

Занятие 4

Хранение данных

Зачем нужно сохранять данные на диск

- Оперативная память дорогая, дисковая - дешевая. Так что можно оперировать значительными объемами данных
- Дисковая память - энергонезависимая
- Технологии резервного копирования

Ну так давайте хранить все в файлах?

Java позволяет записывать и считывать объекты через `ObjectOutputStream/ObjectInputStream`. Или можно записывать в файл данные в произвольном формате.

Основная возникающая проблема - сложность управления сохраненными данными и неэффективность получения данных

СУБД

СУБД - средство, решающее проблему хранения данных на диске, проблему управления сохраненными данными и проблему эффективного получения данных

Обычно Java приложения взаимодействуют с СУБД по сети используя протокол TCP

Виды СУБД

Существует множество видов СУБД, но можно примерно поделить их на 2 основных группы

SQL. Реляционные СУБД, хранящие данные в виде таблиц и позволяющие запрашивать данные на языке запросов SQL

NoSQL. Можно отнести СУБД, с которыми можно оперировать другим способом, кроме SQL и/или хранят данные не в виде таблиц

Обзор SQL

SQL - Structured Query Language

SQL - стандартный язык запросов к табличным данным.

Разработан в 1974(!) году

В данный момент является де-факто стандартом для взаимодействия с реляционными базами данных.

Стоит помнить, что каждая БД может расширять собственный синтаксис SQL, так что запросы от одной базы могут не работать с другой БД

Как хранятся данные в реляционных БД

Колонка - это атрибут сущности.
Например email - атрибут пользователя

Таблица. Совокупность данных
одинаковой структуры

users

Запись в таблице
представляет
собой данные об
одном объекте

id	email	first_name	last_name
1	imask@ylab.ru	Илон	Маск
2	sjobs@ylab.ru	Стив	Джобс
3	bgates@ylab.ru	Билл	Гейтс

В данном примере id - так называемый “первичный ключ”. Искусственно добавляемая колонка, которая обеспечивает уникальность записей. Поиск данных в БД по первичному ключу считается наиболее эффективным

Как получить данные из БД?

users

id	email	first_name	last_name
1	imask@ylab.ru	Илон	Маск
2	sjobs@ylab.ru	Стив	Джобс
3	bgates@ylab.ru	Билл	Гейтс

Для того, чтобы получить данные из БД - необходимо отправить серверу БД специальную команду, в ответ на которую БД вернет данные.

Команда - это выражение на языке SQL, также называется **SQL-запросом**

Пример SQL запроса

users

id	email	first_name	last_name
1	imask@ylab.ru	Илон	Маск
2	sjobs@ylab.ru	Стив	Джобс
3	bgates@ylab.ru	Билл	Гейтс

```
select first_name, last_name  
from users  
where email = 'imask@ylab.ru'
```

Инструкция select сообщает БД, что необходимо вернуть какие то данные. После идет перечисление колонок. Если надо вернуть все колонки - можно использовать *.
select * ..

Имя таблицы из которой надо вернуть данные

Задаются условия, по которым нужно фильтровать данные. В данном случае возвращается только если email совпадает

Добавление данных в таблицу

Имя таблицы, в которую
добавляем данные

insert into

user

(id, email, first_name, last_name)

Список колонок, которые мы
хотим заполнить

values

(4, 'jbesos@ylab.com', 'Jeff', 'Besos');

Значения, которые необходимо
сохранять в колонках. Должны
идти в том же порядке, что и
перечисление колонок

Обновление данных в таблице

Имя таблицы, в которой
обновляем данные

update

user

set

email = 'tcook@ylab.com',

first_name = 'Tim',

last_name= 'Cook'

where

id = 2

Список обновляемых значений
для каждой колонки

Условия, указывающие, какую
именно запись надо обновить.
Если не указать - будут
обновлены все записи в
таблице


Создание таблицы

```
create table users (  
  id int primary key,  
  email varchar(50),  
  first_name varchar(50),  
  last_name varchar(50)  
)
```

Имя создаваемой таблицы



Список колонок с указанием типа колонки
primary key - специальная инструкция,
указывающая, что поле всегда имеет значение и
оно уникальное для всех записей



```
drop table users
```



JDBC

JDBC

JDBC - Java Database Connectivity. Стандарт, описывающий взаимодействие java приложений с SQL-совместимыми базами данных. Каждый вендор БД выпускает собственную реализацию данного стандарта для обеспечения работы со своей БД

`javax.sql.DataSource` - инкапсулирует информацию о типе БД, параметрах подключения

`javax.sql.Connection` - инкапсулирует сеанс работы с БД

`javax.sql.Statement/java.sql.PreparedStatement` - инкапсулирует конкретный запрос, который будет выполняться в БД

`javax.sql.ResultSet` - инкапсулирует результат выполнения запроса

DataSource -> Connection -> Statement -> ResultSet

JDBC Example. PostgreSQL

```
public static void main(String[] args) throws SQLException {  
    DataSource dataSource = initDataSource();  
    try (Connection connection = dataSource.getConnection();  
        Statement statement = connection.createStatement();  
        ResultSet rs = statement.executeQuery("select * FROM users")) {  
        while (rs.next()) {  
            int columnCount = rs.getMetaData().getColumnCount();  
            for (int i = 1; i <= columnCount; i++) {  
                String columnName = rs.getMetaData().getColumnName(i);  
                String columnValue = rs.getString(i);  
                System.out.print(columnName + " = " + columnValue + ", ");  
            }  
            System.out.println();  
        }  
    }  
}
```

```
<dependency>  
    <groupId>org.postgresql</groupId>  
    <artifactId>postgresql</artifactId>  
    <version>42.2.6</version>  
</dependency>
```

Live Coding Section

Пользователь вводит в консоли строки. Окончание ввода - пустая строка.
Для каждой строки создать запись в таблице message

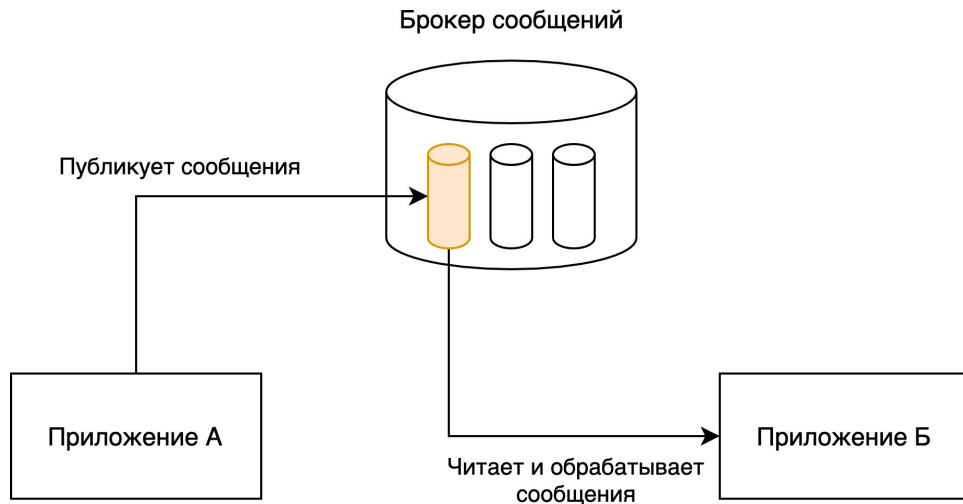
```
create table message (  
  id primary key, - идентификатор записи  
  dt datetime, - время создания записи  
  message text - введенное сообщение  
)
```

Сообщения RabbitMQ

Сообщения

Messaging - это подход к разработке, когда одно приложение публикует сообщения, а другое считывает.

Сообщение - в общем случае любой объект, представленный в сериализованном виде



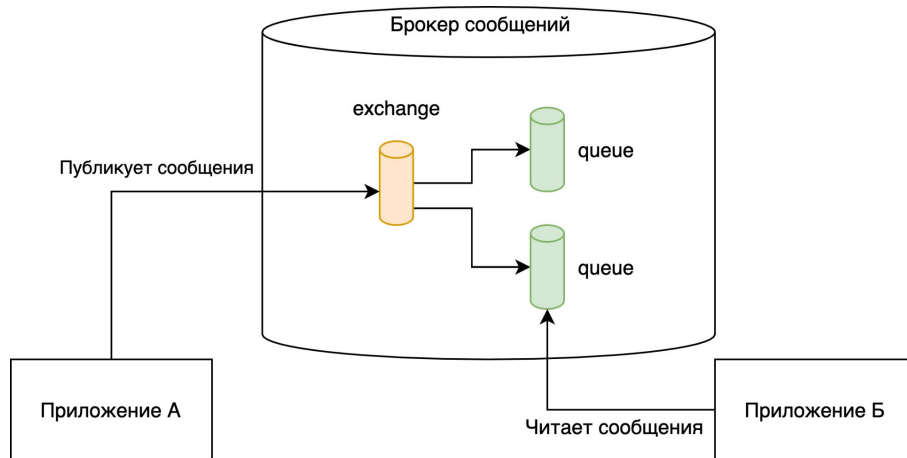
Зачем нужен Messaging?

- Позволяет организовать асинхронную обработку. Одно приложение публикует данные и не ждет окончания обработки
- Позволяет повысить отказоустойчивость. Если приложение упадет, сообщения останутся в брокере
- Позволяет повысить масштабируемость. Можно развернуть 2 экземпляра приложения, читающих сообщения и работать параллельно
- Приложения могут писать и читать сообщения в том темпе, в котором могут. Их производительность не зависит от производительности друг друга

RabbitMQ. Exchange & Queue

Понятия Exchange и Queue описаны в стандарте AMQP. Но в данный момент можно остановиться на следующем описании

- **Queue.** Очередь сообщений. Как бы “физическое” хранилище сообщений. Слушатель, когда необходимо получать сообщения, подписывается именно на очередь. Как только сообщение было прочитано каким-либо слушателем, оно удаляется из очереди
- **Exchange.** Нечто вроде “почтового ящика”. Когда отправитель отправляет сообщение, он указывает не очередь, а exchange. RabbitMQ, при получении сообщения обрабатывает присланный Exchange и сохраняет сообщение в одну (или больше) очередей, откуда потом их сможет прочитать слушатель



RabbitMQ. Отправка сообщений в Java

```
public static void main(String[] args) throws Exception {  
    String exchangeName = "exc";  
    String queueName = "queue";  
  
    ConnectionFactory connectionFactory = initConnectionFactory();  
    try (Connection connection = connectionFactory.newConnection();  
        Channel channel = connection.createChannel())  
    {  
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.DIRECT);  
        channel.queueDeclare(queueName, true, false, false, null);  
        channel.queueBind(queueName, exchangeName, "*");  
  
        channel.basicPublish(exchangeName, "key", null, "Hello World".getBytes());  
    }  
}
```

Имя exchange

Имя очереди

Создаем объект соединения

Создаем канал связи

Объявляем exchange, очередь. Затем связываем

Отправляем массив байт в exchange

```
<dependency>  
    <groupId>com.rabbitmq</groupId>  
    <artifactId>amqp-client</artifactId>  
    <version>5.16.0</version>  
</dependency>
```

RabbitMQ. Получение сообщений в Java

```
public static void main(String[] args) throws Exception {  
    String queueName = "queue";  
  
    ConnectionFactory connectionFactory = initConnectionFactory();  
    try (Connection connection = connectionFactory.newConnection();  
        Channel channel = connection.createChannel()) {  
        while (!Thread.currentThread().isInterrupted()) {  
            GetResponse message = channel.basicGet(queueName, true);  
            if (message == null) {  
                // no messages  
            } else {  
                String received = new String(message.getBody());  
                System.out.println(received);  
            }  
        }  
    }  
}
```

Имя очереди

Создаем объект соединения

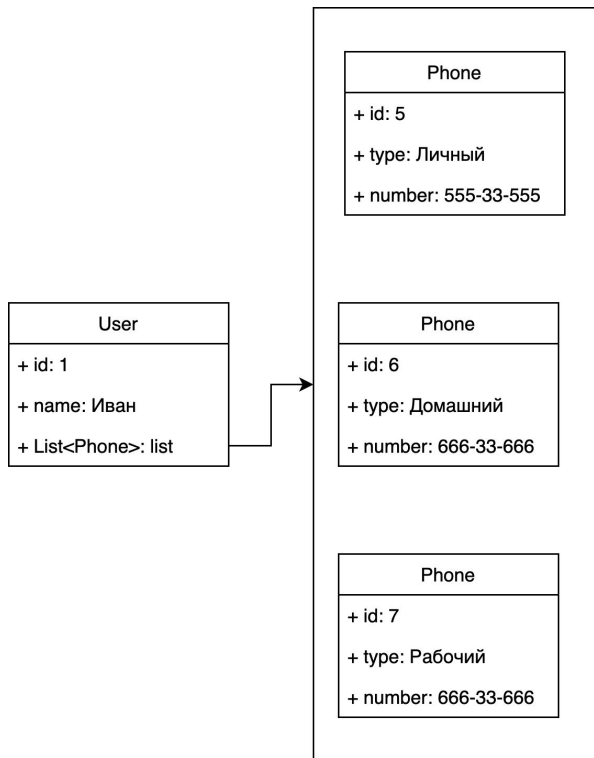
Создаем канал связи

Запускаем бесконечный цикл. И пытаемся получить сообщения из очереди

Если полученный результат null - значит новых сообщений нет

Обработка полученного сообщения

А если отправить что-то сложнее строки?



1. Сериализация -> byte[] -> Base64 -> String
2. CSV-like (1;Иванов;3;1;Личный;555-33-555....)
3.
4. Формат обмена данными (XML, **JSON**, Protobuf, Thrift...)



JSON

JSON

JSON - текстовый формат обмена данными, основанный на JavaScript.

```
{
  "id":1,
  "name":"Иван",
  "phones":[
    {
      "id":5,
      "type":"Личный",
      "number":"555-33-555"
    },
    {
      "id":6,
      "type":"Домашний",
      "number":"666-33-666"
    },
    {
      "id":7,
      "type":"Рабочий",
      "number":"666-33-666"
    }
  ]
}
```

Объект:

```
{
  ...
  "<имя атрибута>": <значение>
  ...
}
```

Значение может быть строкой, числом, boolean (в некоторых спецификациях), массивом (`[]`) или объектом (`{ }`)

JSON в Java

Библиотека Jackson позволяет настроить отображение объектов в строку и обратно. Как вариант - можно использовать Map

```
1 <dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.13.3</version>
</dependency>
```

```
3 public static void main(String[] args) throws Exception {
    String s = ...;
    ObjectMapper objectMapper = new ObjectMapper();
    User user = objectMapper.readValue(s, User.class);
}
```

```
public class User {      2
    private Long id;
    private String name;
    private List<Phone> phones;
    //Getters, Setter, C-tor
}

public class Phone {
    private Long id;
    private String type;
    private String number;
    //Getters, Setters, C-tor
}
```

JSON в Java

```

▼ ■ user = {User@1316}
  > (f) id = {Long@1319} 1
  > (f) name = "Иван"
  ▼ (f) phones = {ArrayList@1321} size = 3
    ▼ ■ 0 = {Phone@1323}
      > (f) id = {Long@1326} 5
      > (f) type = "Личный"
      > (f) number = "555-33-555"
    ▼ ■ 1 = {Phone@1324}
      > (f) id = {Long@1329} 6
      > (f) type = "Домашний"
      > (f) number = "666-33-666"
    ▼ ■ 2 = {Phone@1325}
      > (f) id = {Long@1332} 7
      > (f) type = "Рабочий"
      > (f) number = "666-33-666"

```

Обзор домашнего задания