

// JavaJub

Шпаргалка

Java · Senior

Быстрое повторение перед собесом

Триггер-вопрос → готовый ответ. Уровень Senior

Что внутри

JVM · многопоточность · Spring · PostgreSQL · Kafka · микросервисы

DevOps · безопасность · алгоритмы · System Design · live-coding

версия 2026.07

// содержание

Содержание

01	Java Core и Collections	3
02	JVM под нагрузкой	8
03	Многопоточность	12
04	Spring	17
05	PostgreSQL	22
06	Kafka	27
07	Архитектура микросервисов	30
08	Обмен между сервисами и кэш	34
09	DevOps и эксплуатация	37
10	Безопасность	40
11	Наблюдаемость и инциденты	44
12	AI в продакшене и зрелость	46
13	Алгоритмы и структуры данных	51
14	System Design кейсы	55
15	Live-coding и разбор кода	59
16	Топ-30 Senior-ловушек	63
	Если осталось 5 минут	65

// как читать цвета

- **БАЗА** — обязан знать на Senior. Не знаешь — вопросов к грейду не останется.
- **ЧАСТОЕ** — то, где отличают Senior от Middle: детали и последствия решений.
- **ЛОВУШКА** — каверзные места, на которых валятся даже сильные кандидаты.

Мало времени → иди по красным (ловушки), потом по оранжевым. Синие — освежить в конце.

► 01 · Java Core и Collections

Модель памяти Java

- Зачем нужна модель памяти, если процессор и так согласует кэши?
 - Протокол согласования кэшей держит в порядке сами кэши, но между ядром и кэшем стоят буфер записи и очередь инвалидаций. Они переставляют операции незаметно для протокола. Модель памяти — это контракт языка поверх этого.
- Что такое happens-before?
 - Это частичный порядок над действиями: чтением, записью, захватом и освобождением блокировки. Если действие A предшествует B по этому порядку, все результаты A видны в B. Порядок транзитивен.
- Откуда берётся связь happens-before?
 - Из порядка команд внутри одного потока и из действий синхронизации: освобождение монитора предшествует следующему захвату, запись volatile предшествует её чтению, старт потока предшествует его первому действию, завершение конструктора предшествует чтению final-полей.
- Какие барьеры памяти ставит компилятор?
 - По рецептам спецификации ставятся четыре вида барьеров. Самый дорогой — StoreLoad, на архитектуре x86 это отдельная инструкция синхронизации. Запись volatile закрывается барьерами StoreStore и StoreLoad, чтение — LoadLoad и LoadStore.
- Почему код работает на x86 и ломается на ARM?
 - У x86 сильная модель памяти: переставляется только пара «запись, затем чтение», поэтому гонки часто проходят незамеченными. У ARM модель слабая, там нужен каждый барьер. Код, годами живший на серверах x86, падает на процессорах Graviton или Apple.
- Гарантирует ли Java последовательную согласованность?
 - Только для программ без гонок: если гонок нет, поведение будет как при последовательном выполнении. Есть гонка — допустимо любое поведение, и рассуждать о порядке строк кода уже нельзя.

■ ЛОВУШКА

- Даёт ли synchronized по локальному объекту видимость между потоками?
 - Нет. Барьер компилятор поставит, но связи happens-before не возникает: этот монитор больше никто не берёт. Синхронизация работает только тогда, когда потоки берут один и тот же монитор.
- Двойная проверка без volatile — что происходит на слабой модели памяти?
 - Читающий поток видит ненулевую ссылку, но объект собран лишь частично: публикация ссылки переставилась раньше записи полей. Лечится словом volatile у поля или тем, что все поля объекта final.
- Почему нельзя рассуждать в терминах строк кода?
 - Модель памяти оперирует действиями, а не строками. Одна строка присваивания превращается в чтение и запись, между которыми компилятор вправе вставить что угодно. Думать надо про барьеры, а не про порядок в источнике.

Безопасная публикация объекта

- Что такое небезопасная публикация?
 - Другой поток видит ненулевую ссылку, но объект недостроен: поля содержат значения по умолчанию, хотя конструктор их выставил. Классический пример — двойная проверка без volatile.
- Способ первый: статический инициализатор?
 - Запись вида static final INSTANCE = new Foo() безопасна: виртуальная машина гарантирует связь между инициализацией класса и любым его использованием. На этом построена ленивая инициализация через вложенный класс-держатель.
- Способ второй: volatile или AtomicReference?
 - Запись ссылки работает как освобождение, чтение — как захват. Все записи, сделанные до публикации ссылки, видны читателю. Это основа корректной двойной проверки.

- Способ третий: заморозка final-полей?
 - После завершения конструктора значения final-полей гарантированно видны любому потоку, если ссылка на объект не утекла из конструктора. Работает без volatile и лежит в основе неизменяемых объектов вроде String.
- Способ четвёртый: общий монитор?
 - Публикация и чтение идут под одной блокировкой. Так устроены Hashtable и обёртки из Collections. Дорого по производительности, зато надёжно.
- Почему утечка ссылки на себя из конструктора смертельна?
 - Если в конструкторе зарегистрировать объект в чужом реестре до присваивания полей, другой поток увидит его со значениями по умолчанию. Заморозка final-полей не сработает: ссылка ушла раньше её завершения.

■ ЛОВУШКА

- Класс с final-полем регистрирует себя в реестре до присваивания — что увидит читатель?
 - Он увидит ноль вместо присвоенного значения. Даже final не спасает, потому что ссылка ушла наружу раньше момента заморозки полей.
- Публикация коллекции через volatile-поле — этого достаточно?
 - Достаточно для самой ссылки, но не для содержимого. Другой поток может обходить коллекцию, пока её меняют. Нужна либо внутренняя синхронизация, либо неизменяемая копия.
- Обычная HashMap, заполненная один раз на старте и дальше только читаемая?
 - Без барьера это небезопасно: читатель может увидеть частично собранную таблицу, оставшуюся от расширения. Либо ConcurrentHashMap, либо публикация через volatile или final.

Записи (records) в проектировании API

- Что даёт запись?
 - Финальный класс с финальными полями и автоматически сгенерированными методами сравнения, хеширования, вывода и доступа. Наследоваться от записи нельзя, добавлять обычные поля тоже — только статические.
- Чем канонический конструктор отличается от компактного?
 - Канонический принимает полный список параметров и может быть переопределён целиком. Компактный пишется без списка параметров и без присваивания полей: компилятор подставит присваивания сам после блока. Компактный идеален для проверки и нормализации значений.
- Где запись применять не стоит?
 - В роли сущности Hibernate — там нужны изменяемое состояние и конструктор без аргументов. В роли объекта передачи данных с иерархией наследования. И как контейнер на два десятка полей: подпись становится нечитаемой.
- Как запись ведёт себя при сериализации?
 - Она восстанавливается через канонический конструктор, а не через рефлексию по полям. Значит, проверки инвариантов выполняются всегда и объект в некорректном состоянии получить нельзя. Библиотекам разбора нужен модуль имён параметров и флаг компилятора -parameters.
- Как сделать копию с изменённым полем?
 - В стандартной библиотеке такого пока нет. Пишут метод руками либо берут генератор строителей. От трёх полей и больше генератор обязателен, иначе руками не уследить.
- Как записи сочетаются с сопоставлением по образцу?
 - Начиная с Java 21 запись можно разобрать прямо в условии на компоненты. Компилятор гарантирует, что компоненты совпадают с каноническим конструктором. Вместе с запечатанным интерфейсом это даёт полноценный алгебраический тип.

■ ЛОВУШКА

- Изменяемый список внутри записи — где ломается?
 - Ссылка на список финальная, но содержимое меняется. Значит, хеш объекта плавает, и он теряется в множестве, куда его положили. Оборачивай список в неизменяемую копию прямо в компактном конструкторе.

Запечатанные типы и сопоставление по образцу

- Зачем запечатанные типы?
 - Они закрывают иерархию: перечисляешь разрешённых наследников списком `permits`. Компилятор знает полный список, поэтому переключатель по типу можно писать без ветки по умолчанию.
- Какие правила у наследников?
 - Наследник обязан быть финальным, либо сам запечатанным со своим списком, либо явно распечатанным через `non-sealed`. И он должен лежать в том же пакете или модуле.
- Что даёт исчерпывающий переключатель?
 - Компилятор проверяет, что покрыты все варианты. Добавил нового наследника в список `permits` — и все переключатели по этому типу перестают компилироваться. Это делает рефакторинг безопасным.
- Как ведут себя ветки с дополнительным условием?
 - Ветка вида `case Order o when сумма больше порога` — это фильтр. Компилятор такие ветки не засчитывает при проверке полноты, поэтому нужна ещё безусловная ветка того же типа или ветка по умолчанию.
- Как обрабатывать `null` в переключателе?
 - С Java 21 есть явная ветка `case null`. Без неё переключатель по `null` бросит исключение ещё до сопоставления. Популярная идиома — объединить ветку `null` с веткой по умолчанию.

■ ЛОВУШКА

- Запечатанный тип плюс сериализация — в чём риск?
 - Все разрешённые наследники становятся частью публичного контракта сериализации. Добавил новый вариант — старый клиент упадёт при чтении. Публиковать запечатанную иерархию через сериализацию не стоит.
- Что ломает гарантию полноты?
 - Наследник, объявленный распечатанным: от него может отнаследоваться кто угодно. Компилятор больше не видит всех вариантов и проверить покрытие не может.

Дженерики: вариантность и стирание

- Почему дженерики инвариантны?
 - Иначе через ссылку на список чисел можно было бы положить дробное значение в реальный список целых, а падение случилось бы только при чтении. У массивов такая подстановка разрешена, и там она даёт ошибку уже во время работы.
- Как применять правило PECS в проектировании API?
 - Параметр метода объявляй через `extends` — так клиенту гибче передавать подтипы. Возвращаемый тип обычно оставляют без подстановочных знаков вообще. Приёмник значений объявляют через `super`.
- Что такое несколько ограничений и самоограниченный тип?
 - Ограничений может быть несколько: один класс и сколько угодно интерфейсов, причём класс идёт первым. Самоограниченный тип вида `T расширяет Comparable от T` гарантирует, что объект сравнивается сам с собой. На этом построен базовый класс перечислений.
- Когда нужна явная подсказка типа?
 - Когда вывод типов не справляется — обычно в цепочке потоков или при передаче в обобщённый метод. Начиная с Java 8 вывод сильно улучшили, поэтому подсказка требуется редко.
- Чем массивы отличаются от дженериков во время работы?
 - Массив знает свой тип во время работы, дженерик — нет, его параметр стирается. Поэтому создать массив из параметра типа нельзя: настоящего класса для него в этот момент не существует.

■ ЛОВУШКА

- Обобщённый метод с переменным числом аргументов — в чём подвох?
 - Внутри метод получает массив дженерика, а такой массив в системе типов существовать не должен. Компилятор выдаёт предупреждение. Аннотацию о безопасности ставят, только если метод в этот массив ничего не записывает.

- Ковариантность возвращаемого типа работает с дженериками?
→ При переопределении метода сузить возвращаемый класс можно, и это удобно для текучих интерфейсов. Но с дженериками не выйдет: список чисел на список целых компилятор заменить не даст.

Escape analysis, скалярная замена и TLAB

- Что делает анализ убегания?
→ Компилятор проверяет, покидает ли ссылка на объект границы метода: уходит ли она в поле, в другой поток или наружу через возврат. Если нет, объект можно вообще не создавать в куче.
- Что такое скалярная замена?
→ Компилятор разбирает объект на отдельные поля и держит их в регистрах или на стеке кадра. Выделения памяти в куче нет, сборщику мусора работать не с чем. Типичный выигрыш — Optional и итераторы потоков.
- Правда ли, что объекты размещаются на стеке?
→ Нет, это распространённая неточность. HotSpot объекты на стек не кладёт — он разбирает их на скалярные значения. Скалярная замена и есть единственный работающий механизм.
- Что такое буфер выделения памяти для потока?
→ У каждого потока свой кусок молодого поколения, поэтому выделение памяти сводится к сдвигу указателя без атомарных операций. Если кусок кончился, идёт медленный путь через общий аллокатор.
- Когда анализ убегания не срабатывает?
→ Когда ссылка уходит в статическое поле, в поле объекта, в другой поток, в исключение или наружу через возврат. А также когда виртуальный вызов не удалось встроить, потому что реализаций у метода много.

■ ЛОВУШКА

- Атомарный счётчик в горячем цикле оптимизируется?
→ Нет: объект убегает через семантику volatile, и анализ его не устранил. На состязательном пути обычное поле через VarHandle работает в 5-10 раз быстрее.
- Когда именно работает анализ убегания?
→ Только после того, как метод скомпилирован. Первые тысячи вызовов идут в интерпретаторе, и там выделение памяти вполне реально. В коротких утилитах и тестах оптимизация может не сработать совсем.

Многопоточные коллекции

- Как устроена ConcurrentHashMap?
→ Запись блокирует только один bucket, чтение идёт без блокировок за счёт правильной видимости узлов. Методы computeIfAbsent и merge атомарны. Обход отражает слабо согласованный снимок и не бросает исключений.
- Когда брать CopyOnWriteArrayList?
→ Только когда читают часто, а пишут редко: подписчики, слушатели событий. Каждая запись копирует весь внутренний массив, поэтому тысячи записей в секунду его убьют. Зато чтение полностью свободно от блокировок.
- Зачем нужна ConcurrentSkipListMap?
→ Это единственная ConcurrentHashMap с навигацией: первый ключ, диапазон, ближайший снизу. Все операции логарифмические и построены на атомарных операциях без общей блокировки. На точечном поиске она в 3-5 раз медленнее обычной ConcurrentHashMap.
- Чем она лучше синхронизированного TreeMap?
→ обёртка synchronizedSortedMap над TreeMap держит одну блокировку на всю Map, и потоки выстраиваются в очередь. Здесь же блокировки распределены по узлам, и потоки почти не мешают друг другу.
- Что значит слабо согласованный обход?

→ Итератор не бросает исключение при изменении коллекции, показывает состояние на момент создания и может увидеть или не увидеть новые записи. Обходить такую коллекцию в продакшене безопасно.

■ ЛОВУШКА

- Насколько точен размер `ConcurrentHashMap`?
 - Он приблизительный: считается суммой распределённых счётчиков. Опирается на него в горячем цикле нельзя. Для точного значения есть отдельный метод, возвращающий длинное число.
- Почему в `ConcurrentHashMap` запрещён `null`?
 - Это осознанное решение, а не ограничение реализации. Иначе результат «получили `null`» нельзя было бы отличить от «ключа нет» без потери гарантий чтения без блокировок.

Мягкие, слабые и фантомные ссылки

- Как ведёт себя мягкая ссылка?
 - Сборщик держит объект, пока памяти хватает, и собирает его перед нехваткой по давности использования. Под нагрузкой это плохо: освобождение идёт пачкой и даёт всплеск задержек.
- Как ведёт себя слабая ссылка?
 - Сборщик забирает объект при первой возможности, если сильных ссылок на него не осталось. Применяется как ключ в `WeakHashMap` и в кэшах канонизации значений.
- Зачем фантомная ссылка?
 - Получить объект через неё нельзя, метод доступа всегда возвращает `null`. Польза одна: после сборки ссылка попадает в очередь, и это сигнал освободить внешние ресурсы. На ней построен современный механизм очистки, пришедший на смену устаревшему `finalize`: тот объявлен к удалению и отключается флагом.
- В чём подвох `WeakHashMap`?
 - Ключ слабый, а значение сильное. Если значение ссылается на ключ, объект не соберётся никогда и получится утечка. Нужен либо механизм очистки, либо слабая ссылка и на значение.
- Как локальная переменная потока течёт в пуле?
 - Внутренняя Мар держит слабую ссылку на ключ и сильную на значение. Пока поток живёт в пуле, значение не освобождается. Если оно ссылается на классы приложения, при переразвёртывании утекает целый загрузчик классов.

■ ЛОВУШКА

- Строковый литерал как ключ в `WeakHashMap`?
 - Литерал лежит в общем пуле строк и не собирается никогда, поэтому ключ вечный и запись не удалится. Нужен либо явно созданный объект строки, либо другой тип ключа.
- Почему мягкие ссылки перестают работать на большой куче?
 - Сборщик забирает их только под давлением памяти, а на куче в сотню гигабайт давление наступает поздно. Кэш растёт и не чистится. Надёжнее взять `Caffeine` с вытеснением по размеру.

Память вне кучи

- Зачем выносить данные за пределы кучи?
 - Сборщик мусора такую память не сканирует и не останавливает приложение. Это позволяет держать десятки гигабайт с задержками меньше миллисекунды. Цена — ручное управление жизненным циклом и отсутствие проверки типов.
- Как работает прямой буфер?
 - Память выделяется вне кучи Java, а освобождается через механизм очистки, привязанный к самому объекту буфера. Пока объект достижим, внешняя память занята. Собрали объект — освободилась и она.
- Каков предел памяти вне кучи?
 - По умолчанию он равен максимальному размеру кучи — это историческая ловушка. Превышение даёт отдельный вид нехватки памяти, не связанный с кучей. Следить за ним нужно через отдельную метрику пулов буферов.

- Что даёт отображение файла в память?
 - Файл попадает в адресное пространство процесса, и чтение с записью становятся обычным обращением к памяти, а страницы подгружает ядро. Это даёт передачу данных без лишнего копирования.
- **ЛОВУШКА**
 - Как освободить отображение файла до сборки мусора?
 - Метода закрытия у него нет. Освободить раньше можно только через внутренние средства платформы. На больших отображениях файл при этом нельзя удалить, потому что дескриптор ещё держится.
 - Прямой буфер в локальной переменной потока из пула?
 - Это утечка внешней памяти: пока поток живёт в пуле, очистка не сработает. Нужен явный пул буферов с вытеснением.
 - Предел внешней памяти по умолчанию — чем опасен?
 - При куче в 32 гигабайта приложение может занять ещё столько же вне её, и контейнер убьют по превышению памяти. Значение всегда выставляют явно и следят за ним отдельной метрикой.

Kotlin для Java-Senior

- Что такое платформенные типы?
 - Тип, пришедший из Java без пометок о допустимости null, компилятор Kotlin не проверяет. Ошибка обращения к null вылетает где угодно, и стек указывает мимо. Лечится пометками в Java-коде и строгим режимом компилятора.
- Чем корутины отличаются от виртуальных потоков?
 - Корутины кооперативные: функция с пометкой suspend превращается в пошаговый автомат с продолжениями. Виртуальные потоки планирует среда выполнения, и обычный блокирующий код становится дешёвым без переписывания.
- Заменяют ли они друг друга?
 - Нет. Корутины переносимы за пределы JVM и дают богатый набор средств для потоков данных. Виртуальные потоки не требуют менять код, но ломаются на synchronized. Выбор зависит от того, что важнее.
- Что такое встроенный класс-значение?
 - Обёртка с нулевой стоимостью: типизированный идентификатор компилируется в примитив без упаковки. Ограничение в том, что при использовании как параметра типа или как значения, допускающего null, упаковка всё же появится.
- Как вызвать suspend-функцию из Java?
 - Напрямую нельзя: она компилируется в метод с дополнительным параметром продолжения, и читается это плохо. Практичнее обернуть её в CompletableFuture средствами библиотеки корутин.

■ ЛОВУШКА

- Kotlin-класс данных в сообщениях Kafka — где ломается?
 - Сравнение и хеш считаются по всем полям, включая отметку времени получения. Одно и то же сообщение даёт разный хеш и уходит в разные разделы. Сравнение нужно объявить по бизнес-ключу явно.
- Диспетчер для операций ввода-вывода — в чём предел?
 - По умолчанию это пул на 64 потока. Шестидесят пятый блокирующий вызов встанет в очередь, и обработка остановится. На виртуальных потоках такой предел не нужен.

► 02 · JVM под нагрузкой

Выбор сборщика мусора

- Кому подходит Parallel?

→ Пакетной обработке и выгрузкам данных, где важна общая пропускная способность, а паузы в секунду терпимы. Он был выбором по умолчанию до Java 9: простой, экономный по процессору, без параллельных фаз.

- Кому подходит G1?

→ Это выбор по умолчанию с Java 9 и универсальный вариант для кучи примерно от 4 до 32 гигабайт. Куча делится на регионы, уплотнение идёт частями, а целевую паузу задают отдельным флагом. Паузы предсказуемы, но не меньше миллисекунды.

- Когда брать ZGC?

→ Когда договор об уровне сервиса требует задержек меньше 10 миллисекунд, а куча большая — от нескольких гигабайт до терабайтов. Уплотнение идёт параллельно с приложением. Плата — примерно 5-15 процентов дополнительной нагрузки на процессор и больше памяти.

- Когда Shenandoah?

→ Он близок к ZGC по идее и тоже уплотняет кучу параллельно, но лучше показывает себя на куче до 32 гигабайт и на процессорах ARM. Паузы держатся в пределах одной-двух миллисекунд.

- Как выглядит размен между паузой и ресурсами?

→ ZGC и Shenandoah снижают паузы, но увеличивают расход памяти примерно на 10-20 процентов и нагрузку на процессор. Parallel даёт максимальную пропускную способность при самых длинных паузах. G1 находится посередине.

- Что даёт поколенческий ZGC?

→ Он появился в Java 21 и снимает главный недостаток обычного ZGC: при высокой скорости выделения памяти тот срывался в полную сборку с паузой. С разделением на поколения это ушло.

■ ЛОВУШКА

- «Возьмём ZGC везде, он же быстрее» — где ошибка?

→ На маленьком сервисе с кучей в пару гигабайт и низкой скоростью выделения памяти ZGC даст только накладные расходы на процессор без выигрыша по задержкам. Там с запасом хватит G1.

- Целевая пауза в G1 — это гарантия?

→ Нет, это мягкая цель. Если объём живых данных приблизился к размеру кучи, G1 всё равно уйдёт в полную сборку в один поток, а это уже катастрофа по времени.

Компиляция во время работы

- Что такое многоуровневая компиляция?

→ Нулевой уровень — интерпретатор. Уровни с первого по третий — быстрый компилятор со слабой оптимизацией, который заодно собирает профиль. Четвёртый уровень — медленный, но агрессивный: встраивание вызовов, анализ убегания, развёртывание циклов.

- Что запускает компиляцию метода?

→ Счётчики вызовов метода и повторов цикла. Когда счётчик переваливает порог, метод уходит на компиляцию. На средних уровнях попутно копится профиль типов и ветвлений.

- Что такое замена кода прямо на стеке?

→ Метод уже выполняется в интерпретаторе, а внутри крутится долгий цикл. Ждать выхода из метода незачем, поэтому компилятор подменяет кадр прямо на стеке. В журнале компиляции такие случаи помечаются отдельно.

- Что такое деоптимизация?

→ Компилятор сделал предположение — например, что у вызова всего одна реализация. Загрузился новый подкласс, предположение нарушилось, и код возвращается в интерпретатор с последующей перекомпиляцией.

- Что такое встроенные кэши вызовов?

→ Если у точки вызова одна реализация, идёт прямой переход. Если две, проверяются оба варианта. Если больше, компилятор сдаётся и идёт через таблицу виртуальных методов. Загрузка нового класса ломает быстрый вариант.

- Что делать с прогревом?

→ Первые запросы после старта идут через интерпретатор и быстрый компилятор, поэтому задержка сначала высокая. Помогают разделяемые данные классов при старте либо заранее скомпилированный образ.

■ ЛОВУШКА

- Замер производительности через засечку времени в цикле — что измеришь?
 - Интерпретатор, а не оптимизированный код. Нужен полноценный набор для микрозамеров: он делает прогрев, запускает отдельные процессы и защищает от выбрасывания мёртвого кода.
- Почему огромные методы работают медленно?
 - Метод длиннее порога встраивания компилятор не встраивает в вызывающий код. Гигантские переключатели и лямбды с большим телом молча убивают горячий путь, и в профиле это выглядит загадочно.

Заранее скомпилированный образ (GraalVM)

- В чём суть образа?
 - Приложение целиком компилируется в исполняемый файл заранее. Старт занимает десятки миллисекунд, потребление памяти в разы ниже, но пиковая пропускная способность немного проседает: нет профиля времени выполнения.
- Что такое предположение о закрытом мире?
 - На этапе сборки должны быть известны все классы, методы и ресурсы. Всё, что определяется динамически, придётся объявить отдельными файлами конфигурации.
- Почему ломается рефлексия?
 - Библиотеки разбора JSON, Hibernate и любые контейнеры внедрения зависимостей без нативной поддержки падают уже во время работы с ошибкой поиска класса. Современные фреймворки собирают подсказки за вас, иначе их пишут вручную.
- Зачем агент трассировки?
 - Он запускается на обычной виртуальной машине, наблюдает за прогоном и сам пишет конфигурацию. Прогонять под ним нужно все сценарии, иначе непокрытые ветки найдёт уже продакшен.
- Чем опасна инициализация на этапе сборки?
 - Статические инициализаторы по умолчанию выполняются при сборке. Если так инициализировать генератор случайных чисел, текущее время или соединение, значения зашьются в исполняемый файл навсегда.

■ ЛОВУШКА

- Собрали образ по конфигурации из разработки — что случится в проде?
 - Сработает редкая ветка повторной обработки с полем, которого не было в прогоне, и разбор JSON упадёт на поиске класса. Агент трассировки нужно гонять на полноценной нагрузке, а не только на модульных тестах.
- Значит ли образ, что «Java стала быстрее»?
 - Нет. На устоявшейся нагрузке обычная виртуальная машина обычно даёт большую пропускную способность. Образ выигрывает в скорости старта и потреблении памяти, а не в пиковой производительности.

Профилирование в продакшене

- Сколько стоит встроенный профилировщик?
 - Около одного процента при непрерывной записи. Он пишет события в ring buffer, поэтому снимок за последние минуты можно забрать в любой момент без остановки приложения.
- Зачем тогда внешний профилировщик?
 - Встроенный теряет замеры в нативном коде и семплирует только в безопасных точках, из-за чего смещает картину горячих циклов. Внешний работает через счётчики производительности ядра и даёт честный профиль.
- Как читать пламенный график по процессору?

// JavaJub

Полная версия — в JavaJub Pro.

Шпаргалка Java Senior · Java · Senior

Что внутри полной версии:

- все триггер-вопросы по всем темам
- готовые короткие ответы под грейд
- цветокод: база / частое / ловушка

@Java_Jub_Pro

t.me/Java_Jub_Pro · полная версия по подписке