Documentation

⚠ **Важно:** Нам очень-очень важен любой фидбек по проекту, поэтому обо всех замечаниях, проблемах и пожеланиях, пожалуйста, пишите нам на drd-support@devexperts.com. Мы долго вели разработку и хотим понять насколько она для вас полезна, какие в ней недочеты и куда нам развиваться дальше.

О программе

Data Race Detector (DRD) - это утилита для динамического обнаружения гонок (data races) в Java-программе. DRD реализован в виде java agent-а, то есть он запускается в той же JVM, что и целевое приложение, и динамически ищет фактические и некоторые потенциальные гонки в приложении во время его работы. Результаты работы DRD выводятся в лог-файлы. По каждой обнаруженной гонке печатается информация, которая будет полезна для устранения гонки в коде исходной программы.

Статус: release 0.7

Установка

Delivery bundle представляет собой архив DRD-<version_number>.zip. Его нужно распаковать в какую-нибудь папку, которую далее по тексту мы назваем DRD_HOME_DIR.
В DRD_HOME_DIR находятся

- drd_agent.jar - собственно, сам DRD Java Agent
- папка config - конфигурация DRD
  - drd.properties - файл с настройками, который будет использоваться по умолчанию.
  - config-example.xml, hb-config-example.xml - конфигурационные файлы по умолчанию. Если вы собираетесь использовать их - переименуйте (уберите "-example")

После распаковки архива и переименовывания конфигурационных файлов DRD готов к использованию.

Использование

1. Сконфигурировать DRD - см. DRD configuration.
2. Найти место в вашем приложении, из которого непосредственно идет вызов java или javaw
3. Прописать туда drd agent:

   ```
   java -javaagent:DRD_HOME_DIR/drd_agent.jar -Ddrd.settings.file=<...> ...
   ```

   если drd.setttings.file не указан, он по умолчанию ищется в DRD_HOME_DIR под именем drd.properties.
4. Запустить исходное приложение и убедиться, что DRD Agent стартовал нормально (см. Анализ логов).
5. Все обнаруженные гонки будут фиксироваться в файле drd_races.log

На данный момент в DRD нет интерактивного управляющего интерфейса - весь конфиг читается в момент старта и дальнейшие изменения в нем никак не отслеживаются (что позволяет перенести все проверки из runtime на этап инструментирования байт-кода). Если есть необходимость внести изменения в конфигурацию, то можно внести их и перезапустить программу. Например, можно изменить область поиска гонок, не искать гонки по определенным полям/методам или указать DRD необходимость хранить полные стектрейсы потоков при обращении к определенным полям.

Поиск гонок

DRD выделяет три области в коде: Race Detection Scope ≤ Synchronization Scope ≤ All Application Code

- All Application Code - весь код приложения (непосредственно ваш код, используемые библиотеки/фреймворки, JRE, ...)
- Synchronization Scope - область, в которой DRD отслеживает операции синхронизации
- Race Detection Scope - область, в которой DRD ищет гонки.

blocked URL

Если Race Detection Scope = Synchronization Scope = All Application Code, то приложение под DRD, скорее всего, не запустится из-за накладных расходов, поэтому разумно как-то сокращать количество информации, которую обрабатывает DRD во время работы программы.

Наша рекомендация по умолчанию: **отслеживать операции синхронизации и искать гонки только в свое коде.**

Итак, пусть Race Detection Scope < All Application Code. Код, который не попадает в race detection scope, будем называть foreign-кодом, а который попадает - our-кодом.

DRD репортит два типа гонок:

- гонки по полям our-кода - 2 потока одновременно обращаются к полю какого-то our-объекта, хотя бы один из них - на запись.
- гонки по вызовам методов foreign-объектов. В our-коде есть 2 вызова методов на одном и том же объекте (напр., list.add() и list.get()) из разных потоков и хотя бы один из этих вызовов *трактуется DRD как write.*

Как ограничить Sync/Race Detection scopes, как указать DRD, какие foreign-методы write, а какие нет и т.д. - см. Documentation.

Примерный сценарий использования DRD

1. SyncScope = com.mycompany.*, RaceDetectionScope = com.mycompany.*; остальную конфигурацию не трогаем.
2. Запускаемся, собираем логи. По желанию подключаем JVisualVM или какой-нибудь профайлер, смотрим на потребление памяти/cpu.
3. Анализируем гонки и корректируем конфигурацию в зависимости от конкретной ситуации:
    - если производительность неудовлетворительна (или приложение не запустилось вовсе), то сузить область инструментирования. Например, отслеживать операции синхронизации тоже только в com.mycompany.*.
    - если DRD находит гонки по вызовам методов непроинструментированных классов, которые на самом деле гонками не являются, то стоит откорректировать contracts или убрать эти классы из области поиска гонок через SkipForeignCalls;
    - если используются какие-то внутренние механизмы синхронизации (или обнаружились непокрытые существующие), нужно создать соответствующий синхронизационный контракт;
    - если по какой-то гонке нужен второй стектрейс, то выставить в TraceConfig storeThreadAccesses="true" для соответствующего класса и поля/метода (ну, или написать "*", чтоб отслеживалось по всем полям/методам) и надеяться, что после перезапуска эта гонка произойдет снова.
4. После корректировки конфигурации перезапускаем приложение: go to 2.

Анализ результатов

drd.log: информационные сообщения - ход инструментации, разнообразная статистика
drd_error.log: ошибки
drd_races.log: обнаруженные гонки
Путь к папке для log-файлов можно указать с помощью property "drd.log.dir". По умолчанию лог-файлы создаются в рабочей директории. Путь к лог-файлам пишется в консоли (System.out) сразу же после запуска DRD.
Подробнее о содержимом файлов и о том, как трактовать обнаруженные гонки см. Анализ логов.

> ⚠ **Важно:** при отправке баг-репорта на drd-support@devexperts.com, пожалуйста, прикладывайте оба лог-файла к письму.

Troubleshooting

⚠ **Будет заполняться по мере использования DRD**