DRD configuration

drd.properties

drd.properties - файл, в котором прописаны все остальные настройки. По умолчанию этот файл ищется в "<DRD_HOME>\drd.properties", но путь к нему можно указать через -Ddrd.settings.file=<FULL_PATH_TO_FILE>. Если этого файла нет, используются настройки по умолчанию.

Все настройки перечислены ниже. Кроме файла drd.properties их всегда можно указать через -D<property_name>=<property_value>

- **drd.log.dir** - папка, в которой будут созданы лог-файлы. По умолчанию это "logs" в текущей директории.
- **drd.config.dir** - папка, в которой ищутся конфигурационные файлы (config.xml и hb-config.xml). По умолчанию это <DRD_HOME>\config\
- **drd.log.level** - уровень логгинга (DEBUG/INFO, по умолчанию - INFO).
- **print.races.in.log.file** - печатать ли гонки в drd.log помимо drd_races.log. (true/false, по умолчанию false)
- **drd.data.clock.histogram.limit** - DRD умеет печатать top N классов, в которых создавались векторные часы для разделяемых данных. Данный параметр позволяет выставить N. По умолчанию N = 20. Абсолютно не влияет на перформанс. Уменьшить нельзя, можно только увеличить.
- **drd.races.grouping** - группировка гонок. По умолчанию CALL_CLASS_AND_METHOD.

Логгирование

DRD создает 3 лог-файла:

- drd.log - статистическая информация о ходе работы DRD
- drd-errors.log - ошибки в работе DRD
- drd-races.log - обнаруженные гонки

Папка, в которой эти файлы будут созданы, регулируется свойством "drd.log.dir". Его можно прописать либо в DRD_HOME_DIR\config\drd.properties, либо в строке параметров JVM через -Ddrd.log.dir= ...

config.xml

Это пример конфигурационного файла. Актуальный находится в delivery bundle.

```
<DRDConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="config-scheme.xsd">
    <!--NB: the less instrumentation scope is, the less is overhead, that DRD imposes on target application-->
    <InstrumentationScope>
        <!--Detect sync operations everywhere-->
        <SyncInterception defaultPolicy="include">
            <!--<Rule type="include" path="com/devexperts/"/>-->
        </SyncInterception>
        <!--Detect data operations only in "com.devexperts" package by default-->
        <RaceDetection defaultPolicy="exclude">
            <Rule type="include" path="com/devexperts/"/>
        </RaceDetection>

        <!--Races wouldn't be detected on fields, listed in SkipOurFields tag
        -->
        <SkipOurFields/>

        <!--Races wouldn't be detected on foreign calls, listed in SkipForeignCalls tag
        Foreign call is method call of object not from race detection scope
        -->
        <SkipForeignCalls>
            <!--Well, we believe, that any call to System is thread safe, so do not detect races on it,
            because we have a lot of System.currentTimeMillis() calls from different threads and do not want to
            see them in DRD output file-->
            <Target clazz="java/lang/System" name="*" type="method"/>
        </SkipForeignCalls>
    </InstrumentationScope>

    <!--Contracts determine, if foreign call should be treated as read or as write operation-->
    <Contracts>
        <!--Contracts are analyzed top down until some matches. If no one matches, foreign call would be
treated as write-->
        <!--If write="*", treat all methods, not listed in "read", as writes. Same for read="*"-->
        <!--Methods "keySet","values" and "entrySet" of java.util.Map are reads-->
        <Contract clazz="java.util.Map" read="keySet,values,entrySet"/>
        <!--Methods java.util.List.listIterator() is read too -->
        <Contract clazz="java.util.List" read="listIterator"/>
        <!--If method name is "hashCode", "toString" or "equals", or method name starts with "get", "is",
        "contains", "iter" or "has", than this method is read-->
        <Contract clazz="*" read="get*,toString,hashCode,equals,is*,contains*,iter*,has*"/>
        <!--All other foreign calls are writes-->
    </Contracts>

    <!--When DRD should print additional info on accessing field of instrumented class or foreign call?
    traceDataOperations = trace clocks for each read/write operation on this field/call.
    traceSyncOperations = trace all sync operations in this class
    These two options would help to detect bugs in DRD itself

    storeThreadAccesses = store last thread accesses to specified field/call. If field is racy,
      enabling this option would make it possible to see stack traces of both threads, when race occurs
      (by default only one stack trace is displayed - stack trace of current thread. For second thread only
brief location is available)
    printThreadAccess = print all thread accesses to specified field/call

    By default all these options are true
    -->
    <TraceTracking>
        <!--We've found race in our app on MyBusinessLogicClass.myField field and want to get both stack traces.
        We enable storing Thread Accesses for this field and restart app
                We may optionally specify caller classes: classes, whose method's accesses to myField we want
to track
        -->
        <Target clazz="com/my/MyBusinessLogicClass" type="myField"
                name="activeIssue" caller="com/my/class/that/uses/MyBusinessLogicClass" storeThreadAccesses="
true"/>
    </TraceTracking>
</DRDConfig>
```

Файл состоит из трех частей:

- InstrumentationScope - отвечает за то, где (в каких классах, полях, методах) DRD будет отслеживать операции синхронизации и обращения к разделяемым перенным.
- Contracts - отвечает за то, как будут трактоваться вызовы методов тех объектов, которые не попали в instrumentation scope, - как read или как write.
- TraceTracking - по каким классам/полям/методам печатать доп.информацию в файл (drd.log)

Instrumentation scope

DRD agent модифицирует классы приложения с помощью Java Instrumentation API - перед тем, как класс будет загружен, он в виде массива байтов передается DRD agent-у, а тот, если необходимо, вставляет вызовы внутреннего алгоритма DRD, отвечающего за поиск гонок. Этот процесс называется *инструментированием байт-кода*.

Интересные с точки зрения поиска гонок операции можно разделить на две группы:

- **операции синхронизации**, осуществляющие передачу отношения happens-before (см. JLS chapter 17) - начало и конец synchronized-блоков, чтение/запись volatile-переменных, thread start/join, обращения к средствам пакета java.util.concurrent и т.д.
- **обращения к данным**, доступным нескольким потокам - собственно, ровно в этих местах и могут возникнуть гонки.

Чтобы DRD не отслеживал операции синхронизации во всех классах / не искал гонки во всех классах (например, логично не искать гонки в java.*), можно ограничить его область инструментирования.

```
<SyncInterception defaultPolicy="include"/>
<RaceDetection defaultPolicy="exclude">
   <Rule type="include" path="com/devexperts/"/>
</RaceDetection>
```

Тег "SyncInterception" отвечает за область, в которой будут отслеживаться синхронизационные операции, тег "RaceDetection" - обращения к разделяемым переменным, то есть, непосредственно за поиск гонок.

> ⚠ Поскольку искать гонки в классе X, и при этом не отслеживать в нем операции синхронизации, бессмысленно, DRD пересекает области RaceDetection с SyncInterception, чтобы первая была подножеством второй.

Для каждого загружаемого класса C DRD выполняет следующее:

1. Проходит по списку правил из SyncInterception сверху вниз до первого правила, для которого верно C.getName().startsWith(rule.path).
   Если type="include", то DRD будет отслеживать в классе операции синхронизации, если "exclude", то нет.
   Если такое правило не обнаружено, то используется defaultPolicy(атрибут тега SyncInterception).
2. Если в предыдущем пункте получилось "include", то ровно таким же образом анализируется тег RaceDetection чтобы определить, нужно ли искать гонки в загружаемом классе.

Классы, в которых DRD *ищет гонки*, мы называем *"проинструментированными"*. Соответственно, инстансы этих классов - *проинструментированные объекты*.

Если класс проинструментирован, то он внутри себя содержит вставки кода, проверяющие обращения к полям этого класса (как static, так и instance) на предмет вовлеченности в гонки. Таким образом, если мы видим вызов метода проинструментированного класса, нам ничего не нужно делать - класс внутри себя разберется сам. Однако, мы можем встретить вызов непроинструментированного класса. Например, если мы ищем гонки только в пакете "/com/devexperts/", то вызов java.util.List.get() будет таким примером. Такие вызовы мы далее называем "foreign calls", а непроинструментированные объекты, встреченные в проинструментированном коде - "foreign objects".

В отличие от проинструментированных объектов, foreign-объекты не содержат часов внутри себя для своих полей, поэтому мы ассоциируем отдельные часы с каждым foreign-объектом и каждый foreign-call трактуем как read или write операцию.

Contracts

Тег "Contracts" предназначен для того, чтобы описать контракты foreign классов (например, java.util.Map.put() - это write, а get() - это read).

```
<Contracts>
        <!--Contracts are analyzed top down until some matches. If no one matches, foreign call would be
treated as write-->

        <!--If write="*", treat all methods, not listed in "read", as writes. Same for read="*"-->

        <!--Methods "keySet","values" and "entrySet" of java.util.Map are reads-->
        <Contract clazz="java.util.Map" read="keySet,values,entrySet"/>
        <!--Methods java.util.List.listIterator() is read too -->
        <Contract clazz="java.util.List" read="listIterator"/>
        <!--If method name is "hashCode", "toString" or "equals", or method name starts with "get", "is",
        "contains", "iter" or "has", than this method is read-->
        <Contract clazz="*" read="get*,toString,hashCode,equals,is*,contains*,iter*,has*"/>
        <!--All other foreign calls are writes-->
</Contracts>
```

Контракты внутри этого тега анализируются сверху вниз до первого совпадения: для каждого foreign call X.f() DRD идет по списку контрактов сверху вниз, пока не найдет такой контракт c, что имя класса начинается с contract.clazz и либо название вызываемого метода указано в contract.read/contract.write, либо contract.read/contract.write="*".
Если таких контрактов не найдено, то foreign call трактуется как write.

> **Важно:**
> У класса может быть существенно более сложный контракт, чем простое "трактовать такие-то методы, как read, а остальные, как write". Например, некоторые методы могут быть не просто потокобезопасными, а еще и при вызове их в правильном порядке обеспечивать сихнронизацию потоков (happens-before) - например, ConcurrentHashMap.get() синхронизирован с предыдущем вызовом set() по тому же ключу. Для этого мы разработали конфигуратор синхронизационных контрактов, которые описываются в файле hb-config.xml.

hb-config.xml

В этом файле перечислены все синхронизационные контракты. В частности, описаны контракты классов Unsafe и AbstractQueuedSynchronizer, что позволяет автоматически поддерживать все сложные механизмы синхронизации пакета java.util.concurrent.

Все дескрипторы методов описываются согласно спецификации JVM.
Например, "(Ljava/lang/Object;JLjava/util/List;)Z" описывает дескриптор метода с тремя входными параметрами (Object, long, List), возвращающего значение типа boolean.

На данный момент выделяются 2 типа контрактов:

Контракт пары методов

Такие контракты содержатся внутри тега "Syncs".

> **Этот и следующий xml-фрагменты - примеры частей конфигурационного файла. Актуальный файл hb-config.xml находится в delivery bundle.**

```
<Sync>
            <Links>
                <Link send="owner" receive="owner"/>
                <Link send="param" send-number="0" receive="param" receive-number="0"/>
            </Links>
            <Send>
                <MethodCall owner="java.util.concurrent.ConcurrentMap" name="put"
                            descriptor="(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;"/>
            </Send>
            <Receive>
                <MethodCall owner="java.util.concurrent.ConcurrentMap" name="get"
                            descriptor="(Ljava/lang/Object;)Ljava/lang/Object;"/>
            </Receive>
</Sync>
```

В данном контракте указано, что вызов метода *put* объекта типа ConcurrentMap happens-before вызова *get того же объекта* (link owner-owner) по тому же ключу (link param0-param0).

Контракт класса, являющиегося примитивом синхронизации

Такие контракты содержатся внутри тега "Multiple-Syncs".

```
<Multiple-Sync owner="java.util.concurrent.locks.AbstractQueuedSynchronizer">
            <Multiple-Links>
                <Multiple-Link type="owner"/>
            </Multiple-Links>
            <Call type="receive" name="tryAcquire" descriptor="(I)Z" shouldReturnTrue="true"/>
            <Call type="send" name="tryRelease" descriptor="(I)Z"/>
            <Call type="receive" name="tryAcquireShared" descriptor="(I)Z" shouldReturnTrue="true"/>
            <Call type="send" name="tryReleaseShared" descriptor="(I)Z"/>
            <Call type="receive" name="acquire" descriptor="(I)V"/>
            <Call type="receive" name="acquireInterruptibly" descriptor="(I)V"/>
            <Call type="receive" name="tryAcquireNanos" descriptor="(IJ)Z" shouldReturnTrue="true"/>
            <Call type="send" name="release" descriptor="(I)Z"/>
            <Call type="receive" name="acquireShared" descriptor="(I)V"/>
            <Call type="receive" name="acquireSharedInterruptibly" descriptor="(I)V"/>
            <Call type="receive" name="tryAcquireSharedNanos" descriptor="(IJ)Z" shouldReturnTrue="true"/>
            <Call type="send" name="releaseShared" descriptor="(I)Z"/>
            <Call type="send" name="setState" descriptor="(I)V"/>
            <Call type="receive" name="getState" descriptor="()I"/>
            <Call type="full" name="compareAndSetState" descriptor="(II)Z" shouldReturnTrue="true"/>
</Multiple-Sync>
```

Здесь везде идет речь об одном и том же объекте типа AbstractQueuedSynchronizer (link типа owner). Указано, например, что метод release является левой частью отношения happens-before (имеет семантику monitorexit), а, скажем, "compareAndSetState" - полным отношением happens-before (имеет семантику volatile read и write одновременно), но *только если вернул true*.

Полезные ссылки

JVM method descriptors : http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.3
JLS chapter about threads: http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html
happens-before relation: http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5
java.util.concurrent.atomic documentation: http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html
java.util.concurrent documentation: http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html