INFO:int
-EMPTY_STRING:Stri
+stackTraceSerialNo:in
+bytesLive:int
+instsLive:int
+bytesAlloc:int
+instsAlloc:int
+bytesLiveMark:int
+instsLiveMark:int
+bytesAllocMark:int
+instsAllocMark:int
+traceInfo:TraceInfo
+copyFrom:void
+copyMark:void
+startMark:void
+toTabbedString:String
+toString:String
className:String
threadName:String
frameName:String
thread:ThreadInfo
0..*
CpuSampleInfo
-EMPTY_STRING:Stri
+stackTraceCost:int
+stackTraceSerialNo:in
+traceInfo:TraceInfo
+resolve:void
+toAsciiString:String
+toTabbedString:String
+toString:String
threadName:String
frameName:String
thread:ThreadInfo
0..*
CpuSampleSorter
+addCpuSamples:void
-matchesFilters:boolean
+getCpuSample:CpuSampleI
+getCpuSampleRow:int
+clear:void
+setFilters:void
+showCpuSamples:void
+sort:void
-compare:int
allSamples:CpuSampleInfo[]
filteredSamples:CpuSampleI
sort:int
column:int
ascending:boolean
AllocSiteSorter
+addAllocSites:void
+getFilteredGroups:AllocSite
+getFilteredSites:AllocSiteInf
+getFilteredSites:AllocSiteInf
+getAllSitesByTrace:Vector
+getSitesFromNode:Vector
-initEnumeration:boolean
-nextElement:AllocSiteInfo
-matchesFilters:boolean
-matchesClassFilters:boolean
-matchesMethodFilters:boole
+getAllocGroup:AllocSiteGrou
+getAllocSiteRow:int
+clear:void
+startMark:void
+continueMark:void
+stopMark:void
+setClassFilters:void
+setMethodFilters:void
+showAllocSites:void
-sort:void
-compareWithGC:int
-compareWithoutGC:int
+sort:void
+sort:void
-compareWithGC:int
-compareWithoutGC:int
allSites:AllocSiteInfo[]
allSitesFromRoot:Vector
GCDisabled:boolean
showZeroInsts:boolean
sort:int
column:int
ascending:boolean
0..*
ComparableElemen
AllocSiteData
+obj:Object
+bytes:int
+insts:int
+allocSite:AllocSiteInf
+hasChildren:boolean
+AllocSiteData
+compareTo:int
AllocSiteGroup
-EMPTY_STRING:String
+bytesLive:int
+instsLive:int
+bytesAlloc:int
+instsAlloc:int
+bytesLiveMark:int
+instsLiveMark:int
+bytesAllocMark:int
+instsAllocMark:int
+storage:ObjectStorageI
+isOpened:boolean
+AllocSiteGroup
+addFrom:void
+startMark:void
+toTabbedString:String
+toString:String
className:String


ProfilerNamespace
+heap_primarr:int[]
+lookupName:String
+lookupThread:ThreadInfo
+lookupTrace:TraceInfo
+lookupFrame:FrameInfo
+lookupClass:String
+loadData:void
+saveData:void
nameTable:ObjectStorageInt
threadTable:ObjectStorageInt
traceTable:ObjectStorageInt
frameTable:ObjectStorageInt
classTable:ObjectStorageInt
javaFrameRootTable:java.uti
nativeStackRootTable:java.u
Dumpable
ProfilerNamespaceDefaultI..
+NAMETABLE_ISIZE:int
+THREADTABLE_ISIZE:int
+TRACETABLE_ISIZE:int
+FRAMETABLE_ISIZE:int
+CLASSTABLE_ISIZE:int
+CLSDUMPTABLE_ISIZE:int
+OBJDUMPTABLE_ISIZE:int
#javaFrameTable:Vector
#nativeStackTable:Vector
+lookupName:String
+lookupThread:ThreadInfo
+lookupTrace:TraceInfo
+lookupFrame:FrameInfo
+lookupClass:String
+dump:void
+dump:void
+dump:void
+loadData:void
+saveData:void
nameTable:ObjectStorageInt
threadTable:ObjectStorageInt
traceTable:ObjectStorageInt
frameTable:ObjectStorageInt
classTable:ObjectStorageInt
javaFrameRootTable:Vector
nativeStackRootTable:Vector
interface
ObjectStorageInt
+bind:void
+unbind:void
+lookup:Object
+keys:int[]
+size:int
+clear:void
Dumpable
ObjectStorageIntDefaultIm
#storage:HashIntObjNS
+bind:void
+unbind:void
+lookup:Object
+keys:int[]
+size:int
+clear:void
+dump:void
+dump:void
HashIntObjNS
DEFAULT_CAPACITY:int
-count:int
-step:int
+keys:int[]
+values:Object[]
+next:int[]
-limit:int
-loadFactor:double
+HashIntObjNS
+put:void
+get:Object
+remove:Object
+removeAll:void
+size:int
+keys:int[]
+elements:Object[]
#rehash:void
-find:int
-_put:boolean


LineListener
VisualManager
+VisualManager
-initMenuBar:void
-initToolBar:void
-createConsolePanel:PPanel
-addMenu:PMenu
-addItem:PMenuItem
-addButton:PButton
+saveFramePos:void
+loadFramePos:void
+clearConsole:void
+addToConsole:void
+showConsoleFind:void
+clearSource:void
+showSource:void
+showSource:void
+changePlayDesc:void
+setEnabled:void
+setSupportedCommands:vo
+setCaretLineInFile:void
+setCaretLineInClass:void
+getIcon:Icon
+getIcon:Icon
title:String
titleShort:String
visible:boolean
frame:PFrame
mainPanel:PPanel
consolePanel:PPanel
tabIndex:int
process:Process
sourcePanel:SourcePanel
groupEnabled:boolean
playEnabled:boolean
pauseEnabled:boolean
stopEnabled:boolean
startingEnabled:boolean
dataLoadEnabled:boolean
dataSaveEnabled:boolean
interface
ActionCodes
+ACTION_EXIT:int
+ACTION_LAUNCH:int
+ACTION_LAUNCHQ:int
+ACTION_LISTEN:int
+ACTION_ATTACH:int
+ACTION_PLAY:int
+ACTION_PAUSE:int
+ACTION_STOP:int
+ACTION_PREF:int
+ACTION_FIND:int
+ACTION_FIND_NEXT:int
+ACTION_LOAD_DATA:int
+ACTION_SAVE_DATA:int
+ACTION_CLEAR_DATA:
+ACTION_SAVE_TEXT:in
+ACTION_START_MARK
+ACTION_CONT_MARK:i
+ACTION_STOP_MARK:i
+ACTION_GC:int
+ACTION_SOURCE:int
+ACTION_TREE:int
+ACTION_HEAP_DUMP:i
+ACTION_MONITORS:int
WindowAdapter
ActionListener
ProfilerListener
LookAndFeelListener
ChangeListener
ActionManager
+ActionManager
+ActionManager
+actionHideSite:void
+actionCloseProject:void
+actionPerformed:void
+perform:void
+actionLaunch:void
+actionLaunchQuick:void
+actionListen:void
+actionAttach:void
+actionPlay:void
-actionPlay:void
+runProcess:Process
-createCommand:String
+parseCommandWithClassic
+parseCommand:String
+profileExternal:Process
+profileExternal:Process
+actionPause:void
+actionStop:void
-destroyProcess:void
+actionStartMark:void
+actionStartMemMark:void
+actionStartCpuMark:void
+actionContinueMark:void
+actionContinueMemMark:vo
+actionContinueCpuMark:void
+actionStopMark:void
+actionStopMemMark:void
+actionStopCpuMark:void
+actionGC:void
+actionSource:void
#actionSource:void
#showSource:void
#showSource:void
-isArchiveFile:boolean
-readFromJar:boolean
+addToSourcePath:void
+actionTree:void
+actionHeapDump:void
+actionMonitorDump:void
+actionLoadData:void
+actionSaveData:void
+actionSaveText:void
+actionSaveMemData:void
+actionSaveMemAllData:void
+actionSaveMemDiffData:void
+actionSaveCpuData:int
+actionClearData:void
+actionFind:void
+actionFindNext:void
-initCFindDialog:void
-initFindDialog:void
+actionPrefs:void
+setPreferences:void
+actionExit:void
+profilerStarted:void
+profilerSupport:void
+profilerStopped:void
+profilerError:void
+profilerErrorMsg:void
+waitForAllocSites:void
+eventAllocSites:void
+waitForCpuSamples:void
+eventCpuSamples:void
+eventHeapDump:void
+eventGraphInfo:void
+eventMonitorsDump:void
+eventCommon:void
-setStatus:void
-initSamplers:void
+stateChanged:void
+setTabDependentMasks:void
+windowClosing:void
+lookAndFeelChanged:void
PPanel
Searchable
SourcePanel
+SourcePanel
+init:void
+clear:void
+setSource:void
+setSource:void
+search:void
+replace:void
+replaceAll:void
+setSelectedSearch:vo
selectedText:String
PPanel
ActionListener
KeyListener
MouseListener
FocusListener
TreeExpansionListene
TreeShouldExpandListene
AllocSamplerGuiImpl
+AllocSamplerGuiImpl
+init:void
+eventAllocSites:void
+showAllocSites:void
+showAllocSitesAll:void
+startMark:void
+continueMark:void
+stopMark:void
+actionPerformed:void
+keyPressed:void
+keyReleased:void
+keyTyped:void
+focusGained:void
+focusLost:void
+mouseClicked:void
+mousePressed:void
+mouseReleased:void
+mouseEntered:void
+mouseExited:void
+treeExpanded:void
-expandSingle:void
+treeCollapsed:void
-tableAction:void
-tableUpDown:void
+openTableNode:void
-closeTableNode:void
-moveInTable:void
+showSource:void
+refreshTree:void
+refreshWholeTree:void
-updateNode:boolean
-searchNode:boolean
+treeShouldExpand:boolean
+expandFake:void
-traceFromNode:TraceInfo
+treeShouldCollapse:boolean
-columnToView:int
-viewToColumn:int
-sortColumn:void
-removeColumnHeader:void
-setColumnHeader:void
-selectRow:void
+refreshTableSelect:void
+refreshTable:void
+loadData:void
+saveData:void
PPanel
KeyListener
MouseListener
FocusListener
TreeExpansionListene
CpuSamplerGuiImpl
+CpuSamplerGuiImpl
+init:void
+cont:void
+eventCpuSamples:void
+showCpuSamples:void
+keyPressed:void
+keyReleased:void
+keyTyped:void
+focusGained:void
+focusLost:void
+mouseClicked:void
+mousePressed:void
+mouseReleased:void
+mouseEntered:void
+mouseExited:void
+treeExpanded:void
-expand:void
+treeCollapsed:void
-moveInTree:void
-moveInTable:void
+showSource:void
+refreshTree:void
-updateNode:boolean
-searchNode:boolean
-columnToView:int
-viewToColumn:int
-sortColumn:void
-removeColumnHeader:void
-setColumnHeader:void
+refreshTable:void
+loadData:void
+saveData:void
PPanel
ActionListener
ListSelectionListener
TableSelectionListener
KeyListener
MouseListener
FocusListener
TreeShouldExpandListene
HeapDumpPanel
+HeapDumpPanel
-createRadioButton:PRadioB
+clear:void
+clearClassTable:void
+clearClassInfo:void
+clearObjectInfo:void
+clearAllocStack:void
+showInfo:void
-setClassTable:void
+setSelectedClass:void
+gotoClassLoader:void
-hasStaticReferences:boolean
-matchesClassFilters:boolean
-setDataForCurrentClass:void
-expandOutRefFieldNode:void
-createOutRefsNode:void
-getOutRefStateFor:int
-expandInRefFieldNode:void
-createInRefsNode:void
-getInRefStateFor:int
+valueChanged:void
+valueChanged:void
+showSource:void
+actionPerformed:void
+keyPressed:void
+keyReleased:void
+keyTyped:void
+focusGained:void
+focusLost:void
+mouseClicked:void
+mousePressed:void
+mouseReleased:void
+mouseEntered:void
+mouseExited:void
+treeShouldExpand:boolean
+expand:void
+treeShouldCollapse:boolean
-sortColumn:void
-removeColumnHeader:void
-setColumnHeader:void
+sort:void
-compare:int
PPanel
ItemListener
KeyListener
VMInfoPanel
+VMInfoPanel
+addPoint:void
+clear:void
+itemStateChanged:voi
+keyPressed:void
+keyReleased:void
+keyTyped:void
disabled:boolean
PComponent
AdjustmentListener
VMInfoComponent
+VMInfoComponent
+clear:void
+_clear:void
+addPoint:void
+setBounds:void
-adjustHScale:void
-adjustScroll:void
+repaintGraph:void
+paint:void
-drawScale:void
-convertTime:String
-convertSize:String
+adjustmentValueChanged:v
scale:int
view:int
indexPoint:int
label:String
0..*
VMPoint
#points:long[]
+VMPoint


PPanel
ActionListene
ItemListener
OptionPanel
+OptionPanel
-createEnablePanel:PPanel
-createCmdPanel:PPanel
-createButton:PButton
-createCheckBox:PCheckBox
-createRadioButton:PRadioBu
-createTextField:PTextField
-createTextField:PTextField
+isRemoteMon:boolean
+getRemoteMon:String
+setRemoteMon:void
#backup:void
#reload:void
+load:void
+parseCmd:String
+save:void
+isChanged:boolean
+actionPerformed:void
+itemStateChanged:void
memDisabled:boolean
cpuDisabled:boolean
autoCpu:boolean
VMDisabled:boolean
threadsDisabled:boolean
listen:boolean
attach:boolean
command:String
normalCommand:String
host:String
port:int
options:String
enabledFields:boolean
CButtonsDialog
ActionListene
StartDialog
+StartDialog
-createMainPanel:PPanel
-createSettingsPanel:PPanel
-createClassPanel:PPanel
-createParamsPanel:PPanel
-createButtons:void
-createButton:PButton
-createButton:PButton
-createTextField:PTextField
-createTextField:PTextField
+showDialog:int
+startEnabled:void
-changeTitle:void
+actionPerformed:void
-actionClassName:void
-actionWorkDir:void
-actionClsPath:void
-actionSrcPath:void
+shouldSave:boolean
-actionNew:void
-actionLoad:void
-actionReload:void
-actionSave:void
-actionStart:void
#performEnterAction:void
#okButton_actionPerformed:v
#cancelButton_actionPerform
-backup:void
-reload:void
+doHelp:void
-load:void
-save:void
-load:void
-save:void
visible:boolean
profileName:String
className:String
workDir:String
javaParam:String
progParam:String
classPath:String
sourcePath:String
JDKPath:String
optionPanel:OptionPanel
sourcePathOut:String
javaVersion:int
changed:boolean
PPanel
ActionListene
ListSelectionListene
TextListene
VMUsePanel
+VMUsePanel
-createVMListPanel:PPan
-createCmdPanel:PPanel
-createButton:PButton
-createTextField:PTextFie
+actionPerformed:void
-actionAdd:void
-actionModify:void
-actionRemove:void
-actionSearch:void
#stopSearch:void
-search:boolean
-isJava:boolean
-listContains:int
+valueChanged:void
+textValueChanged:void
-actionEnabled:void
#load:void
#save:void
+checkJavaVersion:int
SearchThread
VMPath:String
javaPath:String
javaVersion:int
CButtonsDialog
ActionListene
ListenDialog
+ListenDialog
-createMainPanel:PPanel
-createEnablePanel:PPanel
-createButtons:void
-createButton:PButton
-createCheckBox:PCheckBox
-createTextField:PTextField
+showDialog:int
+startEnabled:void
+actionPerformed:void
-actionStart:void
#performEnterAction:void
#okButton_actionPerformed:v
#cancelButton_actionPerform
-actionSrcPath:void
+doHelp:void
+load:void
+save:void
visible:boolean
host:String
port:int
sourcePath:String
memDisabled:boolean
cpuDisabled:boolean
autoCpu:boolean
VMDisabled:boolean
threadsDisabled:boolean
CButtonsDialog
ActionListene
TableSelectionListene
PathDialog
+PathDialog
-createMainPanel:PPanel
-createCmdPanel:PPanel
-createButtons:void
-createButton:PButton
+showDialog:int
+actionPerformed:void
+actionAdd:void
+actionMoveUp:void
+actionMoveDown:void
+actionDelete:void
#performEnterAction:void
#okButton_actionPerformed:v
#cancelButton_actionPerform
+doHelp:void
+valueChanged:void
visible:boolean
path:String

Приложение Г (компакт диск с дипломната работа)

Приложеният диск съдържа дипломната работа, програмния код, инсталационни програми и спецификации на използвания помощен софтуер. Директориите са разпределени както следва:

- `/Thesis` - текстът на дипломната работа и на резюметата на български и английски език
- `/Develop` - java файловете на проекта, картинките за менютата и лентата с инструментите, помощните библиотеки, скриптовете за компилиране и спецификацията на интерфейса JVMPI.
- `/ProfilerSetup` - скрипта за стартиране на JProfiler, jar файловете с класовете на проекта, динамичните библиотеки и конфигурационните файлове.
- `/ProfilerSetup/demo` - демонстрационен файл на възможностите на JProfiler
- `/ProfilerSetup/jdk1.2.2` - инсталация на Java 2 1.2.2_017 SDK