

// JavaJub

Шпаргалка

Java · Middle

Быстрое повторение за 2 часа до собеседа

Триггер-вопрос → готовый ответ. Уровень Middle / Middle+

Темы внутри

Core · Collections · JVM · Concurrency · Spring · SQL · System Design · Git · CI/CD · AI/LLM ·
Кэш · DevOps · Security · Tools

версия 2026.07 · обновляется

// содержание

Содержание

01	Java Core	3
02	Collections	7
03	JVM и память	8
04	Многопоточность	9
05	Spring	12
06	Базы данных	15
07	System Design и микросервисы	17
08	Тесты & DevOps	20
09	Git	22
10	CI/CD	24
11	AI / LLM	26
12	Кэширование	28
13	DevOps & Развёртывание	30
14	Безопасность & API Design	32
15	Практические инструменты	34
16	Топ-30 ловушек собеседа	37
	Если осталось 5 минут	39

// как читать цвета

■ **БАЗА** — спросят почти точно. Если не знаешь — провалишься сразу.

■ **ЧАСТОЕ** — частые вопросы средней сложности, где многие плывут.

■ **ЛОВУШКА** — каверзные места, на которых валятся даже знающие.

Мало времени → иди по красным (ловушки) и синим (база), остальное пропускай.

► 01 · Java Core

equals и hashCode

- Какие 5 свойств у equals?
 - Рефлексивность — объект равен сам себе. Симметричность — если a равно b, то и b равно a. Транзитивность — если a равно b и b равно c, то a равно c. Стабильность — повторные вызовы дают тот же ответ. И сравнение с null всегда возвращает false.
- Как связаны equals и hashCode?
 - Если два объекта равны по equals, их hashCode обязаны совпадать. Обратное не требуется: у разных объектов hashCode может совпасть — это называется коллизией и считается нормой.
- Переопределил equals, забыл hashCode — что будет?
 - В коллекциях HashMap и HashSet объект попадёт не в ту корзину, поиск его не найдёт, появятся дубликаты. Переопределять нужно всегда оба метода вместе.
- Что включать в equals и hashCode?
 - Только неизменяемые поля, которые задают смысл объекта. Аннотация Lombok @Data берёт все поля подряд, включая id — лучше явно перечислить нужные через @EqualsAndHashCode(of={...}).

■ ЛОВУШКА

- Сущность JPA со сравнением по id — где ломается?
 - До сохранения id ещё null, поэтому два новых объекта считаются равными, а после сохранения — уже нет. Из-за этого equals ведёт себя нестабильно. Сравнивай по бизнес-ключу, а не по id из базы.

== и equals

- Что сравнивает == у примитивов и у объектов?
 - У примитивов знак == сравнивает сами значения. У объектов он сравнивает ссылки, то есть адреса в памяти, а не содержимое. Поэтому new String("a") == new String("a") даёт false — это два разных объекта.
- Как работает equals по умолчанию?
 - В классе Object метод equals сравнивает ссылки, как и знак ==. Для своих классов его переопределяют. Чтобы не поймать ошибку на null, удобно сравнивать через Objects.equals(a, b) — он корректно обрабатывает null.

■ ЛОВУШКА

- Integer a=100, b=100: a==b? А если 200?
 - Для 100 будет true: Java заранее кэширует объекты Integer от минус 128 до 127, поэтому это одна и та же ссылка. Для 200 будет false — создаются новые объекты. Так же кэшируются Boolean, Byte, Short, Character и Long; у Float и Double кэша нет.
- Integer x = null; int y = x; — что будет?
 - Будет ошибка NullPointerException. При распаковке объекта в примитив Java обращается к значению внутри, а его нет. Такая ошибка легко прячется при работе с коллекциями.

String, StringBuilder и пул строк

- Почему String неизменяемый?
 - Неизменяемость даёт сразу несколько плюсов: можно кэшировать hashCode, строки безопасно использовать как ключи Map и как имена файлов и классов, они потокобезопасны и хранятся в общем пуле. Один и тот же литерал — это одна ссылка на всю программу.
- Строка через new String попадает в пул?
 - Нет, new String создаёт отдельный объект в памяти. В пуле лежит только строковый литерал. Метод intern() вернёт ссылку из пула, а если такой строки там ещё нет — сначала добавит её.
- StringBuilder или StringBuffer?
 - Методы у них одинаковые, но StringBuffer синхронизирован и потому медленнее. В одном потоке всегда берут StringBuilder. Склеивать строки в цикле нужно тоже через него — иначе на каждом шаге создаётся новый объект строки.

- Пароль хранить в String или в массиве char?
 - Лучше в массиве char: его можно очистить сразу после использования. Строка String остаётся в памяти до сборки мусора и видна в дампе памяти.

■ ЛОВУШКА

- Меняет ли s строку вызов s.concat("!")?
 - Нет. Строка неизменяема, а результат метода никуда не сохранён, поэтому переменная s остаётся прежней.

Неизменяемые объекты

- Как сделать класс неизменяемым?
 - Сделать сам класс final, все поля — private final, убрать сеттеры и задавать значения только в конструкторе. А для полей-ссылок (коллекции, Date) в конструкторе и в геттере делать копию, чтобы их нельзя было изменить снаружи.
- Зачем нужны неизменяемые объекты?
 - Такой объект потокобезопасен без блокировок, годится как надёжный ключ в HashMap, позволяет кэшировать hashCode, и его состояние нельзя испортить снаружи.
- Для поля-списка достаточно unmodifiableList?
 - Нет, это лишь обёртка над списком. Метод add кинет исключение, но тот, у кого осталась ссылка на исходный список, всё равно его изменит. Делай копию через List.copyOf(...) или в конструкторе.

■ ЛОВУШКА

- record Person(String name, List roles) — он неизменяемый?
 - Нет: вызов person.roles().add("admin") изменит список снаружи. Нужен компактный конструктор, который скопирует список через List.copyOf(...).
- Делает ли слово final у поля-списка объект неизменным?
 - Нет, final фиксирует только ссылку. В такой список всё равно можно добавлять элементы — меняется содержимое, а ссылка остаётся прежней.

Дженерики и PECS

- Зачем дженерики и что такое стирание типов?
 - Дженерики дают проверку типов при компиляции, без ручного приведения к Object. Но во время работы программы параметр типа стирается: List<String> и List<Integer> превращаются в обычный List. Поэтому нельзя написать new T[], получить T.class или проверить instanceof List<String>.
- Что означает правило PECS?
 - Расшифровка: Producer Extends, Consumer Super. Если из коллекции только читаем — берём List<? extends T>. Если в неё только пишем — берём List<? super T>.
- Почему в List<? extends Number> нельзя добавить число?
 - Компилятор не знает, какой именно подтип лежит внутри — Integer, Double или другой. Поэтому он запрещает добавлять что угодно, кроме null.

■ ЛОВУШКА

- List<Integer> — это List<Number>?
 - Нет, дженерики инвариантны: один тип не считается родственником другому. Иначе через ссылку типа List<Number> можно было бы положить Double в список, который на деле хранит только Integer.

Optional

- Зачем Optional и где его НЕ применять?
 - Optional в типе возвращаемого значения честно говорит: результата может не быть. Не стоит применять его для полей класса, для параметров методов и для ключей Map — там это считается плохой практикой.
- Чем of отличается от ofNullable?
 - Метод of(null) сразу кинет ошибку — его берут, когда точно уверены, что значение не null. Метод ofNullable(null) спокойно вернёт пустой Optional.

- Чем `orElse` отличается от `orElseGet`?
 - Метод `orElse(new Foo())` создаёт объект `Foo` всегда, даже когда значение уже есть. Метод `orElseGet(Foo::new)` вызовет создание только если `Optional` пустой. Это важно, когда значение по умолчанию дорогое.
- Чем плохо `isPresent()` вместе с `get()`?
 - Это по сути та же ручная проверка на `null`, только многословнее. Лучше использовать методы `map`, `orElse`, `orElseThrow` или `ifPresent`.

■ ЛОВУШКА

- В чём баг: `findById(id).orElse(repo.save(...))`?
 - Метод `save` выполнится всегда, даже когда запись уже нашлась в базе. Аргумент `orElse` вычисляется в любом случае — здесь нужен `orElseGet`, чтобы сохранение шло только при пустом результате.

Stream API

- Чем промежуточные операции отличаются от конечных?
 - Промежуточные операции (`filter`, `map`, `sorted`) ленивые: они лишь описывают шаги и возвращают новый `Stream`. Конечные (`collect`, `count`, `reduce`) запускают всю цепочку. Пока нет конечной операции, не выполняется ничего.
- Можно ли дважды запустить один и тот же стрим?
 - Нет — будет ошибка «`stream has already been operated upon`». Поток одноразовый, для нового прохода его создают заново.
- Чем опасен `parallelStream`?
 - Он работает на одном общем пуле потоков (его размер — число ядер минус один). Одна тяжёлая или ждущая задача затормозит все параллельные стримы в программе. Для операций ввода-вывода его использовать нельзя.
- Разница `collect(toList())` и `Stream.toList()`?
 - Метод `Stream.toList()` (появился в Java 16) возвращает неизменяемый список. А `Collectors.toList()` обычно даёт изменяемый `ArrayList`. Разница всплывает, когда попробуешь добавить в него элемент.

■ ЛОВУШКА

- `peek` без конечной операции печатает логи?
 - Нет — без конечной операции стрим вообще не запускается. Частая ошибка при отладке: поставили `peek` для логов, а конечную операцию (например, `collect`) забыли.
- `peek` с `anyMatch` сработает на каждый элемент?
 - Нет: `anyMatch` останавливает поток на первом же подходящем элементе, поэтому `peek` отработает только до него. А если размер известен и побочных действий нет, оптимизатор (начиная с Java 9) может убрать `peek` совсем. Из-за этого логи то появляются, то нет.
- `Stream.iterate(...).sorted().limit(10)` — что не так?
 - Программа зависнет и упадёт по памяти. Сортировка должна сначала собрать все элементы, а `iterate` создаёт их бесконечно. Правильный порядок — сначала `limit(10)`, потом `sorted()`.

Wildcards на практике

- Ещё раз, коротко: правило PECS?
 - `Producer Extends, Consumer Super`. Из `List<? extends T>` только читаем, в `List<? super T>` только пишем значения типа `T`. И помни: `List<Integer>` не является `List<Number>`.
- Как получить `Class<T>` внутри обобщённого класса?
 - Напрямую нельзя: запись `T.class` не работает, потому что тип стирается при компиляции. Варианты — передать `Class<T>` в конструктор, а для сложных типов вроде `List<Foo>` взять `TypeReference` из `Jackson` или `TypeToken` из `Guava` и `Gson`.
- Зачем `super` в `Comparable<? super T>`?
 - Чтобы сортировка работала и когда класс наследует метод сравнения от родителя — например, `Dog` наследует `Animal`, а `Comparable` объявлен у `Animal`. Без `super` каждый тип был бы обязан реализовывать сравнение заново под себя.

- Чем опасен сырой тип List и зачем @SafeVarargs?
 - Сырой тип (без указания дженерика) отключает проверки типов в соседнем коде, из-за чего появляются скрытые ошибки приведения. Аннотацию @SafeVarargs ставят на final, static или private метод с обобщённым varargs, когда в переданный массив реально ничего не записывают.

■ ЛОВУШКА

- Почему List<Integer> нельзя присвоить List<Number>?
 - Иначе через ссылку типа List<Number> можно было бы положить Double в список, который хранит только Integer, и при чтении выпала бы ошибка приведения типа. И учти: List<Object> — это не то же, что List<?>; во второй можно добавить только null.
- Можно ли поймать исключение MyException<String>?
 - Нет, это запрещено: параметр типа стирается, и виртуальная машина не отличит MyException<String> от MyException<Integer>. По той же причине нельзя создать массив new T[10].

Kotlin для Java-разработчика

- Что даёт data class?
 - Он сам создаёт методы equals, hashCode, toString, copy и доступ к полям по порядку — по полям главного конструктора. Это как record в Java, но объект может быть изменяемым (через var) и по умолчанию не финальный.
- Защита от null, а также val и var?
 - Тип со знаком вопроса (String?) допускает null, а без него (String) — нет; компилятор не даст обратиться к возможному null без специальных операторов. val — это неизменяемая ссылка (сам объект менять можно), var — обычная переменная. По умолчанию берут val.
- Корутины и виртуальные потоки — в чём разница?
 - Корутины кооперативные: функции с пометкой suspend компилируются в пошаговый автомат и работают на любой версии JVM. Виртуальные потоки планирует сама среда выполнения, код при этом обычный, но нужна Java 21 или новее.

■ ЛОВУШКА

- Что ломается у Kotlin со Spring и Hibernate?
 - В Kotlin классы по умолчанию final, а Spring и Hibernate создают их наследников-обёртки — для настроек и ленивой загрузки. Наследовать final-класс нельзя, поэтому нужны плагины kotlin-spring, all-open и no-arg, которые их открывают.
- Почему data class — плохая сущность JPA?
 - Потому что equals и hashCode считаются по всем полям, включая id, который сначала равен null — те же проблемы, что и у Lombok @Data. Для сущностей берут обычный class со сравнением по бизнес-ключу.

Исключения

- Иерархия и проверяемые против непроверяемых?
 - В основе класс Throwable, от него идут Error (его не ловят) и Exception. RuntimeException и его потомки непроверяемые (например, NullPointerException). Остальные проверяемые (ошибки ввода-вывода, SQL): их обязательно либо обработать, либо объявить в throws.
- Как работает try-with-resources?
 - Если ресурс реализует интерфейс AutoCloseable, его метод close() вызовется сам даже при ошибке, причём ресурсы закрываются в обратном порядке. Если и при закрытии возникнет ошибка, она станет подавленной, а основная сохранится.
- Блок finally выполняется всегда?
 - Почти всегда. Он не выполнится, если вызвать System.exit(), если поток убьют, если в try бесконечный цикл или взаимная блокировка, либо если упадёт сама JVM.

■ ЛОВУШКА

- return в finally — что станет с return из try?
 - Он его перекроет: если try возвращает 1, а finally возвращает 2, наружу уйдёт 2. Хуже того — исключение, брошенное в try, при этом молча потеряется.

- @Transactional откатит транзакцию на проверяемом исключении?
 - Нет — по умолчанию откат идёт только на RuntimeException и Error. На проверяемом исключении транзакция закоммитится. Чтобы она откатывалась и на нём, укажи rollbackFor = Exception.class.

► 02 • Collections

HashMap

- Что у HashMap внутри?
 - Массив корзин (по умолчанию 16 штук). При совпадении хешей в корзине сначала связный список, а когда элементов становится 8 и больше и размер массива не меньше 64 — список превращается в красно-чёрное дерево. При заполнении на 75 процентов массив увеличивается вдвое.
- Как выбирается корзина?
 - Берётся хеш ключа, его старшие биты подмешиваются к младшим (чтобы плохие хеши распределялись ровнее), а затем по остатку от деления на размер массива определяется номер корзины.
- Можно ли ключ null?
 - Да, один — он всегда ложится в самую первую корзину. А вот у TreeMap null-ключ запрещён: при сравнении будет ошибка. Если у всех ключей хеши совпадут, HashMap продолжит работать, но поиск внутри корзины замедлится до перебора.
- HashMap потокобезопасен?
 - Нет — для нескольких потоков берут ConcurrentHashMap, а старый Hashtable уже не используют. В Java 7 при одновременном изменении из разных потоков увеличение массива могло уйти в вечный цикл и нагрузить процессор на 100 процентов; в Java 8 это поправили.

■ ЛОВУШКА

- Что будет, если изменить ключ после put?
 - Элемент фактически теряется: хеш пересчитается, но объект остался в старой корзине, и поиск его не найдёт. Поэтому ключ обязан быть неизменяемым.

ArrayList и LinkedList

- Что у них внутри?
 - ArrayList — это массив, который при нехватке места вырастает примерно в полтора раза. LinkedList — двусвязный список узлов, разбросанных по памяти; он же используется как очередь с двух концов.
- Какая сложность доступа и вставки?
 - Доступ по индексу: у ArrayList мгновенный, у LinkedList — перебор по ссылкам. Вставка в середину у обоих требует прохода до нужного места. Если размер известен заранее, задавай его в конструкторе new ArrayList<>(размер).

■ ЛОВУШКА

- LinkedList быстрее на вставке в середину?
 - Нет. Поиск нужной позиции требует прохода по всему списку и съедает всё преимущество, а массив ещё и лежит в памяти подряд, что ускоряет обход. На практике ArrayList быстрее уже примерно с 10 элементов.

ConcurrentHashMap

- Зачем он и как блокирует (Java 8 и новее)?
 - Это потокобезопасная Map, которая не блокирует всю структуру целиком. При записи блокируется только одна корзина, а чтение идёт вообще без блокировок за счёт правильной видимости данных.
- Чем он лучше synchronizedMap?
 - В synchronizedMap любое чтение и запись проходят через один общий замок и выстраиваются в очередь. В ConcurrentHashMap операции по разным корзинам идут параллельно. При этом ни ключ, ни значение не могут быть null.

■ ЛОВУШКА

- Проверка «если нет ключа — положить» атомарна?
 - Нет: между проверкой и добавлением может вклиниться другой поток, даже в ConcurrentHashMap. Используй putIfAbsent, computeIfAbsent или merge — они выполняются как единая атомарная операция.

TreeMap и отсортированные Map

- Что внутри и какой порядок?
 - Внутри красно-чёрное дерево, все операции работают за логарифмическое время — это плата за то, что ключи всегда отсортированы. Порядок задаёт либо сам ключ (если реализует Comparable), либо переданный в конструктор компаратор.
- Зачем нужен NavigableMap?
 - Он даёт методы «ближайший снизу», «ближайший сверху» и выборку диапазона ключей — быстро, за логарифмическое время. А LinkedHashMap не сортирует, он лишь запоминает порядок, в котором элементы добавляли.

■ ЛОВУШКА

- Компаратор не согласован с equals — что будет?
 - Элементы начнут «пропадать»: TreeMap и TreeSet считают ключи одинаковыми по результату сравнения, а не по equals. Классический баг, когда сортируешь только по одному полю.

HashSet и обход коллекции

- Что у HashSet внутри?
 - Это обёртка над HashMap: твой объект становится ключом, а значением ставится специальная заглушка. Вся логика и требования (нужны equals и hashCode) берутся из HashMap.
- Что такое ошибка изменения при обходе и как удалять в цикле?
 - Если во время обхода добавить или удалить элемент не через сам итератор, следующий шаг кинет ошибку ConcurrentModificationException. Удалять можно только методом самого итератора remove() или через removeIf().
- Что значит «падает сразу» и «работает по копии»?
 - ArrayList, HashMap и HashSet «падают сразу»: замечают изменение при обходе и бросают ошибку. А CopyOnWriteArrayList и ConcurrentHashMap «работают по копии»: обход идёт по снимку данных, ошибку не бросают, но и свежих изменений могут не увидеть.

■ ЛОВУШКА

- Замена значения по существующему ключу в цикле — будет ошибка?
 - Нет — замена значения не считается изменением структуры. А вот добавление нового ключа или удаление — уже да. При этом такая проверка не гарантирована: в многопоточной работе ошибка может и не сработать.

► 03 · JVM и память

Структура памяти

- Из каких частей состоит память JVM?
 - Куча — общая для всех потоков. Стек — свой у каждого потока. Metaspace, регистр адреса команды, стек нативных методов и прямая (внеобычная) память — тоже отдельные области.
- Как устроена куча?
 - Куча делится на молодое и старое поколение. Новые объекты создаются в молодом; те, что пережили несколько сборок мусора (по умолчанию около 15), переезжают в старое.
- Что лежит в стеке и где хранится static?
 - В стеке лежат кадры вызовов методов: локальные переменные, ссылки, адрес возврата — стек чистится сам, сборщик мусора ему не нужен. Описание классов хранится в Metaspace (с Java 8 вместо старого PermGen), а сами статические поля и их объекты — в куче, внутри объекта класса (так с Java 7).

■ ЛОВУШКА

- Прямая память входит в лимит -Xmx?
→ Нет, она вне кучи (её используют NIO и Netty). Можно получить нехватку памяти по этой области, когда куча ещё почти пустая. Ограничивают её флагом `MaxDirectMemorySize` и следят за ней отдельно.

Сборка мусора

- Какие есть сборщики и какой по умолчанию?
→ Serial, Parallel (был по умолчанию до Java 8), G1 (по умолчанию с Java 9, делит кучу на регионы), а также ZGC и Shenandoah с очень короткими паузами. Старый сборщик CMS убрали в Java 14.
- Как находится мусор?
→ Сборщик идёт от корней (стеки потоков, статические поля) и помечает всё, до чего можно дойти по ссылкам. Если до объекта дойти нельзя — это мусор. Взаимные ссылки друг на друга проблемой не являются, потому что это не подсчёт ссылок.
- Что такое остановка мира (stop-the-world)?
→ Это пауза, во время которой все потоки приложения заморожены. Такая пауза есть у любого сборщика, отличается только длина: у Parallel — сотни миллисекунд, у G1 — десятки, у ZGC — единицы. Полностью беспаузной сборки не бывает.

■ ЛОВУШКА

- Сборщик удалит объект сразу после присваивания null?
→ Нет — только когда запустится сборка мусора, и только если на объект больше нет ссылок. Вызов `System.gc()` — лишь просьба, JVM вправе её проигнорировать.

Виды нехватки памяти (OutOfMemoryError)

- Какие бывают виды?
→ Нехватка кучи; слишком много времени уходит на сборку мусора; переполнение Metaspace (много классов — например, из-за прокси или утечки загрузчика классов); нехватка прямой памяти; невозможность создать новый поток.
- Что делать на проде?
→ Включить сохранение дампа при нехватке памяти и разобрать его в Eclipse MAT или VisualVM. Просто поднимать -Xmx без анализа нельзя — утечка вернётся снова.

■ ЛОВУШКА

- StackOverflowError — это нехватка памяти?
→ Нет. Это переполнение стека потока, обычно из-за бесконечной рекурсии, а не проблема кучи. Размер стека задаётся флагом -Xss. Ошибки нехватки памяти в обычном коде не ловят.

Загрузчик классов

- Иерархия и «сначала спроси родителя»?
→ Порядок такой: Bootstrap (системные классы), затем Platform, затем Application (ваш classpath), затем свои загрузчики. Каждый сначала просит загрузить класс родителя — поэтому подменить, например, `java.lang.String` нельзя. В веб-контейнерах порядок обратный.
- Разница ClassNotFoundException и NoClassDefFoundError?
→ Первая — когда класс ищут динамически через `Class.forName` и не находят. Вторая — когда при компиляции класс был, а во время работы его в classpath нет.

■ ЛОВУШКА

- Что задаёт уникальность класса и как течёт загрузчик?
→ Класс определяется парой: имя плюс его загрузчик. Один и тот же класс, загруженный двумя разными загрузчиками, — это разные классы, и будет ошибка приведения. А статическая ссылка на объект из веб-приложения не даёт при перезапуске освободить старый загрузчик, из-за чего Metaspace растёт до нехватки памяти.

► 04 • Многопоточность

synchronized, volatile и Atomic

- Что гарантирует synchronized?
 - Он даёт две вещи: только один поток одновременно внутри блока, и видимость изменений другим потокам. Один и тот же поток может войти в блок повторно. Обычный метод блокирует текущий объект, статический — сам класс.
- Что даёт volatile?
 - Только видимость: изменение переменной сразу видят другие потоки, и запрещается переставлять операции местами. Атомарности составных действий он не даёт.
- Как работает Atomic и когда что брать?
 - Внутри команда процессора «сравнить и записать»: если значение то, что ожидали, — меняем; иначе повторяем в цикле, без блокировок. Флаг — берём volatile; счётчик — Atomic; группу связанных полей — synchronized или замок.

■ ЛОВУШКА

- volatile long counter; counter++ — это атомарно?
 - Нет, это гонка. Операция counter++ состоит из трёх шагов: прочитать, увеличить, записать обратно. Между ними может вклиниться другой поток, и часть увеличений потеряется. Нужен AtomicLong или synchronized. volatile отвечает за видимость, но не за атомарность.
- synchronized(new Object()) внутри метода — что не так?
 - Синхронизация идёт по локальному объекту, а каждый поток создаёт свой собственный — поэтому никакой блокировки на самом деле нет.

Пул потоков (ExecutorService)

- Почему не создавать поток через new Thread?
 - Создание потока дорогое: свой стек около мегабайта, обращение к операционной системе. Нет переиспользования и нет ограничения сверху — при нагрузке кончится память. Пул же переиспользует уже созданные потоки.
- Разница shutdown() и shutdownNow()?
 - shutdown() перестаёт принимать новые задачи, но доделывает текущие. shutdownNow() пытается прервать все задачи и возвращает те, что не успели выполняться. Если не вызвать shutdown, обычные (не фоновые) потоки не дадут программе завершиться.
- Как выбрать размер пула?
 - Для задач, нагружающих процессор, — примерно по числу ядер. Для задач с ожиданием (сеть, диск) — больше: чем дольше ждём, тем больше потоков. Ставить наугад 200 не стоит: переключения между потоками съедят выигрыш.

■ ЛОВУШКА

- newFixedThreadPool готов к продакшену?
 - Нет: у него очередь без ограничения, и задачи копятся до нехватки памяти. В продакшене создают ThreadPoolExecutor вручную — с ограниченной очередью и правилом, что делать при переполнении.

CompletableFuture

- Чем он лучше обычного Future?
 - У Future есть только ожидание результата, которое блокирует поток. У CompletableFuture — цепочки шагов, объединение нескольких задач и удобная обработка ошибок.
- На каком пуле он работает по умолчанию?
 - На общем пуле для всей программы, его размер — число ядер минус один. Отдавать туда задачи с ожиданием (сеть, диск) нельзя: заблокируется всё приложение.
- Разница thenApply и thenApplyAsync?
 - thenApply выполняется в том же потоке, что завершил предыдущий шаг. thenApplyAsync отправляет шаг в пул (по умолчанию — общий). Для задач с ожиданием передавай туда свой отдельный пул.

■ ЛОВУШКА

// JavaJub

Полная версия — в JavaJub Pro.

Шпаргалка Java Middle · Java · Middle

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