Оглавление

Английский язык

Английский язык

GIT

При разработке обычно используется удаленный сервер (как резервная копия). Если над проектом занимается команда (не один человек), то могут возникнуть проблемы. Например, как загружать изменения на удаленный сервер. В этой ситуации поможет GIT.

GIT берет на себя слияние разных версий файлов (merging) и является системой контроля версий (история изменений с возможностью вернуться любому моменту).

В системе контроля версий существует два подхода к хранению данных: централизированный и распределенный. При централизированном подходе проект храниться только на центральном сервере. При распределенном подходе проект храниться на центральном сервере плюс у каждого разработчика есть копия проекта. Второй подход имеет преимущества, т. к. разрабатывать можно офлайн и, если что-то случиться с центральным сервером, то в этом случае копия проекта останется у разработчиков. Git является распределенной системой.

В отличии от других систем контроль версий, которые хранят список изменений,

GIT хранит изменения снимков проекта во времени.

Статусы файлов

- untracked (неотслеживаемый) — файл создан;
- modified (измененный) — файл изменен;
- staged (подготовленный) — `git add`;
- committed (зафиксированный) — `git commit`.

Указатели

В GIT есть указатель HEAD. Обычно он указывает на последний (текущий) коммит. Этот указатель можно смещать: `HEAD^` или `HEAD~1` (1 коммит), `HEAD^^` или `HEAD~2` (2 коммита) и т. д. Помимо указателей для возврата проекта можно воспользоваться хэшом коммита.

Удаленный репозиторий

Удаленный репозиторий необходим для резервной копии проекта и для того, чтобы другие люди могли видеть наши коммиты.

Популярные удаленные репозитории: GitHub, BitBucket, GitLab. Они предоставляют всю инфраструктуру для хранения и управления GIT-репозиториев.

У одного проекта может существовать несколько удаленных репозиториев с разными именами и адресами (git remote).

SSH

SSH (от анг. «Secure Shell» — «безопасная оболочка») — сетевой протокол, позволяющий производить удаленное управление операционной системой. SSH позволяет безопасно передавать данные в незащищенной среде.

В простом представлении работу SSH можно представить наличием приватных ключей у клиента (локального компьютера) и сервера.

Ссылки для настройки SSH-ключа для github.com:

- проверка наличия SSH-ключа;
- генерация SSH-ключа;
- связка SSH-ключа.

Ветвление

- новые функции разрабатываются в отдельных ветках;
- ветка master содержит стабильную версию проекта, можем вернуться на master в любой момент;
- сразу несколько разработчиков могут работать в своих ветках над своими

задачами, после завершения работы над задачами эти ветки «сливаются» в ветку master.

Слияние веток (merge)

При окончании работы во время слияния текущей ветки в основную с момента ответвления от основной ветки могут быть следующие ситуации:

- Fast-Forward
  - изменений в основной ветке не было,
  - конфликты не могут возникнуть,
  - слияние происходит автоматически,
  - коммит не создается;
- No-Fast-Forward
  - в основную ветку был добавлен коммит,
  - могут возникнуть конфликты,
  - слияние будет происходить автоматически или вручную,
  - создается коммит.

Конфликт происходит при изменении одного и того же файла в разных ветках.
Он решается вручную, т. к. GIT не может самостоятельно слить ветки.

rebase

rebase — альтернатива merge:

- обе команды делают одно и то же — сливают ветки;
- команда merge может создавать merge commit при слиянии (в случае не fast-forward), команда rebase merge commit'а не создает;
- команда merge безопасней, чем rebase — есть отдельный commit, отображающий слияние;
- плюс merge — достоверная полная история commit'ов;
- плюс rebase — лаконичная линейная история без лишних коммитов;
- если в ветке долго велась работа и произошло много изменений лучше использовать merge;
- если ветка была недолгая и произошло мало изменений — можно использовать rebase;
- используйте merge, если вас не просят о rebase.

Команда rebase работает так, будто мы только сделали git pull и сразу добавили в нее изменения. Можно сказать, что новая ветка «перебазировалась» на последний коммит. Или в новую ветку был добавлен

последний коммит из master, а затем, поверх него были добавлены коммиты текущей ветки. Теперь можно делать fast-forward слияние без merge commit'a.

После совершения данной команды коммиты текущей ветки помещаются во временную зону, далее в текущую ветку добавляются все коммиты из ветки master, позже поочередно добавляются все коммиты из временной зоны.

Также можно сделать все наоборот. Можно перейти в ветку мастер и совершить текущую команду из нее. Таким образом сначала добавятся коммиты из новой ветки, а затем новый коммит из мастера.

Разрешение конфликта такое же, как в случае с merge.

Интерактивный rebase

- обычный rebase нужен для манипуляций с ветками, интерактивный rebase работает на одной ветке;
- обычный rebase берет коммиты из другой ветки, перемещает их в нашу ветку и поверх этих коммитов по одному применяет коммиты из временной зоны;
- интерактивный rebase не берет коммиты из другой ветки, он помещает некоторые коммиты из текущей ветки во временную

зону и потом применяет эти коммиты опять к текущей ветке (в момент применения мы можем изменить коммиты);

- несмотря на то, что название команд одинаковое, обычный rebase сильно отличается от интерактивного rebase (разная логика).

Интерактивный rebase работает с коммитами, которые идут после того коммита, который вы указали.

Что можно делать с помощью интерактивного rebase:

- поменять коммиты местами;
- поменять название коммита(ов);
- объединить два коммита в один;
- добавить изменения в существующий коммит;
- разделить коммит на несколько коммитов;
- ...

Команды

Информационные команды:

- `git help` (помощь, документация);
- `git help название_команды` (документация конкретной команды).

Конфигурация:

- `git config --global user.name "имя фамилия"`;
- `git config --global user.email "email"`;
- `git config --global color.ui true`.

Создание нового проекта:

- `mkdir название_проекта` (создание каталога);
- `cd название_проекта` (перейти к данному каталогу);
- `git init` (инициализация репозитория git).

Базовые команды:

- `git status` (узнать текущий статус репозитория);
- `git add` (подготовить файлы к коммиту):
  - `git add .` (все файлы в текущей папке),
  - `git add *.java` (все файлы в текущей папке с расширением .java),
  - `git add someDir/*.java` (все файлы в папке someDir с расширением .java),
  - `git add someDir/` (все файлы в папке someDir),
  - `git add "*.java"` (все файлы в проекте с расширением .java);

- git commit (сделать коммит):
  - git commit -m "сообщение",
  - git commit -a -m "сообщение" (в отличии от примера выше такая вариация позволяет не использовать команду git add),
  - git commit --amend -m "сообщение" (дополняет последний коммит, добавляя в него «свежие» изменения, меняет сообщение последнего коммита, новый коммит не создается);
- git log (история коммитов).

Другие команды:

- git diff (разница между статусами untracked и committed),
  - git diff --staged (staged и committed),
  - git diff COMMIT_ID (текущим состоянием репозитория и указанным коммитом);
- git reset (git reset --mixed HEAD) (отмена изменений, откат к комиту),
  - git reset --hard (git reset --hard HEAD) (возвращает проект к указанному коммиту, при этом полностью удаляет все коммиты после указанного безвозвратно, файлы в статусе untracked остаются без изменений),

- `git reset --mixed` (`git reset --mixed HEAD`) (возвращает проект к указанному коммиту, при этом переводит все коммиты после указанного в неотслеживаемую (unstaged) зону),
- `git reset --soft` (`git reset --soft HEAD`) (возвращает проект к указанному коммиту, при этом переводит все коммиты после указанного в отслеживаемую (staged) зону),
- `git reset HEAD~2`,
- `git reset --soft HEAD^^`,
- `git reset --hard хеш_коммита`,
- файлы в статусе untracked нельзя удалить командой `git reset --hard`, но их можно перевести в любой другой статус, а затем удалить при помощи текущей команды, или воспользоваться следующей командой;

- `git clean` (удаление untracked файлов):
  - `git clean -n` (посмотреть какие файлы будут удалены),
  - `git clean -f` (удалить untracked файлы);

- `git checkout` (перемещения между коммитами, версиями отдельных файлов и ветками):
  - `git checkout хеш_коммита/указатель` (между коммитами, совершать изменения и делать новые коммиты нельзя),
  - `git checkout название_текущей_ветки` (переход обратно к актуальному коммиту);
  - `git checkout хеш_коммита/указатель -- путь_до_файлов` (между версиями файлов в разных коммитах),
  - `git checkout HEAD~3 -- .` (все файлы),
  - `git checkout HEAD^ -- file1 file2` (для файлов file1 и file2),
  - `git checkout -- путь_до_файлов` (`git checkout HEAD -- путь_до_файлов`);
  - `git checkout название_ветки` (между ветками),
  - две черты указывают, что после них идет обычный текст (в нашем случае путь до файла), а не команда или параметр для команды (например, когда наименование ветки совпадет со значением коммита);

- git remote (настройка и просмотр удаленных репозиториев, на компьютере хранится только ссылка на удаленный репозиторий, origin — название этой ссылки):
  - git remote -v (просмотр списка существующих удаленных репозиториев),
  - git remote add название_репозитория URL_репозитория (добавить новый удаленный репозиторий, на компьютере к удаленному репозиторию мы будем обращаться по его названию),
  - git remote remove название_репозитория (удалить репозиторий (ссылку на него)),
  - git remote show название_репозитория (позволяет сравнить актуальность веток локальных и удаленных);
- git push (отправка локального репозитория на удаленный),
  - git push название_репозитория ветка;
  - git push origin master (отправка удаленный репозиторий с именем origin ветку master).
  - git push --delete origin master (удаление ветки в удаленном репозитории);

- git pull (git fetch, git merge) (получения обновлений (новые коммиты) с удаленного репозитория);
  - git pull origin (получить все удаленные ветки и обновления в них),
  - git pull origin название_ветки (получить обновления по определенной ветке).
- git clone URL_репозитория (загрузить репозиторий);
- git branch (работа с ветками),
  - git branch (список веток и ветка с указателем HEAD),
  - git branch -r (список удаленных веток),
  - git branch название_ветки (создание новой ветки),
  - git branch -d название_ветки (удаление ветки),
  - git branch -D название_ветки (удаление ветки, даже если в ней был коммит);
- git merge ветка (слияние текущую и указанную веток);
- git rebase ветка (слияние текущую и указанную веток, разница команд описана выше),

- `git rebase --continue` (принять команду после исправления вручную конфликтов),
- `git rebase --skip` (пропустить коммит, который вызывает конфликт слияния),
- `git rebase --abort` (прекратить слияние),
- `git rebase -i HEAD~3` (интерактивный rebase);

- `git cherry-pick` («взять» коммиты из другой ветки),
  - `git cherry-pick коммит` («взять» один или несколько коммитов из другой ветки, у коммита будет другой хэш),
  - `git cherry-pick --edit коммит` (перенести коммит из другой ветки, но при этом хотим поменять сообщение коммита),
  - `git cherry-pick --no-commit коммит` (хотим перенести изменения из коммита из другой ветки, но при этом не хотим делать коммит в нашей ветке (изменения просто попадут в отслеживаемую зону); это бывает полезно, если мы хотим внести небольшие правки в тот коммит, который мы забираем из другой ветки или слияние двух коммитов из другой ветки в один коммит),

- `git cherry-pick -x коммит` (указывает в сообщении коммита хэш того коммита, из которого мы сделали cherry-pick),
- `git cherry-pick --signoff коммит` (указывает в сообщении коммита имя того пользователя, кто совершил cherry-pick).

Источники

- Курс на Udemy «Git: Полный курс для начинающих и не только», Наиль Алишев, 03.2019.

Операционная система

Linux (bash)

Windows (bat)

Паттерны и алгоритмы

Паттерны

Адаптер

Совмещает несовместимые интерфейсы.

Есть 2 розетки (американская и европейская):

```
interface EuroSocket {
  void getPower();
}
```

```
interface AmericanSocket {
  void getPower();
}
```

И пусть будет реализация американской розетки:

```
class SimpleAmericanSocketImpl
    implements AmericanSocket {
  @Override
  public void getPower() {
    System.out.println("americanSocket");
  }
}
```

Есть радио, которое работает только с европейской розеткой:

```
class Radio {
  public void listenMusic(EuroSocket euroSocket) {
    euroSocket.getPower();
  }
}
```

Вопрос в том, как запустить это радио, если мы имеем только европейскую розетку:

```
AmericanSocket socket =
    new SimpleAmericanSocketImpl();
```

```
Radio radio = new Radio();
radio.listenMusic(???);
```

Здесь и прийдет на помощь адаптер:

```
class SocketAdapter implements EuroSocket {
  private AmericanSocket americanSocket;

  public SocketAdapter(
      AmericanSocket americanSocket) {
    this.americanSocket = americanSocket;
  }

  @Override
  public void getPower() {
    americanSocket.getPower();
  }
}
```

Использование адаптера:

```
AmericanSocket socket =
    new SimpleAmericanSocketImpl();
Radio radio = new Radio();
EuroSocket euroSocket = new SocketAdapter(socket);
radio.listenMusic(euroSocket);
```

Bridge (мост)

Помогает отделить абстракцию от имплементации.

Проблема. У нас есть 2 типа транспортных средств (легковая машина и грузовик):

```
interface Car { }
interface Track { }
```

и их производители:

```
class AudiCar implements Car { }
class AudiTrack implements Track { }
class ToyotaCar implements Car { }
class ToyotaTrack implements Track { }
class MercedesCar implements Car { }
```

```
class MercedesTrack implements Track { }
```

Со временем могут появляться новые транспортные средства (мотоциклы, трактора). И для каждого нового транспортного средства надо будет создавать новый класс соответствующего производителя. Bridge помогает уменьшить рост количества классов.

Решение. У нас будет интерфейс моделей авто:

```
interface Model {
  void drive(String str);
}
```

и абстрактный класс транспортного средства, который содержит в себе интерфейс модели:

```
abstract class Vehicle {
  private Model model;

  public Vehicle(Model model) {
    this.model = model;
  }

  abstract void drive();
}
```

Реализации транспортных средств:

```
class Car extends Vehicle {
  public Car(Model model) {
    super(model);
  }

  @Override
  void drive() {
    System.out.println("drive car");
  }
```

```
}
```

```
class Track extends Vehicle {
  public Track(Model model) {
    super(model);
  }

  @Override
  void drive() {
    System.out.println("drive track");
  }
}
```

Реализации моделей:

```
class Audi implements Model {
  @Override
  public void drive(String str) {
    System.out.println(str + " audi");
  }
}
```

```
class Toyota implements Model {
  @Override
  public void drive(String str) {
    System.out.println(str + " toyota");
  }
}
```

```
class Mercedes implements Model {
  @Override
  public void drive(String str) {
    System.out.println(str + " mercedes");
  }
}
```

Класс Main:

```
Vehicle toyotaCar = new Car(new Toyota());
toyotaCar.drive();
```

```
Vehicle audiTrack = new Track(new Audi());
audiTrack.drive();
```

Теперь добавлять транспортные средства и модели гораздо проще.

Builder (строитель)

Необходим для объектов с несколькими полями чтобы не плодить много конструкторов.

```
public static void main(String[] args) {
  Person person = Person.newBuilder()
      .setFirstName("Василий")
      .setLastName("Иванов")
      .setMiddleName("Сергеевич")
      .build();
  System.out.println(person);
}

@Getter
@ToString
class Person {

  private String firstName;
  private String lastName;
  private String middleName;
//  ...

  private Person() {
  }

  public static Person.Builder newBuilder() {
    return new Person().new Builder();
  }

  public class Builder {

    private Builder() {
    }

    public Person.Builder setLastName(
        String lastName) {
      Person.this.lastName = lastName;
      return this;
    }

    public Person.Builder setFirstName(
```

```
          String firstName) {
      Person.this.firstName = firstName;
      return this;
    }

    public Person.Builder setMiddleName(
        String middleName) {
      Person.this.middleName = middleName;
      return this;
    }

//    ...

    public Person build() {
      return Person.this;
    }
  }
}
```

Chain of responsibility (цепочка ответственности)

Проблема. Есть строка. К ней надо применить фильтры (убрать цифры, добавить символы) в разном порядке, количестве и т. п. Также эти фильтры хотелось бы хранить как-то упорядоченно в одном месте. На помощь здесь и приходит данный паттерн.

```
public class Main {
  public static void main(String[] args) {
    MessageHandler messageHandler =
        new MessageAddExclamationMarkHandler(
            new MessageVerifyHandler(
                new MessagePrintHandler(null)));
    messageHandler.handle("hello world");
  }
}
```

```
abstract class MessageHandler {
  MessageHandler messageHandler;

  public MessageHandler(
```

```
      MessageHandler messageHandler) {
    this.messageHandler = messageHandler;
  }

  public abstract void handle(String message);
}
class MessagePrintHandler extends MessageHandler {

  public MessagePrintHandler(
      MessageHandler messageHandler) {
    super(messageHandler);
  }

  @Override
  public void handle(String message) {
    System.out.println(message);
  }
}
class MessageVerifyHandler
    extends MessageHandler {

  public MessageVerifyHandler(
      MessageHandler messageHandler) {
    super(messageHandler);
  }

  @Override
  public void handle(String message) {
    if (!message.matches(".*\\d.*")) {
      System.out.println(message);
    }
  }
}
class MessageAddExclamationMarkHandler
    extends MessageHandler {

  public MessageAddExclamationMarkHandler(
      MessageHandler messageHandler) {
    super(messageHandler);
  }

  @Override
  public void handle(String message) {
    System.out.println(message + "!");
```

```
    }
}
```

Command (команда)

Он решает проблему связывания через прослойку Command, а он в свою очередь уже и будет вызывать конкретный класс.

В чем преимущества? Во-первых, очень расширяемый (легко добавлять новые классы в любом количестве). Во-вторых, несмотря на кажущую простоту этот паттерн очень гибкий. С его помощью (через единый интерейс) можно выбрать две реализации (LightAndMusicPlayerCommand) или передать переменную (PhoneCommand).

```
public class Main {
  public static void main(String[] args) {
    Command command = new LightCommand(
        new Light());
    Button button = new Button(command);
    button.pressButton();

    command = new MusicPlayerCommand(
        new MusicPlayer());
    button = new Button(command);
    button.pressButton();

    command = new CoffeeMachineCommand(
        new CoffeeMachine());
    button = new Button(command);
    button.pressButton();

    command = new LightAndMusicPlayerCommand(
        new Light(), new MusicPlayer());
    button = new Button(command);
```

```
    button.pressButton();

    command =
        new PhoneCommand(new Phone(), "Иван");
    button = new Button(command);
    button.pressButton();
  }
}
```

```
class Button {
  Command command;

  public Button(Command command) {
    this.command = command;
  }

  public void pressButton() {
    command.execute();
  }
}
```

```
interface Command {

  void execute();
}
```

```
class Light {
  boolean isOn;

  public void swicthLight() {
    this.isOn = !this.isOn;
    System.out.println(
        "light is " + (isOn ? "on" : "off"));
  }
}
```

```
class LightCommand implements Command {
  private Light light;

  public LightCommand(Light light) {
    this.light = light;
  }

  @Override
  public void execute() {
    light.swicthLight();
  }
}
```

```
class MusicPlayer {

  public void playMusic() {
    System.out.println("play music");
  }
}
```

```
class MusicPlayerCommand implements Command {
  private MusicPlayer musicPlayer;

  public MusicPlayerCommand(
        MusicPlayer musicPlayer) {
    this.musicPlayer = musicPlayer;
  }

  @Override
  public void execute() {
    musicPlayer.playMusic();
  }
}
```

```
class CoffeeMachine {

  public void makeCoffee() {
    System.out.println("make coffee");
  }
}
```

```
class CoffeeMachineCommand implements Command {
  private CoffeeMachine coffeeMachine;

  public CoffeeMachineCommand(
      CoffeeMachine coffeeMachine) {
    this.coffeeMachine = coffeeMachine;
  }

  @Override
  public void execute() {
    coffeeMachine.makeCoffee();
  }
}
```

```
class LightAndMusicPlayerCommand
    implements Command {
  private Light light;
  private MusicPlayer musicPlayer;

  public LightAndMusicPlayerCommand(
```

```
      Light light,
      MusicPlayer musicPlayer) {
    this.light = light;
    this.musicPlayer = musicPlayer;
  }

  @Override
  public void execute() {
    light.swicthLight();
    musicPlayer.playMusic();
  }
}
```

```
class Phone {

  public void makeCall(String name) {
    System.out.println("male call to " + name);
  }
}
```

```
class PhoneCommand implements Command {
  private Phone phone;
  private String name;

  public PhoneCommand(Phone phone, String name) {
    this.phone = phone;
    this.name = name;
  }

  @Override
  public void execute() {
    phone.makeCall(name);
  }
}
```

Composite

Паттерн, который представляет из себя древовидную структуру (папки внутри папок).

Реализация папок в папке (любое количество).

```
public class Main {
```

```
  public static void main(String[] args) {
    Folder users = new Folder("users");
    Folder etc = new Folder("etc");
    Folder root = new Folder("root");
    root.addFolder(users, etc);

    Folder mike = new Folder("Mike");
    Folder kent = new Folder("Kent");
    Folder max = new Folder("Max");
    users.addFolder(mike, kent, max);

    Folder one = new Folder("one");
    Folder two = new Folder("two");
    Folder three = new Folder("three");
    mike.addFolder(one, two, three);

    root.printFolders();
  }
}
```

```
class Folder {
  private String name;

  public Folder(String name) {
    this.name = name;
  }

  private List<Folder> list = new ArrayList<>();

  public void addFolder(Folder folder) {
    list.add(folder);
  }

  public void addFolder(Folder... folder) {
    list.addAll(Arrays.asList(folder));
  }

  public void printFolders() {
    for (Folder folder : list) {
      System.out.println(folder.name);
      folder.printFolders();
    }
  }
}
```

Decorator (декоратор)

Позволяет не плодить классы и упростить их сочетания.

```
public class Main {
  public static void main(String[] args) {
    Folder users = new Folder("users");
    Folder etc = new Folder("etc");
    Folder root = new Folder("root");
    root.addFolder(users, etc);

    Folder mike = new Folder("Mike");
    Folder kent = new Folder("Kent");
    Folder max = new Folder("Max");
    users.addFolder(mike, kent, max);

    Folder one = new Folder("one");
    Folder two = new Folder("two");
    Folder three = new Folder("three");
    mike.addFolder(one, two, three);

    root.printFolders();
  }
}
```

Facade (фасад)

Позволяет уменьшить повторяемый код?

```
public class Main {
  public static void main(String[] args) {
    LivingRoom livingRoom = new LivingRoom();
    livingRoom.pressButton("5", "22");

    BadRoom badRoom = new BadRoom();
    badRoom.pressButton("10", "15");
  }
}
```

```
class RoomFacade {
  Tv tv = new Tv();
  AirConditioning airConditioning =
      new AirConditioning();
  Light light = new Light();
```

```
  public void pressButton(
      String channel,
      String temperature) {
    tv.playChannel(channel);
    airConditioning.setTemperature(temperature);
    light.turnLight();
  }
}
```

```
class LivingRoom {
  private RoomFacade roomFacade =
      new RoomFacade();

  public void pressButton(
      String channel,
      String temperature) {
    roomFacade.pressButton(channel, temperature);
  }
}
```

```
class BadRoom {
  private RoomFacade roomFacade =
      new RoomFacade();

  public void pressButton(
      String channel,
      String temperature) {
    roomFacade.pressButton(channel, temperature);
  }
}
```

```
class AirConditioning {

  public void setTemperature(String temperature) {
    System.out.println(
        "set temperature " + temperature);
  }
}
```

```
class Light {

  public void turnLight() {
    System.out.println("turn light");
  }
}
```

```
class Tv {
```

```
  public void playChannel(String channel) {
    System.out.println("play channel " + channel);
  }
}
```

sdf

Алгоритмы

Алгоритм — набор инструкций для выполнения некоторой задачи.

О:

- $O(\log n)$ или $O(\log_2 n)$ — логарифмическое время (бинарный поиск);
- $O(n)$ — линейное время (простой поиск);
- $O(n \log n)$ — эффективные алгоритмы сортировки (быстрая сотрировка);
- $O(n^2)$ — медленные алгоритмы сортировки (сортировка выбором);
- $O(n!)$ — очень медленные алгоритмы (задача о коммивояжере).

Скорость алгоритма измеряется не в секундах, а в темпе роста количества операций. По сути формула описывает, насколько быстро возрастает время выполнения алгоритма с увеличением размера исходных данных.

Бинарный поиск

Работает с отсортированной структурой, делит ее каждый раз на два, время логарифмическое.

Указать источник!!!

Оценка сложности

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.

Допустим, некоторому алгоритму нужно выполнить $3n^3 + 7n$ условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление 7n. Тогда говорят, что временная сложность этого алгоритма равна $O(n^3)$, т. е. зависит от размера входных данных кубически.

Использование заглавной буквы O (или так называемая O-нотация) пришло из математики, где ее применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём

занимаемой памяти) растет в зависимости от объема входных данных не быстрее, чем некоторая константа, умноженная на f(n).

Источники

- Статья «Оценка сложности алгоритмов, или Что такое O(log n)»

Порядок роста

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных. Чаще всего он представлен в виде O-нотации (от нем. «Ordnung» — порядок): O(f(x)), где f(x) — формула, выражающая сложность алгоритма. В формуле может присутствовать переменная n, представляющая размер входных данных. Ниже приводится список наиболее часто встречающихся порядков роста, но он ни в коем случае не полный.

- Константный — O(1)
  Порядок роста O(1) означает, что вычислительная сложность алгоритма не зависит от размера входных данных. Следует помнить, однако, что единица в формуле не значит, что алгоритм выполняется за одну операцию или требует очень мало времени. Он может потребовать и микросекунду, и год. Важно то, что это время не зависит от входных данных.

- Линейный — O(n)
  Порядок роста O(n) означает, что сложность алгоритма линейно растет с увеличением входного массива. Если линейный алгоритм обрабатывает один элемент пять миллисекунд, то мы можем ожидать, что тысячу элементов он обработает за пять секунд.
  Такие алгоритмы легко узнать по наличию цикла по каждому элементу входного массива.
- Логарифмический — O(log n)
  Порядок роста O(log n) означает, что время выполнения алгоритма растет логарифмически с увеличением размера входного массива. (Прим. пер.: в анализе алгоритмов по умолчанию используется логарифм по основанию 2). Большинство алгоритмов, работающих по принципу «деления пополам», имеют логарифмическую сложность. Метод Contains бинарного дерева поиска (binary search tree) также имеет порядок роста O(log n).
- Линеарифметический — O(n·log n)
  Линеарифметический (или линейно-логарифмический) алгоритм имеет порядок роста O(n·log n). Некоторые алгоритмы типа «разделяй и властвуй» попадают в эту категорию. В следующих частях мы

увидим два таких примера — сортировка слиянием и быстрая сортировка.

- Квадратичный — O(n2)
- Время работы алгоритма с порядком роста O(n2) зависит от квадрата размера входного массива. Несмотря на то, что такой ситуации иногда не избежать, квадратичная сложность — повод пересмотреть используемые алгоритмы или структуры данных. Проблема в том, что они плохо масштабируются. Например, если массив из тысячи элементов потребует 1_000_000 операций, массив из миллиона элементов потребует 1_000_000_000_000 операций. Если одна операция требует миллисекунду для выполнения, квадратичный алгоритм будет обрабатывать миллион элементов 32 года. Даже если он будет в сто раз быстрее, работа займет 84 дня.

  Мы увидим пример алгоритма с квадратичной сложностью, когда будем изучать пузырьковую сортировку.

Наилучший, средний и наихудший случаи

Что мы имеем в виду, когда говорим, что порядок роста сложности алгоритма — O(n)? Это усредненный случай? Или наихудший? А может быть, наилучший?

Обычно имеется в виду наихудший случай, за исключением тех случаев, когда наихудший и средний сильно отличаются. К примеру, мы увидим примеры алгоритмов, которые в среднем имеют порядок роста O(1), но периодически могут становиться O(n) (например, ArrayList.add). В этом случае мы будем указывать, что алгоритм работает в среднем за константное время, и объяснять случаи, когда сложность возрастает.

Самое важное здесь то, что O(n) означает, что алгоритм потребует не более n шагов.

Источники

- Статья «Алгоритмы и структуры данных для начинающих: сложность алгоритмов»

Двоичные и красно-черные деревья

Двоичное дерево — иерархическая структура данных, в которой каждый узел имеет не более двух потомков.

Красно-черное дерево:

- корень — черные,
- конечные — черные,
- дочерние от красного — черные,
- глубина в черных узлах одинаковая

Ф

Java Core

ООП

ООП

ООП — методология программирования, основанная на представлении программы в виде совокупности объектов, взаимодействующих между собой, и соблюдении 3-х основных принципов — инкапсуляция, наследование и полиморфизм.

Источники

- Ссылка на подготовку к собесам JM

Основные принципы ООП

Инкапсуляция объединяет поля и методы в классы, скрывает детали реализации и открывает только то, что необходимо при последующем использовании.

Наследование позволяет описать новый класс на основе уже существующего с частичной или полностью заимствующей функциональностью.

Полиморфизм — возможность использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

В Java это возможность хранить в переменной разные типы и перегрузка.

Источники

- Статья на хабре «ООП с примерами (часть 2)»
- Видео с канала Сергея Немчинского «Лекция по основам Java: ООП, объекты, классы, интерфейсы, JVM, JDK, JIT»

Основные понятия ООП (класс, объект, интерфейс)

Класс — способ описания сущности, определяет состояние (набор полей) и поведение (набор методов), задает правила взаимодействия с данной сущностью (контракт).

Объект — экземпляр класса, имеет конкретное состояние и поведение, определяемые классом.

Интерфейс — набор публичных методов (доступных для других классов). Специфицирует класс, определяя все возможные действия над ним.

Абстрагирование — способ выделить набор общих характеристик объекта, исключая из рассмотрения частные и незначимые. Абстракция — набор всех таких характеристик (набор общих характеристик).

Источники

- Статья на хабре «ООП с примерами (часть 1)»
- Статья на хабре «ООП с примерами (часть 2)»

Виды связей («является» и «имеет», композиция и агрегация)

Является (is a) — наследование, имеет (has a) — ассоциация (использование полей в классе).

Ассоциация обозначает связь между объектами. Композиция и агрегация — частные случаи ассоциации «часть-целое».

Авто и двигатель — композиция (может быть только частью чего-то), авто и пассажиры — агрегация (независимая часть).

Источники

Ссылка на подготовку к собесам JM

SOLID

Роберт Мартин

Преимущества SOLID (при соблюдении его условий):

- программу легко дополнять, расширять, изменять и поддерживать;
- код легче читать (находить ошибки и переиспользовать).

1. Принцип единственной ответственности (SRP, The Single Responsibility Principle)

Роберт Мартин: каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс.

2. Принцип открытости/закрытости (OCP, The Open Closed Principle)

Бертран Мейер: программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения.

3. Принцип подстановки Барбары Лисков (LSP, The Liskov Substitution Principle)

Роберт Мартин: функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Поведение наследующих классов не должно противоречить поведению, заданного базовым классом.

4. Принцип разделения интерфейса (ISP, The Interface Segregation Principle)

Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения для всех.

5. Принцип инверсии зависимостей (DIP, The Dependency Inversion Principle)

Зависимость на абстракциях, нет зависимости на что-то конкретное. Абстрактность не должна зависеть на конкретное, конкретность должна завесить на абстракциях.

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракции.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Источники:

5 видео с канала Немчинского:

- SOLID принципы: SRP (Принцип единственной ответственности, Single Responsibility Principle)

- SOLID принципы: OCP (Открытости/закрытости (Open Closed Principle)
- SOLID: Принцип подстановки Барбары Лисков/ LSP (The Liskov Substitution Principle)
- SOLID принципы: ISP (Принцип Разделения Интерфейса (The Interface Segregation Principle)
- SOLID принципы: DIP (Принцип инверсии зависимостей (The Dependency Inversion Principle)

Java Basics (ООП, синтаксис)

Generics

String

Concurrency (Thread, Thread Pool, Patterns)

Java 8 Features (Lambda, Stream, Optional, FP)

IO (Networking (Socket), File, Java NIO/NIO2)

Java Collection Framework

Java Collection Framework — это набор связанных классов и интерфейсов, реализующих широко используемые структуры данных (коллекции).

List

Iterable -> Collection -> List.

ArrayList

List -> ArrayList.

ArrayList — это класс, который реализует функционал динамического массива, т. е. массива, который расширяется при добавление новых объектов.

Основные поля из реализации:

- Object[] EMPTY_ELEMENTDATA (массив);

- `int DEFAULT_CAPACITY` (дефолтный размер массива);
- `int size` — текущий размер ArrayList.

Конструкторы:

- `ArrayList()` (размер по умолчанию);
- `ArrayList(int initialCapacity)` (указывается размерность);
- `ArrayList(Collection<? extends E> c)`.

Интересно будет так же разобрать основные методы (добавление, получение по индексу, удаление). Можно попробовать написать свою реализацию.

LinkedList

`List -> LinkedList`.

LinkedList — двухсвязанный список. Есть указатели (head) в начале и в конце списка.

В отличии от ArrayList здесь всего 2 конструктора:

- `LinkedList()`;
- `LinkedList(Collection<? extends E> c)`.

Основные поля из реализации:

- `int size`;
- `Node<E> first` (первый узел цепочки);
- `Node<E> last` (последний узел цепочки).

Здесь можно разобрать Node, метод добавления, получения и удаления. Было бы полезно написать свою реализацию (Алишев описывал в своем курсе).

Скорость:

- добавление миллиона элемента — ArrayList;
- получение по индексу — ArrayList;
- добавление в начало — LinkedList.

Set

Iterable -> Collection -> Set.

Коллекция, которая хранит в себе уникальные элементы.

Реализации:

- HashSet;
- LinkedHashSet (сохраняет порядок добавления);
- TreeSet (отсортированные элементы).

Можно разобрать методы (add, contains, isEmpty, addAll, retainAll, removeAll)

Map

Object -> Map.

Map (карта, отображение). Элементу одному множеству сопоставляем элемент другого множества.

Реализации:

- HashMap;
- LinkedHashMap (сохраняет порядок добавления);
- TreeMap (сортировка по ключу).

Обход через Entry.

Нужно знать реализацию HashSet. Можно еще раз написать реализацию.

Comparator и Comparable

Comparator реализуется в отдельном классе, а Comparable — в самом объекте.

Iterator

Iterator позволяет удалять элементы из коллекции во время обхода.

Источник

- Курс на Udemy «Продвинутая Java», Наиль Алишев, 06.2019.

Многопоточность

Все, что выполняется в методе main() — выполняется в своем потоке. Этот поток главный.

Многопоточность позволяет выполнять какой-то код параллельно с другим кодом.

Для многопоточно программы не имеет значение сколько у вас ядер. В java многопоточность представлена в виде виртуальной параллельности. Т. е. если у вас одноядерный компьютер. В этом случае процессор будет переключаться быстро переключаться между этими параллельными участками кода. Достигается видимость, что задание выполняется параллельно, хотя идет просто быстрое переключение между потоками.

Если у вас многоядерный процессор, то многопоточный код java сама распределить на разные ядра процессора.

Важно понимать, что ускорить работу программы — не основная цель многопоточности. Основная цель многопоточного программирования в том, чтобы реализовать какой-то функционал (клиент не ждал чего-то).

Способы создания потоков:

- в новом классе наследуемся от Thread и переопределяем run() (куда и помещаем логику), для запуска этого потока создаем объект нашего класса и запускаем поток при помощи метода start();
- в новом классе реализуем Runnable и переопределяем метод run(), далее создаем объект класса Thread и передаем в конструктор созданный класс, запускаем поток через метод start().

Потоки в java не синхронизированы, т. е. они могут выполняться в любом порядке. Выдача совершенно не предсказуема.

Методы:

- void sleep(long millis) (засыпает на указанное количество мс);
- void join() (ждет выполнения потока до перехода к следующему участку кода);

Резюме: потоки не синхронизированы, существует несколько методов создания потоков, метод sleep().

Приложение остановится, когда закончат работу все потоки (а не когда завершится метод main()).

Для того, чтобы структурировать и упорядочить работу потоков существует синхронизация потоков.

Ключевое слово volatile. Это ключевое слово необходимо для того случая, когда одна переменная делится между несколькими потоками. И когда один поток пишет в переменную, а другие читают эту переменную, тогда при помощи этого слово мы можем настроить взаимодействие этих потоков.

Как работает. Если у нашего компьютера несколько ядер, у каждого ядра процессора есть свой кэш (свой отдельный участок памяти). Каждый поток может выполняться в своем ядре. Таким образом можно сказать: у потоков может быть своя память (кэш ядра) и общая память. Если мы изменим какую-либо переменную, то она сначала измениться в своей памяти и не сразу передаст значение общей памяти. В этот момент к переменной может обратиться другой поток (у этого потока свой кэш). Т. о. мы можем не увидеть изменение. Ключевое слово volatile запрещает хранить переменную в кэше. Т. о. значение переменной будет только в общей памяти, и мы не сможем считать устаревшее значение.

Состояние гонки (англ. race condition). Несколько потоков пишут что-то одновременно (одним потоком 100 раз

прибавили, а вторым 100 раз вычислили и не получили ноль). Решение — заключение блока кода в synchronized.

Ключевое слово синхронизирует работу нескольких потоков при записи в общую переменную.

Как работает synchronized? Только один поток в один момент времени получает доступ к выполнению телу метода.

Как устроено ключевое слово? В java каждому объекту присваивается специальная сущность. Эта сущность называется монитор. Эта сущность в один момент времени может быть только у одного потока. Важный момент: монитор привязан к объекту, т. е. для синхронизации нужен объект. Если мы не указываем объект явно, то значит синхронизируется на объекте this.

Ключевое слово может быть как в блоке, так и на уровне метода. Если оно на уровне метода, то это всегда this. Бывают случаи, когда у нас 2 метода находятся в одном классе (один объект), но в текущей задаче нам не требуется, чтобы 2 метода выполнялись независимо относительно друг друга (понятие монитор). В этом случае можно создать произвольный объект, затем в блок synchronized поместить код

и заключить созданный объект в круглые скобки.

Способ создания потоков, называемый Thread pool (пул потоков). В некоторых случаях этот способ более удобен. Суть: создается n число потоков и выполняют какое-то задание параллельно.

```
ExecutorService executorService =
    Executors.newFixedThreadPool(2);
```

ExecutorService — есть пул потоков. В данному случае мы создали пул из 2 потоков.

```
for (int i = 0; i < 5; i++) {
  executorService.submit(new Work(i));
}
```

Передаем 5 заданий созданному пулу потоков. Work — класс, реализующий интерфейс Runnable.

```
executorService.shutdown();
```

Перестаем принимать новые задание и начинаем выполнять, что было прописано раньше (submit()). Можно сказать, что shutdown() — это аналогия метода start(). В данном случае каждый поток будет брать по одному заданию и выполнять его.

```
executorService
    .awaitTermination(1, TimeUnit.DAYS);
```

Указываем сколько мы готовы ждать, пока наши потоки выполнять задание, и пойти

дальше. Т. е. мы ждем столько времени и потом начинаем выполнять все то, что находится ниже этой строки (или раньше, если потоки завершат свою работу раньше). Это аналоги метода join().

Паттерн producer-consumer. Часто встречается в многопоточно программировании. Часто бывает такое, что один или более потоков производят что-то, и один или более потоков потребляют что-то (одни дает какие-то примеры машине, а другие решают для каждого пользователя). Реализация при помощи ArrayBlockingQueue:

```
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class Main {
  private static BlockingQueue<Integer> queue =
      new ArrayBlockingQueue<>(10);

  public static void main(String[] args)
      throws InterruptedException {
    Thread thread1 = new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          produce();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    });

    Thread thread2 = new Thread(new Runnable() {
      @Override
```

```
      public void run() {
        try {
          consumer();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    });

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();
  }

  private static void produce()
      throws InterruptedException {
    Random random = new Random();

    while (true) {
      queue.put(random.nextInt(100));
    }
  }

  private static void consumer()
      throws InterruptedException {
    while (true) {
      Thread.sleep(100);
      if (random.nextInt(10) == 5) {
        System.out.println(queue.take());
        System.out.println("size" + queue.size());
      }
    }
  }
}
```

В чем суть. Мы задаем `ArrayBlockingQueue<>(10)` фиксированного размера (10). Далее 2 метода `produce()` и `consumer()`. В `consumer()` было специально добавлено условие, чтобы показать

что produce() будет ждать, пока в коллекции не освободится время. Пример из реальной жизни: есть сервер и на этом сервере люди шлют запросы (арифметические операции); они послали множество арифметических операций, и если сервер не справляется — система ограничит количество запросов, которые могут прийти; остальные люди, которые пришли позже, будут ждать.

Методы wait() и notify() определены в Object.

Метод wait() вызывается только внутри синхронизованного блока, иначе не имеет смысла. При вызове этого метода отдаем синхронизированный блок (т. е. вышли из этого блока потоком, поток может заниматься чем-то другим). А также ждем вызова notify() на этом объекте для продолжения работы. Если объект ждут несколько потоков, то в этом случае надо вызвать notifyAll() для продолжения работы всех потоков (а не только одного).

Важное замечание: notify() не освобождает монитор. Т. е. сначала выполниться код под этим методом, а потом перейдет к выполнению кода после метода wait().

Метод wait() с параметрами продолжит свою работу после вызова notify() на объекте

(при свободном мониторе) или после истечения времени.

Неважно на каком объекте вызывается внутри блока synchronized вызываются wait() и notify(). Они будет действовать на объект, внутри класса которого они были вызваны. Либо надо явно это указать (указать на конкретный объект).

Все объекты, на которых мы синхронизируемся должны быть константой.

Реализация того же паттерна producer-consumer, но с методами wait() и notify().

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class Main {

  public static void main(String[] args)
      throws InterruptedException {
    ProduserConsumer pc = new ProduserConsumer();

    Thread thread1 = new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          pc.produce();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    });
    Thread thread2 = new Thread(new Runnable() {
      @Override
      public void run() {
```

```
        try {
          pc.consume();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    });

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();
  }
}

class ProduserConsumer {
  private Queue<Integer> queue =
      new LinkedList<>();
  private final int LIMIT = 10;
  private final Object lock = new Object();

  public void produce()
      throws InterruptedException {
    int value = 0;
    while (true) {
      synchronized (lock) {
        while (queue.size() == LIMIT) {
          lock.wait();
        }
        queue.offer(value++);
        lock.notify();
      }
    }
  }

  public void consume()
      throws InterruptedException {
    while (true) {
      synchronized (lock) {
        while (queue.size() == 0) {
          lock.wait();
        }
        int value = queue.poll();
```

```
            System.out.println(value);
            System.out.println("size = "
                + queue.size());
            lock.notify();
        }

        Thread.sleep(3000);
      }
    }
}
```

Здесь важный момент: `lock.notify()` всегда вызывается в конце метода.

`CountDownLatch` — защелка обратного отсчета. При создании объекта класса указываем количество итераций в конструкторе, которые мы должны отсчитать назад прежде, чем защелка нам отопрется. Класс потокобезопасный. Метод `countDown()` декрементирует переменную на единицу.

`CountDownLatch` — класс, при создании объекта которого мы передаем в конструкторе в качестве аргумента число. Это число говорит о том, сколько раз мы должны вызвать метод `countDown()` из любого количества потоков, чтобы `await()` больше не ждал.

Класс `ReentrantLock` нужен для синхронизации потоков. Действует аналогично ключевому слову `synchronized`. Создаем переменную `Lock lock = new ReentrantLock()` и заключаем участок

кода в блок lock.lock() и lock.unlock(). lock.unlock() лучше заключить в блок finally.

Класс Semaphore. Предположим, у нас есть ресурс, и несколько потоков используют этот ресурс. Данный класс мы будем использовать, когда нам необходимо ограничить доступ к этому ресурсу, передав нужное число в конструктор. Метод release() — отдает разрешение. acquire() — когда мы начинаем взаимодействовать с ресурсом. availablePermits() — возвращает количество разрешений, который у нас свободно.

Преимущество ReentrantLock перед synchronized в следующем:

- нет вложенных конструкций (синхронизированный блок внутри синхронизированного блока);
- можно избежать дедлока.

Дедлок — взаимная блокировка ресурсов.

```
import java.util.Random;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Main {
  public static void main(String[] args)
      throws InterruptedException {

    Runner runner = new Runner();

    Thread thread1 = new Thread(new Runnable() {
```

```
      @Override
      public void run() {
        runner.firstThread();
      }
    });

    Thread thread2 = new Thread(new Runnable() {
      @Override
      public void run() {
        runner.secondThread();
      }
    });

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();

    runner.finished();
  }
}

class Runner {
  private Account account1 = new Account();
  private Account account2 = new Account();
  Random random = new Random();
  Lock lock1 = new ReentrantLock();
  Lock lock2 = new ReentrantLock();

  public void firstThread() {
    for (int i = 0; i < 10000; i++) {
      lock1.lock();
      lock2.lock();
      try {
        Account.transfer(account1, account2,
            random.nextInt(100));
      } finally {
        lock1.unlock();
        lock2.unlock();
      }
    }
  }
```

```
  public void secondThread() {
    for (int i = 0; i < 10000; i++) {
      lock2.lock();
      lock1.lock();
      try {
        Account.transfer(account2, account1,
            random.nextInt(100));
      } finally {
        lock2.unlock();
        lock1.unlock();
      }
    }
  }
  public void finished() {
    System.out.println(account1.getBalance());
    System.out.println(account2.getBalance());
    System.out.println("summ: "
        + (account1.getBalance()
        + account2.getBalance()));
  }
}

class Account {
  private int balance = 10000;

  public void deposit(int amount) {
    balance += amount;
  }

  public void withdraw(int amount) {
    balance -= amount;
  }

  public int getBalance() {
    return balance;
  }

  public static void transfer(Account acc1,
      Account acc2, int amount) {
    acc1.withdraw(amount);
    acc2.deposit(amount);
  }
}
```

Способы избежания:

- лочить объекты в одинаковом порядке (если позволяет логика программы);
- решение на уровне кода:

```
import java.util.Random;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Main {
  public static void main(String[] args)
      throws InterruptedException {
    Runner runner = new Runner();

    Thread thread1 = new Thread(new Runnable() {
      @Override
      public void run() {
        runner.firstThread();
      }
    });

    Thread thread2 = new Thread(new Runnable() {
      @Override
      public void run() {
        runner.secondThread();
      }
    });

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();

    runner.finished();
  }
}

class Runner {

  private Account account1 = new Account();
  private Account account2 = new Account();
  Random random = new Random();
  Lock lock1 = new ReentrantLock();
  Lock lock2 = new ReentrantLock();
```

```
  private void takeLocks(Lock lock1, Lock lock2) {
    boolean firstLockTaken = false;
    boolean secondLockTaken = false;
    while (true) {
      try {
        firstLockTaken = lock1.tryLock();
        secondLockTaken = lock2.tryLock();
      } finally {
        if (firstLockTaken && secondLockTaken) {
          return;
        }
        if (firstLockTaken) {
          lock1.unlock();
        }
        if (secondLockTaken) {
          lock2.unlock();
        }
      }
      try {
        Thread.sleep(1);
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }

  public void firstThread() {
    for (int i = 0; i < 10000; i++) {
      takeLocks(lock1, lock2);
      try {
        Account.transfer(account1, account2,
            random.nextInt(100));
      } finally {
        lock1.unlock();
        lock2.unlock();
      }
    }
  }
  public void secondThread() {
    for (int i = 0; i < 10000; i++) {
      takeLocks(lock2, lock1);
      try {
        Account.transfer(account2, account1,
```

```
            random.nextInt(100));
      } finally {
        lock2.unlock();
        lock1.unlock();
      }
    }
  }
  public void finished() {
    System.out.println(account1.getBalance());
    System.out.println(account2.getBalance());
    System.out.println("summ: "
        + (account1.getBalance()
        + account2.getBalance()));
  }
}

class Account {
  private int balance = 10000;

  public void deposit(int amount) {
    balance += amount;
  }

  public void withdraw(int amount) {
    balance -= amount;
  }

  public int getBalance() {
    return balance;
  }

  public static void transfer(Account acc1,
      Account acc2, int amount) {
    acc1.withdraw(amount);
    acc2.deposit(amount);
  }
}
```

-

Источник

- Курс на Udemy «Продвинутая Java», Наиль Алишев, 06.2019.

Инструменты

IDE (VSCode, Eclipse, IDEA, vim)

Build (Maven, Gradle)

Docker (k8s, Docker Compose, Docker swarm)

Jenkins

Фреймворки

Spring

Spring Framework

Framework

Framework (англ. — «каркас») — платформа, которая определяет структуру системы (приложения) и облегчает разработку компонентов системы и их интеграцию.

Фреймворк — это больше, чем просто библиотека (определяет структуру системы, предоставляет определенные паттерны разработки).

Популярные фреймворки:

- Spring Framework;
- Node.js;
- Django;
- Ruby on Rails;
- ASP.NET;
- и множество других.

Spring Framework

Востребованность Spring:

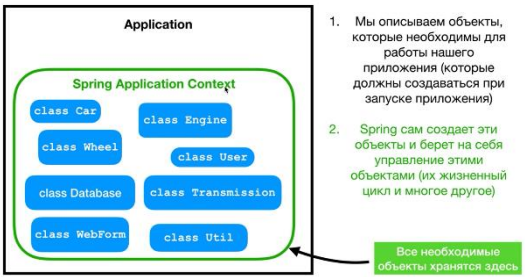
- один из самых популярных web-фреймворков в мире;
- самый популярный Java-фреймворк;

- Java — один из самых популярных ЯП в мире, Spring обычно используется везде, где используется Java;
- очень востребован работодателями по всему миру — достаточно зайти на любой сайт по поиску работы.

Spring Framework состоит из множества компонентов, т. е. облегчает множество аспектов разработки приложений на Java:

- контекст приложения (Application Context) и Внедрение Зависимости (Dependency Injection);
- удобный и эффективный доступ к БД (замена JDBC);
- компонент для разработки web-приложений на Java (Spring MVC);
- множество других полезных компонентов (spring.io).

Application Context и Dependency Injection



Spring решает следующие проблемы

- Мы описываем объекты, которые необходимы для работы нашего приложения (которые должны создаваться при запуске приложения).
- Spring сам создает эти объекты и берет на себя управление этими объектами (их жизненный цикл и многое другое).
- Spring сам внедряет все необходимые зависимости в объекты (связывает объекты между собой). Нам необходимо только описать эту связь, Spring делает все остальное. Это называется Dependency Injection.

Удобный и эффективный доступ к БД

- Большинство приложений используют БД.
- JDBC — примитивный и неудобный способ взаимодействия с БД.
- JDBC не подходит для сложных приложений. Он слишком низкоуровневый.
- Spring предоставляет множество инструментов для взаимодействия с БД.

Spring MVC

- Компонент Spring Framework, который позволяет создавать web-приложения.
- Огромное количество web-приложений в интернете работают на Spring MVC.

- Помимо этого, Spring MVC часто используется в качестве backend-API для работы мобильных приложений.

Другие компоненты Spring Framework

- Spring Security;
- Spring Boot;
- Spring Webflow;
- и множество других.

spring.io/projects

Bean

- Это просто Java объект.
- Когда Java объекты создаются с помощью Spring'а, они называются бинами (beans).
- Бины создаются из Java классов (так же, как и обычные объекты).
- и множество других.

Spring можно конфигурировать с помощью:

- XML файла конфигурации (старый способ, но многие существующие приложения до сих пор его используют).
- Java аннотации и немного XML (современный способ).
- Вся конфигурация на Java коде (современный способ).
- и множество других.

Способы внедрения зависимостей:

- Через конструктор.
- Через setter.
- Есть множество конфигураций того, как внедрять (scope, factory method и т. д.).
- Можно внедрять через XML, аннотации или Java-код.
- Процесс внедрения можно автоматизировать (Autowiring).

Setter'ы

Setter'ы — это «специальные методы», которые нужны для присвоения значения полям. Они позволяют нам не нарушать инкапсуляцию.

Scope

Scope задает то, как Spring будет создавать ваши бины.

Singleton

Scope, который используется по умолчанию.

- По умолчанию создается один объект (он создается до вызова метода `getBean()`).
- При всех вызовах `getBean()` возвращается ссылка на один и тот же единственный объект.

Scope Singleton чаще всего используется тогда, когда у нашего бина нет изменяемых состояний (stateless). Потому что если будем изменять состояние у Singleton бина, столкнемся с проблемой.

Prototype

Scope, который каждый раз создает новый объект при каждом вызове `getBean()`.

Scope Prototype чаще всего используется тогда, когда у нашего бина есть изменяемое состояние (stateful).

Другие scope'ы: request, session, global-session.

Жизненный цикл бина (Bean Lifecycle):

- вы запускаете Spring приложение;
- запускается Spring контейнер;
- создается объект бина;
- в бин внедряются зависимости (DI);
- вызывается init-method;
- бин готов к использованию;
- вызывается указанный destroy-method;
- остановка Spring приложения.

init-method и destroy-method

init-method:

- метод, который запускается в ходе инициализации бина;
- инициализация ресурсов, обращение к внешним файлам, запуск БД.

destroy-method:

- метод, который запускается в ходе уничтожения бина (при завершении приложения);
- очищение ресурсов, закрытие потоков ввода-вывода, закрытие доступа к БД.

Тонкости init и destroy методов:

- Модификатор доступа. У этих методов может быть любой модификатор доступа.
- Тип возвращаемого значения. Может быть любой, но чаще всего используется void (т. к. нет возможности получить возвращаемое значение.
- Название метода. Название может быть любым.
- Аргументы метода. Эти методы не должны принимать на вход какие-либо аргументы.
- Для бинов со scope «prototype» Spring не вызывает destroy метод. Spring не берет на себя полный жизненный цикл

бинов со scope «prototype». Spring отдает prototype бины клиенту и больше о них не заботиться (в отличие от singleton бинов).

factory-method

Фабричный метод (англ. Factory Method) — это паттерн программирования.

Вкратце: паттерн «фабричный метод» предлагает создавать объекты не напрямую, используя операторы new, а через вызов особого фабричного метода. Объекты все равно будут создаваться при помощи new, но делать это будет фабричный метод (иногда это бывает полезно).

Если объекты класса создаются фабричным методом, то можно определить factory-method.

Закончил:

https://www.youtube.com/watch?v=aXDMYy930b4&ab_channel=alishev

Источник

- Цикл видео на YouTube «Spring Framework», alishev, 02.2019.

String Core

sdfsd

String Data

sdfds

String Cloud

sdfsd

Spring Security

Ссылка на GitHub

Источники

- Сайт baeldung.com: «Security with Spring», 2022

Обработка исключений в контроллерах

@ExceptionHandler

Пример.

Основной недостаток подхода в том, что он определяется для каждого контроллера отдельно, а не глобально для всего приложения. Это ограничение можно обойти путем наследования от класса, содержащий метод с аннотацией @ExceptionHandler.

HandlerExceptionResolver

Либо отсутствует тело ответа, либо длинное решение.

@ControllerAdvice

Пример.

Решает недостатки двух способов выше. Также позволяет подключать лишь некоторые контроллеры. В примере данный код закомментирован.

Источник

Статья на habr.com «Обработка исключений в контроллерах Spring», Alexey Kutepov, 11.2020.

Spring validation

Ссылка на GitHub

Источники

- Видео на youtube «Java EE 66: Bean validation 4: Кастомный валидатор», Alexey Kutepov, 04.2018.

REST vs SOAP

Enterprise Application (корпоративные приложения) и его проблемы:

- объемы данных;
- устаревшие приложения;
- монолитность систем и интеграций;
- внешняя интеграция.

Интеграция — обмен данными между двумя системами.

Интеграция и ее история:

- интеграция через БД;
- интеграция через вызов методов:
  - DCOM, RPC, RMI;
  - CORBA (Common Object Request Broker Architecture, общая архитектура брокера объектных запросов);
  - SOAP, REST;
- SOA.

Интеграция через БД

Преимущества:

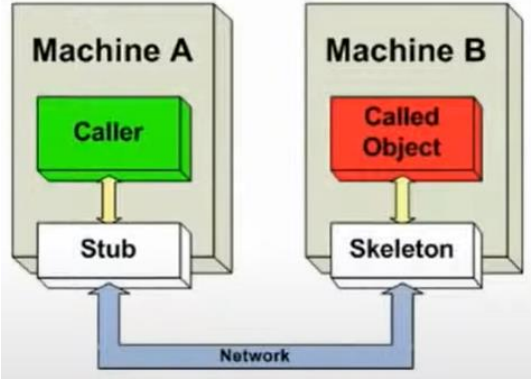
- простота;
- дешевизна.

Недостатки:

- разрастание;

- отсутствие масштабирования;
- нарушения целостности.

RPC (Remote Procedure Call, вызов удаленных процедур)



Machina A вызывает Stub локально, а Stub дальше уже через сеть обращается к Skeleton. Skeleton локально вызывает код на сервере (Machina B). Можно сказать на обеих машинах совершается локальных вызов, но они невидимым образом превращаются в сетевые вызовы.

К подвиду RPC можно отнести Java RMI (Remote Method Invocation, удаленный вызов метода). В отличие от RPC поддерживал объектно-ориентированный подход, но использовал большое количество портов.

DCOM (Distributed Component Object Model, объектный режим распределенных компонентов)

Преимущества:

- технология Microsoft;
- HTTP поддержка.

SOAP, REST

В SOAP интеграция происходит через шину, а не напрямую между приложениями. Таким образом, программист не пишет интеграцию для каждого приложения, а делает только одну интеграцию — это шина. А потом ее средствами перенаправляет эти данные на вход другого сервиса.

REST

REST — это не протокол и не стандарт, а архитектурный стиль.

Принципы стиля:

- Единый интерфейс
  Ресурсы должны быть однозначно идентифицированы посредством одного URL-адреса и только с помощью базовых методов сетевого протокола (DELETE, PUT, GET, HTTP).
- Клиент-сервер

Должно быть четкое разграничение между клиентом и сервером: пользовательский интерфейс и вопросы сбора запросов — на стороне клиента; доступ к данным, управление рабочей нагрузкой и безопасность — на стороне сервера.

- Сохранение состояния
  Все клиент-серверные операции должны быть без сохранения состояния. Любое необходимое управление состоянием должно осуществляться на клиенте, а не на сервере.
- Кэширование
  Все ресурсы должны разрешать кэширование, если явно не указано, что оно невозможно.
- Многоуровневая система
  REST API допускает архитектуру, которая состоит из нескольких уровней серверов.
- Запрос кода
  В большинстве случаев сервер отправляет обратно статические представления ресурсов в формате XML или JSON. Однако при необходимости серверы могут отправлять исполняемый код непосредственно клиенту.

Сравнение

- SOAP — это целое семейство протоколов и стандартов;
- SOAP использует HTTP как транспортный протокол, в то время как REST базируется на нем;
- есть мнение, что разработка RESTful сервисов намного проще;
- REST может быть представлен в различных форматах, а SOAP привязан к XML;
- «REST vs SOAP» можно перефразировать в «Простота vs Стандарты»;
- обработка ошибок;
- SOAP работает с операциями, а REST — с ресурсами.

Выводы

Где REST лучше использовать и почему:

- В сервисах, которые будут использоваться из javascript. Тут и говорить нечего, javascript хорошо работает с json, поэтому именно его и надо предоставлять.
- В сервисах, которые будут использоваться из языков, в которых нет возможности сгенерировать прокси клиента. Это Objective-C, например. Не нужно парсить вручную SOAP-конверт, это незачем.

- Когда существуют очень высокие требования к производительности. Это, как правило, очень интенсивно используемые API, вроде Twitter API или Google API.

Где SOAP лучше использовать и почему:

- Во внешней интеграции между большими (Enterprise) системами.
- В случае использования при пересылке сложных (от сотни полей) объектов, требующих автоматической валидации (например, имеющих небольшое количество консистентных состояний).
- В случае, когда технический диалог с командой, поддерживающей другую часть интеграции, затруднен (например, гос органы).
- Короче — в сложных случаях.

REST

Ни HTTP методы (GET, POST, PUT DELETE), ни коды ответов никак не сопоставляются с бизнесом.

Транзакция REST:

- метод запроса (GET);
- путь запроса (/object/list);
- тело запроса (форма);

- код ответа (200 OK);
- тело ответа (данные в формате JSON).

HATEOAS

HATEOAS (Hypermedia As The Engine Of Application State) — архитектурные ограничения для REST-приложений.

С помощью HATEOAS клиент взаимодействует с сетевым приложением, сервер которого обеспечивает динамический доступ через гипермедиа. REST-клиенту не требуется заранее знать, как взаимодействовать с приложением или сервером за пределом гипермедиа.

JSON Schema

- один из языков описания структуры JSON-документа;
- использует синтаксис JSON;
- базируется на концепциях XML Schema, RelaxNG, Kwalify.

Преимущества RESTful API:

- простота;
- скорость;
- легкость в написании.

Недостатки RESTful API:

- до сих пор нет общего согласования того, что такое RESTful API;
- словарь REST поддерживается не полностью;
- словарь REST недостаточно насыщен и не расширяем;
- RESTful API очень трудно дебажить;
- RESTful API привязан к протоколу.

SOAP

SOAP (Simple Object Access Protocol);

- SOAP (Simple Object Access Protocol, простой протокол доступа к объектам). Основан на XML.
- Рекомендует к использованию W3C (World Wide Web Consortium, Консорциум Всемирной паутины);
- SOAP не зависит ни от платформы, ни от языка.

XML-RPC

- XML-RPC — стандарт/протокол вызова удаленных процедур, использующий XML для кодирования своих сообщений и HTTP в качестве транспортного механизма.
- Является прародителем SOAP, отличается исключительной простотой в применении.

- Был отвергнут Microsoft, как излишне простой.
- Существует по сей день и даже набирает популярность.

WSDL (Web Services Description Language, язык описания веб-сервисов)

- Для описания используется документ формата XML. В нем описываются технические детали: URL-адреса, порт, имена методов, аргументы и типы данных.
- Поскольку WSDL представляет собой XML, он читается человеком и может использоваться машиной, что помогает динамически вызывать службы и привязываться к ним.

UDDI (Universal Description, Discovery and Integration, универсальное описание, обнаружение и интеграция)

- Это служба каталогов.
- Веб-сервис может зарегистрироваться в UDDI и сделать себя доступным через него для обнаружения.

Завершение

- В простых случаях и когда у нас критична скорость работы — REST.
- В сложных случаях, когда у нас не критична скорость, но критична

автоматическая поддержка от технологии — SOAP.

Источник

- Видео на YouTube «Rest web-services vs SOAP Services», Sergey Nemchinskiy, 05.2020.

SOAP

Сервер

Проект Spring (https://start.spring.io) с зависимостями Spring Web и Spring Web Services.

К текущему проекту добавить зависимости

```
<dependency>
  <groupId>wsdl4j</groupId>
  <artifactId>wsdl4j</artifactId>
</dependency>
```

а также

```
<dependency>
  <groupId>javax.activation</groupId>
  <artifactId>activation</artifactId>
  <version>1.1.1</version>
</dependency>
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.1</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>2.3.4</version>
</dependency>
```

и плагин:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <version>2.5.0</version>
  <executions>
    <execution>
      <id>xjc</id>
      <goals>
        <goal>xjc</goal>
```

```
      </goals>
    </execution>
  </executions>
  <configuration>
    <sources>
      <source>
        ${project.basedir}/src/main/resources
            /countries.xsd
      </source>
    </sources>
  </configuration>
</plugin>
```

Ссылка на POM файл.

В ресурсы поместить файл схемы XML (XSD), из которого при помощи плагина будет создан WSDL: ссылка.

Ссылка на проект. Компилируем его и переходим по ссылке.

Клиент

Так же создаем проект Spring (https://start.spring.io) с зависимостями Spring Web и Spring Web Services.

Для Java 11 добавляем профиль:

```
<profiles>
  <profile>
    <id>java11</id>
    <activation>
      <jdk>[11,)</jdk>
    </activation>
    <dependencies>
      <dependency>
        <groupId>org.glassfish.jaxb</groupId>
        <artifactId>jaxb-runtime</artifactId>
      </dependency>
    </dependencies>
  </profile>
```

```
</profiles>
```

Добавляем плагин:

```
<plugin>
  <groupId>org.jvnet.jaxb2.maven2</groupId>
  <artifactId>maven-jaxb2-plugin</artifactId>
  <version>0.14.0</version>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <schemaLanguage>WSDL</schemaLanguage>
    <generatePackage>
      localhost.devSpringSoapClient.wsdl
    </generatePackage>
    <schemas>
      <schema>
        <url>
          http://localhost:8080/ws/countries.wsdl
        </url>
      </schema>
    </schemas>
  </configuration>
</plugin>
```

Ссылка на POM файл.

Api уже генерируется в папке target. Ссылка на проект. Сначала компилируем, потом запускаем.

Источники

- Видео YouTube «SOAP Spring Boot Web Service на примере с нуля. SOAP UI тестирование», Artemy, 10.2021;

- Официальная документация Spring «Producing a SOAP web service»;
- Официальная документация Spring «Consuming a SOAP web service»;

Общий репозиторий

Ссылка на гит:

ссылка.

Источники

- Статья на хабре «Абстрактный CRUD от репозитория до контроллера: что ещё можно сделать при помощи Spring + Generics», xpendence, 10.2018.

Quarkus

dsfds

Vert.X

sdfsf

Micronaut

sdsdf

Desktop (JavaFX, Swing AWT)

sdfsdf

Web basic

JavaScript (React, Vue.js, Angular)

HTML5

CSS3 (Bootstrap)

Тестирование

Unit Testing (JUnit, Mockito, EasyMock)

Auto Testing (Selenium)

Integration Testing (Selenium, Robot)

Utils

Logs (Log4j, SLF4j)

Docs (JavaDoc)

Java Analyze (Heap Analyze, Thread Dump Analyzer)

Jira

GitLab/GitGub/BitBucked

AWS

HTTP (Postman)

Kubernetes

Урок 1. Знакомство с Kubernetes

Инструкции по доступу к кластеру закреплены в чате (курс давний, как найти?).

Kubernetes — оркестратор контейнеров, т. е. инструмент, который позволяет нам управлять множество контейнеров на разных серверах централизовано.

Почему Kubernetes:

- Kubernetes вырос из Google;
- удачные архитектурные решения;
- большое комьюнити (79 000 звезд на GitHub, 102 000 коммитов).

Преимущества Kubernetes:

- immutable (неизменяемость);
- declatative (есть два подхода: императивный и декларативный; императивный — говорим как именно надо выполнить задачу; декларативный — говорим что выполнить, а как — система знает сама; в декларативном подходе, в отличие от императивного, можно использовать обычные инструменты разработки, такие как git, unit тесты и т. д.);

- self-healing (умеет лечить себя; каждый компонент отвечает за свою часть всей инфраструктуры и постоянно поддерживает ее в актуальном состоянии);
- decoupling (каждый компонент инфраструктуры независим от других и полагается на их SLA).

Kubernetes очень сложный. Ну и зачем он разработчику?

- прогнозируемое поведение;
- удобство разработки;
- DevOps!!!
- Time-to-market и польза бизнесу.

Прогнозируемое поведение:

- Все аспекты взаимодействия с приложением контролируемы:
  - Абстракции, которые позволяют контролировать настройки работы
  - ... конфигурацию приложения;
  - ... хранение токенов и паролей;
  - ... внутреннее и внешнее сетевое взаимодействие;
  - ... автоскейлинг/stateful/etc.
- Возможность определить необходимые ресурсы вашему приложению (CPU, RAM).
- Healthchecks вашего приложения.

- Декларативность.

Удобство разработки:

- инструментарий разработчика (Minikube, Ksync, Telepresence, Skaffold);
- отладка (kubectl-debug, kubectl logs/describe);
- темплейтирование (helm, kustomize);
- быстрое поднятие окружения себе под разработку.

DevOps!!!

Пока разработка не знает, куда деплоить код, а инженеры не знают, для какого кода готовят инфраструктуру и строят пайпланы — никакого DevOps не будет!

Time-to-market и польза бизнесу:

- не нужно ждать адимнов;
- огромные возможности по автоматизации процессов деплоя/тестирования;
- возможность организовать необходимый уровень отказоустойчивости и SLO.

Kubernetes на «железках»

Плюсы:

- свое, родное;
- возможность гибкой кастомизации;

- возможность организовать нужный уровень безопасности.

Минусы:

- требуются серьезные компетенции;
- отсутствует или труднореализуем ряд фич;
- избыточно для небольших компаний.

Kubernetes на публичном облаке

Плюсы:

- львиную долю эксплуатации провайдер берет на себя;
- быстро разворачивается;
- множество уже встроенных удобных фич;
- отлично подходит для разработчиков.

Минусы:

- ограничены возможности кастомизации;
- не всегда очевидная ценовая политика;
- может быть куча всяких дополнительных абстракций и практик, что даже отдельные сертификации есть.

Kubernetes в приватном облаке

Включает в себя все указанные ранее плюсы, но главный минус — существенное усложнение инфраструктуры и эксплуатации. Сильно подумайте, нужно ли вам городить свое облако внутри компании.

Урок 2. Абстракции приложений

фва

asdfsadf

Источник

- Ссылка на репозиторий;
- Серия видео на YouTube «Открытая вечерняя школа. Kubernetes для разработчиков (Осень 2021)», Слёрм, 25.03.2022.

База данных

SQL (DML, DCL, TCL, DDL)

JDBC

Hibernate

NoSQL

БД (MySQL, MSSQL, PostgreSQL, MongoDB, ClickHouse)

Java Performance

JMH

Benchmarks

Optimizations

JIT

warmup

JVM (OpenJDK, Zulu, AdobeJDK, GraalVM)