Часть пятая, общие вопросы разработки ПО.

GoncharovA Follow
Jan 3 · 7 min read

Заключительная часть списка замечательных вопросов. Здесь собраны точные и наиболее ёмкие ответы по теме программирования в общем.

. . .

66. Thread pool vs creating your own threads:

Пул потоков обеспечит преимущества для частых и относительно коротких операций. Повторное использование уже созданных потоков вместо создания новых (что приставляет собой сравнительно дорогостоящую операцию). Самостоятельное создание нового потока было бы более уместным, если бы задание длилось относительно долго (возможно, около секунды или двух, но это зависит от конкретной ситуации).

67. Принципы ООП:

Наследование — механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.

Абстракция — означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.

Инкапсуляция — означает ограничение доступа к данным и возможностям их изменения.

Полиморфизм — это возможность работать с несколькими типами так, будто это один и тот же тип. При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат.

68. SOLID:

Принцип единственной ответственности (SRP) Соответствует букве S акронима SOLID. Согласно этому принципу, не должно быть более одной причины для изменения класса, или класс должен всегда обрабатывать одну функциональность.

Принцип открытости/закрытости (OCP) Соответствует букве O акронима SOLID. Принцип можно выразить так: «Классы, методы или функции должны быть открыты для расширения (добавления новой функциональности) и закрыты для модификации».

Принцип подстановки Барбары Лисков (LSP) Соответствует букве L акронима SOLID. Согласно этому принципу подтипы должны быть заменяемыми для супертипа.

Принцип разделения интерфейса (ISP) Соответствует букве I акронима SOLID. Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы программные сущности маленьких интерфейсов знали только о методах, которые необходимы им в работе. Этот принцип подразумевает, что интерфейс, который не используется, не должен быть реализован. В основном это происходит, когда один интерфейс содержит несколько функциональностей, и клиенту нужна только одна из них, а другие — нет.

Принцип инверсии зависимостей (DIP) Соответствует букве D акронима SOLID. “Зависимость на Абстракциях. Нет зависимости на что-то конкретное”. Прелесть этого принципа проектирования в том, что любой класс легко тестируется с помощью фиктивного объекта и проще в обслуживании, потому что код создания объекта централизован, а клиентский код не перегружен им.

69. Паттерны проектирования:

Порождающие: Простая фабрика, Фабричный метод, Абстрактная фабрика, Строитель, Прототип, Одиночка.

Структурные: Адаптер, Мост, Компоновщик, Декоратор, Фасад, Заместитель.

Поведенческие: Цепочка ответственности, Команда, Итератор, Посредник, Хранитель, Наблюдатель, Посетитель, Стратегия, Состояние, Шаблонный метод.

70. POST vs GET запросы:

Используйте GET для безопасных действий и POST для небезопасных. RFC указывает Интернет браузерам на то, что пользователей необходимо предупреждать о повторном использовании POST запроса, потому что это действие потенциально небезопасно (к примеру оформление онлайн оплаты).

Используйте POST для операций с важной информацией.

Используйте POST для операций с большими данными.

Используйте GET в AJAX приложениях.

71. Lru Cache — рассказать общие принципы:

LRU (least recently used) — это алгоритм, при котором вытесняются значения, которые дольше всего не запрашивались. Соответственно, необходимо хранить время последнего запроса к значению. И как только число закэшированных значений превосходит N необходимо вытеснить из кеша значение, которое дольше всего не запрашивалось.

72. Сложность алгоритмов — определение и виды:

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

O(1) — время работы алгоритма вообще не зависит от размера входных данных.

O(n) — линейная сложность — Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

O(log n) — логарифмическая сложность — Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

$O(n^2)$ — квадратичная сложность — Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n * n, т. е. n^2.

73. Git flow — рассказать общие принципы:

Ветвь master создаётся при инициализации репозитория, что должно быть знакомо каждому пользователю Git. Параллельно ей также мы создаём ветку для разработки под названием develop. Мы считаем ветку origin/master главной. То есть, исходный код в ней должен находиться в состоянии production-ready в любой произвольный момент времени.

Ветвь origin/develop мы считаем главной ветвью для разработки. Хранящийся в ней код в любой момент времени должен содержать самые последние изданные изменения, необходимые для следующего релиза. Эту ветку также можно назвать «интеграционной». Она служит источником для сборки автоматических ночных билдов. Когда исходный код в ветви разработки (develop) достигает стабильного состояния и готов к релизу, все изменения должны быть определённым способом влиты в главную ветвь (master) и помечены тегом с номером релиза.

Помимо главных ветвей master и develop, наша модель разработки содержит некоторое количество типов вспомогательных ветвей, которые используются для распараллеливания разработки между членами команды, для упрощения внедрения нового функционала (features), для подготовки релизов и для быстрого исправления проблем в производственной версии приложения. В отличие от главный ветвей, эти ветви всегда имеют ограниченный срок жизни. Каждая из них в конечном итоге рано или поздно удаляется. Ветви функциональностей (feature branches) Могут порождаться от: develop Должны вливаться в: develop. Соглашение о наименовании: всё, за исключением master, develop, release-* или hotfix-*. Ветви функциональностей (feature branches), также называемые иногда тематическими ветвями (topic branches), используются для разработки новых функций, которые должны появиться в текущем или будущем релизах. При начале работы над функциональностью (фичей) может быть ещё неизвестно, в какой именно релиз она будет добавлена. Смысл существования ветви функциональности (feature branch) состоит в том, что она живёт так долго, сколько продолжается разработка данной функциональности (фичи). Когда работа в ветви завершена, последняя вливается обратно в главную ветвь разработки (что означает, что функциональность будет добавлена в грядущий релиз) или же удаляется (в случае неудачного эксперимента).

Ветви релизов (release branches) Могут порождаться от: develop Должны вливаться в: develop и master Соглашение о наименовании: release-*. Ветви релизов (release branches) используются для подготовки к выпуску новых версий продукта. Они позволяют расставить финальные точки над i перед выпуском новой версии. Кроме того, в них можно добавлять минорные исправления, а также подготавливать метаданные для очередного релиза (номер версии, дата сборки и т.д.). Когда вся эта работа выносится в ветвь релизов, главная ветвь разработки (develop) очищается для добавления последующих фич (которые войдут в следующий большой релиз).

Ветви исправлений (hotfix branches) Могут порождаться от: master Должны вливаться в: develop и master Соглашение о наименовании: hotfix-*. Ветви для исправлений (hotfix branches) весьма похожи на ветви релизов (release branches), так как они тоже используются для подготовки новых выпусков продукта, разве лишь незапланированных. Они порождаются необходимостью немедленно

исправить нежелательное поведение производственной версии продукта. Когда в производственной версии находится баг, требующий немедленного исправления, из соответствующего данной версии тега главной ветви (master) порождается новая ветвь для работы над исправлением.

74. Сложности вставки и поиска в ArrayList, LinkedList, HasMap:

ArrayList реализован внутри в виде обычного массива. Поэтому при вставке элемента в середину, приходится сначала сдвигать на один все элементы после него, а уже затем в освободившееся место вставлять новый элемент. Зато в нем быстро реализованы взятие и изменение элемента — операции get, set, так как в них мы просто обращаемся к соответствующему элементу массива.

LinkedList реализован внутри по-другому. Он реализован в виде связного списка: набора отдельных элементов, каждый из которых хранит ссылку на следующий и предыдущий элементы. Чтобы вставить элемент в середину такого списка, достаточно поменять ссылки его будущих соседей. А вот чтобы получить элемент с номером 130, нужно пройтись последовательно по всем объектам от 0 до 130. Другими словами операции set и get тут реализованы очень медленно.

У HashMap время доступа к отдельному элементу — O(1) при условии, что хэш-функция (Object.hashCode()) определена нормально (что является правдой в случае Integer).

75. Виды сортировок — просто перечислить:

Простейшая сортировка (Bubble Sort), Сортировка выбором (Selection Sort), Сортировка вставками (Insertion Sort), Челночная сортировка (Shuttle Sort), Сортировка Шелла, Сортировка слиянием (merge sort), Сортировка подсчётом (Counting Sort), Поразрядная сортировка (Radix Sort), Быстрая сортировка (Quick Sort).

76. Принципы функционального программирования:

Все функции — чистые. Это правило безусловно является основным в функциональном программировании. Все функции являются чистыми, если они удовлетворяют двум условиям: Функция, вызываемая от одних и тех же

аргументов, всегда возвращает одинаковое значение. Во время выполнения функции не возникают побочные эффекты.

Все функции — первого класса и высшего порядка. Эта концепция — не особенность ФП (она используется в Javascript, PHP и других языках) — но его обязательное требование. Для того, чтобы функция была первоклассной, у неё должна быть возможность быть объявленной в виде переменной. Это позволяет управлять функцией как обычным типом данных и в то же время исполнять её. Функции высшего порядка же определяются как функции, принимающие другую функцию как аргумент или возвращающие функцию. Типичными примерами таких функций являются map и filter.

Переменные неизменяемы. В функциональном программировании вы не можете изменить переменную после её инициализации. Вы можете создавать новые, но не можете изменять существующие — и благодаря этому вы можете быть уверены, что никакая переменная не изменится.

Относительная прозрачность функций — если вы можете заменить вызов функции на возвращаемое значение, и состояние при этом не изменится, то функция относительно прозрачна.

Функциональное программирование сильно опирается на математическую систему, называющуюся лямбда-исчислением. Два ключевых принципа лямбда-исчисления, которые формируют самое понятие функционального программирования: В лямбда-исчислении все функции могут быть анонимными, поскольку единственная значимая часть заголовка функции — это список аргументов. При вызове все функции проходят процесс каррирования. Он заключается в следующем: если вызывается функция с несколькими аргументами, то сперва она будет выполнена лишь с первым аргументом и вернёт новую функцию, содержащую на 1 аргумент меньше, которая будет немедленно вызвана. Этот процесс рекурсивен и продолжается до тех пор, пока не будут применены все аргументы, возвращая финальный результат. Поскольку функции являются чистыми, это работает.

. . .

Все ответы собраны из следующих источников:

https://tproger.ru/

https://askdev.ru/

https://developer.android.com/

https://www.codeproject.com/

https://proglib.io/

http://javastudy.ru/interview/collections/

Development Programming Android App Development