otvorot2008 9 марта 2020 в 13:47

Вопросы к собеседованию Java-backend, Jav вопросов)

Java, Карьера в IT-индустрии

Добрый день! Представляю вашему вниманию список вопросов к собеседованию Java Backend, которые я оформлял на протяжении около 2х лет.

Вопросы разбиты по темам: **core**, **collections**, **concurrency**, **io**, **exceptions**, которые задают основные направления хода технического собеседования. Звездочками отмечен субъективный (с точки зрения автора) уровень сложности вопроса, в сноске спойлера — краткий ответ на вопрос. Ответ представляет для интервьюера правильное направления развития мысли кандидата.

Опросник не претендует на роль исчерпывающего средства оценки технических навыков кандидата, это скорее еще одна компиляция вопросов и тем по Java и сопуствующим технологиям, принципам и концепциям разработки, коих в сети десятки. Она преследует цель собрать большое число технических вопросов, возникающих на собеседованиях, в удобном для читателей Хабра формате. Некоторые ответы следует воспринимать как мнемоники, «размечивающие» пространство поиска, так что глубже копать нужно уже в документации.

Также не стоит обращать внимание на слишком большое кол-во списков, так как это все-таки не вольное переложение авторских знаний с примесью литературной экспрессии (что на хабре котируется и собирает аудиторию, особенно в пятницу). Этот список я составил для самого себя как способ структурировать основные тематики и типичные вопросы с собеседований, так что все дополнения и правки только приветствуются.

Это также не руководство к действию — не надо закидывать бедных кандидатов всеми вопросами из списка.

Секция Core:

- Назвать методы Object. (*)
  Краткий ответ
  - toString()
  - equals()
  - hashCode()
  - wait()
  - notify()
  - notifyAll()
  - finalize() — deprecated в Java 9+
  - getClass()

  Про toString(), equals(), hashCode() и их контракт знать нужно обязательно

- Что такое string-pool? В чем отличие создания строки через new от литерала? Что такое String.intern()? (*)
  Краткий ответ

  string-pool — структура в памяти, хранящая массив всех строк-литералов программы.
  String.intern(), соответственно, вернет строку из пула, при наличии таковой. Полезно при сравнениях вида:

  ```
  new String("hello").intern() == new String("hello").intern()
  ```

  Т.к без интернирования пришлось бы сравнивать строки через equals, что может быть медленнее при наличии длинных строк. В данном случае возвращается ссылка на один и тот же объект строки из пула, и проверка проходит с true.

- Почему хранить пароль предпочтительнее в char[]/byte[], а не в String? (**)
  Краткий ответ
  - Строка в виде литерала сразу раскрывает пароль, плюс она всегда хранится в string-пуле
  - byte[]/char[] возможно сбросить после использования, и удалить все ссылки на него

- Привести пример плохой реализации hashCode() (*)
  Краткий ответ

  Метод, возвращающий константу, или значения хэшкодов с неравномерным распределением, приводящим к коллизиям

- Примитивы, врапперы. Package/unpackage (boxing/unboxing). (*)
  Краткий ответ
  - Типы примитивы не создаются в куче, их жизненный цикл ограничен жизненным циклом стек-фрейма

    - Package — создание типа-обертки в хипе для аналогичного типа-примитива, например при объявлении аргумента как Integer, и при передаче int в качестве аргумента. Unpackage — обратная операция

- Сравнение по == и по equals (*)
  Краткий ответ

    - Сравнение по "==" — сравнение по ссылкам
    - Сравнение по «equals» — если переопределен equals, то это сравнение эквивалентности объектов по их полям, если нет — по ссылкам на объекты

- Свойства, которым должен удовлетворять equals (**)
  Краткий ответ

    - Рефлексивность: a==a
    - Симметричность: a==b, b==a
    - Транзитивность: a==b, b==c, a==c
    - Консистентность: Множественные вызовы equals должны возвращать один и тот же результат

- Отличия String/StringBuilder/StringBuffer (**)
  Краткий ответ

    - String — иммутабельный байтовый массив
    - StringBuilder — helper-класс для построения строк, не предоставляет гарантий синхронизации
    - StringBuffer — то же, что и StringBuilder, с synchronized методами

- Приведите пример нарушения симметрии equals (**)
  Краткий ответ

    1. Создать класс Point2D с полями x,y: double
    2. Унаследовать от него класс ColoredPoint2D с доп. полем color
    3. a: Point2D
    4. b: ColoredPoint2D
    5. a.equals(b), !b.equals(a)

- Interface vs Abstract Class. (*)
  Краткий ответ

    - Интерфейс есть средство наследования API, абстрактный класс — средство наследования реализации
    - Через интерфейсы возможно осуществлять множественное наследование, абстрактный класс можно наследовать в одном экземпляре.
    - В интерфейсе нет возможности определить поля и конструкторы

- override vs overload (*)

Краткий ответ

- override — возможность переопределениия поведения метода в типах-потомках
- overload — возможность переопределять метод с одним именем, но разным набором аргументов

- Как в Java сделать утечку памяти? (**)
  Краткий ответ

  - Используя самописный класс стека, при выполнении операции pop() не присваивать предыдущей ссылке значение null.
  - Также можно неверно использовать HashMap вместо WeakHashMap для кэширования чего-нибудь большого, например картинок ваших товаров, пользователей и.т.д в. Т.к ссылки на ключи сильные (strong references), значения по этим ключам будут висеть в хипе до ~~морковкиного заговенья~~ следующей перезагрузки jvm процесса или удаления ключа из мапы и обнуления ссылки на него. Вообще, кэширование — тема для отдельного разговора
  - Также, статья (но староватая)

- Как вернуть псевдо-случайную последовательность целых чисел/чисел с плавающей запятой? (**)
  Краткий ответ

  java.util.Random

- В чем проблемы Random? (**)
  Краткий ответ

  Random возвращает псевдо-случайную числовую последовательность, основанную на линейном конгруэнтном методе и seed'e, основанном на timestamp'e создания *j.u.Random*.
  Соотвественно, зная время создания, можно предсказать такую последовательность. Такой генератор является детерминированным, и криптографически нестойким. Для исправления этого лучше использовать SecureRandom

- GC и различные его виды в JVM. Какой объект считать достижимым. Как происходит сборка мусора (своими словами).(**)
  Краткий ответ

  Виды GC:

  - Serial Stop the World
  - Parallel
  - CMS (В чем недостаток по сравнению с Parallel?)
  - G1 (Назвать отличие от CMS)
  - Shenandoah

  Если объект является достижимым из стека или статической области, то он не поддается сборке мусора

- Java 8: стримы, функциональные интерфейсы, Optional (**)
  Краткий ответ

  Stream — интерфейс, предоставляющий функциональные возможности обработки коллекций (filter, map, reduce, peek)
  Операции на стримах делятся на терминальные и нетерминальные. Нетерминальные операции модифицируют pipeline операций над коллекцией, при этом не изменяя саму коллекцию, терминальные (например, collect) — проводят действия

pipeline'а, возвращают результат и закрывают Stream.

FunctionalInterface — аннотация, предоставляющая возможность использовать лямбды на месте интерфейсов (например, при передаче лямбды в качестве аргумента в метод)

Optional — интерфейс, предохраняющий пользовательский код от nullable ссылок. Оборачивает исходный nullable объект, и предоставляет возможность понять, хранит ли non-nullable объект или нет.

- Java 8: Что такое capturing/non-capturing lambda (**)
  Краткий ответ
  - capturing lambda захватывает локальные переменные/аргументы/поля объекта из внешнего скоупа
  - non-capturing lambda — не захватывает контекст внешнего скоупа, не инстанцируется каждый раз при использовании

- Новые возможности Java 9 — 11 (**)
  Краткий ответ
  - Новые методы в String
  - Java 9: Модульность
  - Java 9: Методы в Objects: requireNonNullElse() и requireNonNullElseGet()
  - Java 9: List.of(), Set.of(), Map.of(), Map.ofEntries()
  - Java 9: Optional.ifPresentOrElse(), Optional.stream()
  - Java 10: var type-inference
  - Java 11: Files.readString(), Files.writeString()
  - Java 11: Local-Variable Syntax for Lambda Parameters — выведение типов у var-аргументов в лямбда-параметрах
  - Java 11: JEP 321: HTTP Client

  Можно как бонус назвать какие-нибудь:
  - JEP 328: Flight Recorder
  - JEP 335: Deprecate the Nashorn JavaScript Engine
  - JEP 320: Remove the Java EE and CORBA Modules

  но это совершенно необязательно, покажет лишь вашу въедливость при чтении JDK'шных Release Notes :)

- Swing: рассказать про EDT, как им пользоваться (**)
  Краткий ответ

  EDT — тред в котором производится обработка пользовательских действий на UI: движение курсора, нажатие клавиш, скролл, drag'n'drop и.т.д. Соотвественно, все «тяжелые» по времени и ресурсам операции нужно выносить в отдельный worker-тред (`SwingUtils.invokeLater(...)`), чтобы не фризить EDT.

- Swing: перечислить все виды Layout, которые знаете (**)

Краткий ответ

- FlowLayout
- GridLayout
- BoxLayout
- BorderLayout

- Generics: В чем преимущество, как работают? Что такое type-erasure? В чем отличие от шаблонов C++? (**)
  Краткий ответ
  - Типы дженерики обеспечивают параметрический полиморфизм, т.е выполнение идентичного кода для различных типов. Типичный пример — коллекции, итераторы
  - type-erasure — это стирание информации о типе-параметре в runtime. Таким образом, в байт-коде мы увидим List, Set вместо List<Integer>, Set<Integer>, ну и type-cast'ы при необходимости
  - В отличие от дженериков в Java, в C++ шаблоны в итоге приводят к компиляции метода или типа для каждого специфицированного типа параметра (специализация шаблона). Да простят меня здесь адепты C++.

- Generics: Метод принимает ссылку на List<Parent>. Child наследуется от Parent. Можно ли в метод передать List<Child>? (**)
  Краткий ответ

  В типе аргумента нужно указать List<? extends Parent>

- Generics: Ковариантность/контравариантность. Спросить про принцип PECS как бонус
  Краткий ответ
  - Ковариантность — List<? extends T>, если B extends T, то и List<B> extends List<T>
  - Контраваринтность — List<? super T>, если B super T, то и List<B> super List<T>
  - PECS — Producer-Extends-Consumer-Super, метод отдаёт ковариантный тип, принимает контравариантный (прим. автора — последнее интуитивно не очень понятно)

- Регионы памяти в JVM (**)
  Краткий ответ

  Java 8: Metaspace, Old Generation, Young Generation (Eden Space/Survivor Space), Stack, Constant Pool, Code Cache, GC Area.

- Hard-references, weak references, soft-references, phantom-references (***)
  Краткий ответ
  - Hard-references — стандартные ссылки на объекты, которые становится eligible for collection после недостижимости из root set
  - Weak-references — объекты могут быть удалены при наличии слабой ссылки на него в любое время
  - Soft-references — объекты могут удалятся GC при недостатке памяти
  - Phantom-references — объекты не доступны напрямую по ссылкам, перед удалением помещаются в очередь на удаление. Нужны для более безопасной финализации ссылок (вместо finalize)

- Рассказать про classloader'ы и их иерархию. Из за чего, например, может возникать NoClassDefFoundError, NoSuchMethodError? (***)
  Краткий ответ

  Иерархия classloader'ов

  1. Bootstrap
  2. System
  3. Application

  - NoClassDefFoundError может возникнуть, если нужной библиотеки с этим классом нет в classpath
  - NoSuchMethodError может возникнуть из-за несовместимости ваших библиотек, если зависимая библиотека A вызывает метод из старой версии библиотеки B, но в classpath есть более новая версия библиотеки B, с другой сигнатурой этого метода

- Какими способами можно сконструировать объект в Java? (**)
  Краткий ответ

  - Через конструктор
  - Через статический factory-method
  - Через паттерн Builder

- Как идентифицируется класс в Java? (**)
  Краткий ответ

  По его FQDN и classloader'у

- Bytecode: назовите какие-нибудь инструкции и опишите их (**).
  Краткий ответ

  Здесь только краткий список команд:

  - aload
  - aconst
  - astore

  * Попросить описать принцип действия стековой машины, как бонус. Допустим, на примере вызова метода.

- Bytecode: invokevirtual, invokestatic, invokespecial — когда используются?
  Краткий ответ

  - invokevirtual — вызовы методов (в Java все методы виртуальные)
  - invokestatic — вызовы статических методов
  - invokespecial — вызовы конструкторов и приватных методов

Секция Concurrency:

- synchronized. wait/notify/notifyAll. Как есть примитивы аналоги из пакета j.u.c? (**)
  Краткий ответ

  Дальше тезисы:

  - synchronized — ключевое слово, обозначающее скоуп критической секции. Можно ставить напротив объявления метода, или в виде блока в коде.
  - wait() — ожидание треда до тех пор, пока он не будет разбужен другим тредом через notify/notifyAll.
  - У wait() есть перегруженные версии с таймаутами.
  - Тред ставится в wait-set на объекте
  - Перед вызовом wait() нужно захватить монитор на данном объекте (через synchronized)
  - Магия wait() — он отпускает лок на мониторе объекта после вызова, так чтобы в дальнейшем другой тред мог захватить монитор и вызвать notify/notifyAll
  - notify() — будит **один из** ожидающих тредов, но **Важно!** — лок на объекте не отпускает, т.е ожидающий тред разбужен будет, но с ожиданием входа в критическую секцию объекта (т.к как будто остановился на synchronized). Так что если после notify есть тяжелые операции, это затормозит ожидающий тред, т.к тред с notify еще не отпустил монитор
  - notifyAll() — будут разбужены все треды в wait-set, но при этом далее между тредами происходит contention («сражение») за монитор
  - Тред на wait() может быть разбужен также через interrupt, или через spurious wake-up, или по таймауту
  - Так что условие выполнения, которого ожидает тред, проверяется в цикле while, а не в if
  - Примитив-аналог — Condition

- volatile. happens-before. (**)
  Краткий ответ

  - Ключевое слово volatile устанавливает отношение happens-before над операциями записи-чтения на поле
  - Таким образом, операции чтения из читающих тредов будут видеть эффекты записи пишущих тредов.
  - В частности, решается проблема double checked locking. Для double/long типов есть проблема атомарности, она решается через атомики

- AtomicInteger, AtomicLong, AtomicBoolean, AtomicDouble (**)
  Краткий ответ

  - Атомики предоставляют возможность изменения переменной в нескольких потоках без эффекта гонок.
  - Например, 10 тредов инкрементят AtomicInt = 0, основной тред ждет их выполнения через countdown-latch, далее проверка атомика должна показать 10.
  - Основной механизм под капотом атомиков — цикл cas (compare-and-set). На примере increment:
    1. Читаем старое значение

2. Перед set'ом проверяем старое значение, если оно не изменилось, сетаем старое + 1

3. Если изменилось, в след. итерации получаем «новое» старое, далее см. п. 1

- Редкий вопрос — как поймать exception из другого треда? (***)
  Краткий ответ

  Зарегистрировать Thread.UncaughExceptionHandler

- ReentrantLock (**)
  Краткий ответ

  Примитив синхронизации, с помощью которого можно установить границы критической секции. Тред, перед входом в критическую секцию должен сделать захват с операцией `lock()`, после выхода из крит. секции — сделать `unlock()`. Другой тред в это время ожидает на lock'е (можно указывать таймаут ожидания), либо может проверить доступность через `tryLock()`.

  ReentrantLock обязательно нужно освобождать (такое кол-во раз, сколько раз он был захвачен), в противном случае будет thread starvation у других тредов, ожидающих у границы критической секции.
  ReentrantLock может быть «честным» (fairness = true), тогда приоритет отдается тредам, ждущих на нем наибольшее кол-во времени, но это вроде как уменьшает производительность

```
lock.lock();  // block until condition holds
try {
        // ... method body
} finally {
        lock.unlock()
}
```

- Countdown Latch/Cyclic Barrier (**)
  Краткий ответ

  CountdownLatch («защелка») — примитив синхронизации, с помощью которого, например, основной thread может ожидать выполнения работы остальных N тредов. Треды, выполняющие работу, выполняют countDown() на защелке, основной тред ожидает на операции await(). Когда счетчик достигает нуля, основной тред продолжает работу.

  Для синхронизации N тредов (все ждут всех) и переиспользования используется CyclicBarrier, ему также можно указывать действие (через Runnable), выполняемое после синхронизации всех-со-всеми

- ThreadLocal (**)
  Краткий ответ

  Класс, предоставляющий доступ к операциям get/set в области видимости треда. Под капотом содержит кастомную реализацию мапы со слабыми ссылками на ключи-треды. Каждый тред имеет доступ только к своим данным.

- Создание singleton? (**)
  Краткий ответ

  - Наивным способом, с проверкой на null статического поля

- Double checked locking (объяснить проблемы double checked locking)
- Простой — инициализация статического поля, или через enum, т.к ленивая инициализация thread-safe по-умолчанию

- Способы запустить поток? (***)
  Краткий ответ
  - Переопределить Thread#run(), запустить через Thread#start()
  - new Thread(Runnable).start()
  - Через ExecutorService, используя utility-класс Executors

- ConcurrentHashMap (**)
  Краткий ответ

  ConcurrentHashMap имеет разные реализации в 1.7 и 1.8 (что стало для меня неожиданностью).
  Раньше параллелизм основывался на идее сегментирования хэштаблицы на основе заданного уровня параллелизма.
  Начиная с Java 8 — это единый массив бакетов с lock-free (локинг на первой ноде бакета с cas-циклом) и конкурентным ресайзингом.
  Частичная реализация ConcHashMap (аля Java 8) с нуля от @kuptservol

- ConcurrentSkipListMap
  Краткий ответ

  Lock-free структура данных, хранящая упорядоченный набор элементов, с O(log N) временем доступа/удаления/вставки и weakly-consistent итераторами. Под капотом содержит структуру SkipList, предоставляющую собой слои связных списков, от верхнего к нижнему, элементы верхнего списка ссылаются на элементы нижнего списка под ними. Вероятность попадания элемента при вставке в самый нижний список — 1.0, далее она равняется p (либо 1/2, либо 1/4 как правило) — вероятности попадания элемента из нижнего списка в верхний. Таким образом, на самом верхнем будет вставлено минимальное кол-во элементов. Skiplist — вероятностная структура данных. Подробнее и доходчиво про skiplist описано здесь. Полезна, если стоит задача отсортировать поток событий, одновременно читаемый несколькими тредами, которым нужно делать срез по временному интервалу. Более медленные операции по сравнению с ConcurrentHashMap

- Thread states (**)
  Краткий ответ
  - NEW
  - RUNNABLE
  - BLOCKED(monitor lock)
  - WAITING(Thread.join)
  - TERMINATED

- Deadlocks, условия наступления, как избежать: (***)
  Краткий ответ
  - Условия наступления — эксклюзивность доступа к ресурсам, наличие циклов в графе ожиданий ресурсов, отсутствие таймаутов на ожидание

- Как избежать — задать порядок доступа к ресурсам. Всегда обращение в порядке либо Thread1->Thread2, либо Thread2->Thread1

- ThreadPoolExecutor — описать механизм работы, св-ва, частности (fixed threadpool, scheduled, single thread executor) (***)
  Краткий ответ

  ThreadPoolExecutor — средство контроля исполнения параллельных задач, задействует один из свободных тредов в общем пуле, или ставит задание в очередь, если таковых нет, или достигнуты определенные условия (ниже)

  Основными св-вами ThreadPoolExecutor являются *corePoolSize* и *maxPoolSize*. Если текущее количество тредов в пуле < *corePoolSize* — новый тред будет создаваться в независимости от того, есть ли в пуле незанятые треды. В промежутке между *corePoolSize* и *maxPoolSize* тред будет создаваться в том случае, если заполнена очередь задач, и удаляться спустя *keepAliveTime*. Если кол-во тредов стало >= *maxPoolSize* — новые треды не создаются, а задачи ставятся в очередь.

  Есть возможность регулировать поведение очереди:

  - Direct handoffs: немедленная передача задачи тредпулу. Нет понятия очереди задачи. Если свободных тредов нет — кидается exception. Применимо при неограниченных maxPoolSize, но нужно понимать проблему при быстрых записях и медленных чтениях, что может спровоцировать непомерное потребление ресурсов

  - Unbounded queues: очередь без ограничений. Задачи будут добавляться в нее при превышении corePoolSize, при этом *maxPoolSize* будет игнорироваться. Unbounded queues имеют проблемы потребления ресурсов при больших нагрузках, но сглаживают рост тредов при пиках.

  - Bounded queues: очередь с ограничениями. Задачи будут добавляться в очередь до достижения некоего capacity. Для достижения наилучшей производительности нужно понимать размеры *corePoolSize* и самой очереди и чем можно пожертвовать — перфомансом (малый *corePoolSize* и большая очередь), или же памятью (ограниченная очередь, большой corePoolSize)

  Частности ThreadPoolExecutor:

  - ScheduleThreadPoolExecutor — применяется для периодичных по времени задач

  - fixed thread pool — частность ScheduleThreadPoolExecutor'a с настроенным *corePoolSize* и unbounded queue

  - single thread executor — тредпулл, с *corePoolSize* = 1, гарантирующий последовательное выполнение задач из очереди

Секция Collections:

- Рассказать про java.util.collection. (*)
  Краткий ответ

  - Iterable — реализуют коллекции, по которым можно проитерироваться

  - Collection — общий интерфейс для коллекций

  - List (стандартная реализация ArrayList) — список с массивом элементов, с возможностью случайного доступа элемента по индексу за O(1), вставкой/удалением со сложностью O(n)

  - Set (стандартная реализация HashSet) — мн-во элементов без дубликатов. Нет возможности доступа по индексу, есть вставка и удаление за O(1)

- Map (стандартная реализация HashMap) — мн-во пар элементов «ключ-значение». Доступ по ключу/добавление/удаление за O(1) при оптимальном случае, O(n) — при вырожденном

Обзор по коллекциям от JournalDev

- Устройство ArrayList, LinkedList, HashMap, HashSet. Когда следует использовать. Контракт equals/hashcode для Map, Set (*)
  Краткий ответ

  Обзор по коллекциям от JournalDev

- Итератор по коллекции, его св-ва и интерфейс (*)
  Краткий ответ

  - Может только один раз проходить по коллекции. Для прохождения в двух направлениях есть ListIterator
  - Если в foreach цикле структурно модифицировать коллекцию, при последующем обращению к элементу (неявно через итератор) получим ConcurrentModificationException (fail-fast)
  - hasNext(), next() — основные методы

- Hashtable vs HashMap (*)
  Краткий ответ

  - Hashtable — legacy, thread safe, методы синхронизированы, поэтому работа с Hashtable (обращение, удаление, добавление) в целом накладнее
  - HashMap — не thread-safe

- Java 8,11: новые методы в Map (**)
  Краткий ответ

  - Java 8: compute
  - Java 8: computeIfAbsent
  - Java 8: computeIfPresent
  - Java 8: forEach
  - Java 8: putIfAbsent
  - Java 11: factory-методы of()

- LinkedHashMap, зачем он нужен (**)
  Краткий ответ

  - Позволяет сохранять порядок вставки пар key-value в Map
  - Каждый entry содержит помимо hash, value, next (следующий элемент в бакете) также поля, указывающие на предыдущий и следующий элементы относительно порядка вставки

- Устройство TreeMap (**)
  Краткий ответ

  - Сбалансированное красно-черное дерево

- Реализует интерфейс NavigableMap, что позволяет возвращать из него элементы, больше (меньше) указанного, либо range элементов, находящийся в определенных границах

- Какой контракт Comparator должен соблюдать?
  Краткий ответ

  Быть согласованным с equals()

- Есть ли способ сделать enum ключом Map? (**)
  Краткий ответ

  EnumMap — массив, по размеру соотвествующий кол-ву элементов в enum'e. Индекс элемента массива соотвествуют ordinal'y из enum'a

- Расскажите про CopyOnWriteArrayList/CopyOnWriteHashSet (**)
  Краткий ответ

  - CopyOnWriteArrayList — иммутабельный list, при добавлении/апдейте/удалении элементов из которого пользователь получает новую модифицированную копию данного списка
  - CopyOnWriteHashSet — иммутабельный set, при добавлении/апдейте/удалении элементов из которого пользователь получает новую модифицированную копию данного set'a

- IdentityHashMap — когда используется? (**)
  Краткий ответ

  IdentityHashMap — используется, только если нужно проверять идентичность двух ссылок, а не эквивалентность двух объектов по ним. Например, если нужно отслеживать уже посешенные ноды в графе, или строить карту объекты-прокси. IdentityHashMap представляет из себя не классическую хэштаблицу со связанными списками, это linear probing map (бонус за объяснение работы linear probing)

- Интерфейсы Queue/Deque и их реализации (***)
  Краткий ответ

  - Queue — коллекция, предоставляющая возможности упорядочения элементов в порядке вставки согласно принципу FIFO. Поддерживает два набора операций добавления/удаления/взятия элемента с конца — с возвращением спец. значения при исключительной ситуации (пустая или заполненная очередь), и с киданием exception'a
  - Deque — коллекция, предоставляющая возможность вставки значений в начала и в конец, позволяющая организовать очереди по принципам FIFO/LIFO. Предпочтительна для реализации стэка вместо Stack. Ниже приведен набор операций для Deque:

    Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

  - BlockingQueue — очередь, блокирующая операции чтения take при пустой очереди или операции записи put при полной очереди. Есть наборы операций и с неблокирующей семантикой. Ниже определен полный список операций

чтения/записи.

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

- ArrayBlockingQueue — кольцевой bounded-buffer с внутренним capacity и fairness policy. Все операции чтения и записи защищены внутренним lock'ом (*j.u.c.l.ReentrantLock*), неявно связанным с двумя condition'ами — notFull, notEmpty. Соотвественно, если пишущий тред, захвативший общий lock, застрял на condition'e notFull, он неявно освобождает lock, чтобы читающий тред мог освободить очередь, и сигнализировать producer'у о том, что можно продолжать. Ровно и наоборот, читающий тред также захватывает общий lock, застревает на notEmpty, неявно освобождает lock, чтобы пишущий тред мог положить новый элемент и просигналить о непустой очереди. Т.е семантика такая же, как и у synchronized/wait/notify/notifyAll. Fairness = true позволяет предотвратить ситуации thread starvation для консьюмеров и продюсеров, но снизит производительность.

- LinkedBlockingQueue — очередь, в которой элементы добавляются в связанный список, основанная на two-lock queue, описанного в статье Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms by Maged M. Michael and Michael L. Scott.

Секция IO:

- InputStream, OutputStream и их buffered версии (*)
  Краткий ответ

  Далее, для краткости *InputStream* — is, *OutputStream* — os
  is — побайтное чтение из сокета/файла/строки/другого байтового массива
  os — побайтная запись в сокет/файл/другой байтовый массив
  Buffered-версии нужны для оптимизации чтения/записей через отдельный буффер

- Зачем нужен Reader? (*)
  Краткий ответ

  Reader позволяет указать Charset при чтении

- Serializable, serialVersionUID (*)
  Краткий ответ

  Классы, чьи объекты подвергаются сериализации/десериализации должны реализовывать marker интерфейс Serializable (и иметь статическое поле serialVersionUID для указании при сериализации, с какой версией класса данный объект был сериализован. Если serialVersionUID из сериализованного представления не совпадает с serialVersionUID класса «на том конце провода» — то кидается exception)

  На практике, уже довольно редко используется, т.к тем же Jackson/GSON не обязательно наличие данного интерфейса для сериализации

- try-with-resources. AutoCloseable (*)
  Краткий ответ

try-with-resources — краткая замена стандартному try..catch..finally. Закрывает ресурс после выхода из секции try-with-resources. Ресурс должен имплементить интерфейс AutoCloseable.

«Ресурс» в данном контексте — это класс, представляющий собой соединение/сокет/файл/поток

```
try (InputStream is = new FileInputStream("/path/to/file.txt")) {
        ...
}
```

Секция Exceptions:

- Отличие checked-exception/unchecked-exception. Error, Exception, RuntimeException (*)
  Краткий ответ
  - Checked exceptions (проверяемые исключения). В JDK представлены классом Exception. Исключения, которые нельзя проигнорировать, их обязательно нужно обрабатывать, либо специфицировать в сигнатуре метода, для обработки выше. Как правило, считаются дурным тоном, т.к код со мн-вом конструкций try..catch плохо читабелен, к тому же добавление новых пробрасываемых исключений в сигнатуре метода может сломать контракт вызова у пользователей данного метода.
  - Unchecked exceptions (непроверяемые исключения). В JDK это класс RuntimeException. Можно игнорировать, не требуют обработки через try..catch, или указания в сигнатуре через throws. Минус такого подхода — у вызывающей стороны нет никакого понимания, как обрабатывать ситуацию, когда под капотом «рванет»
  - Error — ошибки, кидаемые JVM в результате нехватки памяти (OutOfMemoryError), переполнения стэка (StackOverflowError) и.т.д

Полезные ссылки:

1. Утечки памяти — статья на TopTal.com
2. Частичная реализация ConcHashMap (аля Java 8) с нуля от @kuptservol
3. Структура SkipList
4. Обзор коллекций от JournalDev
5. Обзор коллекций от @vedenin1980
6. Обзор пакета java.util.concurrency (Java 7) от @Melnosta
7. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms by Maged M. Michael and Michael L. Scott (для сильных духом)

Теги: java, backend, собеседования, программирование

Хабы: Java, Карьера в IT-индустрии

Лучше алгоритмы на доске спрашивайте. По этим вопросам можно взять человека который потратил пару дней на подготовку к ним, и не взять всех остальных.

В вопросах в основном мелочи которые при необходимости гуглятся за 15 минут, но наизусть не помнятся если совсем недавно именно с ними не работал.

Sm1le291 9 марта 2020 в 17:29 0

Вы предлагаете другую крайность, мало кто в состоянии на доске написать алгоритмы, если заранее не готовился. Опять же проверите готовился ли человек к алгоритмам, которые никогда не будет использовать на практике

BugM 9 марта 2020 в 17:38 +1

Вы fizzbuzz или определение простое число или нет (любым способом) на доске не напишите? Не Гугл же.

Это гораздо лучше чем спрашивать «Generics: Ковариантность/контравариантность.» большая часть разработчиков слов то таких не знает. Ибо не надо. А если понадобится то быстрогугл сразу решает пробелму.

sshikov 9 марта 2020 в 17:54 +1

Проблему непонимания ковариантности гугл не решает. Насчет «не надо» — ну в общем да, не особо часто.

BugM 9 марта 2020 в 18:19 +7

Определение из Вики:

```
Ковариантностью называется сохранение иерархии наследования исходных типов в производных типах в том же пор
ядке. Так, если класс Cat наследуется от класса Animal, то естественно полагать, что перечисление IEnumerab
le<Cat> будет потомком перечисления IEnumerable<Animal>. Действительно, «список из пяти кошек» — это частны
й случай «списка из пяти животных». В таком случае говорят, что тип (в данном случае обобщённый интерфейс)
IEnumerable<T> ковариантен своему параметру-типу

Контравариантностью называется обращение иерархии исходных типов на противоположную в производных типах. Та
к, если класс String наследуется от класса Object, а делегат Action<T> определён как метод, принимающий объ
ект типа T, то Action<Object>  наследуется от делегата Action<String>, а не наоборот. Действительно, если
«все строки — объекты», то «всякий метод, оперирующий произвольными объектами, может выполнить операцию над
строкой», но не наоборот. В таком случае говорят, что тип (в данном случае обобщённый делегат) Action<T> ко
нтравариантен своему параметру-типу T.
```

Чтобы найти прочитать и понять 5 минут достаточно. В работе, разговорах итд не используется никогда. И через неделю-месяц это определение забудется за ненадобностью.

И там все в списке примерно такое. «Назвать методы Object» equals и hashCode я назову сразу. toString немного подумав. Остальное только по документации или исходникам JDK. Ибо оно не используется практически никогда.

sshikov 9 марта 2020 в 19:11 0

>Чтобы найти прочитать и понять 5 минут достаточно.
Не совсем согласен. Во-первых, увы не всем достаточно, а главное, оно может и появляется, какое-то понимание, но очень краткосрочное.

>И через неделю-месяц это определение забудется за ненадобностью.
Ну я собственно этоже самое и имел в виду, когда говорил, что гугление не решает проблему понимания этой темы. Потому что настоящее понимание позволяет такое же определение написать/вывести, исходя по большей части из «здравого смысла».

12,0 Карма 0,0 Рейтинг

Олег @otvorot2008
Java разработчик

ПОХОЖИЕ ПУБЛИКАЦИИ

2 апреля 2018 в 17:12
Программирование согласно контракту на JVM
+15 5,2k 27 8

22 апреля 2014 в 10:23
Горизонтальное масштабирование небольших Web-приложений на Java (вопросы собеседований)
+35 30,7k 350 87

17 апреля 2010 в 01:29
Java-ассемблер, мета-программирование и JPA
+24 8,5k 48 24

Комментарии 74

BugM 9 марта 2020 в 16:48 +5

BugM 10 марта 2020 в 00:20 +1

> Потому что настоящее понимание позволяет такое же определение написать/вывести, исходя по большей части из «здравого смысла».

Представим себе собеседование. Судя по формулировкам на нем прямо так и спросят. А расскажите о Ковариантность/контравариантность в дженериках Джавы. Тут любой кто не помнит определение будет в тупике и больше чем «Эээ. А что эти слова значат?» Выдавить из себя не сможет. В итоге собеседование завалено, хотя человек просто не знает определение и слов. Которые не используются в работе.

И опять таки остальное ровно такое же.
«Почему хранить пароль предпочтительнее в char[]/byte[], а не в String? (**)»
Вот прямо так ведь и спросят. Я честно затруднюсь ответить. На мой вкус хуже. Потому что граблей на которые можно наступить заметно больше, а плюсов нет.

И даже после их «правильного» ответа

> Строка в виде литерала сразу раскрывает пароль, плюс она всегда хранится в string-пуле
> byte[]/char[] возможно сбросить после использования, и удалить все ссылки на него

я все еще затрудняюсь. И скорее всего буду спорить. Если у зломышленника есть доступ к памяти процесса в котором пароли проверяются, то уже спасать пароли поздно. Пароли точно утекли. Исходим из этого.
Защита «Строка в виде литерала сразу раскрывает пароль, а переменная password четко видимая в любом отладчике видимо не раскрывает?» как-то по детски выглядит.

sshikov 10 марта 2020 в 22:28 0

>Которые не используются в работе.
Вы никогда List что-ли не пишете? А как только вы его написали, и ежели у вашего MySuperObject есть наследники, вы должны понимать, что у вас имеет место, ко/контра либо инвариантность. А если не понимаете — вероятность накосячить весьма велика. Названий можете не знать, не вопрос. Но реально не понимать разницу, хотя бы на интуитивном уровне...? А аббревиатуру LSP вы тоже никогда не видели что-ли? Я если что не конкретно про кого-то лично, пусть будет абстрактный разработчик в вакууме.

Ну то есть, я могу себе представить деятельность, где это не нужно совсем. И легко могу представить разработчика, который вообще не понимает generics, при этом достаточно долго и сравнительно успешно работает. Но могу и наоборот. В принципе, второй случай — это наверное скорее разработка фреймворка. А первый — использование.

>И опять таки остальное ровно такое же.
Ну, про остальное в целом — скорее согласен.

BugM 11 марта 2020 в 00:19 0

> Вы никогда List что-ли не пишете? А как только вы его написали, и ежели у вашего MySuperObject есть наследники, вы должны понимать, что у вас имеет место, ко/контра либо инвариантность. А если не понимаете — вероятность накосячить весьма велика. Названий можете не знать, не вопрос. Но реально не понимать разницу, хотя бы на интуитивном уровне...?

Особенность джавы в том что это работает в одну сторону. А мы о Джаве говорим. И соответственно эти определения получаются чисто академическими. Так что, что как и кого наследует в шаблонах расскажу без проблем. А вот про вариативности извините, не смогу. Они были в лучше случае на каком-то курсе и успешно забылись за ненадобностью. Соответственно вопрос на собеседовании в котором будут они звучать вызовет замешательство и непонимание.

> А аббревиатуру LSP вы тоже никогда не видели что-ли?

```
Значения LSP:

    LSP — англ. label switch path — виртуальный канал, туннель, путь в протоколе MPLS.
    LSP — англ. layered service provider — технология ОС Windows.
    LSP — англ. lightest supersymmetric particle — легчайшая суперсимметричная частица.
    LSP — англ. Liskov substitution principle — принцип подстановки Барбары Лисков.
    LSP — белорусский музыкант и одноимённый белорусский музыкальный коллектив.
```

sshikov 11 марта 2020 в 14:37 0

>Так что, что как и кого наследует в шаблонах расскажу без проблем. А вот про вариативности извините, не смогу.
Так этож одно и тоже, по сути. То есть, фактически вы не знаете названий для этого?

LSP — это принцип подстановки Лисков. Ну то есть, я опять же вполне допускаю вариант, что не зная ничего этого вы можете успешно много лет работать. Но только до какого-то предела — примерно пока вам не потребуется написать свою «коллекцию».

BugM 11 марта 2020 в 19:47 0

Если по русски "Наследники в @Override функциях не должны делать ничего неожиданного." Я даже больше сказать могу. Весь код должен работать самым ожидаемым способом. Не надо форматировать диск в функции getValue() оверрайдит она что-то или нет не важно.

Это действительно такая странная и редкая вещь вещь что ее надо знать и обсуждать отдельно? Вроде обычный здравый смысл.

sshikov 11 марта 2020 в 19:48 0

То что вы говорите — это совсем не про ко/контравариантность.

BugM 11 марта 2020 в 20:47 0

Это про написание нормального кода. Высказанное обычным языком. Без терминов которые не используются в реальности. Ну и максимально близко к теме.

Начали с одного из предлагаемых вопросов. Где предполагалось что пришедший на собеседование знает что означает слово ковариантность и выдаст определение из учебника. И моего возражения что это очень странный вопрос для собеседования.

Neikist 9 марта 2020 в 18:09 0

Я вот слова эти знаю, даже знаю примерно какую ситуацию они описывают, но вечно путаю какое из них что значит ибо они черт побери почти одинаково написаны/звучат. А на практике я их встречал только в документации да книгах, в разговорах с коллегами ни разу не слышал.

vladimir_dolzhenko 9 марта 2020 в 16:56 +7

Используете `String.intern()`? Спасибо, до свидания.

sshikov 9 марта 2020 в 17:53 +2

Не, ну почему же сразу? Но вот вопрос к автору по этому поводу у меня тоже имеется — а нафига спрашивать вот такое? Вот вы прям без String.intern() ну никак, вообще невозможно разрабатывать в вашей компании?

Если что, я видел замечание, что автор не предлагает задавать все вопросы из списка, но причина их появления в списке все-же осталась пока до конца не ясной.

poxvuibr 9 марта 2020 в 17:57 +3

> Не, ну почему же сразу?

Потому что это создаёт проблемы на ровном месте, а что хорошего получаешь взамен — непонятно.

sshikov 9 марта 2020 в 18:09 0

Я поэтому и хотел бы задать вопрос автору — нафига? А точнее, переформулировал бы подход к подобным вопросам вообще. Не «Что такое String.intern()?», как у автора, а акцентировать на то, для чего это применяется.

>а что хорошего получаешь взамен — непонятно.
И вот уже в зависимости от ответа на этот вопрос — обсуждать дальше технические тонкости. Например, почему вам непонятно, что можно получить потенциальный прирост производительности (в определенных условиях, разумеется)? А дальше уже обсуждаем, готовы ли вы купить этот прирост взамен на потенциальные проблемы (и расспросить, какие именно).

Ну т.е., любой из вопросов этого списка, на первый взгляд, как раз для меня выглядит лишь поводом, чтобы поговорить. А чтоб поговорить — сойдет любая в общем тема — но все-таки, лучше что-то более широко используемое.

poxvuibr 9 марта 2020 в 18:37 +1

> Например, почему вам непонятно, что можно получить потенциальный прирост производительности (в определенных условиях, разумеется)?

Скажу честно, я не очень понимаю, как можно получить прирост в производительности с помощью String.intern(), а вот, как получить с его помощью уменьшение производительности в общем-то понятно.

С помощью String.intern() можно сэкономить память, но если вопрос производительности вас хоть как-то волнует, лучше экономить память другими методами.

> Ну т.е., любой из вопросов этого списка, на первый взгляд, как раз для меня выглядит лишь поводом, чтобы поговорить

Поддерживаю

BugM 9 марта 2020 в 18:40 0

Ну чисто в теории сравнение указателей дешевле чем сравнение строк. Прирост производительности получить можно. Но в проде так не надо делать, естесвенно.

poxvuibr 9 марта 2020 в 18:50 0

> Ну чисто в теории сравнение указателей дешевле чем сравнение строк.

Это конечно правда.

> Прирост производительности получить можно.

Да? Каким образом?

> Но в проде так не надо делать, естесвенно.

Если прирост в производительности действительно будет, почему не воспользоваться этим приёмом на проде?

BugM 9 марта 2020 в 19:27 0

Давайте натяну сову на глобус.
Есть у нас стейт машина. Большая и сложная. У которой состояние это строка. И внутри море функций проверяющих текущее состояние и если оно нужное что-то делающих. Замена .equals на == ускорит работу.

Потому что всегда есть более хороший способ сделать тоже самое.

poxvuibr 9 марта 2020 в 19:35 0

> Замена .equals на == ускорит работу.

Если вы будете где-то постоянно прогонять строки через String.intern(), то не ускорит. А если не будете, то при чём тут String.intern() ?

BugM 9 марта 2020 в 19:39 0

Так один раз на входе в эту стейт машину. И один раз при загрузке.
И дальше много-много раз ==.

poxvuibr 9 марта 2020 в 19:45 0

> Так один раз на входе в эту стейт машину. И один раз при загрузке.

Получается 2 раза за всё время работы программы? Или я что-то не понял?

Если я всё понял правильно, то String.intern() тут не нужен, нужен справочник внутри стейт машины.

BugM 9 марта 2020 в 19:54 0

При загруке intern на все варианты делаем. Для каждой входной строки один раз и далее == много раз.

Вы просили пример где быстрее будет. Я натянул сову на глобус, раз так надо. Естесвенно есть варианты сделать тоже самое лучше.

poxvuibr 9 марта 2020 в 20:27 0

> Вы просили пример где быстрее будет. Я натянул сову на глобус, раз так надо. Естественно есть варианты сделать тоже самое лучше.

Чтобы в описанном вами случае применить String.intern() нужно во-первых знать о том, что такое существует, а во-вторых зачем-то решить применить именно его, несмотря на то, что первая мысль тут — применить словарь.

Для пояснения приведу пример из другой области, где хорошее решение не так очевидно, как плохое. ,

Вот есть у нас объект, который нужно собрать, а потом больше он меняться не будет. Очевидное и плохое решение тут — сделать ему сеттеры. Неочевидное и хорошее — применить паттерн билдер.

Я не могу придумать ситуации, где плохое решение со String.intern() напрашивается само по себе.

TheKnight 10 марта 2020 в 01:42 +1

Мне кажется собирать стейтмашину на строках — плохая идея. Чем вам Enum не угодили?

sshikov 9 марта 2020 в 19:29 0

>Но в проде так не надо делать, естесвенно.
Почему не надо? Я видел утверждения, что этого добра полно как в коде runtime, так и в продуктах Apache. Это не гарантия конечно, но и считать всех авторов дураками заранее тоже не приходится.

vladimir_dolzhenko 9 марта 2020 в 20:43 0

Учитывая того как работает String.intern() авторам Jackson были предоставлены доказательства (Jackson Issue #332) того, что не надо использовать этот подход для экономии памяти, вместо этого лучше использовать свой дедупликатор (который контроллируемо может экономить память), но автор оказался упёртым, даже не смотря на предоставление прод профиля.

Читать https://shipilev.net/jvm/anatomy-quarks/10-string-intern/

sshikov 9 марта 2020 в 21:35 0

Ну, в принципе логично. Хотя ваш подход к интервью я все равно не понимаю. У любого механизма есть свои ограничения, свои достоинства и недостатки. Если человек их изложит — мы возможно получим представление о том, как он думает. Даже если в конкретном частном случае он ошибается.

vladimir_dolzhenko 9 марта 2020 в 23:08 0

Знать о вещи, которая вредная, можно конечно, но спрашивать такое на собеседовании — это полная дичь.

Если человек не знает её, значит поставить человека в неловкое положение, на вопрос потратили время (минимально 5 минут из обычно часа собеседования), у кандидата появится неприятное ощущение того, что его валят — и главный вопрос — зачем? Реальная практическая ценность близка к нулю.

Как по мне, кто начинает разговор о intern как правило сами не знают о чём оно.

sshikov 9 марта 2020 в 23:40 0

Ну если вы посмотрите комменты — то вопрос «зачем» был первым, который я тут задал. На мой взгляд, спрашивать надо вообще не фичи языка/фреймворка и т.п., а то, как человек решает задачи — т.е. задача должна быть исходной точкой, а способ решения — вторичен (и если уж человек предложит intern как способ, и сможет это как-то обосновать — мы почти наверное многое узнаем о его уровне).

sshikov 9 марта 2020 в 19:05 0

В некоторых случаях экономия памяти — это уже практически прямое повышение производительности.

>>Прирост производительности получить можно.
>Да? Каким образом?
Сравнение длинных строк — это операция намного более дорогая, чем сравнение указателей. Ну т.е. в зависимости от средней длины можно получить очень даже ого-го.

Ну а в целом — хороший же вопрос, если к нему подойти вот с такой стороны, разве нет?

poxvuibr 9 марта 2020 в 19:17 0

> В некоторых случаях экономия памяти — это уже практически прямое повышение производительности.

Да, но только если действия, которые предприняты для экономии памяти не убивают производительность.

> Сравнение длинных строк — это операция намного более дорогая, чем сравнение указателей.

Небольшая поправочка. Сравнение длинных строк **одинаковой длины** — операция намного более дорогая, чем сравнение указателей. Если строки разной длины, то выигрыш уже не такой большой.

> Ну т.е. в зависимости от средней длины можно получить очень даже ого-го.

Если в приложении много строк одинаковой длины и оно в основном занимается сравнением строк, то может быть получится что-то выиграть. Но выигрыш будет сожран вызовами String.intern() и в результате производительность скорее всего упадёт.

> Ну а в целом — хороший же вопрос, если к нему подойти вот с такой стороны, разве нет?

В целом, да, хороший, единственное что обычно получается обсудить не сам String.intern(), а то, как он мог бы работать и какие преимущества мог бы дать.

 **vladimir_dolzhenko** 9 марта 2020 в 20:28 0

Вы точно понимаете, что знаете как устроен String.intern() внутри ?

 **sshikov** 9 марта 2020 в 20:33 0

А почему вы решили, что нет? Если что, про память — это было совсем о другом.

 **poxvuibr** 9 марта 2020 в 17:53 +1

volatile не только устанавливает отношение happens-before, это ключевое слово ещё и делает чтение и запись переменных атомарными. Необходимости использовать для этого атомики нет.

И сделать ключом Enum можно, даже если используется обыкновенный HashMap, работат будет только не очень быстро

 **kacetal** 9 марта 2020 в 19:38 −1

Нет не делает, i++ волатайл не являются атомарной операцией в отличии от AtomicInteger с его методом incrementAndGet

 **poxvuibr** 9 марта 2020 в 19:41 +1

> Нет не делает

Нет, делает))

> i++ волатайл не являются атомарной операцией

I++ это не одна операция. Это сначала чтение переменной, а потом запись в переменную. volatile делает атомарными чтение и запись, но не их комбинации.

 **kacetal** 9 марта 2020 в 20:34 −1

Извините, но чтение или запись и так являются атомарными операциями и без волатайл. Волатайл нам дает слабую синхронизацию, но не атомарность.

 **poxvuibr** 9 марта 2020 в 20:39 +1

> Извините, но чтение или запись и так являются атомарными операциями и без волатайл.

Только для переменных размером четыре байта и меньше. Для long и double гарантии атомарности чтения и записи нет.

> Волатайл нам дает слабую синхронизацию, но не атомарность.

У volatile двойная семантика. Это ключевое слово создаёт отношение happens-before между записью и чтением и делает запись и чтение атомарными.

kacetal 10 марта 2020 в 00:10 −1

Вы правы насчет примитивных double и long, но это скорее исключение. Все остальные операции чтения и записи атомарны и без волатайл.
Еще раз, happens-before это не про атомарность, это про видимость значения. Волатайл обеспечивает видимость, но не атомарность.

poxvuibr 10 марта 2020 в 01:11 0

> Вы правы насчет примитивных double и long, но это скорее исключение. Все остальные операции чтения и записи атомарны и без волатайл.

Мне сложно оспорить ваше утверждение, особенно если учесть, что вы практически процитировали мой комментарий, на который отвечаете))

> Еще раз, happens-before это не про атомарность, это про видимость значения.

Не совсем. Отношение happens-before даёт гарантии, что изменения в памяти, сделанные до события, будут видны после события, так что речь идёт не о видимости одного значения, а о видимости значений всех переменных, даже если на них нет модификатора volatile.

> Волатайл обеспечивает видимость, но не атомарность.

volatile создаёт отношение happens-before между записью и чтением и делает запись и чтение атомарными. Такие дела.

kacetal 10 марта 2020 в 09:25 0

Мне кажется мы с вами говорим о разных вещах, если отбросить примитивы long double, можете мне объяснить что по по вашему значит не атомарное чтение или запись например int или String?

poxvuibr 10 марта 2020 в 11:03 0

> Мне кажется мы с вами говорим о разных вещах

Я говорю о том, что делает volatile. Вы, мне кажется, тоже.

> если отбросить примитивы long double

Не надо отбрасывать long и double)), это неотъемлемая часть джавы

> можете мне объяснить что по по вашему значит не атомарное чтение или запись например int или String?

Я могу объяснить, что значило бы неатомарное чтение int или String, но, как вы неоднократно повторили, большого смысла в этом нет, так как оно атомарно по спецификации.

Iqorek 10 марта 2020 в 13:33 +2

> операции чтения и записи атомарны и без волатайл.

Чтение без волатайл, читает значение из кэш, если оно там есть, которое может отличаться от значения в основной памяти. Такое чтение безопасно, пока с данным куском памяти работает один поток, потому что, когда он сделает

запись, обновится так же и значение в кэше. Но если есть 2 потока и которые имеют разный кэш, когда один поток поменяет значение, в кэше второго останется старое.
Волатайл говорит о том, что значение всегда нужно читать из основной памяти, а не из кэш. Поэтому чтение с волатайл будет медленней чем без, но при этом есть гарантия, что другой поток получит корректное значение. Запись всегда идет в основную память.
По крайней мере так все было лет 10 назад, возможно с тех пор что то изменилось.

kacetal 12 марта 2020 в 00:10 0

Спасибо, но как это противоречит моему определению атомарности?

lqorek 12 марта 2020 в 00:57 0

Нашел хорошую цитату «под атомарностью операции зачастую подразумевают видимость ее результата всем участникам системы, к которой это относится (в данном случае — потокам)».

poxvuibr 12 марта 2020 в 02:20 +1

Ну я бы сказал, что атомарная операция это такая операция, у которой нет видимого кому-либо промежуточного состояния.

dim2r 9 марта 2020 в 22:39 0

volatile применяет все барьеры памяти перед записью или чтением для обеспечения полной когерентности кэша процессоров. Но для рядовых джавистов протокол MESI не объясняют.

poxvuibr 9 марта 2020 в 22:55 0

> volatile применяет все барьеры памяти перед записью или чтением для обеспечения полной когерентности кэша процессоров.

А ещё он обеспечивает атомарность записи и чтения))

dim2r 10 марта 2020 в 09:45 0

Это да.

Только атомарность операций типа i++ уже не обеспечивается, так как там отдельно чтение и отдельно запись и уже между ними может кто-то вклиниться.

А в классах типа AtomicInteger наблюдаются бесконечные циклы.

```
 public
final int getAndSet(int newValue) {
        for (;;) {   <-====================*
            int current = get();
            if (compareAndSet(current, newValue))
                return current;
        }
    }
```

Я как-то эксперментировал, чтобы посчитать, — сколько итераций этот цикл может проходить, если два потока лезут к одной переменной. В самых худших случаях получалось аж по 200 раз. Это еще с учетом того, что сам CAS(asm cmpxchg) занимает намного больше тактов процессора, чем простые операции работы с памятью.

Поэтому получается, что если слишком тупо использовать многопоточность, то можно нарваться на падение производительности в десятки раз. А чтобы умно использовать, — надо знать барьеры памяти.

poxvuibr 10 марта 2020 в 11:32 +1

> Поэтому получается, что если слишком тупо использовать многопоточность, то можно нарваться на падение производительности в десятки раз.

Да, легко. Но для разумного использования многопоточности в первую очередь надо знать теорию, а потом уже выяснять про барьеры памяти и всё такое. Из низкоуровневых штук, наверное нужно знать только про такие явления, как попадание конкуретно изменяемых данных в одну линию кеша.

> А чтобы умно использовать, — надо знать барьеры памяти.

Вот, кстати, вы привели хороший пример. Знание о борьерах памяти в описаной вами ситуации не очень сильно помогает. Сильно помогает знаение о том, что такое оптимистическая блокировка и пессимистическая блокировка и когда какую использовать.

С помощью CAS делается оптимистическая блокировка, когда потоков, которые изменяют переменную много, она, как вы справедливо заметили работает не очень. Тут нужно применить пессимистическую блокировку и вставить критическую секцию.

Теория рулит, короче, хотя и про барьеры памяти узнать тоже не помешает.

awfun 10 марта 2020 в 00:32 +2

Шипилев поэтому призывает не аппелировать к барьерам на собеседованиях, как к частной детали реализации jmm. Так как практически опираться все равно приходится именно на гарантии, которые обеспечивает jmm, а не нижележащие слои.

tsypanov 24 марта 2020 в 12:49 0

Караул! Комментарии воруют! :)

tsypanov 24 марта 2020 в 12:48 0

> Но для рядовых джавистов протокол MESI не объясняют.

И правильно делают. Это низкоуровневые детали реализации, которые могут поменяться.

dim2r 26 марта 2020 в 22:13 0

Лучше тогда сразу от многопоточности уйти в многопроцессность, как сделал google chrome. Там намного меньше проблем.

somurzakov 9 марта 2020 в 19:39 +4

а теперь представьте, что этот список вопросов и ожидаемый правильный ответ какой-нибудь сеньор-помидор дал HR-у, и он/она будет эти вопросы задавать и фильтровать кандидатов на первоначальном скрининге...

otvorot2008 9 марта 2020 в 20:55 –3

В статье уже есть пометка :)

> Это также не руководство к действию — не надо закидывать бедных кандидатов всеми вопросами из списка.

Andrey_Dolg 10 марта 2020 в 01:33 +8

Вот вам пистолет только вы никогда из него не стреляйте пожалуйста. =)

MIKEk8 10 марта 2020 в 13:39 +2

Значит не надо идти в такую контору. Если профессиональные вопросы задаёт не профессионал, то и профессиональность ответов он не сможет оценить.
Мало того, что надо оценивать правильность ответов, но надо понимать в чём конкретно была ошибка. Возможно собеседник термином ошибся, и описал полный ответ на другой вопрос.

IvanPonomarev 10 марта 2020 в 12:34 +6

Тут в комментариях все набросились на volatile, а в статье требуют апдейта гораздо более базовые вещи.

1. Пожалуйста, пожалуйста, **пожалуйста** замените Random на ThreadLocalRandom. Класс ThreadLocalRandom существует начиная с Java 7. Он в 3-4 раза быстрее, потокобезопасный, более удобный. О его абсолютном преимуществе перед Random написано в книге «Effective Java». Использовать Random сейчас — то же самое, что использовать какой-нибудь Vector вместо ArrayList.
2. Для чего нужен LinkedHashMap? — почему-то не увидел в ответе «для реализации LRU cache», хотя это один из основных сценариев использования, о чём даже сказано в его Javadoc
3. Секция про ввод-вывод крайне куцая и устаревшая. Вообще ничего нет про java.nio, что странно в 2020м. Хотя бы включите вопрос «какие знаете способы прочитать текстовый файл?»

frozen_coder 10 марта 2020 в 13:18 +1

Офтопик немного, но какая же хорошая презентация! Забрал в закладки.

Мне кажется, что тут должна быть ссылка на все.

jetcar 11 марта 2020 в 14:17 0

Помоему вопросы это только на экзамене годяться чтоб проверить учил или не учил последние пару дней, а подбору работника они никак не помогут.
Всегда давал тестовое задание на простой кейс, где базу нужно создать со связями сохранить туда данные из реквеста и потом достать то что нужно.
По итогам всегда видно подумал человек об удобстве тестирования или нет, перформанс, ну и как сами таблицы выглядят и в спеке есть небольшие детали которые показывают на сколько кандидат хорошо умеет спеку читать, а не делает только то что хочет видеть.
Хороший сениор обычно делает всё по максимуму хорошо, ленивый что-то упустит, а джуниоры сделают как умеют особо не парясь.

Maccimo 12 марта 2020 в 04:09 0

> Назвать методы Object

Это что-то навроде проверки, знает ли студент, на экзамен по какому предмету он пришёл?

shadowofmenin 13 марта 2020 в 18:27 0

А в ответе на вопрос про Thread states не должно быть еще TIMED_WAITING?

ogx 13 марта 2020 в 18:28 0

Сразу бросилось в глаза отсутствие Object.clone()

otvorot2008 13 марта 2020 в 18:35 0

Вы часто используете Object.clone()?

ogx 13 марта 2020 в 21:55 +3

Нет, но и finalize() не использую, а его вы добавили.

john_soft 13 марта 2020 в 18:38 0

```
PECS — Producer-Extends-Consumer-Super, метод отдаёт ковариантный тип, принимает контравариантный (прим. автора — пос
леднее интуитивно не очень понятно)
```

Интуитивно можно понять контрвариантность как это обращение иерархии наследования по отношению к действию над иерархией, например, все операции, которые можно выполнить над object можно выполнить и над string

alexhooray 13 марта 2020 в 18:43 0

устройство атомиков и конкурентных коллекций вы оцениваете в 2 звёзды, а способы запуска потока в 3? серьезно?

KopeMorta 13 марта 2020 в 18:43 0

Я действительно не могу понять зачем эти вопросы.
Что вы хотите ими выяснить? Что человек знает? Ну эти вопросы не про это, эти вопросы больше про то, как он готовился к вашему интервью, по этим вопросам можно понять только как он относится к такого рода вещам.
Как говорилось выше — эти вопросы не нужно всегда держать в голове, потому что большинство из них действительно не так часто нужны и они быстро гугляться.

А некоторые вопросы так вообще вызывают недоумение, так и хочется на них ответить: «Да.».

qideil 13 марта 2020 в 18:43 0

> volatile. happens-before. (**) Для double/long типов есть проблема атомарности, она решается через атомики

Как раз для volatile это утверждение неверно. JLS гарантирует атомарность чтения/записи volatile long/double.
docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.7

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Рафаел Саргсян: «В лаборатории значились 23 человека, но все работы делали три–четыре сотрудника»

+48 16k 36 29

Россия — месторождение слонов

+114 16k 44 39

Болею Коронавирусом. Обзор и тестирование, плюсы и минусы и сравнение с конкурентами из той же ценовой категории

+96 28,5k 56 247

Вы просили подсказку? Мы ее вам дадим

+37 12,4k 24 25

О российской open source платформе для создания ИС

Мегапост