

# Dao DTrace

## Часть III

*Модест Матвеевич, это не диван!  
Это транслятор универсальных превращений!*

*Аркадий и Борис Стругацкие,  
«Понедельник начинается в субботу»*



**Евгений Ильин  
Филипп Торчинский**

**Нежданно-негаданно в контексте разговора о DTrace всплыла тема Digital Rights Management (DRM). В остальном продолжаем разговор о теории и практике DTrace. В теории вас ждет рассказ о спекулятивной трассировке, которая пригодится для анализа причин возникновения спорадических ошибок. Практика посвящена применению провайдеров для Java, PHP и баз данных.**

**В** продолжение цикла статей планировался обещанный ещё в первой части (см. №12 за 2007 г.) рассказ о том, как оснастить исходный код приложения датчиками DTrace. То есть продолжить разговор о различных типах провайдеров DTrace на примере sdt (statically defined tracing) и других провайдеров на базе sdt, например, hotspot и hotspot\_jni.

Однако в процессе сбора материалов для очередной статьи я заглянул на блог одного из авторов и ведущих разработчиков DTrace, Ада-

ма Левентала (Adam Leventhal, <http://blogs.sun.com/ahl>). Во второй половине января он опубликовал заметку о некоторых особенностях реализации DTrace в Mac OS X – «Mac OS X and the missing probes», которая получила очень большое количество откликов, потому что затронула тему, актуальную как для DTrace, так и для Open Source-сообщества. Рекомендую взглянуть в блог Адама всем тем, кого интересует DTrace в принципе. А тем, кто использует DTrace на Mac OS X, стоит прочесть упомянутую статью

просто непременно, если только вы этого уже не сделали.

В начале статьи я вкратце перескажу историю, поведанную Адамом. По этому поводу разговор о sdt отложим до лучших времён, а «теоретическая» часть статьи будет состоять из рассказа о методе вылавливания спорадических ошибок средствами, которые DTrace предоставляет в качестве помощи в этом непростом занятии. Практическая часть расскажет о том, как использовать DTrace для анализа приложений на Java, PHP и SQL.

## DTrace, DRM и шапка-невидимка

Адам уже писал о реализации DTrace на Mac OS X в заметках на своём блоге (к примеру, [http://blogs.sun.com/ahl/entry/dtrace\\_firefox\\_leopard](http://blogs.sun.com/ahl/entry/dtrace_firefox_leopard)). Дадим волю воображению и представим, как солнечным калифорнийским утром он наконец-то выкроил время, запустил iTunes, поставил любимую музыку и приступил к более пристальному изучению реализации своего детища на Mac OS X. В процессе экспериментов Адам обратил внимание на некоторые очень странные результаты. Запустив на двухядерном MacBook Pro скрипт:

```
profile-1000
{
    @ = count();
}

tick-1s
{
    printa(@);
    clear(@);
}
```

он с удивлением обнаружил, что profile-1000 срабатывает существенно меньшее количество раз вместо ожидаемых 2000 на процессоре с двумя ядрами.

В попытке пролить свет на это загадочное явление он вывел список всех работающих приложений при помощи команды:

```
# dtrace -n profile-997 '{ @[execname] = count(); }'
```

В данных, которые были выведены в результате работы команды, не оказалось ничего подозрительного. За исключением одной небольшой детали: iTunes не оказалось в списке, а в наушниках продолжала играть музыка как ни в чём не бывало!

Это навело его на мысль, что некоторые приложения каким-то образом могут быть «спрятаны» от ока DTrace. Разобраться можно было только посмотрев исходный код, и вот какой примечательный отрывок был обнаружен в теле функции dtrace\_probe():

```
#if defined(__APPLE__)
/*
 * If the thread on which this probe has fired
 * belongs to a process marked P_LNOATTACH then
 * this enabling is not permitted to observe it.
 * Move along, nothing to see here.
 */
if (ISSET(current_proc()->p_lflag, P_LNOATTACH)) {
    continue;
}
#endif /* __APPLE__ */
```

То есть процессы, которые помечены флагом P\_LNOATTACH, явным образом выводятся из под наблюдения.

Понятно, что этот код был вставлен для того, чтобы дать возможность тем, кто продаёт музыку или видео, защищать свои права. Беда в другом: способ, которым это было реализовано, слишком сильно затрагивает корневые механизмы средства трассировки. Настолько, что полученные в итоге результаты уже не являются полностью объективным отображением происходящего в системе. Наверное, возможны другие реализации, которые могут и защитить права продавцов, и не повлиять на объектив-

ность получаемых результатов. Что поделаешь — дополнительные граничные условия у задач встречаются, увы, довольно часто.

## Немного о поиске швейных принадлежностей в больших кучах высушенной травы

Вряд ли кто будет сомневаться в том, что умение выдавать подробную информацию о системе является неоспоримым достоинством DTrace. В некоторых случаях даже настолько подробную, что во множестве произошедших событий информацию, которую действительно нужно проанализировать, надо будет искать как иголку в стоге сена. Что делать, например, в тех случаях, когда необходимо выявить причину проблемы, которая проявляется не всегда, а только в некоторых случаях, и когда условия возникновения сбоя не ясны? Или, что даже хуже, наоборот, в заведомо ошибочной ситуации вместо сообщения об ошибке система работает как ни в чём не бывало?

Понятно, что надо трассировать все события, которые могут представлять интерес, как и понятно то, что надо постараться ограничить вывод только лишь теми данными, которые представляют непосредственный интерес для дальнейшей отладки. DTrace, а точнее его архитектура и выразительные средства языка D, предоставляют несколько механизмов ограничения возможного вывода. Первый из них — это, несомненно, предикаты, о которых рассказывалось в первой части статьи. Вкупе с thread-local-переменными, о которых тоже уже была речь, предикаты предоставляют хороший способ фильтрации вывода.

Допустим, мы имеем дело с ситуацией, когда некоторый системный вызов время от времени не срабатывает, выдавая код какой-то стандартной ошибки (EIO или EINVAL) по причине, зависящей не то от фаз луны, не то от направления или силы ветра в данный момент. И скажем для примера, что соотношение корректных обработок вызова к ошибочным находится в соотношении двести к одному. В подобных случаях имеет смысл рассмотреть с чувством, толком и расстановкой стек вызовов функций между входом и выходом из системного вызова. Это один из классических вариантов анализа, который позволяет выявить условия возникновения ошибки.

Каким образом это можно сделать при помощи DTrace? Для большей наглядности давайте попробуем смоделировать ситуацию поломки физического устройства ввода-вывода. Поскольку под рукой оказался DVD с последним OpenSolaris Express Community Edition, то эксперимент будем проводить на нём. Вставим диск, подождём, пока он подмонтируется, и симулируем сбой диска следующей последовательностью команд:

```
# ls /media/SOL_11_X86/* & (sleep 1 && eject -f cdrom)
```

Если у вас есть внешний диск под управлением ZFS и немного смелости (ZFS более устойчива к сбоям, которые могут возникнуть в процессе brutальных экспериментов, но если смелость в избытке, то можно рискнуть данными на флешке под FAT), можно прибавить реализма, аккуратно выдернув шнур USB в процессе чтения с диска. В пер-

вой части статьи приводился пример использования провайдера fbt, который как раз и пригодится для анализа подобной ситуации. Немного модифицируем его.

Во-первых, заменим системный вызов ioctl на open в первой версии скрипта; во-вторых, при помощи предиката ограничим вывод только теми событиями, которые связаны с запуском команды ls, а в-третьих, выделим вывод при возникновении ошибки:

```
#!/usr/bin/dtrace -s

#pragma D option flowindent

syscall::open:entry
/ execname == "ls" /
{
    self->follow = 1;
    printf( "open (%s)\n ", ␣
        copyinstr(arg0) );
}

fbt:::
/self->follow/
{ }

syscall::open:return
/self->follow && errno == 0/
{
    self->follow = 0;
}

syscall::open:return
/self->follow && errno != 0/
{
    self->follow = 0;
    printf( "ERROR" )
    exit(0);
}
```

Данная программа на D скорее всего выдаст многокилобайтовый отчёт, в котором интересующий нас стек вызовов будет в самом конце. Хорошо, если нам повезёт и та единственная причина, по которой происходит ошибка, будет выявлена сразу. А что делать, если окажется, что причина не одна и последний стек вызовов будет каждый раз отличаться от предыдущего? И кто знает, может, причины возникновения сбоя имеют регулярный характер по времени или количеству вызовов. Тогда ничего не останется, как убрать действие exit(0) и анализировать ещё более длинный вывод. Неплохой способ, в таких случаях го-

ворят, что grep и less в помощь, а также флаг и палки от барабана.

Если вы запустите этот скрипт на D, то скоро увидите, что выполнение остановится очень быстро и скорее всего это произойдет так же быстро, как и на той системе, где проверялись примеры для этой статьи. Дело в том, что команда ls использует динамически подключаемые библиотеки. Следовательно, при запуске команды начинает работу динамический линковщик, который смотрит некий стандартный набор файлов, в которых могут храниться параметры его конфигурации. Одно из таких мест в файловой системе – это /var/ld/ld.config. Такого может не оказаться в системе, и поэтому в итоге хоть и получаем ошибку при исполнении системного вызова (попытка открыть несуществующий файл), но это вполне штатный случай, который ничего не расскажет о том, что же на самом деле произошло.

Немного отвлекаясь от темы, в качестве упражнения для тех, кто только начинает изучать Solaris и DTrace, хочу предложить такую задачу: попробуйте модифицировать предложенный скрипт так, чтобы он мог трассировать не только системный вызов open, но и остальные системные вызовы тоже. Вывод имён открываемых файлов должен остаться. Получившийся в итоге скрипт позволит вам увидеть, что происходит на уровне системных вызовов при неполадках с устройствами ввода-вывода.

Какие ещё могут быть методы поиска иголки в стоге сена? Можно добавить в компоненту датчика fbt код, который будет сохранять в предопределённых динамических массивах информацию о вызовах и выводить её только в том случае, когда сам системный вызов отработает с ошибкой. Это тоже хороший способ, но и в этом случае и флаг, и палки от барабана лишними не окажутся потому, что, как минимум, не избежать дополнитель-

ной работы с форматированием вывода. Да и стоит ли изобретать велосипед, если DTrace обладает штатным механизмом, который называется спекулятивной трассировкой?

## Спекулятивная трассировка

Я хочу лишний раз подчеркнуть, что оба рассмотренных метода: обработка вывода Dtrace другими утилитами (см. №1 за 2008 год) и буферизация данных при помощи переменных – не являются ущербными. И в некоторых случаях могут оказаться необходимым подспорьем. Некоторая ирония в рассказе о них вызвана тем, что в DTrace предусмотрена возможность «припрятать» трассировочные данные до поры до времени, чтобы позже принять решение: стоит отдать собранные данные в буфер трассировки или же просто выбросить. Собственно, второй из ранее рассмотренных методов как раз и является неким доморощенным вариантом такого буфера. Однако, используя спекулятивную трассировку, получить интересующий вывод без последующей обработки и с минимальными затратами будет значительно проще.

Вкратце принципы спекулятивной трассировки можно описать так: выделяется специальный буфер, именуемый спекулятивным, в который будет заноситься вся информация, которая может пригодиться в дальнейшем, а может и нет; затем данные из этого буфера либо выкидываются совсем, либо переносятся в главный буфер, откуда и попадают на вывод. Интерфейс спекулятивных функций DTrace выглядит следующим образом (см. таблицу 1).

## speculation

Теперь опишу процесс более подробно. Функция speculation() выделяет спекулятивный буфер и возвращает его идентификатор, который должен использоваться в последующих вызовах функции speculate(). Спекулятивные буферы являются конечным ресурсом, и если при очередном вызове speculation() не окажется доступного буфера, то функция вернёт нулевой идентификатор, DTrace увеличит счёт ошибок и известит пользователя об этом факте примерно таким сообщением:

Таблица 1. Интерфейсные функции спекулятивной трассировки

Имя функции	Аргументы	Описание
speculation	Нет	Возвращает идентификатор нового спекулятивного буфера
speculate	Идентификатор	Указывает, что последующий вывод компоненты будет перенаправлен в спекулятивный буфер с указанным идентификатором
commit	Идентификатор	Заносит данные из спекулятивного буфера, связанного с указанным идентификатором, в главный буфер
discard	Идентификатор	Сбрасывает данные, прежде занесенные в спекулятивный буфер, связанный с указанным идентификатором

```
#dtrace: 2 failed speculations (no speculative buffer space available)
```

Нулевой идентификатор, хоть и является всегда недопустимым, может быть передан в качестве параметра функциям `speculate()`, `commit()` или `discard()`. Количество спекулятивных буферов по умолчанию равно единице, но может быть увеличено. Для этого надо увеличить значение параметра `nspec`, установив его значение в требуемое количество буферов. Это можно сделать прямо в скрипте при помощи прагмы:

```
#pragma D option nspec=3
```

## speculate

Для использования спекулятивного буфера идентификатор, возвращенный вызовом функции `speculation()`, обязан быть передан в качестве аргумента функции `speculate()` до того момента, как в компоненте будут выполняться какие-либо действия, связанные с записью данных. Все последующие действия подобного характера в компоненте, содержащей вызов `speculate()`, будут трассироваться спекулятивно. Если же вызову `speculate()` в коде компоненты будут предшествовать какие-либо действия записи данных, то компилятор с языка D выдаст ошибку времени компиляции. Таким образом, компоненты могут быть предназначены либо для спекулятивной, либо для неспекулятивной трассировки, но не для обоих видов трассировки одновременно.

Из вышеизложенного факта очевидным образом следует, что деструктивные действия не могут использоваться при спекулятивной трассировке. Ведь они существуют для того, чтобы предоставить возможность модификации системы, что суть непосредственное изменение данных, которые влияют на состояние системы. Чуть менее очевиден тот факт, что по тем же причинам это относится к агрегирующим действиям и действию `exit`. Попытка использования любого из этих трёх типов действий в компоненте, содержащей вызов функции `speculate()`, опять же приведёт к ошибке времени компиляции. Завершая описание `speculate()`, стоит еще отметить, что в каждой компоненте не может присутствовать более одного вызова этой функции (что, как вы думаете, произойдёт, если в некоторой компоненте обнаружится несколько вызовов `speculate()`?).

## commit

Разобрались, как происходит сбор данных, теперь о том, что с ними можно сделать в дальнейшем. Собственно, варианта всего два: либо эти данные попадут в вывод, либо будут отброшены. Для первого варианта используется функция `commit()`, которая копирует данные из спекулятивного буфера в главный. Если случилось так, что в спекулятивном буфере данных больше, чем места в главном буфере, то никакие данные не копируются, а соответствующий счётчик ошибок увеличивается на единицу. (В первой части статьи рассказывалось о том, что на системах с несколькими CPU для каждого из них выделяется отдельный буфер для хранения данных, полученных при трассировке; то же самое относится и к спекулятивным буферам.) В случае, когда при спекулятивной трассировке были использо-

ваны буферы, принадлежащие нескольким CPU, незамедлительно копируются только спекулятивные данные из буфера того CPU, на котором была вызвана функция `commit()`, а информация из буферов других CPU – только по прошествии некоторого времени. Таким образом, данные из буфера одного из CPU и остальных будут разнесены по времени при переносе в главный, однако это время гарантированно не больше, чем значение параметра настройки DTrace, который называется `cleanrate` (по умолчанию – `cleanrate=101`). При необходимости его можно изменить, увеличив частоту, с которой происходит очистка буферов CPU. Как и в случае с `nspec`, это можно сделать или при помощи прагмы, или из командной строки:

```
ostap#dtrace -x cleanrate=303 -s speculation.d
```

Стоит рассказать про один подводный камень, на который можно натолкнуться при использовании спекулятивной трассировки на системе с несколькими CPU. До тех пор пока на каждом CPU не будут переписаны данные в главный буфер из спекулятивного, последний будет недоступен для использования функциями `commit()`, `discard()` и `speculate()`. В этом случае вызовы этих функций тихо «завалются». Последнее, что стоит сказать про `commit()`, это то, что компонента, в которой присутствует вызов этой функции, не может содержать действия, записывающие данные. Для того чтобы обеспечить слияние нескольких спекулятивных буферов в одной компоненте, может быть несколько вызовов `commit()`.

## discard

Осталась последняя функция – `discard()`, которая предназначена для чистки спекулятивного буфера, если информация, находящаяся в данном буфере, не представляет интереса. Поскольку можно сказать, что функционально `commit()` – это чистка спекулятивных буферов плюс что-то еще (копирование данных), то всё, что относилось к «первому слагаемому» `commit()`, также справедливо и для `discard()`. И если, к примеру, все спекулятивные буферы заняты и `discard()` не успеет отработать до очередного вызова функции `speculation()`, то DTrace выдаст сообщение об ошибке наподобие такого:

```
#dtrace: 1 failed speculation (available buffer(s) still busy)
```

Изменяя значение параметра `cleanrate`, можно избавиться от этого сообщения или же сделать так, чтобы оно появлялось реже. Стоит заметить, что изменение параметров `cleanrate` и `nspec` увеличивает количество ресурсов для DTrace как приложения, увеличивая нагрузку на систему. Теперь можно вернуться к ранее приведенному скрипту и переписать его с использованием спекулятивной трассировки.

```
#!/usr/bin/dtrace -s

#pragma D option flowindent

syscall::open:entry
/ execname == "ls" /
{
    self->spec = speculation();
    printf( "open (%s)\n ", copyinstr(arg0) );
}
```



```

}

fibt:::
/self->spec/
{
    speculate(self->spec);
}

syscall::open:return
/self->spec/
{
    speculate( self->spec );
    trace( errno );
}

syscall::open:return
/self->spec && errno == 0/
{
    discard( self->spec );
    self->spec= 0;
}

syscall::open:return
/self->spec && errno != 0/
{
    commit( self->spec );
    self->spec = 0;
}

```

После подробного рассказа вряд ли стоит подробно комментировать изменения. Ограничусь только одним замечанием. В этом примере идентификатор спекулятивного буфера присваивается thread-local переменной, которая впоследствии фигурирует в качестве аргумента в предикатах. Это один из распространённых шаблонов при написании скриптов на D.

## Вглубь Java

Мы уже знаем, что с помощью DTrace можно выяснить, какие конкретно функции вызываются, в каком порядке, какие аргументы передаются и сколько времени потрачено на их выполнение. Все это могло воодушевить системных администраторов и разработчиков на языках C и C++, а интересы программистов на Java оставались в стороне. В этом разделе мы рассмотрим, как можно использовать DTrace для отладки приложений на Java.

Для предоставления такой возможности Sun Microsystems встроила в код JVM (начиная с JDK 6.0) датчики двух провайдеров – hotspot и hotspot\_jni. В JDK 5.0 был провайдер dvm. Для тех, кто привык им пользоваться, есть хорошая новость – датчики провайдера hotspot носят такие же имена, как и датчики провайдера dvm.

Провайдер hotspot имеет ряд датчиков, с помощью которых можно отслеживать запуск и работу сборщика мусора в JVM (garbage collector). Если количество запусков сборщика мусора растёт (особенно если быстро растёт) во время работы приложения или время работы самого приложения постоянно увеличивается, это верный признак утечки памяти в приложении.

Провайдер hotspot\_jni требуется для отслеживания событий, связанных с обращениями через JNI (Java native interface – интерфейс Java с машинным кодом, т.е. ранее написанными и скомпилированными методами, реализованными на языке C). Строго говоря, это может быть программа не только на языке C, ибо JNI описывает интерфейс между программой на Java и машинным кодом и в нём не указано, компилятор с какого языка должен создать этот код.

Уже было упомянуто, что провайдеры hotspot и hotspot\_

jni основаны на провайдере sdt. На их датчики можно ссылаться только с упоминанием идентификатора процесса самой машины Java (JVM). Вот пример скрипта, который показывает частоту вызова GC:

```

hotspot$target::gc-begin
{
    printf("GC called at %Y\n", walltimestamp);
}

```

Если запустить выполнение программы на java, скажем, демонстрационный пример /usr/jdk/instances/jdk1.6.0/demo/jfc/Java2D/Java2Demo.jar, а затем этот скрипт, то он покажет, в какие моменты запускался сборщик мусора:

```

# java -jar /usr/jdk/instances/jdk1.6.0/demo/jfc/Java2D/Java2Demo.jar &

```

```
[1] 1147
```

```

# dtrace -qn 'hotspot$target::gc-begin
{
    printf("GC called at %Y\n", walltimestamp);
}' -p 1147

```

```

GC called at 2008 Feb 7 01:50:15
GC called at 2008 Feb 7 01:50:16
GC called at 2008 Feb 7 01:50:17
GC called at 2008 Feb 7 01:50:18
GC called at 2008 Feb 7 01:50:19
GC called at 2008 Feb 7 01:50:20
GC called at 2008 Feb 7 01:50:21
GC called at 2008 Feb 7 01:50:25

```

А что если нам надо выяснить, какие методы вызываются в приложении Java? Возьмем вот такое приложение:

```

# cat Greeting.java
public class Greeting {
    public void greet() {
        System.out.println("Hello DTrace!");
    }
}

# cat TestGreeting.java
public class TestGreeting {
    public static void main(String[] args) {
        Greeting hello = new Greeting();
        while (true) {
            hello.greet();
            try {
                Thread.currentThread().sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}

```

Скомпилируем его:

```
javac TestGreeting.java
```

Теперь надо запустить получившееся приложение (мирно лежащее в TestGreeting.class). Однако прежде надо разобраться, как включать датчики в Java SE 6. По умолчанию в виртуальной машине Java доступен лишь небольшой набор датчиков – тех, которые практически не влияют на производительность. Эти датчики позволяют отследить не все события, а только сборку мусора, компиляцию методов, создание нового потока команд и загрузку класса.

Вызов метода, например, в этот список не входит, потому что соответствующий датчик оказывает значительное влияние на производительность и оттого по умолчанию недоступен. К таким датчикам, кроме датчиков вызова методов, также относятся датчики создания объектов и датчики событий, связанных с мониторами Java. Чтобы сделать такой датчик доступным и «видимым» для DTrace, надо запустить Java с ключами, разрешающими использование этих датчиков, или (если JVM уже запущена) использовать команду jinfo для управления JVM.

Доступны следующие ключи при запуске java:

- -XX:+DTraceAllocProbes
- -XX:+DTraceMethodProbes
- -XX:+DTraceMonitorProbes
- -XX:+ExtendedDTraceProbes

Первые три ключа разрешают использование датчиков создания объектов, вызова методов и датчиков событий, связанных с мониторами Java соответственно, а четвертый разрешает использование всех датчиков в JVM. Фактическое влияние включенных датчиков на производительность может быть различным в зависимости от частоты вызова кода, содержащего встроенный датчик.

Запускаем наше приложение:

```
# java -XX:+ExtendedDTraceProbes TestGreeting &

[1] 1448
Hello DTrace!
Hello DTrace!
...
```

Возьмем скрипт hs.d, вот такой:

```
# cat hs.d
hotspot$target:::method-entry
{
    printf("%s.%s %s\n", copyinstr(arg1, arg2), " ",
        copyinstr(arg3, arg4), copyinstr(arg5, arg6));
}

tick-5ms
{
    exit(0);
}
```

Конструкция hotspot\$target означает, что вместо \$target будет подставлен аргумент, переданный с ключом -r при запуске DTrace (это будет идентификатор процесса Java).

Из описания датчиков провайдера hotspot (<http://java.sun.com/javase/6/docs/technotes/guides/vm/dtrace.html>) известно, что аргументы датчика method-entry следующие (см. таблицу 2).

Сигнатура метода – это совокупность имени функции, типа возвращаемого значения и списка аргументов с указанием порядка их следования и типов. Подробнее об обозначениях, применяемых в сигнатурах, можно прочесть, например, в разделе Type Signatures в спецификации JNI, размещенной по адресу <http://java.sun.com/j2se/1.3/docs/guide/jni/spec/types.doc.html>.

Наш скрипт должен при каждом срабатывании датчика method-entry, т.е. при каждом вызове любого метода в нашем приложении на Java, выводить имя класса, имя метода и подпись. Обратите внимание на использование функции copyinstr для вывода строк.

Таблица 2. Аргументы датчика method-entry провайдера php

args[0]	Идентификатор того потока команд (Java thread ID), который вызывает метод
args[1]	имя класса метода, указатель на строку в кодировке UTF-8
args[2]	длина имени класса метода (в байтах)
args[3]	имя метода, указатель на строку в кодировке UTF-8
args[4]	длина имени метода (в байтах)
args[5]	сигнатура метода, указатель на строку
args[6]	длина сигнатуры метода (в байтах)

Для автоматического завершения скрипта через 5 миллисекунд добавлена конструкция tick-5ms.

Запустим скрипт в соседнем окне терминала:

```
# dtrace -qs hs.d -p 1448

Greeting.\greet ()V
java/io/PrintStream.\println (Ljava/lang/String;)V
java/io/PrintStream.\print (Ljava/lang/String;)V
java/io/PrintStream.\write (Ljava/lang/String;)V
java/io/PrintStream.\ensureOpen ()V
java/io/Writer.\write (Ljava/lang/String;)V
java/io/BufferedWriter.\write (Ljava/lang/String;II)V
java/io/BufferedWriter.\ensureOpen ()V
java/io/BufferedWriter.\min (II)I
java/lang/String.\getChars (II[CI)V
java/lang/System.\arraycopy (Ljava/lang/Object;ILjava/lang/Object;II)V
java/io/BufferedWriter.\flushBuffer ()V
java/io/BufferedWriter.\ensureOpen ()V
java/io/OutputStreamWriter.\write ([CII)V
sun/nio/cs/StreamEncoder.\write ([CII)V
sun/nio/cs/StreamEncoder.\ensureOpen ()V
sun/nio/cs/StreamEncoder.\implWrite ([CII)V
java/nio/CharBuffer.\wrap ([CII)Ljava/nio/CharBuffer;
java/nio/HeapCharBuffer.\<init> ([CII)V
java/nio/CharBuffer.\<init> (IIII[CI)V
java/nio/Buffer.\<init> (IIII)V
java/lang/Object.\<init> ()V
```

## В недрах скриптов на PHP

DTrace становится все популярнее, и поддержку новых датчиков добавляют не только в Java, но и в другие приложения. Далее мы рассмотрим изучение работы скриптов на PHP и СУБД, с которой они общаются, с помощью DTrace.

Как и в случае с JVM, нам потребуется PHP с встроенными в код датчиками. Для этого потребуется скачать и установить модуль dtrace.so для PHP. В Solaris Express Developer Edition 1/08 модуль уже есть – /usr/php5/5.2.4/modules/dtrace.so.

Когда модуль уже готов к работе, надо вписать в php.ini строку:

```
extension=dtrace.so
```

Проверяем, есть ли поддержка DTrace в PHP:

```
# /usr/php5/5.2.4/bin/php -i | grep -i dtrace

/etc/php5/5.2.4/conf.d/dtrace.ini,
dtrace
dtrace support => enabled
```

Теперь предположим, что у нас есть база данных, которая состоит из единственной таблицы, в которой хранится наша записная книжка. Чтобы облегчить задачу, предположим, что в ней записаны только имена, фамилии и телефоны наших знакомых:

```
CREATE TABLE notebook (
id INT AUTO INCREMENT PRIMARY KEY,
name VARCHAR(50),
lastname VARCHAR(50),
phone VARCHAR(15)
);
```

Теперь напишем скрипт на PHP, который будет извлекать из этой таблицы сведения:

```
<?php
$dbname="personal";
$dbhost="localhost";
$dblogin="flip";
$dbpass="9bdx57";

$link = mysql_pconnect($dbhost, $dblogin, $dbpass);
mysql_select_db ($dbname,$link) or die ('Connection Error.');
```

```
$query = "SELECT name , lastname , phone FROM notebook";
$result = mysql_query($query) or
die ('Execution error. ' . mysql_error());

$message = '';

while ($row = mysql_fetch_array($result)) {
    $message = $message . sprintf("%s %% %s %% %s \n",
    $row[0], $row[1], $row[2]);
}
echo $message;
mysql_free_result($result);
?>
```

Пропустим здесь этап заполнения базы данных сведениями и предположим, что данные в таблице notebook уже есть. Интересно, много ли можно узнать с помощью датчиков DTrace в PHP? Провайдер php предоставляет два датчика, срабатывающих при вызове функции и окончании ее работы: function-entry и function-return.

Аргументы этих датчиков:

- **arg0** – имя вызванной функции (указатель на строку внутри PHP);
- **arg1** – имя файла скрипта (указатель на строку внутри PHP);
- **arg2** – номер строки, откуда вызвана функция.

Возьмем простой скрипт для трассировки работы нашего приложения на PHP:

```
#!/usr/sbin/dtrace -Zqs
php*:::function-entry
{
    printf("called %s() in %s at line %d\n",
    copyinstr(arg0), copyinstr(arg1), arg2)
}
```

Ключ Z позволяет запустить скрипт, даже если дескриптор (php\*:::function-entry) не соответствует ни одному датчику (а это в момент запуска скрипта будет именно так, потому что мы еще на запустили скрипт и модуль dtrace.so не загружен).

В одном окне запустим наше приложение на PHP:

```
php simpledb.php
```

```
Philip % Rock % +74951110808
Ivan % Markov % +74951110808
```

а в другом – скрипт dtrace для отслеживания его работы:

```
./php-trace.d
```

```
called mysql_pconnect() in /export/home/flip/simpledb.php at line 7
called mysql_select_db() in /export/home/flip/simpledb.php at line 8
called mysql_query() in /export/home/flip/simpledb.php at line 11
called mysql_fetch_array() in /export/home/flip/simpledb.php at line 15
called sprintf() in /export/home/flip/simpledb.php at line 16
called mysql_fetch_array() in /export/home/flip/simpledb.php at line 15
called sprintf() in /export/home/flip/simpledb.php at line 16
called mysql_fetch_array() in /export/home/flip/simpledb.php at line 15
called mysql_free_result() in /export/home/flip/simpledb.php at line 20
```

Теперь мы знаем больше о том, как вызываются функции в нашем скрипте на PHP. А еще можно выяснить, какие команды наше приложение передает серверу баз данных и сколько времени он на это тратит.

## Внутри баз данных: PostgreSQL и MySQL

Мы рассмотрим возможности по динамическому наблюдению за сервером СУБД, которые предоставляет вживленный в MySQL и PostgreSQL код. По имеющимся сейчас сведениям от разработчиков MySQL, провайдер mysql планируется включить в код СУБД в версии 6.0.4. Однако уже в версии 6.0 можно обнаружить предварительный вариант кода этого провайдера, и если вы хотите его попробовать уже сейчас, это можно сделать, указав при установке из исходных текстов ключ --enable-dtrace при вызове ./configure.

Более детально можно познакомиться с инструкциями по использованию DTrace в MySQL 6.0 на форуме пользователей DTrace по адресу <http://www.opensolaris.org/jive/forum.jspa?forumID=7>.

Здесь мы рассмотрим более хитрую методику отлова команд, переданных серверу MySQL, а затем перейдем к работе с postgresql, который уже давно поставляется со встроенными датчиками DTrace.

Дело в том, что с помощью провайдера pid мы можем получить информацию о вызове любой функции в любом приложении. Кстати, если нас больше интересует производительность ввода-вывода СУБД, то лучше обратиться к провайдеру io, но это уже предмет другой статьи.

В MySQL обработка запросов выполняется функцией mysql\_parse:

```
mysql_parse(THD *thd, char *inBuf, uint length)
```

Название ее второго аргумента звучит многообещающе. Попробуем следующий скрипт для того, чтобы получить список запросов, направленных серверу MySQL:

```
cat mysql-catch.d

pid$target:*mysql_parse*:entry
{
    printf("%Y    %s\n", walltimestamp, copyinstr(arg1));
}
```

Запустим его в одном окне:

```
# dtrace -qs mysql-catch.d -p 1019
```

А в другом запустим уже знакомый скрипт на PHP для получения содержимого нашей записной книжки:

```
# php simpledb.php
```

```
Philip % Rock % +74951110808
Ivan % Markov % +74951110808
```

В первом окне мы получим ответ скрипта:

```
2008 Feb 7 22:58:35 SELECT name , lastname , phone FROM notebook
```

Готово!

Только надо помнить, что, если на сервере запущено несколько экземпляров mysqld, надо позаботиться об указании правильного идентификатора процесса: это обязательно при работе с датчиками провайдера pid. Представленный скрипт справляется с тем, чтобы сообщить нам запрос, отправленный серверу. Может случиться так, что сервер отвергнет запрос – из-за отсутствия права доступа к данным у запрашивающего или из-за некорректности запроса.

Для анализа значения, которое возвращает функция, можно воспользоваться датчиком:

```
pid$target::*mysql_execute_command*:return
```

В arg1 у датчиков провайдера pid находится код возврата. Успешное завершение операции сервером MySQL даст код 0, и ненулевое значение – в противном случае.

Для контроля за выполнением операций достаточно добавить в скрипт mysql-catch.d еще один компонент:

```
pid$target::*mysql_execute_command*:return
{
    printf("%Y %d \n", walltimestamp, arg1);
}
```

Теперь разберемся с PostgreSQL.

Начиная с версии postgresql 8.2 в него встроен код провайдера DTrace – postgresql.

Стало быть, для анализа запросов к этой СУБД можно использовать встроенные датчики, не исследуя исходный код сервера. Вот перечень этих датчиков:

- probe transaction\_\_start(int);
- probe transaction\_\_commit(int);
- probe transaction\_\_abort(int);
- probe lwlock\_\_acquire(int, int);
- probe lwlock\_\_release(int);
- probe lwlock\_\_startwait(int, int);
- probe lwlock\_\_endwait(int, int);
- probe lwlock\_\_condacquire(int, int);
- probe lwlock\_\_condacquire\_\_fail(int, int);
- probe lock\_\_startwait(int, int);
- probe lock\_\_endwait(int, int);

Воспользуемся датчиками transaction-start и transaction-end для получения распределения времени транзакций:

```
#!/usr/sbin/dtrace -Zqs
postgresql*:::transaction-start
{
    self->ts=timestamp;
    @cnt[pid]=count();
}

postgresql*:::transaction-commit
{
    @avg[pid]=avg(timestamp - self->ts);
}
tick-5sec
{
    normalize(@avg, 1000000);
    printf("%15s %30s %30s\n", "PID", "Total queries", "Avegrage time (ms)");
}
```

```
printf("\t===== \n\n");
printa("%15d %30d %30d\n", @cnt, @avg);

printf("\t===== \n\n");
clear(@cnt);
clear(@avg);
}
```

Запуск этого скрипта даст нам реальную картину затрат времени на транзакции. Возможно, в вашем случае распределение будет иным – все зависит от фактической загрузки сервера:

```
# ./postgres_avg_query_time.d
```

PID	Total queries	Avegrage time (ms)
23814	46	57
23817	58	34
23816	59	32
23815	59	33
23818	75	26

А теперь обратимся к уже хорошо знакомому нам провайдеру pid и напишем скрипт для отслеживания запросов к СУБД (обратите внимание: имя функции в postgresql иное, нежели в mysql):


```
#!/usr/sbin/dtrace -Zwqs
BEGIN
{
    freopen("sql.trace");
}
pid$1::pg_parse_query:entry
{
    printf("%s\n", copyinstr(arg0));
}
```

Обратите внимание на вызов freopen("sql.trace") – так мы перенаправим вывод скрипта с экрана в файл с указанным именем.

## Заключение

Сегодня вы познакомились с тем, как использовать DTrace для анализа приложений на Java, PHP и SQL. Вкупе с рассмотренной в предыдущей части статьи методикой отладки приложений на JavaScript (см. №1 за 2008 год) это дает нам полный набор инструментов для отладки как клиентской части приложений Web 2.0, так и серверной части.

Мы постарались обратить ваше внимание на то, что даже если какое-то приложение не имеет встроенных датчиков DTrace, вы всегда можете либо добавить их в исходный код (как уже сделали разработчики PostgreSQL, например), либо использовать известные вам функции для анализа передаваемых ими аргументов и возвращаемых значений – с помощью провайдера pid.

Отдельно стоит поговорить о визуализации результатов работы DTrace, и в одной из следующих статей мы обязательно к этому вернемся. 

1. Блог Адама Левентеля – <http://blogs.sun.com/ahl>.
2. Mac OS X and the missing probes – [http://blogs.sun.com/ahl/entry/mac\\_os\\_x\\_and\\_the](http://blogs.sun.com/ahl/entry/mac_os_x_and_the).
3. Dtrace/Firefox/Leopard – [http://blogs.sun.com/ahl/entry/dtrace\\_firefox\\_leopard](http://blogs.sun.com/ahl/entry/dtrace_firefox_leopard).