**Многопоточность**

*вопросы-ответы Хилькевич Игорь 15.04.2020*

Вопросы из “all question”

2. Потокобезопасность, конкуренция, параллелизм, корпоративная и

вытесняющая многозадачность.

3. Ordering, as-if-serial semantics, sequental consistency, visibility,

atomicity, happense-before, mutual exclusion, safe publication.

16. Thread.holdsLock()

19. Приоритет потока (от 1 до 10, default 5).

23. CyclicBarrier vs. CountDownLatch.

25. Исключение в потоке.

26. Пулл потоков.

27. Executor, Executors.

30. ThreadLocal-переменная.

31. synchronized vs. ReentrantLock.

32. ReedWriteLock.

33. Блокирующий метод.

34. Busy spin.

**Ответы**

1. Java Memory Model (JMM).

**В чем заключаются различия между cтеком (stack) и кучей (heap) с точки зрения многопоточности?**

Cтек – участок памяти, тесно связанный с потоками. У каждого потока есть свой стек, которые хранит локальные переменные, параметры методов и стек вызовов. Переменная, хранящаяся в стеке одного потока, не видна для другого.

Куча – общий участок памяти, который делится между всеми потоками. Объекты, неважно локальные или любого другого уровня, создаются в куче. Для улучшения производительности, поток обычно кэширует значения из кучи в свой стек, в этом случае для того, чтобы указать потоку, что переменную следует читать из кучи используется ключевое слово volatile.

**Как поделиться данными между двумя потоками?**

Данными между потоками возможно делиться, используя общий объект или параллельные структуры данных, например BlockingQueue.

**Какой параметр запуска JVM используется для контроля размера стека потока?**

-Xss

**Как получить дамп потока?**

Среды исполнения Java на основе HotSpot генерируют только дамп в формате HPROF. В распоряжении разработчика имеется несколько интерактивных методов генерации дампов и один метод генерации дампов на основе событий.

Интерактивные методы:

* Использование Ctrl+Break: если для исполняющегося приложения установлена опция командной строки -XX:+HeapDumpOnCtrlBreak, то дамп формата HPROF генерируется вместе с дампом потока при наступлении события Ctrl+Break или SIGQUIT (обычно генерируется с помощью kill -3), которое инициируется посредством консоли. Эта опция может быть недоступна в некоторых версиях. В этом случае можно попытаться использовать следующую опцию: -Xrunhprof:format=b,file=heapdump.hprof
* Использование инструмента jmap: утилита jmap, поставляемая в составе каталога /bin/ комплекта JDK, позволяет запрашивать дамп HPROF из исполняющегося процесса.
* Использование операционной системы: Для создания файла ядра можно воспользоваться неразрушающей командой gcore или разрушающими командами kill -6 или kill -11. Затем извлечь дамп кучи из файла ядра с помощью утилиты jmap.
* Использование инструмента JConsole. Операция dumpHeap предоставляется в JConsole как MBean-компонент HotSpotDiagnostic. Эта операция запрашивает генерацию дампа в формате HPROF.

Метод на основе событий:

Событие OutOfMemoryError: Если для исполняющегося приложения установлена опция командной строки -XX:+HeapDumpOnOutOfMemoryError, то при возникновении ошибки OutOfMemoryError генерируется дамп формата HPROF. Это идеальный метод для «production» систем, поскольку он практически обязателен для диагностирования проблем памяти и не сопровождается постоянными накладными расходами с точки зрения производительности. В старых выпусках сред исполнения Java на базе HotSpot для этого события не устанавливается предельное количество дампов кучи в пересчете на одну JVM; в более новых выпусках допускается не более одного дампа кучи для этого события на каждый запуск JVM.

19. Приоритет потока (от 1 до 10, default 5).

**Что значит «приоритет потока»?**

Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда какому из потоков будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритетные. Практически объем времени процессора, который получает поток, часто зависит от нескольких факторов помимо его приоритета.

Чтобы установить приоритет потока, используется метод класса Thread: final void setPriority(int level). Значение level изменяется в пределах от Thread.MIN\_PRIORITY = 1 до Thread.MAX\_PRIORITY = 10. Приоритет по умолчанию - Thread.NORM\_PRlORITY = 5.

Получить текущее значение приоритета потока можно вызвав метод: final int getPriority() у экземпляра класса Thread.

23. CyclicBarrier vs. CountDownLatch.

**В чем заключаются различия между CyclicBarrier и CountDownLatch?**

CountDownLatch (замок с обратным отсчетом) предоставляет возможность любому количеству потоков в блоке кода ожидать до тех пор, пока не завершится определенное количество операций, выполняющихся в других потоках, перед тем как они будут «отпущены», чтобы продолжить свою деятельность. В конструктор CountDownLatch(int count) обязательно передается количество операций, которое должно быть выполнено, чтобы замок «отпустил» заблокированные потоки.

Примером CountDownLatch из жизни может служить сбор экскурсионной группы: пока не наберется определенное количество человек, экскурсия не начнется.

CyclicBarrier реализует шаблон синхронизации «Барьер». Циклический барьер является точкой синхронизации, в которой указанное количество параллельных потоков встречается и блокируется. Как только все потоки прибыли, выполняется опционное действие (или не выполняется, если барьер был инициализирован без него), и, после того, как оно выполнено, барьер ломается и ожидающие потоки «освобождаются». В конструкторы барьера CyclicBarrier(int parties) и CyclicBarrier(int parties, Runnable barrierAction) обязательно передается количество сторон, которые должны «встретиться», и, опционально, действие, которое должно произойти, когда стороны встретились, но перед тем когда они будут «отпущены».

CyclicBarrier является альтернативой метода join(), который «собирает» потоки только после того, как они выполнились.

CyclicBarrier похож на CountDownLatch, но главное различие между ними в том, что использовать «замок» можно лишь единожды - после того, как его счётчик достигнет нуля, а «барьер» можно использовать неоднократно, даже после того, как он «сломается».

25. Исключение в потоке.

**Что происходит, когда в потоке выбрасывается исключение?**

Если исключение не поймано – поток «умирает» (переходит в состяние мёртв (dead)).

Если установлен обработчик непойманных исключений, то он возьмёт управление на себя. Thread.UncaughtExceptionHandler – интерфейс, определённый как вложенный интерфейс для других обработчиков, вызываемых, когда поток внезапно останавливается из-за непойманного исключения. В случае, если поток собирается остановиться из-за непойманного исключения, JVM проверяет его на наличие UncaughtExceptionHandler, используя Thread.getUncaughtExceptionHandler(), и если такой обработчик найдет, то вызовет у него метод uncaughtException(), передав этот поток и исключение в виде аргументов.

26. Пулл потоков.

**Что такое «пул потоков»?**

Создание потока является затратной по времени и ресурсам операцией. Количество потоков, которое может быть запущено в рамках одного процесса также ограниченно. Чтобы избежать этих проблем и в целом управлять множеством потоков более эффективно в Java был реализован механизм пула потоков (thread pool), который создаётся во время запуска приложения и в дальнейшем потоки для обработки запросов берутся и переиспользуются уже из него. Таким образом, появляется возможность не терять потоки, сбалансировать приложение по количеству потоков и частоте их создания.

**Какого размера должен быть пул потоков?**

Настраивая размер пула потоков, важно избежать двух ошибок: слишком мало потоков (очередь на выполнение будет расти, потребляя много памяти) или слишком много потоков (замедление работы всей систему из-за частых переключений контекста).

Оптимальный размер пула потоков зависит от количества доступных процессоров и природы задач в рабочей очереди. На N-процессорной системе для рабочей очереди, которая будет выполнять исключительно задачи с ограничением по скорости вычислений, можно достигнуть максимального использования CPU с пулом потоков, в котором содержится N или N+1 поток. Для задач, которые могут ждать осуществления I/O (ввода - вывода) - например, задачи, считывающей HTTP-запрос из сокета – может понадобиться увеличение размера пула свыше количества доступных процессоров, потому, что не все потоки будут работать все время. Используя профилирование, можно оценить отношение времени ожидания (WT) ко времени обработки (ST) для типичного запроса. Если назвать это соотношение WT/ST, то для N-процессорной системе понадобится примерно N\*(1 + WT/ST) потоков для полной загруженности процессоров.

Использование процессора – не единственный фактор, важный при настройке размера пула потоков. По мере возрастания пула потоков, можно столкнуться с ограничениями планировщика, доступной памяти, или других системных ресурсов, таких, как количество сокетов, дескрипторы открытого файла, или каналы связи базы данных.

**Что будет, если очередь пула потоков уже заполнена, но подается новая задача?**

Если очередь пула потоков заполнилась, то поданная задача будет «отклонена». Например - метод submit() у ThreadPoolExecutor выкидывает RejectedExecutionException, после которого вызывается RejectedExecutionHandler.

**В чём заключается различие между методами submit() и execute() у пула потоков?**

Оба метода являются способами подачи задачи в пул потоков, но между ними есть небольшая разница.

execute(Runnable command) определён в интерфейсе Executor и выполняет поданную задачу и ничего не возвращает.

submit() – перегруженный метод, определённый в интерфейсе ExecutorService. Способен принимать задачи типов Runnable и Callable и возвращать объект Future, который можно использовать для контроля и управления процессом выполнения, получения его результата.

27. Executor, Executors.

Начиная с Java 1.5 Java API предоставляет фреймворк Executor, который позволяет создавать различные типы пула потоков:

Executor - упрощенный интерфейс пула, содержит один метод для передачи задачи на выполнение;

ExecutorService - расширенный интерфейс пула, с возможностью завершения всех потоков;

AbstractExecutorService - базовый класс пула, реализующий интерфейс ExecutorService;

Executors - фабрика объектов связанных с пулом потоков, в том числе позволяет создать основные типы пулов;

ThreadPoolExecutor - пул потоков с гибкой настройкой, может служить базовым классом для нестандартных пулов;

ForkJoinPool - пул для выполнения задач типа ForkJoinTask;

... и другие.

Методы Executors для создания пулов:

newCachedThreadPool() - если есть свободный поток, то задача выполняется в нем, иначе добавляется новый поток в пул. Потоки не используемые больше минуты завершаются и удаляются из кэша. Размер пула неограничен. Предназначен для выполнения множество небольших асинхронных задач;

newCachedThreadPool(ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

newFixedThreadPool(int nThreads) - создает пул на указанное число потоков. Если новые задачи добавлены, когда все потоки активны, то они будут сохранены в очереди для выполнения позже. Если один из потоков завершился из-за ошибки, на его место будет запущен другой поток. Потоки живут до тех пор, пока пул не будет закрыт явно методом shutdown().

newFixedThreadPool(int nThreads, ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

newSingleThreadScheduledExecutor() - однопотоковый пул с возможностью выполнять задачу через указанное время или выполнять периодически. Если поток был завершен из-за каких-либо ошибок, то для выполнения следующей задачи будет создан новый поток.

newSingleThreadScheduledExecutor(ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

newScheduledThreadPool(int corePoolSize) - пул для выполнения задач через указанное время или переодически;

newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory) - аналогично предыдущему, но с собственной фабрикой потоков;

unconfigurableExecutorService(ExecutorService executor) - обертка на пул, запрещающая изменять его конфигурацию;

30. ThreadLocal-переменная.

**Что такое ThreadLocal-переменная?**

ThreadLocal - класс, позволяющий имея одну переменную, иметь различное её значение для каждого из потоков.

У каждого потока - т.е. экземпляра класса Thread - есть ассоциированная с ним таблица ThreadLocal-переменных. Ключами таблицы являются cсылки на объекты класса ThreadLocal, а значениями - ссылки на объекты, «захваченные» ThreadLocal-переменными, т.е. ThreadLocal-переменные отличаются от обычных переменных тем, что у каждого потока свой собственный, индивидуально инициализируемый экземпляр переменной. Доступ к значению можно получить через методы get() или set().

Например, если мы объявим ThreadLocal-переменную: ThreadLocal<Object> locals = new ThreadLocal<Object>();. А затем, в потоке, сделаем locals.set(myObject), то ключом таблицы будет ссылка на объект locals, а значением - ссылка на объект myObject. При этом для другого потока существует возможность «положить» внутрь locals другое значение.

Следует обратить внимание, что ThreadLocal изолирует именно ссылки на объекты, а не сами объекты. Если изолированные внутри потоков ссылки ведут на один и тот же объект, то возможны коллизии.

Так же важно отметить, что т.к. ThreadLocal-переменные изолированы в потоках, то инициализация такой переменной должна происходить в том же потоке, в котором она будет использоваться. Ошибкой является инициализация такой переменной (вызов метода set()) в главном потоке приложения, потому как в данном случае значение, переданное в методе set(), будет «захвачено» для главного потока, и при вызове метода get() в целевом потоке будет возвращен null.

31. synchronized vs. ReentrantLock.

**Назовите различия между synchronized и ReentrantLock?**

В Java 5 появился интерфейс Lock предоставляющий возможности более эффективного и тонкого контроля блокировки ресурсов. ReentrantLock – распространённая реализация Lock, которая предоставляет Lock с таким же базовым поведением и семантикой, как у synchronized, но расширенными возможностями, такими как опрос о блокировании (lock polling), ожидание блокирования заданной длительности и прерываемое ожидание блокировки. Кроме того, он предлагает гораздо более высокую эффективность функционирования в условиях жесткой состязательности.

Что понимается под блокировкой с повторным входом (reentrant)? Просто то, что есть подсчет сбора данных, связанный с блокировкой, и если поток, который удерживает блокировку, снова ее получает, данные отражают увеличение, и тогда для реального разблокирования нужно два раза снять блокировку. Это аналогично семантике synchronized; если поток входит в синхронный блок, защищенный монитором, который уже принадлежит потоку, потоку будет разрешено дальнейшее функционирование, и блокировка не будет снята, когда поток выйдет из второго (или последующего) блока synchronized, она будет снята только когда он выйдет из первого блока synchronized, в который он вошел под защитой монитора.

Lock lock = new ReentrantLock();

lock.lock();

try {

// update object state

}

finally {

lock.unlock();

}

Реализация ReentrantLock гораздо более масштабируема в условиях состязательности, чем реализация synchronized. Это значит, что когда много потоков соперничают за право получения блокировки, общая пропускная способность обычно лучше у ReentrantLock, чем у synchronized. JVM требуется меньше времени на установление очередности потоков и больше времени на непосредственно выполнение.

У ReentrantLock (как и у других реализаций Lock) блокировка должна обязательно сниматься в finally блоке (иначе, если бы защищенный код выбросил исключение, блокировка не была бы снята). Используя синхронизацию, JVM гарантирует, что блокировка автоматически снимаются.

Резюмируя можно сказать, что когда состязания за блокировку нет либо оно очень мало, то synchronized возможно будет быстрее. Если присутствует заметное состязание за доступ к ресурсу, то скорее всего ReentrantLock даст некое преимущество.

32. ReedWriteLock.

**Что такое ReadWriteLock?**

ReadWriteLock – это интерфейс расширяющий базовый интерфейс Lock. Используется для улучшения производительности в многопоточном процессе и оперирует парой связанных блокировок (одна - для операций чтения, другая - для записи). Блокировка чтения может удерживаться одновременно несколькими читающими потоками, до тех пор пока не появится записывающий. Блокировка записи является эксклюзивеной.

Существует реализующий интерфейс ReadWriteLock класс ReentrantReadWriteLock, который поддерживает до 65535 блокировок записи и до стольки же блокировок чтения.

ReadWriteLock rwLock = new ReentrantReadWriteLock();

Lock rLock = rwLock.readLock();

Lock wLock = rwLock.writeLock();

wLock.lock();

try {

// exclusive write

} finally {

wLock.unlock();

}

rLock.lock();

try {

// shared reading

} finally {

rLock.unlock();

}

33. Блокирующий метод.

**Что такое «блокирующий метод»?**

Блокирующий метод – метод, который блокируется, до тех пор, пока задание не выполнится, например метод accept() у ServerSocket блокируется в ожидании подключения клиента. Здесь блокирование означает, что контроль не вернётся к вызывающему методу до тех пор, пока не выполнится задание. Так же существуют асинхронные или неблокирующиеся методы, которые могут завершится до выполнения задачи.

34. Busy spin.

**Что такое busy spin?**

busy spin – это техника, которую программисты используют, чтобы заставить поток ожидать при определённом условии. В отличие от традиционных методов wait(), sleep() или yield(), которые подразумевают уступку процессорного времени, этот метод вместо уступки выполняет пустой цикл. Это необходимо, для того, чтобы сохранить кэш процессора, т.к. в многоядерных системах, существует вероятность, что приостановленный поток продолжит свое выполнение уже на другом ядре, а это повлечет за собой перестройку состояния процессорного кэша, которая является достаточно затратной процедурой.