1. Сетевое взаимодействие - клиент-серверная архитектура, основные протоколы, их сходства и отличия.

Сетевое взаимодействие - это процесс обмена данными между различными компьютерами или устройствами, подключенными к сети.

Один из основных способов сетевого взаимодействия - это клиент-серверная архитектура. Клиент-серверная архитектура - это модель распределенной системы, состоящей из двух типов компонентов: клиентов и серверов.

Клиенты - это программы или устройства, которые обращаются к серверам за выполнением определенных задач или получением данных.

Серверы - это программы или устройства, которые обрабатывают запросы клиентов и предоставляют им нужные данные или услуги.

То есть клиент отправляет пакет с ip, содержимым и хэш-суммой, а сервер его должен получить и обработать.

Для чего нужны протоколы? Протоколы используются для передачи данных между клиентом и сервером. Они также используют набор различных команд для управления передачей данных. Однако у каждого протокола есть свои уникальные особенности и ограничения, которые определяются его спецификацией и назначением.

Отличия протоколов заключаются в их назначении и способности передавать определенные типы данных и сигналов. Например, протокол HTTP предназначен для передачи гипертекстовых документов, а протокол FTP - для передачи файлов. Протокол SMTP используется для передачи электронной почты, а протокол IMAP для доступа к почтовому ящику на удаленном сервере. Каждый протокол имеет свой набор правил и синтаксис команд для взаимодействия между клиентом и сервером.

1. Протокол TCP. Классы Socket и ServerSocket.

Transmission Control Protocol - это протокол, который обеспечивает надежную передачу данных. Он гарантирует, что данные будут доставлены получателю в правильном порядке и без потерь.

В языке Java для работы с протоколом TCP используются классы Socket и ServerSocket. Класс Socket представляет клиентский сокет, который может осуществлять соединение с сервером. Класс ServerSocket представляет серверный сокет, который может принимать клиентские соединения.

Для создания клиентского сокета нужно создать объект класса Socket, указав IP-адрес и порт сервера:

Socket socket = new Socket("localhost", 8080);

Для создания серверного сокета нужно создать объект класса ServerSocket, указав порт, на котором сервер будет принимать соединения:

ServerSocket serverSocket = new ServerSocket(8080);

После создания серверного сокета можно начинать принимать клиентские соединения с помощью метода accept():

Socket socket = serverSocket.accept();

Метод accept() блокирует выполнение программы, пока не будет установлено соединение с клиентом. После установки соединения можно получать и отправлять данные с помощью потоков ввода-вывода:

// получение потока ввода

InputStream inputStream = socket.getInputStream();

// получение потока вывода

OutputStream outputStream = socket.getOutputStream();

Также для коммуникации между клиентом и сервером можно использовать классы BufferedWriter и BufferedReader:

BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));

BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(outputStream));

// отправка сообщения на сервер

writer.write("Hello from client!");

writer.newLine(); // добавление перевода строки

writer.flush(); // очистка буфера и отправка сообщения

// получение ответа от сервера

String response = reader.readLine();

System.out.println("Server response: " + response);

Классы Socket и ServerSocket позволяют создавать приложения, которые могут обмениваться данными между собой по протоколу TCP.

**Скорость саморегулируется, иногда замедляется, когда ответ о подтверждении получения не доходит до клиента.**

1. Протокол UDP. Классы DatagramSocket и DatagramPacket.

Протокол UDP (User Datagram Protocol) - это простой протокол транспортного уровня, который обеспечивает доставку ненадежных датаграмм (данных), не гарантирующих сохранения их последовательности и успешной доставки.

Для работы с протоколом UDP в Java используются классы DatagramSocket и DatagramPacket. Класс DatagramSocket представляет сокет для отправки и получения датаграмм, а класс DatagramPacket - датаграмму, которая может быть отправлена или получена.

Протокол UDP не гарантирует доставку данных и сохранение их порядка, поэтому приложение должно быть готово к потере и дублированию пакетов. Однако, протокол UDP часто используется для реализации реального времени или потоковой передачи данных, где скорость передачи имеет большое значение, а возможные потери не критичны.

1. Отличия блокирующего и неблокирующего ввода-вывода, их преимущества и недостатки. Работа с сетевыми каналами.

1. Отличия блокирующего и неблокирующего ввода-вывода:

- Блокирующий ввод-вывод: когда приложение запрашивает данные из внешнего источника, выполнение приложения блокируется до тех пор, пока данные не будут получены. То же самое происходит и при передаче данных во внешний источник.

- Неблокирующий ввод-вывод: когда приложение запрашивает данные из внешнего источника, выполнение не блокируется, приложение продолжает работу, даже если данные еще не получены.

2. Преимущества неблокирующего ввода-вывода:

- Неблокирующий ввод-вывод не блокирует выполнение приложения, что позволяет увеличить производительность

- Неблокирующий ввод-вывод позволяет экономить время на запросе и передаче данных, что также ускоряет процесс

- Неблокирующий ввод-вывод позволяет использовать меньшее количество потоков (если их количество было увеличено в блокирующем вводе-выводе для обработки запросов)

3. Недостатки неблокирующего ввода-вывода:

- Неблокирующий ввод-вывод требует большего количества кода

- Неблокирующий ввод-вывод сложнее в использовании и отладке

- По сравнению с блокирующим вводом-выводом неблокирующий имеет меньшее количество операций, которые могут быть выполнены одновременно

4. Работа с сетевыми каналами в JAVA:

Java предоставляет пакет java.nio для работы с неблокирующим вводом-выводом и сокетами. Для работы с сетевыми каналами в Java можно использовать классы Selector, Channel, Buffer и другие.

С помощью класса Selector можно осуществлять мониторинг нескольких каналов со своими состояниями, а классы Channel и Buffer позволяют выполнять операции чтения и записи данных. Для работы с сетевыми каналами необходимо определить протокол обмена данными, создать сокет и открыть каналы, через которые будут передаваться данные. После установления соединения необходимо создать буфер, который будет хранить данные, и выполнять запись и чтение данных.

1. Классы SocketChannel и DatagramChannel.

Классы `SocketChannel` и `DatagramChannel` в Java являются частью пакета `java.nio.channels`.

Класс `SocketChannel` обеспечивает поддержку канала сокета, который можно использовать для установления соединения TCP с сервером или для прослушивания входящих соединений от клиентов. После установления соединения, можно использовать объект `Socket` для передачи данных между клиентом и сервером.

Класс `DatagramChannel` обеспечивает поддержку канала датаграммы, который используется для передачи датаграмм в UDP-протоколе. Датаграммы являются пакетами данных, которые отправляются независимо и могут быть потеряны, повторно переданы или получены в случайном порядке.

Как и другие классы в пакете `java.nio.channels`, `SocketChannel` и `DatagramChannel` поддерживают неблокирующий режим работы, что позволяет принимать и отправлять данные только в том случае, если они доступны. Этот режим позволяет управлять множеством соединений на одном потоке и уменьшить нагрузку на процессор.

В целом, классы `SocketChannel` и `DatagramChannel` предоставляют эффективный способ передачи данных с помощью сокетов, который можно использовать в различных приложениях, включая веб-серверы, чаты, онлайн-игры и другие.

1. Передача данных по сети. Сериализация объектов.

Передача данных по сети - одно из самых важных приложений Java. Для передачи данных по сети используется механизм сериализации объектов.

Сериализация - это процесс преобразования объекта в последовательность байтов для сохранения или передачи в другой процесс или на другой узел.

**Сериализация** — это процесс сохранения состояния объекта в последовательность байт.

**Десериализация** — это процесс восстановления объекта из этих байт.

Для того чтобы объект можно было сериализовать, он должен реализовывать интерфейс Serializable. Кроме того, все поля объекта, которые мы хотим сериализовать, также должны быть сериализуемыми. После этого **объект может быть сериализован при помощи ObjectOutputStream:**

public class MyClass implements Serializable {

private String field1;

private int field2;

// Конструктор, геттеры и сеттеры

public static void main(String[] args) throws IOException {

MyClass myObject = new MyClass("Hello", 42);

FileOutputStream fileOutputStream = new FileOutputStream("serializedObject");

ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);

objectOutputStream.writeObject(myObject);

objectOutputStream.close();

fileOutputStream.close();

}

}

В этом примере мы создаем объект MyClass и сериализуем его в файл "serializedObject". Можно также отправлять сериализованный объект по сети.

**Для десериализации объекта используется класс ObjectInputStream:**

```java

public static void main(String[] args) throws IOException, ClassNotFoundException {

FileInputStream fileInputStream = new FileInputStream("serializedObject");

ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);

MyClass myObject = (MyClass)objectInputStream.readObject();

objectInputStream.close();

fileInputStream.close();

}

```

Здесь мы считываем сериализованный объект из файла и десериализуем его в объект myObject типа MyClass. Для успешной десериализации класс MyClass также должен быть доступен в сборке. Если класс не найден, будет выброшено исключение ClassNotFoundException.

1. Интерфейс Serializable. Объектный граф, сериализация и десериализация полей и методов.

Интерфейс Serializable в Java является маркерным интерфейсом, который говорит о том, что класс может быть сериализован. Сериализация - это процесс преобразования объекта в поток байтов, который может быть сохранен и передан на другой компьютер. Десериализация - это процесс обратного преобразования потока байтов в объект.

При сериализации объекта также сериализуются все его поля и методы, которые являются сериализуемыми. Если какое-то поле не является сериализуемым, то оно не будет сохранено в потоке байтов.

**Если объект содержит ссылки на другие объекты, то весь объектный граф будет сериализован**. То есть, при сериализации объекта также будут сериализованы все объекты, на которые он ссылается.

Для того чтобы класс был сериализуемым, его необходимо пометить ключевым словом implements Serializable. Например:

public class MyClass implements Serializable {

// поля и методы

}

При сериализации объекта можно использовать стандартные механизмы Java, например, ObjectOutputStream и ObjectInputStream. Например:

MyClass myObj = new MyClass();

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("object.bin"));

**out.writeObject(myObj);**

out.close();

При десериализации объекта необходимо использовать метод readObject() класса ObjectInputStream. Например:

ObjectInputStream in = new ObjectInputStream(new FileInputStream("object.bin"));

MyClass myObj = (MyClass) in.**readObject();**

in.close();

1. Java Stream API. Создание конвейеров. Промежуточные и терминальные операции.

Stream API – новый инструмент в Java, который предлагает использование функционального стиля при работе с различными структурами данных.

Создание конвейера Java Stream API выполняется в несколько шагов:

1. Создание исходного потока объектов. Это может быть коллекция, массив или любой другой тип Java, который реализует интерфейс Iterable.

Пример:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

2. Применение промежуточных операций к потоку объектов. Эти операции не изменяют исходный поток, а создают новый поток.

Примеры промежуточных операций:

- filter() - фильтрация элементов по заданному условию:

Stream<Integer> filterStream = numbers.stream()

.filter(n -> n > 2);

- map() - преобразование каждого элемента из исходного потока в новый объект:

Stream<String> mapStream = numbers.stream()

.map(n -> "Number " + n);

- sorted() - сортировка элементов потока:

Stream<Integer> sortedStream = numbers.stream()

.sorted();

3. Применение терминальной операции к потоку объектов. Эта операция выполняет действия над элементами потока и возвращает результат. После терминальной операции поток закрывается и больше нельзя применять промежуточные операции.

Примеры терминальных операций:

- count() - подсчет количества элементов в потоке:

long count = numbers.stream()

.count();

- forEach() - выполнение заданного действия над каждым элементом в потоке:

numbers.stream()

.forEach(n -> System.out.println(n));

- reduce() - свертывание элементов в один объект, используя заданную функцию:

Optional<Integer> sum = numbers.stream()

.reduce((a, b) -> a + b);

1. Шаблоны проектирования: Decorator, Iterator, Factory method, Command, Flyweight, Interpreter, Singleton, Strategy, Adapter, Facade, Proxy.

**Порождающие** – позволяют создавать объекты и классы с разными параметрами и применяются для определения структуры классов и управления их созданием.

Например, Фабричный метод, Одиночка.

Шаблон Factory Method - используется для создания объектов, не задавая их конкретных типов. Фабричный метод задает интерфейс создания объектов, но позволяет подклассам решать, какой класс объекта создать. Создание объекта переносится из основного кода в иерархию классов.

Шаблон Singleton - используется для создания объекта, который может существовать только в единственном экземпляре. Это достигается через создание приватного конструктора и статической переменной, которая будет хранить единственный экземпляр объекта.

**Структурные** – определяют способы организации классов и объектов в более крупные структуры.

Например, Декоратор, Легковес, Адаптер, Фасад и Заместитель.

1. Декоратор - позволяет динамически добавлять новую функциональность объекту, не изменяя его исходную структуру.
2. Фасад - предоставляет упрощенный интерфейс для сложной системы, скрывающий ее внутреннюю структуру.
3. Прокси - предоставляет объект-заместитель для управления доступом к другому объекту.

Поведенческие – определяют различные способы взаимодействия между объектами и управление их взаимодействием.

Например, Итератор, Команда, Стратегия.

Команда - инкапсулирует запрос в виде объекта, позволяя настраивать параметры вызова их отделить от объекта, который выполняет команду.