## Оглавление



## Даты и календарь

java.util.Date хранит дату и время. В его наследниках: java.sql.Date хранит только дату (время на начало дня), в java.sql.Time отсекается дата и хранится время с начала дня, в java.sql.TimeStamp хранится и дата и время.

### Long time → java.sql.Date

Создать java.sql.Date можно задав время в мс от 1973 года (UNIX BD):

```
java.sql.Date(Long time)
```

### java.util.Date → java.sql.Date

```
java.util.Date utilDate = xxx;
java.sql.Date sqlDate = new java.sql.Date(xxx.getTime());
```

### String → java.sql.Date

Строкой по шаблону даты:

```
java.text.DateFormat formatter = new java.text.SimpleDateFormat(“yyyy-MM-dd”);
java.util.Date utilDate = (java.util.Date)formatter.parse(“2000-01-01”);
java.sql.Date sqlDate = new java.sql.Date(utilDate.getTime());
```

### java.util.Date → String

```
java.util.Date utilDate = xxx;
java.text.DateFormat formatter = new java.text.SimpleDateFormat(“yyyy-MM-dd”);
String dataStr = formatter.format(utilDate);
```

### Прибавление к дате N дней

```
java.util.Date utilDate = xxx;
int N = yyy;
java.util.Calendar calendar = java.util.Calendar.getInstance();
calendar.setTimeInMillis(utilDate.getTime());
calendar.add(Calendar.DAY_OF_MONTH, N);
java.sql.Date sqlDate = new java.sql.Date(calendar.getTimeInMillis());
```

### Calendar. Временные пояса и локали

Calendar – абстрактный класс представления абсолютного времени (временной метки) от p.U. (Unix Epoch, 01.01.1973 00:00:00 по UTC). Временная метка = количеству секунд, прошедших с p.U.

UTC – всемирное координированное время ~ GMT +0 (время по Гринвичу, 0-меридиан).

Представление этой временной метки учитывает часовой пояс и национальные форматы (локали). Реализации Calendar производятся его же статистическими методами. В JSE доступна реализация GregorianCalendar.

*Rem: Etc/GMT – "стиль POSIX" часового пояса. Он обратен обычному GMT. Т.е. Etc/GMT-3 = GMT/+3*

Ex: пример получения текущей временной временной метки и ее представления в заданной TZ

```
// текущая временная метка, TimeZone по-умолчанию, но она не важна
java.util.Date utilDate = new java.util.Date();
// задание новой TZ и локали для представления
String timeZone = "Europe/Moscow"; // "GMT+3";
String locale = "ru";
Calendar calendar = Calendar.getInstance(TimeZone.getTimeZone(timeZone), new Locale(locale));
// передача в календарь временной метки (абсолютного Unix-времени) полученной даты
calendar.setTime(utilDate);
// вывод локальных параметров представления даты в новой TZ и локали
System.out.println("Time Zone  = "+timeZone);
System.out.println("Locale = "+locale);
System.out.println("Year = "+calendar.get(Calendar.YEAR));
System.out.println("Month (start from 0!) = "+calendar.get(Calendar.MONTH));
System.out.println("Day = "+calendar.get(Calendar.DAY_OF_MONTH));
System.out.println("Day of week = "+calendar.get(Calendar.DAY_OF_WEEK));
System.out.println("Hour = "+calendar.get(Calendar.HOUR_OF_DAY));
System.out.println("Minute = "+calendar.get(Calendar.MINUTE));
System.out.println("Second = "+calendar.get(Calendar.SECOND));
```

Также календарь можно использовать для получения временной метки по заданному представлению в заданной TimeZone.

Ex: пример получения текущей временной временной по ее представлению в заданной TZ. Дано: String tzStr="GMT+5", String dateStr = "2000-01-01", String timeStr = "22:33:14". Требуется получить абсолютное Unix-время в миллисекундах.

```
//Создаем полную дату в текущей (неправильной) TZ по заданному представлению
java.text.DateFormat formatter = new java.text.SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
java.util.Date date = (java.util.Date)formatter.parse(dateStr+" "+timeStr);
// Создаем календарь в текущей (неправильной) TZ
Calendar c0 = Calendar.getInstance();
//Передаем в этот текущий календарь текущую дату
//Временная метка при этом будет сдвинута, но локальные параметры представления
//год/месяц/число/часы/минуты/секунды останутся правильными
c0.setTimeInMillis(date.getTime());
//Создаем новый календарь в заданной (правильной) TZ
Calendar c1 = Calendar.getInstance(TimeZone.getTimeZone(tzStr));
//Передаем в этот календарь с правильной TZ правильные локальные параметры представления
//Тем самым формируется правильная временная метка
c1.set(Calendar.YEAR, c0.get(Calendar.YEAR));
c1.set(Calendar.MONTH, c0.get(Calendar.MONTH));
```

```
c1.set(Calendar.DAY_OF_MONTH, c0.get(Calendar.DAY_OF_MONTH ));
c1.set(Calendar.HOUR_OF_DAY, c0.get(Calendar.HOUR_OF_DAY));
c1.set(Calendar.MINUTE, c0.get(Calendar.MINUTE));
c1.set(Calendar.SECOND, c0.get(Calendar.SECOND));
//Получаем правильную временную метку
Long time = c1.getTimeInMillis();
```

## Logger

Для ведения логов приложения в JSE предусмотрено дерево логгеров java.util.logging.Logger, которые пишут различные сообщения о ходе выполнения приложения в указанный источник (по-умолчания в System.out). Каждый из них характеризуется ID и уровнем логгирования (Log Level).

ID логгера – строка вида xxx.yyy.zzz, задающая имя логгера в иерархическом пространстве имен. Часто в качестве имени используют F.Q.N. класса-хозяина логгера.

Log Level – константа логгера типа java.util.logging.Level, задающая уровень логгирования, т.е. определяющая те вызовы методов этого логгера, которым разрешено писать лог. Уровни логгирования поддерживают порядок приоритета, в порядке возрастания Level = {FINEST, FINER, FINE, CONFIG, INFO, WARNING, SEVERE}. Общеприкладной уровень логгирования для каждого логгера или их совокупности задается одной из этих констант классом LogManager при инициализации приложения. По-умолчанию общеприкладной уровень логгирования берется из предкового логгера. Каждый из методов логгирования также вызывается с одним из уровней Level, который сравнивается с общеприкладным для данного логгера: если общеприкладной выше, то лог не пишется, в противном случае пишется.

Общеприкладная настройка логгирования осуществляется при инициализации приложения. Конфиг можно подсунуть LogManager'у в виде файла:

```
# logger.conf
# Настройки глобального логгера: приложение логирует все уровни и использует консольный обработчик логов
handlers =java.util.logging.ConsoleHandler
.level = ALL
# Конфигурация файлового обработчика логов (вывод всех консольных логов от SEVERE и выше в файл log.txt в каталоге команды java)
java.util.logging.FileHandler.level = SEVERE
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.limit = 1000000
java.util.logging.FileHandler.pattern = log.txt
# Конфигурация консольного обработчика логов (вывод всех консольных логов)
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.pattern = log.log
java.util.logging.ConsoleHandler.formatter =java.util.logging.SimpleFormatter
```

Файл можно задать в строке запуска:

```
java -Djava.util.logging.config.file=../src/test/logtest/logger.conf -classpath ../bin
test.logtest.LoggerExperiment
```

Файл можно задать в статическом блоке точки входа, перед вызовом первого конструктора:

```
{
 try {
   LogManager.getLogManager().readConfiguration(LoggerExperiment.class.
     getResourceAsStream("logger.conf"));
 } catch(Exception ex) {
   ex.printStackTrace();
 }
}
```

Можно также задать конфиг программно перед вызовом первого конструктора:

```
StringBuilder sb = new StringBuilder();
   sb.append("handlers = java.util.logging.ConsoleHandler\n")
           .append(".level = ALL\n")
           .append("java.util.logging.ConsoleHandler.level = INFO\n");
   try (InputStream in = new ByteArrayInputStream(sb.toString().getBytes());) {
     LogManager.getLogManager().readConfiguration(in);
   } catch (Exception ex) {
     ex.printStackTrace();
   }
```

В JEE можно указать общеприкладной уровень логгирования для каждого конкретного логгера в настройках сервера.

Получить логгер можно, например, статическими вызовами класса Logger (они будут дергать LogManager):

```
Logger logger = Logger.getLogger(ExClass.class.getName());
```

Писать сообщения в лог можно многочисленными  методами Logger, при этом надо указывать уровень логгирования:

```
logger.log(Level.CONFIG,”Log message”);
```

```
try {
  ...
} catch (Exception ex) {
  logger.log(Level.SEVERE, null, ex);
}
```

```

```

Если лог затратный, то для экономии ресурса перед логгированием можно сверять текущий уровень логгирования с общеприкладным при помощи метода isLoggable(level): если общеприкладной уровень выше, то писать лог смысла нет:

```
Level level = Level.INFO;
if (logger.isLoggable(level)) {
  try (ByteArrayOutputStream out = new ByteArrayOutputStream()) {
    ...
    logger.log(level,out.toString("UTF-8"));
  }
}
```

## Перехват сетевого трафика. WireShark

WireShark – анализатор перехваченного трафика. Непосредственный перехват байтовых потоков осуществляют сетевые перехватчики вроде wincap для windows. Для захвата в windows7 локального трафика (localhost 127.0.0.1) надо установить сетевой перехватчик с поддержкой loop-интерфейса. На официальном сайте вместо стандартного wincap советуют установить npcap, работает стабильно версия npcap-0.07-r17. Устанавливать его надо в wincap – режиме, запуск стандартный для wincap:

```
root →
sc start npf
sc stop npf
```

После запуска под админом npcap можно запускать анализатор WireShark. Выбрать требуемый сетевой интерфейс. Далее Capture → Start и Capture → Stop.

**Анализ**

Отфильтровать пакеты можно при помощи Filter → xxx, где xxx – одно из поисковых выражений. Например:

- http – отсечка пакетов по протоколу http
- tcp.port==8080 and ip.addr==127.0.0.1 – отсечка пакетов к приложению, слушающему порт 8080 на локальном хосте
- tcp.port==1521 – отсечка пакетов, идущих на порт Oracle DB

Для склеивания пакетов сессии кликнуть ПКМ по строке одного из них в INFO → Follow TCP Stream.

Можно также сохранить перехваченный трафик в текстовый файл и просматривать его в редакторе, например просмоторщике Far (F3, поиск — F7 + пробел).

## Oracle в Windows.

### SQL+

Запуск cmd → sqlplus.

Для того, чтобы русские буквы нормально отображались в sqlplus надо поставить в cmd (ПКМ по заголовку окна, свойства) шрифт Lucida Console и выполнить команды

```
chcp 1251
sqlplus
```

Вход под администратором:

```
login = system as sysdba
passw = 12345
```

Смена пароля для пользователя xxx:

```
SQL>alter user xxx identified by 123;
```

Для вывода названий пользовательских таблиц можно использовать команду:

```
SQL>select * from tab;
```

Если информации очень много, можно вывести в файл командами:

```
SQL>spool c:\\user\temp.txt;
SQL>select ... ;
SQL>spool off;
```

Тестовая схема (user = scott, pass = tiger) – вход и выход:

```
SQL>CONNECT scott/tiger;
SQL>DISCONNECT scott/tiger;
```

*Rem: если при входе сыпется ошибка ORA-01034, то следует запустить БД: войти из под админа, выполнить команду startup;*

Вход без регистрации:

```
sqlplus /nolog
```

Подключение к удаленной базе. Информация о подключении имеет разные форматы для разных модификаций СУБД. Ее можно вносить в строке запуска sqlplus, в команде

CONNECT или прописать в конфигурационном файле tnsnames.ora клиента (c:\Oracle\db\12.2.0\dbhome1\network\admin\tnsnames.ora):

```
ExConnection =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = xxx.yyy.zzz)(PORT = 1521))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = xxx.yyy.zzz)
    )
  )
```

Если информация о соединении уже прописана в клиентском tnsnames.ora, то можно просто использовать ее алиас в строке соединения:

```
sqlplus user@exconnection
…
```

Или прописать алиас в команде CONNECT:

```
SQL>connection user@exconnection
…
```

**Создание схемы и БД**

В Oracle понятие схема данных обозначает совокупность объектов БД, которая привязывается к определенному пользователю.

Т.е. это описание БД на логическом уровне - таблицы, связи, процедуры и т.д. Т.о. для создания новой БД в Oracle надо определить пользователя и схему.

**Создание схемы в SQL+**

Входим из под администратора:

```
login = system as sysdba
passw = 12345
```

Для Oracle 12c надо либо вводить встраиваемый контейнер pdb (в системном cdb создавать обычных пользователей нельзя), либо использовать недокументированный скрипт (http://www.dba-oracle.com/t_ora_65096_create_user_12c_without_c_prefix.htm):

```
alter session set "_ORACLE_SCRIPT"=true;
```

Создаем пользователя (пользователь = jpa, пароль = 123, область хранения схемы = users (default), объем схемы = 20m, область временных данных = temp):

```
CREATE USER jpa IDENTIFIED BY 123
DEFAULT TABLESPACE users QUOTA 20M ON users
TEMPORARY TABLESPACE temp;
```

Наделяем пользователя jpa правом создавать сессию с сервером:

```
GRANT CREATE SESSION TO jpa;
```

Проверяем создание пользователя (имя пользователя должно быть написано БОЛЬШИМИ БУКВАМИ) JPA:

```
SELECT USERNAME, USER_ID, PASSWORD, ACCOUNT_STATUS, DEFAULT_TABLESPACE,
TEMPORARY_TABLESPACE, PROFILE
FROM DBA_USERS
WHERE USERNAME = 'JPA'
```

Разрешаем пользователю jpa создавать объекты:

```
GRANT CREATE TABLE TO jpa
GRANT CREATE PROCEDURE TO jpa
GRANT CREATE TRIGGER TO jpa
GRANT CREATE VIEW TO jpa
GRANT CREATE SEQUENCE TO jpa;
```

Разрешаем менять объекты пользователю jpa:

```
GRANT ALTER ANY TABLE TO jpa
GRANT ALTER ANY PROCEDURE TO jpa
GRANT ALTER ANY TRIGGER TO jpa
GRANT ALTER PROFILE TO jpa;
```

Разрешаем удалять объекты пользователю jpa:

```
GRANT DELETE ANY TABLE TO jpa
GRANT DROP ANY TABLE TO jpa
GRANT DROP ANY PROCEDURE TO jpa
GRANT DROP ANY TRIGGER TO jpa
GRANT DROP ANY VIEW TO jpa
GRANT DROP PROFILE TO jpa;
```

При необходимости профиль пользователя можно менять:

```
ALTER USER jpa IDENTIFIED BY new_password;
```

```
ALTER USER jpa QUOTA 50M ON USERS;
```

**Создание БД**

Подключаемся под пользователем/схемой:

```
CONNECT jpa/123;
```

Создаем таблицы:

```
CREATE TABLE foo
(
	id INTEGER PRIMARY KEY,
	name VARCHAR2(30),
	bar_id INTEGER
);
CREATE TABLE bar
(
	id INTEGER PRIMARY KEY,
	name VARCHAR2(30)
);
```

Создаем связь между таблицами, обозначая столбец bar_id таблицы foo как внешний ключ для таблицы bar:

```
ALTER TABLE foo ADD
FOREIGN KEY (bar_id) REFERENCES bar (id);
```

Вносим в таблицы тестовые записи:

```
INSERT INTO bar(id, name) VALUES(1, 'bar1');
INSERT INTO bar(id, name) VALUES(2, 'bar2');
INSERT INTO bar(id, name) VALUES(3, 'bar3');

INSERT INTO foo(id, name, bar_id) VALUES(1, 'foo1', 1);
INSERT INTO foo(id, name, bar_id) VALUES(2, 'foo2', 2);
INSERT INTO foo(id, name, bar_id) VALUES(3, 'foo3', 3);
```

При необходимости строки можно менять:

```
UPDATE bar SET name = 'xxx' WHERE id = 3;
DELETE FROM bar b WHERE bar.id = 44;
INSERT INTO foo (id, name, bar_id) VALUES (1, 'foo1', 1);
ALTER TABLE foo DROP COLUMN bar_id;
```

http://www.firststeps.ru/sql/oracle/
http://www.sql.ru/docs/sql/u_sql/ch15.shtml#15.8

**Oracle. Создание DataSource в Glassfish 4.x**

Нужно: имя РБД, логин, пароль, запущенную СУБД Oracle, Glassfish. Имя РБД совпадает с названием каталога, содержащего ее (задается при установке СУБД Oracle).

```
Имя РБД = orcl
Логин = jpa
Пароль = 123
```

Сначала создаем пул соединений Glassfish (DataSource). Лезем в web-консоль, там выбираем Resources → JDBC → JDBC Connection Pools.
Создаем новый пул: имя=xxx, тип = javax.sql.DataSource, вендор = Oracle
Далее все по-умолчанию → финиш

Заходим в созданный пул → вкладка Additional Properties → удаляем все свойства по-умолчанию. Создаем тут же (Add Property) 3 новых: URL, Password, User (найти их можно потом в \glassfish\domains\domain1\config\domain.xml):

```
URL = “jdbc:oracle:thin:@localhost:1521:orcl”
User = “jpa”
Password = “123”
```

Далее Save → вкладка General → ping → Success

Теперь создаем JNDI-имя DataSource. Лезем в  Resources → JDBC → JDBC Resources → New JDBC Resource → JNDI Name = yyy, Pool Name =xxx. Имя yyy рекомендуется выбирать по шаблону jdbc/zzzDS.

*Rem: Создание Data Source в Jdeveloper. Создать фабрику DataSource, настроить ее и зарегистрироват в JNDI можно при помощи инструментов JDeveloper.*
*Windows -> Data Base -> New Connetction*
*Create Connection in = xxx (имя приложения)*
*Connection Name = yyy (автоматически создаваемое JNDI ENC имя DataSource = yyyDS)*
*Connection Type = Oracle (jdbc)*
*User Name = hr*
*Password = PgsrCc*
*Driver = thin (default)*
*Host Name = localhost (default)*
*SID = orcl*
*JDBC port = 1521 (default)*
*Проверка -> Test Conntction = success!*

**Привязка DataSource к проекту NetBeans**

Выбрать JPA-проект, ПКМ → New → Persistence → Persistence Unit. Выбрать имя, JPA-

вендора, указать JNDI-имя DataSource в Glassfish. В ответ будет создан persistence.xml, если его еще нет и там появится элемент навроде

```
<persistence-unit name="ExPU" transaction-type="JTA">
   <jta-data-source>jdbc/exDS</jta-data-source>
   <exclude-unlisted-classes>false</exclude-unlisted-classes>
   <properties>
     <property name="javax.persistence.schema-generation.database.action" value="create"/>
   </properties>
 </persistence-unit>
```

Далее этот PU можно использовать для создания EM в EJB или JPA-клиентах.

```
@PersistenceContext(unitName="ExPU")
private EntityManager em;
```

**Тестирование БД в JPA-приложении Netbeans/Glassfish**

Перед работой с БД надо сначала настроить соединение, DataSource в Glassfish, подвязать последний к проекту.

Далее надо проверить состояние СУБД в sqlplus (под админом) — команды запуска/остановки СУБД:

```
startup;
shotdown;
```

После можно стартовать сервер Glassfish, развертывать приложение (deploy).

При проверке результатов тестирования в sqlplus следует помнить, что открытая во время тестирования сессия sqlplus не увидит результаты внешней транзакции. Поэтому следует перезапускать sqlplus перед просмотром текущего актуального состояния.

```
exit;
```

И наоборот, чтобы изменения через sqlplus были видны JPA, надо зафиксировать транзакцию. Команды фиксации/отката транзакции commit/rollback:

```
commit;
rollback;
```

После завершения теста надежнее сделать undeploy проекта, остановить Glassfish. Иначе могут посыпаться ошибки вроде «Attempting to execute an operation on a closed EntityManagerFactory».

# Postgres

## Установка в Linux (Ubuntu 12.04)

```
#apt-get update
#apt-get install postgresql postgresql-contrib
```

Будет создан UNIX-пользователь — учетка сервера postgres, по-умолчанию его имя postgres. Для проверки запускаем postgres-терминал psql в UNIX-терминале, в сессии пользователя postgres:

```
xxx$sudo -i -u postgres
postgres$psql
 postgres=#
```

## Установка в Windows 7

Качаем инсталлятор с офсайта во временный каталог:

```
https://www.enterprisedb.com/downloads/postgres-postgresql-downloads#windows
```

Запускаем инсталлятор, разрешив админские привилегии. В процессе установки:

- Выбрать каталог установки (на всякий случай не Program Files)
- Выбрать каталог кластера БД (область данных)
- Назначить единый пароль системной  учетки postgres и роли суперпользователя. Системная учетка postgres тут - отдельный Windows-пользователь (по-умолчанию с именем postgres), под которым postgres-сервер будет установлен и в чьем процессе затем этот postgres-сервер будет запускаться. Роль суперпользователя - внутренняя учетка postgres с полномочиями администрирования СУБД
- порт по-умолчанию 5432
- локаль, т.е. кодировка для всего кластера (шаблонов бд). По-умолчанию UTF-8

После установки, для запуска утилит postgres из Win-терминала (cmd), надо прописать в переменной среды path путь к каталогу bin сервера postgres:

```
...\postgres\9.6\bin
```

Для проверки запускаем postgres-терминал psql в Win-терминале (cmd) под ролью postgres с вводом ее пароля:

```
psql -U postgres
 postgres=#
```

### Старт/стоп сервера в Linux (Ubuntu 12.04)

Запуск, остановка, перезапуск postgres:

```
#service posgresql start
#service posgresql stop
#service posgresql restart
```

### Старт/стоп сервера в Windows 7

В windows postgres-сервер стартует как служба, узнать ее можно в списке служб, имя типа postgresql-x64-9.6. Вызываем win-терминал от имени админа (cmd, ПКМ → от имени админа). Запуск, остановка postgres:

```
net stop postgresql-x64-9.6
net start postgresql-x64-9.6
```

### Терминал psql

Терминал psql — интерактивная символьная оболочка для работы с сервером postgres. Запуск psql создает соединение с заданной базой данных под заданной ролью, в котором можно делать вещи, например, менять таблицы при помощи sql-выражений. Разрешение на такое соединение выдается службой аутентификации и авторизации postgres.

Простейший способ соединения по-умолчанию через psql – под ролью суперпользователя postgres:

```
psql -U postgres
```

Приглашение на ввод состоит из строки xxx=# (след. строки xxx-#) для роли суперпользователя xxx либо yyy=> (след. строки yyy->) для обычной роли yyy.

```
postgres=#
```

Метакоманды postgres дополняют или дублируют SQL-выражения, начинаются с символа '\'. Выход из терминала postgres:

```
postgres=#\q
```

Подсказка для команды xxx (выход из режима просмотра ролей Shift + Q, листать PgUp/PgDn):

```
postgres=#\h xxx
```

**Аутентификация и авторизация. Настройка pg_hba.conf**

Сервер postgres использует ролевую модель делегирования прав клиентам. Роли в postgres есть внутренние логические аналоги групп и пользователей ОС UNIX или Windows. Аутентификация заключается в опознавании внешнего клиента и привязывании его к определенной внутренней роли с подключением к определенной БД. Авторизация заключается в наделении роли различными привилегиями работы с СУБД. Под пользователем postgres обычно понимается роль с привилегией входа (подключения к бд).

При установке postgres по-умолчанию в UNIX/Windows создается одноименный UNIX/Windows-пользователь postgres, одноименная роль с привилегиями суперпользователя postgres и база данных postgres. Кроме того создается файл pg_hba.conf с политикой аутентификации по-умолчанию.

Для каждой роли назначается политика аутентификации, т.е. способ связывания с нею клиента. Политика настраивается в файле pg_hba.conf (postgres host-based autentification). Он содержит строки вида:

```
[подключение][бд][роль][адрес*][аутентификация]
```

- подключение - бывает local (локальный UNIX или Windows сокет) и host (сетевой сокет)
- бд - имя бд или all для всех бд
- роль - имя роли или группы ролей или all
- адрес (* только для host подключений) - диапазон допустимых IP-адресов сетевых клиентов вида AAA.BBB.CCC.DDD/NN, где NN маска от 0 до 32, указывающая число старших битов клиентского адреса для сравнения с шаблоном AAA.BBB.CCC.DDD
- аутентификация - метод проверки допуска клиента к нужной базе данных под требуемой ролью

Некоторые методы аутентификации:

- trust - метод полного доступа любых клиентов. По сути это вариант перекладывания проверки на ОС. Локальная внутрисистемная безопасность обеспечивается защитой файла локального сокета с запретом сетевых подключений через localhost. Внешняя безопасность обеспечивается запретом сетевых подключений не с localhost
- ident - допускает сетевых клиентов, одноименных с запрашиваемой ролью. Имя клиента устанавливает сервер ident
- peer - аналогичен ident, но для локальных клиентов. Имя клиента - имя пользователя ОС, использующего локальный сокет
- md5 - метод аутентификации пользователя по хэшу пароля. При этом клиент

посылает не свой пароль а его md5-хэш. Если подключаемой по md5 роли не назначено пароля, то в парольной базе ей присваивается null и вход по паролю не пройдет
- password - аналог md5, но без защитного хэширования

Настроечный файл pg_hba.conf можно найти при помощи sql-команды (из под суперпользовательской роли):

```
postgres=#SHOW hba_file;
```

В Windows файл pg_hba.conf м.б. в каталоге области данных. В Linux – по пути вроде /etc/postgresql/9.2/main/pg_hba.conf.

По-умолчанию в Windows установлена аутентификация через сетевой сокет под любой ролью с паролем:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
```

В Linux по-умолчанию доступна политика peer, позволяющая подключаться локально без пароля из сессии одноименного с ролью пользователя Linux, либо по сетевому сокету под любой ролью с паролем:

```
# TYPE DATABASE USER ADDRESS METHOD
local all postgres peer
# "local" is for Unix domain socket connections only
local all all peer
# IPv4 local connections:
host all all 127.0.0.1/32 md5
# IPv6 local connections:
host all all ::1/128 md5
```

Для того, чтобы в Linux можно было соединяться локально под произвольной ролью с паролем, надо добавить локальную политику md5 и убрать более приоритетные локальные политики peer:

```
#local all postgres peer
#local all all peer
local all all md5
```

После изменения файла pg_hba.conf надо перезапустить сервер postgres.

**Вход через psql**

Политика аутентификации по-умолчанию различна для Windows и Linux.

В Windows по-умолчанию можно соединиться с базой данных postgres только через сетевой сокет под админской ролью postgres, с тем паролем, что был задан при установке:

```
psql -U postgres -d postgres -h 127.0.0.1 -W
 postgres#>
```

- -U задает имя требуемой роли
- -d задает имя бд
- -h 127.0.0.1 указывает на локальное сетевое соединение, разрешенное по-умолчанию
- -W указывает на парольную политику (md5/password) аутентификации, разрешенную по-умолчанию

*Rem: если имя бд совпадает и именем роли, то его можно опустить:*
*psql -U xxx -h 127.0.0.1 -W*

*Rem: в Windows параметры -h 127.0.0.1 и -W можно опустить:*
*psql -U xxx -d yyy*

```
psql -U postgres
 postgres#>
```

В Linux по-умолчанию можно соединиться с базой данных postgres локально из сессии пользователя postgres:

```
xxx$sudo -i -u postgres
postgres$psql -U postgres -d postgres
 postgres#>
```

*Rem: в Linux по-умолчанию можно подключаться через UNIX-сокет под одноименным с ролью пользователем:*
*xxx$psql -d db_name*

```
postgres$psql
 postgres#>
```

После изменения политики аутентификации, добавления пользователей, наделения их паролями можно входить по-другому.

Пример соединения для политик md5 и password. Соединение с нужной бд db_name под

нужной ролью role_name из под любого пользователя ОС через сетевой и локальный сокеты:

```
psql -U role_name -d db_name -h 127.0.0.1 -W
 postgres=>
```

```
psql -U role_name -d db_name
 postgres=>
```

Если у требуемой роли role_name присутствует пароль и он совпадает с введенным, а также у нее есть право подключения к бд db_name, то соединение будет установлено. Если для роли не заводился пароль, то ей сопоставляется null-пароль и -W соединение не устанавливается.

Для удаленного подключения надо найти файл postgresql.conf. Можно командой (под суперпользовательской ролью):

```
postgres=#SHOW config_file;
```

В найденом файле (например /etc/postgresql/9.2/main/postgresql.conf) надо прописать прослушиваемый адрес:

```
listen_addresses = 'localhost, 192.168.0.1'
```

Добавить разрешение на вход для подсети в файл pg_hba.conf:

```
host    all    all    192.168.0.1/16    md5
```

И перегрузить postgres. Подключение с чужой машины выглядит примерно так:

```
psql -h 192.168.0.1 -d db_name  -U role_name
```

**Управление ролями**

Управлять ролями в psql можно под ролью с  привилегией суперпользователя.

Просмотр ролей:

```
postgres=#\du

 List of roles
 Role name | Attributes | Member of
 postgres  | Superuser, Create role, Create DB, Replication | {}
```

```
postgres=#SELECT rolname FROM pg_roles;
```

Создание роли sql-командой (создает роль с привилегией входа LOGIN):

```
postgres=#CREATE USER role_name;
```

Аналогично:

```
postgres=#CREATE ROLE role_name WITH xxx;
```

где xxx - привелегии вроде LOGIN. Полный список привелегий в хелпе по команде CREATE ROLE:

```
postgres=#\h CREATE ROLE
```

(Выход из режима просмотра ролей Shift + Q, листать PgUp/PgDn)

Можно создать роль xxx с привилегией LOGIN в терминале ОС из под учетки postgres при помощи скрипта createuser:

```
createuser role_name
  ... n ... n ... n
```

Удалить роль можно в терминале psql командой:

```
postgres=#DROP ROLE role_name;
```

Наделение роли привилегиями xxx в терминале psql:

```
postgres=#ALTER ROLE role_name WITH xxx;
```

Установить пароль для роли xxx можно так:

```
postgres=#\password xxx
```

```
postgres=#ALTER ROLE xxx WITH PASSWORD 'yyy';
```

или при создании сразу:

```
postgres=#CREATE ROLE xxx WITH PASSWORD 'yyy';
```

Можно передавать привилегии другим ролям. Пусть создана таблица tab под ролью zzz:

```
zzz=>CREATE TABLE tab (id serial, name varchar(20));
```

Передать права xxx на таблицу tab или базу данных db любой роли yyy можно командами:

```
zzz=>GRANT xxx ON tab TO yyy;
zzz=>GRANT xxx ON DATABASE db TO yyy;
```

Права xxx м.б. UPDATE, INSERT и т.д., ALL - все права. Вместо роли yyy можно задать PUBLIC, это даст права xxx на таблицу tab  всем ролям.

Просмотреть таблицы можно:

```
zzz=>\d
```

Просмотреть привелегии можно

```
zzz=>\z
```

Отнять права можно командой вида

```
zzz=>REVOKE xxx ON tab FROM yyy;
```

**Создание бд и подключение к бд**

База данных содержит схемы и таблицы уникально. Также бд является уникальной для клиентского соединения. Бд изолированы друг от друга.

Можно создавать бд в терминале ОС из под учетки postgres:

```
postgres$\createdb -O role_name db_name
```

Можно создавать бд xxx в psql из под роли zzz с привилегией создания бд, параметр OWNER указывает роль yyy (zzz должна быть членом yyy), которая будет владеть бд xxx:

```
zzz=>CREATE DATABASE xxx OWNER yyy;
```

*Rem: роль суперпользователя может создавать бд для любых чужих ролей*

Посмотреть список созданных бд можно командами:

```
zzz=>SELECT datname FROM pg_database;
```

```
zzz=>\l
```

*Rem: template0 и template1 – системные бд-шаблоны, которые копируются при создании всех новых бд*

Подключиться к бд yyy можно в UNIX-консоли из под аутентифицируемого postgres пользователя под ролью xxx, при наличии у нее привелегии входа в yyy, командой вроде

```
psql -U xxx -d yyy
```

Узнать текущую роль и способ подключения можно в psql  командой

```
postgres=>\conninfo
```

**Примеры работы со схемами/таблицами**

Получить список доступных роли zzz таблиц и их атрибутов можно командами:

```
zzz=>\d
zzz=>\z
```

Получить сведения о структуре таблицы xxx (доступной роли zzz) в psql можно командами:

```
zzz=>\d xxx
zzz=>\z xxx
```

Создать таблицы и связи между ними можно примерно так:

```
zzz=>CREATE TABLE foo (
  id integer PRIMARY KEY NOT NULL,
  name varchar(30),
  bar_id integer
);

zzz=>CREATE TABLE bar (
 id integer PRIMARY KEY NOT NULL,
 name varchar(30)
);

zzz=>ALTER TABLE foo
ADD CONSTRAINT fk_per_bar
FOREIGN KEY (bar_id)
REFERENCES bar (id);
```

```
zzz=>INSERT INTO bar (id,name) VALUES (1,'bar1');
zzz=>INSERT INTO foo (id,name,bar_id) VALUES (1,`foo1`,1);

SELECT foo.name, bar.name FROM foo, bar WHERE foo.id=bar.id;
```

Типы данных как у всех, но нет autoincrement, вместо этого используется тип serial.

**Работа с датами**

Полная дата/время записывается временной меткой – тип timestamp. Существует она в варианте с временной зоной (тип timestamp with time zone) и без оной (тип timestamp).

Локальное человечье представление временной метки описывается типом timestamp и по-умолчанию имеет в postgres примерно такой вид:

```
yyyy-MM-dd HH:mm:ss
1988-05-02 10:22:11
```

К локальному времени можно добавить временную зону, такое расширенное представление называется абсолютным (тип timestamp with time zone).

```
yyyy-MM-dd HH:mm:ssX
1988-05-02 10:22:11+03
1988-05-02 07:22:11+00
```

В абсолютном представлении даты +03 определяет временную зону, т.е. означает сколько часов надо прибавить к абсолютному несмещенному (+00 или UTC) представлению этой же даты, чтобы получить текущую локальную часть. Существуют национальные часовые пояса, которые зависят от географии и внутренних законов вроде 'Europe/Moscows'. Есть унифицированный пояс UTC (+00), представление по Гринвичу (GMT +3).

По-умолчанию абсолютным датам присваивается текущая временная зона сервера postgres. Через psql можно установить и посмотреть текущую временную зону сервера postgres командами вроде:

```
zzz=>SET TIME ZONE 'Europe/Moscow';
zzz=>SET TIME ZONE 'UTC';
SHOW TIME ZONE;
```

При занесении в таблицу человечьих представлений дат, postgres вычисляет по ним количество миллисекунд с 01.01.1970. При этому для абсолютного представления учитывается смещение, определяемую этим часовым поясом, а локальное представление считается эквивалентным абсолютному в зоне UTC со смещением 0.

Пример.

```
zzz=>SET TIME ZONE 'GMT+5';
zzz=>CREATE TABLE test (id integer, date timestamp, date1 timestamp with time zone);
zzz=>INSERT INTO test (id, date, date1) VALUES (1, '1975-11-22 22:45:55', '1975-11-22 22:45:55');
zzz=>INSERT INTO test (id, date, date1) VALUES (1, '1980-05-06 09:30:00+06', '1980-05-06
09:30:00+06');
```

Выставляем текущий пояс GMT +5 (-5 часов к метке в UTC), создаем таблицу со столбцами date – локальным временем и date1 – с абсолютным временем.

Вносим запись с id=1 и одинаковыми локальными датами: временная метка для поля date будет посчитана в зоне UTC (+0), а временная метка для date1 будет посчитана в текущей зоне со смещением на -5 часов относительно date.

Вносим запись с id=2 и одинаковыми абсолютными датами в зоне GMT -6: для date эта зона будет проигнорирована, т.е. метка будет считаться по-прежнему в UTC, а для получения метки date1 эта зона перекроет текущую зону postgres, т.е. даст метку +6 часов относительно метки date.

При выводе таблиц postgres проводит обратное преобразование временной метки к локальному или абсолютному человечьему представлению. Для локального представления временная зона не учитывается (зона всегда считается UTC), а для абсолютного используется текущая временная зона сервера postgres.

Смотрим что получилось в текущей зоне GMT +5:

```
zzz=>SELECT * FROM test;
```

```
id | date | date1
----------------------------------------------------------------
  1 | 1975-11-22 22:45:55  |  1975-11-22 22:45:55-05
  2 |  1980-05-06 09:30:00 |  1980-05-06 10:30:00-05
```

В записи id=1 локальная часть date1 совпадает с date, т.к. временная зона (-05) не изменилась с момента записи. В записи id=2 локальная часть date1> date на +1 час т.к. date1 записывалась в зоне со смещением на +6 часов к UTC, а выводится в зоне со смещением -5 часов к UTC.

Выставляем текущую зону в UTC (+0) и смотрим на изменение локальных представлений в столбце date1 (прибавилось +5 часов):

```
zzz=>SET TIME ZONE 'UTC';
zzz=>SELECT * FROM test;
```

```
id | date | date1
-----------------------------------------------------------------
  1 | 1975-11-22 22:45:55  |  1975-11-23 03:45:55+00
  2 |  1980-05-06 09:30:00 |  1980-05-06 15:30:00+00
```

**Хранимые функции**

В postgresql нет полноценных хранимых процедур. Вместо этого присутствуют пользовательские функции. Они не могут возвращать > 1 набора данных, не могут запускать подтранзакции (за исключением блоков исключения). Кроме того, запросы внутри потенциально более медленные, т.к. скомпилированный код должен быть универсальным и не учитывает конкретику запроса при оптимизации. Тем не менее функции могут использоваться для выполнения типовых задач, для уменьшения дерганья РБД, снижения сетевого трафика.

Пример функции, вставляющей запись в таблицу:

```
zzz=>CREATE OR UPDATE FUNCTION  bar_insert (id_ integer, name_ varchar(30))
zzz->RETURNS void AS $$
zzz->BEGIN
zzz->INSERT INTO BAR (id,name) VALUES (id_,name_);
zzz->END
zzz->$$ LANGUAGE plpgsql;
```

```
zzz=>PERFORM bar_insert(777,'bar777');
```

Пример функции, дающей число записей таблицы с заданным именем name_:

```
zzz=>CREATE FUNCTION  bar_count (name_ varchar(30) )
zzz->RETURNS integer AS $$
zzz->DECLARE result integer;
zzz->BEGIN
zzz->SELECT COUNT(b) INTO result FROM BAR b WHERE b.name=name_;
zzz->RETURN result;
zzz->END;
zzz->$$ LANGUAGE plpgsql;
```

```
zzz=>SELECT bar_count();
```

Пример функции, дающей все записи таблицы:

```
zzz=>CREATE FUNCTION  bar_all ()
zzz->RETURNS TABLE (id integer, name varchar(30)) AS $$
zzz->BEGIN
zzz->RETURN QUERY (SELECT * FROM BAR);
zzz->END;
```

```
zzz->$$ LANGUAGE plpgsql;
```

```
zzz=>SELECT bar_all();
```

Для удаление функции нужно имя и сигнатура:

```
zzz=>DROP FUNCTION bar_count(varchar(30));
```

**Создание DataSource в GlassFish. PersistenceUnit в NetBeans**

1. Скачать драйвер jdbc с официального сайта https://jdbc.postgresql.org
Называется он примерно так: xxx.jdbc4.jar

2. Положить в [glassfish_home]/glassfish/domains/domain1/lib/

*Rem: какой используется домен нужно проверять в настройках запущенного сервера, обычно это domain1, но все же.*

3. На всякий случай рестартануть GlassFish

4. Зайти в web-морду GlasFish (localhost:4848), там в Resources/JDBC/JDBC Connection Pools, создать новый пул соединений:

```
name = xxx
type = javax.sql.ConnectionPoolDataSource
vendor = postgresql
```

Далее next -> шаг 2, заполнить свойства:

```
Datasorce classname = org.postgresql.ds.PGConnectionPoolDataSource
DatabaseName = yyy
Password = zzz
PottNumber = 5432
ServerName = 127.0.0.1
User = aaa
```

Далее finish, зайти в созданный Connection Pool и проверить его ping (success - норма).

Теперь создаем JNDI-имя для созданного DataSource. Лезем в Resources/JDBC/JDBC Resources/New JDBC Resource

```
JNDI Name =  jdbc/zzzDS
Pool Name =xxx
```

JNDI-имя рекомендуется выбирать по шаблону jdbc/zzzDS (без jdbc/ может не

сработать).

5. Создать PU в JPA-приложении можно в дескрипторе persistence.xml, в рамках JEE-сервера, примерно так:

```
<persistence-unit name="uuu" transaction-type="JTA">
  <jta-data-source>jdbc/zzzDS</jta-data-source>
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <properties>
  <property name="javax.persistence.schema-generation.database.action" value="create"/>
  </properties>
</persistence-unit>
```

В NetBeans можно воспользоваться мастером: new → persistence → PersistenceUnit → ...

## SQL

Заменить значение поля на xxx в записи с id=yyy:

```
UPDATE bar SET col_name = xxx WHERE id = yyy;
```

Грохнуть таблицу и каскадно FK, которые на нее ссылаются:

```
DROP tableBAR CASCADE CONSTRAINTS;
```

Грохнуть запись id=xxx:

```
DELETE FROM bar b WHERE b.id=xxx;
```

Добавить в конец таблицы xxx столбец yyy и грохнуть его:

```
ALTER TABLE xxx ADD yyy varchar(30) ;
ALTER TABLE xxx DROP COLUMN yyy;
```

Добавить в таблицу xxx столбец yyy на нужную позицию - создать новую таблицу zzz нужной структуры, скопировать туда столбцы таблицы xxx в порядке перечисления, грохнуть xxx каскадно, переименовать zzz в xxx, при необходимости восстановить связи с другими таблицами:

```
CREATE TABLE zzz (id integer NOT NULL PRIMARY KEY, zzz varchar(30), col integer);
INSERT INTO zzz (id, col) SELECT id, col FROM xxx;
DROP TABLE xxx CASCADE;
ALTER TABLE zzz RENAME TO xxx;
```

При каскадном удалении таблицы уничтожаются ее связи. Для восстановления этих

связей можно использовать примерно следующие выражения (восстановление связи xxx с соединительной таблицей xxx_yyy):

```
ALTER TABLE xxx_yyy
ADD CONSTRAINT fk_xxx
FOREIGN KEY (xxx_id)
REFERENCES xxx (id);
```

**Виды соединений JOIN**

Пусть есть 2 таблицы L = {id, name, R_id} и R = {id, name}, соединяемые по соответствующим столбцам R_id и id в таблицу LR.

- INNER JOIN (или просто JOIN) – соединение по пересечению соединительных столбцов, т.е. соединяет те строки, в которых строго R_id = id
- LEFT OUTER JOIN (или просто LEFT JOIN) – соединение по левому соединительному столбцу, т.е. соединяет строки, в которых R_id = id и добавляет строки L0, если равного id в R не нашлось.
- RIGHT OUTER JOIN (или просто RIGHT JOIN) – соединение по правому соединительному столбцу, т.е. соединяет строки, в которых R_id = id и добавляет строки вида 0R, если равного R_id не нашлось.
- FULL JOIN возвращает объединение по соединительным столбцам, т.е. соединяет строки с R_id = id и добавляет L0, 0R, если равных значений не нашлось.
- CROSS JOIN возвращает декартово произведение таблиц целиком, т.е. соединяет каждую строку L с каждой строкой R

```
Связанные по R_id/id таблицы

L = <id, name, R_id> = {[11, L1, 1], [21, L2, 1], [3, L3, 55]}
R = <id, name> = {[1, R1], [2, R2], [3, R3]}

SELECT L.id, L.name, R.id, R.name FROM L JOIN R ON L.R_id = R.id;
LR  = {[11,L1,1,R1], [21,L2,1,R1]}

SELECT L.id, L.name, R.id, R.name FROM L LEFT JOIN R ON L.R_id = R.id;
LR  = {[11,L1,1,R1], [21,L2,1,R1], [3,L3,null,null]}

SELECT L.id, L.name, R.id, R.name FROM L RIGHT JOIN R ON L.R_id = R.id;
LR = {[11,L1,1,R1], [21,L2,1,R1], [null,null,2,R2], [null,null,3,R3]}

SELECT L.id, L.name, R.id, R.name FROM L FULL JOIN R ON L.R_id = R.id;
LR = {[11,L1,1,R1], [21,L2,1,R1], [3,L3,null,null], [null,null,2,R2], [null,null,3,R3]}
```

Допускается множественное соединение:

```
SELECT f.name, b.name FROM foo f JOIN foo_bar fb ON f.id=fb.foo_id JOIN bar b ON
b.id=fb.bar_id;
```

**Фрукты. DISTINCT. JOIN. ORDER BY. GROUP BY**

Выделить из таблицы фруктов уникальные пары id с одинаковыми name и отсортировать их.

```
CREATE TABLE fruits (id int NOT NULL PRIMARY KEY, name varchar(64));
```

```
INSERT INTO fruits (id,name) VALUES (1,'apple'), (2,'apple'), (3,'pear'), (4,'apple'), (5,'orange'), (6,'pear');
```

```
id | name
----------
  1 |  apple
  2 |  apple
  3 |  pear
  4 |  apple
  5 |  orange
  6 |  pear
```

Вариант 1 (подавление дублей DISTINCT):

```
SELECT DISTINCT LEAST(f1.id,f2.id) id1, GREATEST(f1.id,f2.id) id2 FROM fruits f1 JOIN fruits f2 ON f1.name=f2.name WHERE f1.id<>f2.id ORDER BY LEAST(f1.id,f2.id), GREATEST(f1.id,f2.id);
```

Вариант 2 (подавление дублей группировкой GROUP BY):

```
SELECT  LEAST(f1.id,f2.id) id1, GREATEST(f1.id,f2.id) id2 FROM fruits f1 JOIN fruits f2 ON f1.name=f2.name WHERE f1.id<>f2.id GROUP BY LEAST(f1.id,f2.id), GREATEST(f1.id,f2.id) ORDER BY LEAST(f1.id,f2.id), GREATEST(f1.id,f2.id);
```

```
id1 | id2
----------
  1 |  2
  1 |  4
  2 |  4
  3 |  6
```

## Регулярные выражения в Java. RegEx

Regular Expression – формальный язык для работы с подстроками текста (on-line движок https://regex101.com/). В его синтаксисе используются следующие символы:

| | |
|---|---|
| \ | любой языковый символ преобразуется в обычный, а любой обычный — в спецсимвол<br>ex.: \. = точка; \s = пробел |
| **\w** | буквенно-цифровой символ или "_" |

| | |
|---|---|
| **\W** | не \w |
| **\d** | цифровой символ |
| **\D** | не \d |
| **\s** | любой "пробельный" символ (по умолчанию - [ \t\n\r\f]) |
| **\S** | не \s |
| **^** | искать в начале строки<br>ex.: ^A: an A → null; ^A: An A → A |
| **$** | искать  в конце строки<br>ex: $A: eat A → A; $A: A eat → null |
| **[xyz]** | находит в тексте символ, попадающий во множество x,y,z. Допускаются диапазоны [a-d]<br>ex: [xX]: aX → X; [a-c][def]: obed → be |
| **[^xyz]** | находит в тексте символ, не попадающий во множество x,y,z. Допускаются диапазоны<br>ex: [^xX]: aX → a; [^a-c][^def]: obed → ob |
| ***** | повтор предшествующего символа >= 0 раз<br>ex: bo*: booo → booo; o*: bar → b |
| + | повтор предшествующего символа >= 1 раз<br>ex: bo+: booo → booo; o+: bar → null |
| **.** | любой непустой символ кроме \n \r<br>ex: .o: booo → bo; .b: booo → null |
| **?** | предшествующий символ 0 или 1 раз<br>ex: e?le?: angel → el; e?le?: angle → le |
| **+?**<br>***? ??** | нежадный (минимальная длина) поиск по соответствующим кванторам<br>ex: '.+?': 'foo' and 'bar' → 'foo', 'bar'; жадный - '.+': 'foo' and 'bar' → 'foo' and 'bar' |
| **(x)** | группировка с запоминанием найденного, $0 – вся строка, $1, …, $n – группы/скобки<br>ex: (foo): foo bar → foo, $1 = foo; (?.lob): clob and blob → $1 = clob, $2 = blob |
| **(?:x)** | группировка без запоминания<br>ex: (?:foo).+(bar): foo and bar → $0 = foo and bar, $1 = bar |
| **x\|y** | x или y<br>ex: x\|y : xor yet → x, y ) |
| **{n}** | n > 0 повторов предшествующего символа<br>ex:  a{2}: can → null; a{2}b: saab → aab |
| **{n,m}** | предшествующий символ повторяется от n до m раз, m>n>0<br>ex: a{2,3}: aa aaaaaaaaaaaaa →  aa, aaa |
| **{n,}** | эквивалентен {n,бесконечность} |

В Java существует разные формы работы: find – ищет подстроку; match – разбирает строку на подстроки по шаблону, при этом для каждой найденной подстроки могут храниться ее группы; split – разбивает строку на подстроки по регулярному выражению разделителя.

java.util.regex.Pattern – класс для задания шаблонов регулярных выражений
java.util.Matcher – разбор строки по заданному шаблону

java.util.Scanner – проход строки с получением результатов разных типов

*Rem: в Java '\' - спецсимвол, поэтому в RegEx используются \\.*

### Примеры RegEx

Типичные ошибки - всегда положительные шаблоны:

```
(.*) или \d* или [\|]*
```

Правильные шаблоны:

```
\d+(.*) или ([\|]*)\|
```

Удаление пустых строк в текстовом редакторе заменой:

```
найти:      \n\s*(\n)
заменить: $1
```

### Схема работы Matcher

```
Pattern p = Pattern.compile("regexStr");
Matcher m = p.matcher("sourceStr");
```

Разбор строки дает множественные подстроки, соответствующие шаблону, в каждой такой частной подстроке могут храниться ее группы.

```
Ex: найти расширения вида .xxx

String str = "start.bat start.exe";
Pattern p = Pattern.compile("\\.([^\\.]+)");
Matcher m = p.matcher(str);
while(m.find()){
  System.out.println(m.group(1));
}
```

Группа (скобка) тут одна, подстрок будет две. На выходе получим bat и exe.

## Hadoop

Hadoop - распределенный фреймворк для суперэнтерпрайзных приложений.

HDFS - hadoop распределенная файловая система. Это виртуальная фс поверх реальных, расположенных на разных носителях. Оперирует крупными блоками данных, хранящимся локально, поддерживает их сквозную индексацию. Поддерживает

дублирование данных для надежности и лучшего распараллеливания работы с ними. Есть сервер имен и сервера данных. Каждый файл в HDFS хранит путь, список его блоков и их копий (реплик). Сами файлы образуют юниксообразное дерево, команды HDFS похожи на posix.

MapReduce - парадигма распараллеливания вычислений. В основе лежит идея таскать не большие данные к маленькому коду, а код к исходным данным для локального выполнения и пересылке небольшого его результата. Это дает выигрыш в уменьшении задержек при передачах, надежности и скорости расчетов за счет кэширования промежуточных результатов и дублирования их на критических участках. Каждая задача выполняется в 2 фазы. На фазе Map выполнение распараллеливается на операции, которые выполняются локально на узлах, содержащих нужные блоки данных при помощи разосланного им кода обработки. На фазе Reduce проводится агрегирование локальных результатов. Может существовать промежуточная фаза Combine, на которой проводится некоторая локальная доп. обработка кэшируемого результата более общей локальной операции.

Hadoop есть реализация MapReduce на HDFS. Hadoop-кластер состоит обычно из мастера MapReduce (JobTracker), сервера имен, кучи рабочих узлов, на каждом из которых крутится сервер данных и обработчик TaskTracker.

Spark. Техника MapReduce создавалась для решения пула сходных изначально распараллеленых задач, кэширование их общих промежуточных результатов давало выигрыш в скорости и надежности. Но при декомпозиции сложной задачи на последовательность задач MapReduce избыточное кэширование становится тормозом. Поэтому применяется техника ленивых операций над RDD (resilient distributed dataset, гибко-распределенным массивом данных). Суть ее в том, что цепочка задач Map и Reduce выполняется не последовательно, а анализируется и распараллеливается целиком. Это позволяет сократить объем промежуточных результатов и их кэширования. В качестве примера можно рассмотреть подсчет денег по Лебегу и по Риману. По Лебегу надо разложить монетки по номиналу, потом посчитать количество монет в каждой кучке и умножить на номинал, результаты сложить. На каждом узле будет проводиться раскладывание на кучки (выборка, Map), получение сумм каждой кучки (промежуточное агрегирование, Combine), затем сложение значений кучек каждого номинала и их окончательное суммирование (конечное агрегирование, Reduce). MapReduce на каждом узле будет создавать кучки и их суммы, кэшировать все это. По Риману надо брать монетку, смотреть ее номинал и добавлять в общую сумму. На узлах достаточно кэшировать только сумму монеток блока данных.

Распараллеливание реляционной структуры данных. Берем всю реляционную бд и строим из нее одну супертаблицу при помощи последовательных прямых произведений связанных табличек. Получаем массив допустимых записей, содержащий все возможные результаты запросов на выборку к бд. SQL-запрос надо распарсить и извлечь требуемые ключи столбцов, т.е. для каждого поискового столбца определяется множество требуемых значений (ключей). Каждое такое множество ключей столбца

задает подмножество записей супертаблицы. Задача поиска сводится к нахождению пересечения этих подмножеств. В булевой алгебре это эквивалентно конъюнкции условий попадания каждого поискового подключа записи в требуемое множество. Поиск записей по композитному ключу является распараллеливаемой задачей и выполняется на фазе Map.  На деле SQL не используется, просто меняется язык запросов. Минусом данного подхода является требование статичности данных бд, т.к. издержки на создание и модификацию супертаблицы могут быть велики. Поэтому кэширующая супертаблица обновляется относительно редко и используется лишь для общих поисковых запросов, дающих предположительный результат. А для конкретных точных расчетов используется точная реляционная структура. На реляционном уровне можно организовать некое hadoop-образное подобие путем разнесения таблиц по отдельным экземплярам субд, шардирования таблиц. Тогда каждый узел будет параллельно формировать подмножество по ключам определенных столбцов. Но sql-запросы  надо будет парсить каждый раз.