Java企业级开发

Java企业级开发

1. 开发构建工具

1.1Nginx 的作用

- Nginx 是一个高性能的 HTTP 和反向代理 Web 服务器
- Nginx 的特点是：
  - 内存占用少
  - 并发能力强（可支持大约 50000 个并发连接）
  - 配置简洁
  - 服务稳定
- **反向代理**是 Nginx 作为 Web 服务器最常用的功能之一
  - **正向代理是代理客户端**，通过访问代理服务器，间接正常访问目的服务器（VPN）
  - **反向代理是代理服务器，让大量的请求均衡地访问到某一台服务器上
- 负载均衡
  - **Nginx 内置了轮询和加权轮询来达到负载均衡的目的**，不同服务器根据能力分配不同权重

- 动静分离
  - 软件开发中，有些请求是需要后台处理的；有些请求是不需要后台处理的，比如说 css、js 这些文件请求，这些不需要经过后台处理的文件就叫静态文件
  - 我们可以根据一些规则，把动态资源和静态资源分开，然后通过 Nginx 把请求分开，静态资源的请求就不需要经过 Web 服务器处理了，从而提高整体上的资源的响应速度

1.2 Nginx 的安装

- 针对不同的操作系统，Nginx 的安装各不相同
- 通过 `brew info nginx` 命令查看 Nginx 是否安装
- 通过 `brew install nginx` 命令安装 Nginx
  - 默认端口是 8080
- 通过 `nginx` 命令启动 Nginx
- 通过 `localhost:8080` 访问

1.3 Nginx 常用命令

- 一般Nginx 一旦启动后，我们是很少让它退出的，使用最多的就是 reload 命令。修改了配置文件，是需要执行一次 reload 命令让 Nginx 生效的

```
nginx  启动
nginx -s stop  停止
nginx -s quit  安全退出
nginx -s reload  重新加载配置文件
ps aux|grep nginx  查看nginx进程
```

1.4 Nginx 的配置

- Nginx 的配置结构图：

```
main        # 全局配置
├── events  # 配置网络连接
├── http    # 配置代理、缓存、日志等
|   ├── upstream # 配置负载均衡
|   ├── server   # 配置虚拟主机，可以有多个 server
|   ├── server
|   |   ├── location  # 用于匹配 URI（URL 是 URI 的一种），可以有多个 location
|   |   ├── location
|   |   └── ...
|   └── ...
└── ...
```

- Nginx 的默认配置

```
worker_processes  1; # Nginx 进程数，一般设置为和 CPU 核数一样

events {
```

```
    worker_connections  1024; # 每个进程允许最大并发数
}
http {
    include       mime.types; # 文件扩展名与类型映射表
    default_type  application/octet-stream;

    sendfile        on; # 开启高效传输模式
    keepalive_timeout  65; # 保持连接的时间，也叫超时时间，单位秒

    server {
        listen       8080; # 配置监听的端口
        server_name  localhost; # 配置的域名

        location / {
            root   html; # 网站根目录
            index  index.html index.htm; # 默认首页文件
        }

        error_page   500 502 503 504  /50x.html; # 默认50x对应的访问页面
        location = /50x.html {
            root   html;
        }
    }
    include servers/*; # 加载子配置项
}
```

1.5 Nginx面试问题

- Nginx如何处理HTTP请求的
  - 它结合多进程机制(单线程)和 异步非阻塞方式
  - 多进程机制(单线程)
    - 服务器每当收到一个客户端时，就有服务器主进程 (master process) 生成一个子进程 (worker process)出来和客户端建立连接进行交互，直到连接断开，该子进程就结束了
    - master 进程负责管理 Nginx 本身和其他 worker 进程，实际上的业务处理逻辑都在 worker 进程
    - 每一个Worker进程都维护一个线程(避免线程切换)，处理连接和请求。worker 进程中有一个函数，执行无限循环，不断处理收到的来自客户端的请求，并进行处理，直到整个 Nginx 服务被停止
  - 异步非阻塞机制
    - 每个工作进程使用异步非阻塞方式，可以处理多个客户端请求
    - 运用了epoll模型，提供了一个队列，排队解决。当某个工作进程接收到客户端的请求以后，调用 IO 进行处理，如果不能立即得到结果，就去处理其他请求(即为非阻塞)；而客户端在此期间也无需等待响应，可以去处理其他事情(即为异步)
  - 同步和异步

        - 同步是指在发送方发出消息后，需要等待接收到接收方发回的响应，或者通过回调函数来接收到对方响应信息
        - 异步是指在发送方发出请求后，不等待返回消息
    - 阻塞和非阻塞
        - 在网络通讯中，阻塞和非阻塞主要是指Socket的阻塞和非阻塞方式，而socket的实质是IO操作
        - 阻塞是指在IO操作返回结果之前，当前的线程处于被挂起状态，直到调用结果返回后才能处理其它新的请求
- nginx功能
    - 代理
        - 正向代理：特定情况下，代理用户访问服务器，需要用户手动的设置代理服务器的ip和端口号
        - 反向代理：是用来代理服务器，代理用户要访问的目标服务器。代理服务器接受请求，然后将请求转发给内部网络的服务器(服务集群模式)，并将从服务器上得到的结果返回给客户端，此时代理服务器对外就表现为一个服务器
        - Nginx在反向代理上，提供灵活的功能，可以根据不同的正则采用不同的转发策略
    - 负载均衡
        - Nginx可使用的负载均衡策略有：轮询（默认）、权重、ip_hash、url_hash(第三方)、fair(第三方)
    - 动静分离
        - 常用于前后端分离，把动态请求和静态请求分离开，合适的服务器处理相应的请求，使整个服务器系统的性能、效率更高
        - 常用于前后端分离，把动态请求和静态请求分离开，合适的服务器处理相应的请求，使整个服务器系统的性能、效率更高

1.6 Maven

- Maven，自动化构建，抽象构建过程，提供构建任务实现；跨平台，对外提供了一致的操作接口
- Maven 3 点优点
    - **依赖管理**：Maven 能帮助我们解决软件包依赖的管理问题，不再需要提交大量的 jar 包、引入第三方库
    - **规范目录结构**：Maven 标准的目录结构有助于项目构建的标准化，通过配置 profile 还可以根据不同的环境（开发环境、测试环境，生产环境）读取不同的配置文件
    - **方便集成**：能够集成在 IDE 中更方便使用

1.7 Maven 配置文件

- Maven 是基于 POM（Project Object Model） 进行的，项目的所有配置都会放在 pom.xml 文件中，包括项目的类型、名字，依赖关系，插件定制等等

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.itwanger</groupId>
    <artifactId>MavenDemo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>MavenDemo</name>
</project>
```

  - 第一行是XML头，指定了该xml文档的版本和编码方式
  - project 是根元素，声明了一些POM相关的命名空间及xsd元素
  - modelVersion指定了当前POM的版本，对于Maven 3来说，值只能是4.0.0
  - groupId定义了项目属于哪个组织，通常是组织域名的倒序
  - artifactId定义了项目在组织中的唯一ID
  - version指定了项目当前的版本，SNAPSHOT意为快照，说明该项目还处于开发中
  - name 声明了一个对于用户更为友好的项目名称

- groupId、artifactId和version这三个元素定义了一个项目的基本坐标，Maven任何的jar和pom都是以基于这些坐标进行区分的

```
<project>
...
<dependencies>
    <dependency>
        <groupId>实际项目</groupId>
        <artifactId>模块</artifactId>
        <version>版本</version>
        <type>依赖类型</type>
        <scope>依赖范围</scope>
        <optional>依赖是否可选</optional>
        <!-主要用于排除传递性依赖-->
        <exclusions>
            <exclusion>
                <groupId>…</groupId>
                <artifactId>…</artifactId>
            </exclusion>
        </exclusions>
   </dependency>
<dependencies>
...
</project>
```

  - scope 指定了依赖的范围（详情见下面**依赖范围**部分）。
  - optional 标记了依赖是否是可选的（详情见下面**依赖可选**部分）。

    - exclusions 用来排除传递性依赖（详情见下面**依赖排除**部分）。
- **依赖范围**有以下几种：
    - compile，默认的依赖范围，表示依赖需要参与当前项目的编译，后续的测试、运行周期也参与其中，是比较强的依赖。
    - test，表示依赖仅仅参与测试相关的工作，包括测试代码的编译和运行。比较典型的如 junit。
    - runntime，表示依赖无需参与到项目的编译，不过后期的测试和运行需要其参与其中。
    - provided，表示打包的时候可以不用包进去，别的容器会提供。和 compile 相当，但是在打包阶段做了排除的动作。
    - system，从参与程度上来说，和 provided 类似，但不通过 Maven 仓库解析，可能会造成构建的不可移植，要谨慎使用
- **依赖可选**
    - 项目中A依赖B，B依赖于X和Y，如果所有这三个的范围都是compile的话，那么X和Y就是A的compile范围的传递性依赖
- **依赖排除**：
    - 引入的依赖中包含不想要的依赖包，想引入自己想要的，这时候就要用到排除依赖
    - 声明exclustion的时候只需要groupId和artifactId，不需要version元素，因为groupId和artifactId就能唯一定位某个依赖

1.8 使用 Maven

- **Maven 常见命令**
    - `mvn clean`：表示运行清理操作（会默认把target文件夹中的数据清理）。
    - `mvn clean compile`：表示先运行清理之后运行编译，会将代码编译到target文件夹中。
    - `mvn clean test`：运行清理和测试。
    - `mvn clean package`：运行清理和打包。
    - `mvn clean install`：运行清理和安装，会将打好的包安装到本地仓库中，以便其他的项目可以调用。
    - `mvn clean deploy`：运行清理和发布（发布到私服上面）。
    - `mvn help:effective-settings`：查看 Maven 的有效配置信息。
- **Maven 常用 POM 属性**
    - `${project.build.sourceDirectory}`：项目的主源码目录，默认为`src/main/java/`
    - `${project.build.testSourceDirectory}`：项目的测试源码目录，默认为 `/src/test/java/`
    - `${project.build.directory}`：项目构建输出目录，默认为 `target/`
    - `${project.build.outputDirectory}`：项目主代码编译输出目录，默认为 `target/classes/`
    - `${project.build.testOutputDirectory}`：项目测试代码编译输出目录，默认为 `target/testclasses/`
    - `${project.groupId}`：项目的 groupId.
    - `${project.artifactId}`：项目的 artifactId.

- `${project.version}`：项目的 version，于 `${version}` 等价
- `${project.build.finalName}`：项目打包输出文件的名称，默认为 `${project.artifactId}${project.version}`

2. Git 和 Docker

2.1 Git 起源

- Git 是一个分布式版本控制系统，Git 最初的目的是为了能更好的管理 Linux 内核源码
- Git 这三种状态：
  - 已提交（committed），表示数据已经安全的保存在本地数据库中
  - 已修改（modified），表示修改了文件，但还没保存到数据库中
  - 已暂存（staged），表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中
- Git 的三个工作区域：
  - Git 仓库，用来保存项目的元数据和对象数据库
  - 工作目录，对项目的某个版本进行独立提取
  - 暂存区域，保存了下次将提交的文件列表信息，也可以叫“索引”
- Git 的工作流程：
  - 在工作目录中修改文件
  - 暂存文件，将文件的快照放入暂存区域
  - 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录

2.2 Git 的数据模型

- 尽管 Git 的接口有些难懂，但它底层的设计和思想却非常的优雅。难懂的接口只能靠死记硬背，但优雅的底层设计则非常容易理解。我们可以通过一种自底向上的方式来学习 Git，先了解底层的数据模型，再学习它的接口
- 快照
  - Git 将顶级目录中的文件和文件夹称作集合，并通过一系列快照来管理历史记录。
  - 在 Git 的术语中，文件被称为 blob 对象(数据对象)，也就是一组数据。目录则被称为 tree(树)，目录中可以包含文件和子目录
- 历史记录建模：关联快照
  - 在 Git 中，历史记录是一个由快照组成的有向无环图。这代表 Git 中的每个快照都有一系列的父辈，也就是之前的一系列快照。这些快照通常被称为“commit”
  - o 表示一次 commit，也就是一次快照。箭头指向了当前 commit 的父辈。在第三次 commit 之后，历史记录分叉成了两条独立的分支，这可能是因为要同时开发两个不同的特性，它们之间是相互独立的。开发完成后，这些分支可能会被合并为一个新的 commit，这个新的 commit 会同时包含这些特性

```
o <-- o <-- o <-- o <---- o
           ^            /
            \          v
             --- o <-- o
```

  - Git 中的 commit 是不可改变的。当然了，这并不意味着不能被修改，只不过这种“修改”实际上是创建了一个全新的提交记录

- 对象和内存寻址
  - Git 中的对象可以是 blob、tree 或者 commit：`type object = blob | tree | commit`
  - Git 在存储数据的时候，所有的对象都会基于它们的安全散列算法进行寻址
  - blob、tree 和 commit 一样，都是对象。当它们引用其他对象时，并没有真正在硬盘上保存这些对象，而是仅仅保存了它们的哈希值作为引用
- 仓库: 对象 和 引用
  - 在硬盘上，Git 仅存储对象和引用，因为其数据模型仅包含这些东西。所有的 git 命令都对应着对 commit 树的操作

2.3 Git 常用命令

- 新建代码库

```
# 在当前目录新建一个Git代码库
$ git init
# 新建一个目录，将其初始化为Git代码库
$ git init [project-name]
# 下载一个项目和它的整个代码历史
$ git clone [url]
```

- 配置

```
# 显示当前的Git配置
$ git config --list

# 编辑Git配置文件
$ git config -e [--global]

# 设置提交代码时的用户信息
$ git config [--global] user.name "[name]"
$ git config [--global] user.email "[email address]"
```

- 增加/删除文件

```
# 添加指定文件到暂存区
$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录
$ git add [dir]
```

```
# 添加当前目录的所有文件到暂存区
$ git add .

# 添加每个变化前，都会要求确认
# 对于同一个文件的多处变化，可以实现分次提交
$ git add -p

# 删除工作区文件，并且将这次删除放入暂存区
$ git rm [file1] [file2] ...

# 停止追踪指定文件，但该文件会保留在工作区
$ git rm --cached [file]

# 改名文件，并且将这个改名放入暂存区
$ git mv [file-original] [file-renamed]
```

- 代码提交

```
# 提交暂存区到仓库区
$ git commit -m [message]

# 提交暂存区的指定文件到仓库区
$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次commit之后的变化，直接到仓库区
$ git commit -a

# 提交时显示所有diff信息
$ git commit -v

# 使用一次新的commit，替代上一次提交
# 如果代码没有任何新变化，则用来改写上一次commit的提交信息
$ git commit --amend -m [message]

# 重做上一次commit，并包括指定文件的新变化
$ git commit --amend [file1] [file2] ...
```

- 分支

```
# 列出所有本地分支
$ git branch

# 列出所有远程分支
$ git branch -r

# 列出所有本地分支和远程分支
$ git branch -a

# 新建一个分支，但依然停留在当前分支
$ git branch [branch-name]
```

```
# 新建一个分支，并切换到该分支
$ git checkout -b [branch]

# 新建一个分支，指向指定commit
$ git branch [branch] [commit]

# 新建一个分支，与指定的远程分支建立追踪关系
$ git branch --track [branch] [remote-branch]

# 切换到指定分支，并更新工作区
$ git checkout [branch-name]

# 切换到上一个分支
$ git checkout -

# 建立追踪关系，在现有分支与指定的远程分支之间
$ git branch --set-upstream [branch] [remote-branch]

# 合并指定分支到当前分支
$ git merge [branch]

# 选择一个commit，合并进当前分支
$ git cherry-pick [commit]

# 删除分支
$ git branch -d [branch-name]

# 删除远程分支
$ git push origin --delete [branch-name]
$ git branch -dr [remote/branch]
```

- 标签

```
# 列出所有tag
$ git tag

# 新建一个tag在当前commit
$ git tag [tag]

# 新建一个tag在指定commit
$ git tag [tag] [commit]

# 删除本地tag
$ git tag -d [tag]

# 删除远程tag
$ git push origin :refs/tags/[tagName]

# 查看tag信息
$ git show [tag]
```

```
# 提交指定tag
$ git push [remote] [tag]

# 提交所有tag
$ git push [remote] --tags

# 新建一个分支，指向某个tag
$ git checkout -b [branch] [tag]
```

- 查看信息

```
# 显示有变更的文件
$ git status

# 显示当前分支的版本历史
$ git log

# 显示commit历史，以及每次commit发生变更的文件
$ git log --stat

# 搜索提交历史，根据关键词
$ git log -S [keyword]

# 显示某个commit之后的所有变动，每个commit占据一行
$ git log [tag] HEAD --pretty=format:%s

# 显示某个commit之后的所有变动，其"提交说明"必须符合搜索条件
$ git log [tag] HEAD --grep feature

# 显示某个文件的版本历史，包括文件改名
$ git log --follow [file]
$ git whatchanged [file]

# 显示指定文件相关的每一次diff
$ git log -p [file]

# 显示过去5次提交
$ git log -5 --pretty --oneline

# 显示所有提交过的用户，按提交次数排序
$ git shortlog -sn

# 显示指定文件是什么人在什么时间修改过
$ git blame [file]

# 显示暂存区和工作区的差异
$ git diff

# 显示暂存区和上一个commit的差异
$ git diff --cached [file]
```

```
# 显示工作区与当前分支最新commit之间的差异
$ git diff HEAD

# 显示两次提交之间的差异
$ git diff [first-branch]...[second-branch]

# 显示今天你写了多少行代码
$ git diff --shortstat "@{0 day ago}"

# 显示某次提交的元数据和内容变化
$ git show [commit]

# 显示某次提交发生变化的文件
$ git show --name-only [commit]

# 显示某次提交时，某个文件的内容
$ git show [commit]:[filename]

# 显示当前分支的最近几次提交
$ git reflog
```

- 远程同步

```
# 下载远程仓库的所有变动
$ git fetch [remote]

# 显示所有远程仓库
$ git remote -v

# 显示某个远程仓库的信息
$ git remote show [remote]

# 增加一个新的远程仓库，并命名
$ git remote add [shortname] [url]

# 取回远程仓库的变化，并与本地分支合并
$ git pull [remote] [branch]

# 上传本地指定分支到远程仓库
$ git push [remote] [branch]

# 强行推送当前分支到远程仓库，即使有冲突
$ git push [remote] --force

# 推送所有分支到远程仓库
$ git push [remote] --all
```

- 撤销

```
# 恢复暂存区的指定文件到工作区
$ git checkout [file]
```

```
# 恢复某个commit的指定文件到暂存区和工作区
$ git checkout [commit] [file]

# 恢复暂存区的所有文件到工作区
$ git checkout .

# 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变
$ git reset [file]

# 重置暂存区与工作区，与上一次commit保持一致
$ git reset --hard

# 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变
$ git reset [commit]

# 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致
$ git reset --hard [commit]

# 重置当前HEAD为指定commit，但保持暂存区和工作区不变
$ git reset --keep [commit]

# 新建一个commit，用来撤销指定commit
# 后者的所有变化都将被前者抵消，并且应用到当前分支
$ git revert [commit]

# 暂时将未提交的变化移除，稍后再移入
$ git stash
$ git stash pop
```

- 其他

```
# 生成一个可供发布的压缩包
$ git archive
```

2.4 Docker 简介

- Docker是一个用于 构建 运行 传送应用程序的平台
- Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化
- 容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）,更重要的是容器性能开销极低
- Docker 包括三个基本概念
    - 镜像（Image）：Docker 镜像（Image），就相当于是一个 root 文件系统。比如官方镜像 ubuntu:16.04 就包含了完整的一套 Ubuntu16.04 最小系统的 root 文件系统
    - 容器（Container）：镜像（Image）和容器（Container）的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等

    - 仓库（Repository）：仓库可看成一个代码控制中心，用来保存镜像
- Docker 使用客户端-服务器 (C/S) 架构模式，使用远程API来管理和创建Docker容器
    - 客户端(Client): docker run; docker build; docker pull
    - 服务端(Docker Host): Daemon; Container; Image
    - 仓库(Registry): Docker hub

3. Spring

3.1 Spring AOP

- AOP，也就是 Aspect-oriented Programming，译为面向切面编程，是计算机科学中的一个设计思想，旨在通过切面技术为业务主体增加额外的通知（Advice），从而对声明为“切点”（Pointcut）的代码块进行统一管理和装饰
- 这种思想非常适用于将那些与核心业务不那么密切关联的功能添加到程序中
    - 典型的案例 - 日志功能
- AOP 是对面向对象编程 (OOP) 的一种补充，OOP 的核心单元是类（class），而 AOP 的核心单元是切面（Aspect）
    - 利用 AOP 可以对业务逻辑的各个部分进行隔离，从而降低耦合度，提高程序的可重用性，同时也提高了开发效率

3.2 AOP 的相关术语

- **横切关注点**，从每个方法中抽取出来的同一类非核心业务
- **切面（Aspect)**: 对横切关注点进行封装的类，每个关注点体现为一个通知方法；通常使用 @Aspect 注解来定义切面
- **通知（Advice)**: 切面必须要完成的各个具体工作，比如我们的日志切面需要记录接口调用前后的时长，就需要在调用接口前后记录时间，再取差值。通知的方式有五种：
    - @Before：通知方法会在目标方法调用之前执行
    - @After：通知方法会在目标方法调用后执行
    - @AfterReturning：通知方法会在目标方法返回后执行
    - @AfterThrowing：通知方法会在目标方法抛出异常后执行
    - @Around：把整个目标方法包裹起来，在被调用前和调用之后分别执行通知方法
- **连接点（JoinPoint)**: 通知应用的时机，比如接口方法被调用时就是日志切面的连接点
- **切点（Pointcut)**: 通知功能被应用的范围，比如本篇日志切面的应用范围是所有 controller 的接口
    - 通常使用 @Pointcut 注解来定义切点表达式
    - `@Pointcut("execution(* com.codingmore.controller.*.*(..))")`
- **目标对象 (target)**：代理的目标对象

3.3 Spring IoC

- Inversion of Control 概念
  - 控制反转就是把创建和管理 Bean 的过程转移给了第三方，Spring IoC Container
  - 对于 IoC 来说，最重要的就是**容器。**容器负责创建、配置和管理 bean，也就是它管理着 bean 的生命，控制着 bean 的依赖注入
    - 因为项目中每次创建对象很麻烦，所以使用 Spring IoC 容器来管理这些对象
  - bean 是什么
    - Bean 是包装了的 Object，无论是控制反转还是依赖注入，它们的主语都是 object，bean 是由第三方包装好的 object
- IoC 能给我们带来什么好处
  - 解藕, 它把对象之间的依赖关系转成用配置文件来管理，由 Spring IoC Container 来管理
- IoC 容器
  - Spring 容器使用 `ApplicationContext`，它是 `BeanFactory` 的子类，更好的补充并实现了 `BeanFactory`
  - `BeanFactory` 简单粗暴，可以理解为 HashMap，一般只有 get, put 两个功能，所以称之为“低级容器”：
    - Key - bean name
    - Value - bean object
  - `ApplicationContext` 多了很多功能，因为它继承了多个接口，可称之为“高级容器”：
    - `ClassPathXmlApplicationContext` - 从 class path 中加载配置文件，更常用一些
    - `FileSystemXmlApplicationContext` - 从本地文件中加载配置文件，如果再到 Linux 环境中，还要改路径，不是很方便
- 深入理解 IoC
  - 用经典 `class Rectangle` 来举例：
    - 两个变量：长和宽
    - 自动生成 `set()` 方法和 `toString()` 方法
- **依赖注入 Dependency Injection**
  - Inversion of Control (IoC) is also known as Dependency Injection (DI 依赖注入)
    - IoC 是设计思想，DI 是具体的实现方式
    - IoC 是理论，DI 是实践
  - **依赖**：程序运行需要依赖外部的资源，提供程序内对象所需要的数据、资源
  - **注入**：配置文件把资源从外部注入到内部，容器加载了外部的文件、对象、数据，然后把这些资源注入给程序内的对象，维护了程序内外对象之间的依赖关系

3.5 Spring Framework 八大模块

- Data Access/Integration
  - JDBC
  - OXM
  - ORM
  - JMS
  - Transactions
- Web
  - WebSocket
  - Servlet
  - Web
  - Portlet
- AOP
- Aspects
- Instrumentation
- Messaging
- Core Container: IoC 容器, 核心
  - Beans
  - Core
  - Context
  - SpEL: spring express language
- Test

3.6 搭建Spring项目

- 创建springboot 项目with maven
- 创建bean class，添加注解@Component
- 创建Service layer，添加注解@Service，like DAO
- 创建Controller，链接Service ，添加注解@RestController
- 创建config，添加注解@Configutation，@ComponentScan
  - @ComponentScan会扫描package所有Component
- 创建main class，IoC Container `ApplicationContext context = new AnnotationConfigAplontext(Configclass):`
  - 得到某个bean `context.getBean(Car.class)`
- 搭建Spring项目

4. Spring Boot

4.1 搭建第一个Spring Boot项目

- Spring 官方提供了 Spring Initializr 的方式来创建 Spring Boot 项目
  - https://start.spring.io/
  - Project：项目的构建方式，可以选择 Maven（构建脚本基于XML） 和 Gradle（构建脚本基于 Groovy 或者 Kotlin 等语言）
  - Language：项目的开发语言，可以选择 Java、Kotlin (可以在 JVM 上运行的编程语言)、Groovy(可以作为 Java 平台的脚本语言)
  - Spring Boot：项目使用的 Spring Boot 版本
  - Project Metada：项目的基础设置，包括包名、打包方式(默认 Jar 包)、JDK 版本等
  - Dependencies：项目所需要的依赖和 starter, 默认是核心模块 spring-boot-starter 和测试模块 spring-boot-starter-test
- Spring Boot 项目结构
  - Spring Boot 项目的目录结构

```
.
├── ./idea
├── ./mvn
├── ./src
|   ├── ./src/main
|   |   ├── ./src/main/java      # 项目的开发目录，业务代码在这里写
|   |   ├── ./src/main/resources # 配置文件目录，静态文件、模板文件和配置文件都放在这里
|   |   |   ├── ./static    # 用于存放静态资源文件，比如说 JS、CSS 图片等
|   |   |   ├── ./templates # 用于存放模板文件，比如说 thymeleaf 和 freemarker 文件
|   ├── ./src/test
|   |   ├── ./src/test/java # 测试类文件目录
├── ./HELP.md
├── ./mvnw
└── ./mvnw.cmd
└── ./pom.xml # 用来管理项目的依赖和构建
```

  - 处理 Web 请求, 增加 Controller 文件

```
@Controller
public class HelloController {

    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "hello, springboot";
    }
}
```

  - 热部署

- 每次修改代码后，代码能够自动编译，服务能够自动重新加载
- 把spring-boot-devtools添加到项目当中后，无论是代码修改，还是配置文件修改，服务都能够秒级重载（俗称热部署）
- 在 pom.xml 文件中添加依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

4.2 整合 MySQL 和 Druid

- Spring Boot 整合 MySQL 数据库
  - Spring Boot 整合 MySQL 数据库非常简单，只需要添加 MySQL 依赖并在配置文件中添加数据库配置即可
  - 新建一个 Spring Boot 项目，添加 MySQL 的 Java连接驱动依赖和 JDBC Starter

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

- 通过 Navicat 连接 MySQL
  - 通过 Navicat 可以轻松连接数据库，并执行增删改查操作
  - 连接数据库也非常的简单，只需要填写主机 IP 地址、端口、用户名和密码即可。
- 通过 Intellij IDEA 连接 MySQL
  - 点击「database」面板，在左上角选择 + 号，选择 DataSource，再选择 MySQL。
  - 如果是第一次连接 MySQL 的话，记得点击「download」下载 MySQL 驱动，之后点击「test connection」测试是否链接成功

4.3 整合 JPA

4.4 整合 Thymeleaf 模板引擎

4.5 开启事务支持

4.6 过滤器、拦截器、监听器

- 各自作用
  - 过滤器（Filter）：当有一堆请求，只希望符合预期的请求进来
  - 拦截器（Interceptor）：想要干涉预期的请求
  - 监听器（Listener）：想要监听这些请求具体做了什么
- 区别
  - 过滤器是在请求进入容器后，但还没有进入 Servlet 之前进行预处理的
  - 拦截器是在请求进入控制器（Controller） 之前进行预处理的
  - 过滤器依赖于 Servlet 容器，而拦截器依赖于 Spring 的 IoC 容器，因此可以通过注入的方式获取容器当中的对象
  - 监听器用于监听 Web 应用中某些对象的创建、销毁、增加、修改、删除等动作，然后做出相应的处理
- 过滤器
  - 过滤敏感词汇（防止sql注入）；设置字符编码；URL级别的权限访问控制；压缩响应信息
  - 过滤器的创建和销毁都由 Web 服务器负责，Web 应用程序启动的时候，创建过滤器对象，为后续的请求过滤做好准备
  - 过滤器可以有很多个，一个个过滤器组合起来就成了 FilterChain，也就是过滤器链
  - 在 Spring 中，过滤器都默认继承了 OncePerRequestFilter，顾名思义，OncePerRequestFilter 的作用就是确保一次请求只通过一次过滤器，而不重复执行
  - 利用 Spring Initializr 来新建一个 Web 项目 codingmore-filter-interceptor-listener

```
@WebFilter(urlPatterns = "/*", filterName = "myFilter")
public class MyFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
        long start = System.currentTimeMillis();
        chain.doFilter(request,response);
        System.out.println("Execute cost="+(System.currentTimeMillis()-
start));
    }

    @Override
    public void destroy() {
        Filter.super.destroy();
    }
}
```

    - @WebFilter 注解用于将一个类声明为过滤器，urlPatterns 属性用来指定过滤器的 URL 匹配模式，filterName 用来定义过滤器的名字
    - MyFilter 过滤器的逻辑非常简单，重写了 Filter 的三个方法，在 doFilter 方法中加入了时间戳的记录
    - 在项目入口类上加上 @ServletComponentScan 注解，这样过滤器就会自动注册
- 拦截器
    - 登录验证，判断用户是否登录
    - 权限验证，判断用户是否有权限访问资源，如校验token
    - 日志记录，记录请求操作日志（用户ip，访问时间等），以便统计请求访问量
    - 处理cookie、本地化、国际化、主题等
    - 性能监控，监控请求处理时长等
    - 写一个简单的拦截器 LoggerInterceptor：

```
@Slf4j
public class LoggerInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        log.info("preHandle{}...",request.getRequestURI());
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        HandlerInterceptor.super.postHandle(request, response, handler,
modelAndView);
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws Exception
{
        HandlerInterceptor.super.afterCompletion(request, response, handler,
ex);
    }
}
```

    - 一个拦截器必须实现 HandlerInterceptor 接口，preHandle 方法是 Controller 方法调用前执行，postHandle 是 Controller 方法正常返回后执行，afterCompletion 方法无论 Controller 方法是否抛异常都会执行
    - 只有 preHandle 返回 true 的话，其他两个方法才会执行。如果 preHandle 返回 false 的话，表示不需要调用Controller方法继续处理了，通常在认证或者安全检查失败时直接返回错误响应
    - InterceptorConfig 对拦截器进行配置：

```
@Configuration
public class InterceptorConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new
LoggerInterceptor()).addPathPatterns("/**");
    }
}
```

- @Configuration 注解用于定义配置类，干掉了以往 Spring 繁琐的 xml 配置文件
- 编写一个用于被拦截的控制器 MyInterceptorController：

```
@RestController
@RequestMapping("/myinterceptor")
public class MyInterceptorController {
    @RequestMapping("/hello")
    public String hello() {
        return "沉默王二是傻X";
    }
}
```

- @RestController 注解相当于 @Controller + @ResponseBody 注解，@ResponseBody 注解用于将 Controller 方法返回的对象，通过适当的 HttpMessageConverter 转换为指定格式后，写入到 Response 对象的 body 数据区，通常用来返回 JSON 或者 XML 数据，返回 JSON 数据的情况比较多

- 监听器

4.7 整合 Redis 缓存

- Redis 是使用 C 语言开发的一个高性能键值对数据库，是互联网技术领域使用最为广泛的存储中间件，它是「Remote Dictionary Service」的首字母缩写，也就是「远程字典服务」
- Redis 数据类型
  - Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set 有序集合)
  - Redis 教程
  - string
    - string 是 Redis 最基本的数据类型，一个key对应一个value
    - set [key] [value] / get [key]
    - del [key]
    - EXISTS [key] / EXISTS [value]
  - hash
    - Redis hash 是一个键值对集合，值可以看成是一个 Map
    - hset [key] [field] [value] 将哈希表 key 中的字段 field 的值设为 value

        - hmset [key] [field1] [value1] [field2] [value2] 同时将多个 field-value (域-值)对设置到哈希表 key 中
        - hkeys [hashName] / hvals [hashName] 获取哈希表中所有字段/ 所有值
        - hlen [key] 获取哈希表中字段的数量
        - hget [key] [field] 获取存储在哈希表中指定字段的值
        - hgetall [key]获取在哈希表中指定 key 的所有字段和值
    - list
        - list 是一个简单的字符串列表，按照插入顺序排序, left是头，right是尾
        - lpush [key] [value1] [value2] 将一个或多个值插入到列表头部
        - rpush [key] [value1] [value2] 将一个或多个值插入到列表尾部
        - lrange [key] [start] [stop] 获取列表指定范围内的元素
        - lpop [key] 移出并获取列表的第一个元素
        - rpop [key] 移除并获取列表最后一个元素
    - set
        - set 是 string 类型的无序集合，不允许有重复的元素
        - sadd [key] [member1] [member2]
        - smembers [key] 返回集合中的所有成员
        - srem [key] [member1] [member2] 移除集合中一个或多个成员
        - sdiff [key1] [key2] / sinter [key1] [key2] 返回给定所有集合的差集 / 交集
    - sorted set
        - sorted set 是 string 类型的有序集合，不允许有重复的元素, 不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序. 有序集合的成员是唯一的,但分数(score)却可以重复
        - zadd [key] [score1] [member1] [score2] [member2] 向有序集合添加一个或多个成员，或者更新已存在成员的分数
        - zrem [key] [member1] [member2] 移除有序集合中的一个或多个成员
        - zrange [key] [start] [stop] [WITHSCORES] 通过索引区间返回有序集合成指定区间内的成员
        - zrangebyscore [key] [min] [max] [WITHSCORES] [LIMIT] 通过分数返回有序集合指定区间内的成员
        - zrevrange [key] [start] [stop] [WITHSCORES] 返回有序集中指定区间内的成员，通过索引，分数从高到底
        - zrevrangebyscore [key] [min] [max] [WITHSCORES] [LIMIT] 返回有序集中指定分数区间内的成员，分数从高到低排序
- Spring Boot 和 Redis
    - 在 pom.xml 文件中添加 Redis 的 starter

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

- 在 application.yml 文件中添加 Redis 的配置信息

```yaml
spring:
  redis:
    host: xxx.xxx.99.232 # Redis服务器地址
    database: 0 # Redis数据库索引（默认为0）
    port: 6379 # Redis服务器连接端口
    password: xxx # Redis服务器连接密码（默认为空）
```

- 测试代码

```java
public void testRedis() {
    // 添加
    redisTemplate.opsForValue().set("name","沉默王二");
    // 查询
    System.out.println(redisTemplate.opsForValue().get("name"));
    // 删除
    redisTemplate.delete("name");
    // 更新
    redisTemplate.opsForValue().set("name","沉默王二的狗腿子");
    // 查询
    System.out.println(redisTemplate.opsForValue().get("name"));

    // 添加
    stringRedisTemplate.opsForValue().set("name","沉默王二");
    // 查询
    System.out.println(stringRedisTemplate.opsForValue().get("name"));
    // 删除
    stringRedisTemplate.delete("name");
    // 更新
    stringRedisTemplate.opsForValue().set("name","沉默王二的狗腿子");
    // 查询
    System.out.println(stringRedisTemplate.opsForValue().get("name"));
}
```

- RedisTemplate 和 StringRedisTemplate 都是 Spring Data Redis 提供的模板类，可以对 Redis 进行操作，后者针对键值对都是 String 类型的数据，前者可以是任何类型的对象
- RedisTemplate 和 StringRedisTemplate 除了提供 opsForValue 方法来操作字符串之外，还提供了以下方法
  - opsForList：操作 list
  - opsForSet：操作 set
  - opsForZSet：操作有序 set
  - opsForHash：操作 hash

4.8 整合 Logback 定制日志框架

4.9 整合Swagger-UI实现在线API文档

4.10 整合Knife4j，美化强化丑陋的Swagger

4.11 整合 Spring Task 实现定时任务

4.12 整合Quartz实现编程喵定时发布文章

4.13 整合 MyBatis

- 介绍
  - Object Relational Mapping (对象关系映射 ORM) 框架的本质是简化操作数据库的编码工作，常用的框架有两个: 一个是可以灵活执行动态 SQL 的 MyBatis；一个是崇尚不用写 SQL 的 Hibernate
    - Hibernate 的特点是所有的 SQL 通过 Java 代码生成，发展到最顶端的是 Spring Data JPA，根据方法名就可以生成对应 SQL
    - MyBatis 早些时候用起来比较繁琐，需要各种配置文件，需要实体类和 DAO 的映射关联，经过不断地演化和改进，可以通过 generator 自动生成实体类、配置文件和 DAO 层代码，简化了不少开发工作
  - 经过 MyBatis-Plus 的增强，开发者只需要简单的配置，就可以快速进行单表的 CRUD 操作；同时，MyBatis-Plus又提供了代码生成、自动分页、逻辑删除、自动填充等丰富功能，进一步简化了开发工作
- 整合 MyBatis
  - 在 pom.xml 文件中引入 starter

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.2.2</version>
</dependency>
```

  - 在 application.yml 文件中添加数据库连接配置

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password: Huicheng123**
    url: jdbc:mysql://localhost:3306/codingmore-mybatis
```

  - 新建 User.java 实体类
    - User 包含 id; age; name; password
  - 新建 UserMapper.java 接口:

```
public interface UserMapper {
    @Select("SELECT * FROM user")
    List<User> getAll();

    @Select("SELECT * FROM user WHERE id = #{id}")
    User getOne(Integer id);

    @Insert("INSERT INTO user(name,password,age) VALUES(#{name}, #
{password}, #{age})")
    void insert(User user);

    @Update("UPDATE user SET name=#{name},password=#{password},age=#{age}
WHERE id =#{id}")
    void update(User user);

    @Delete("DELETE FROM user WHERE id =#{id}")
    void delete(Integer id);
}
```

- @Select 注解用来查询
- @Insert 注解用来插入
- @Update 注解用来修改
- @Delete 注解用来删除

- 在启动类 CodingmoreMybatisApplication 上添加 @MapperScan 注解来扫描 mapper
  - 如果没有指定 @MapperScan 的扫描路径，将从声明该注解的类的包开始进行扫描。

```
@SpringBootApplication
@MapperScan
public class CodingmoreMybatisApplication {
    public static void main(String[] args) {
        SpringApplication.run(CodingmoreMybatisApplication.class, args);
    }

}
```

- 通过 MyBatis-Plus 增强
  - MyBatis 属于半自动的 ORM 框架，实现一些简单的 CRUD 也是需要编写 SQL 语句
  - MyBatis-Plus提供了诸多优秀的特性:
    - 强大的 CRUD 操作：内置了通用的 mapper、service，可通过少量的配置实现大部分常用的 CRUD，不用再编写 SQL 语句
    - 支持主键自动生成
    - 支持 ActiveRecord 模式：实体类只需继承 Model 类即可进行强大的 CRUD 操作
    - 可快速生成 Mapper 、 Model 、 Service 、 Controller 层代码
    - 内置分页插件
    - 内置性能分析插件：可输出 SQL 语句以及其执行时间

- 整合 MyBatis-Plus
  - 在 pom.xml 文件中添加 MyBatis-Plus 的 starter

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.2</version>
</dependency>
```

  - 新建例子PostTag 实体类，mp自动做映射关联

```
@Data
public class PostTag {
    private Long postTagId;
    private String description;
}
```

  - 新建 PostTagMapper 继承 BaseMapper，无需编写 mapper.xml 文件，即可获得CRUD功能

```
public interface PostTagMapper extends BaseMapper<PostTag> {}
```

4.14 一键部署 Spring Boot 到远程 Docker 容器

- https://tobebetterjavaer.com/springboot/docker.html#%E5%AE%89%E8%A3%85-docker

4.15 Spring Boot 面试题

Spring AOP

- 通过运行期动态代理的方式实现不修改源代码的情况下给程序动态统一添加功能的技术，针对业务处理过程中的切面进行提取，将日志、事务、异常处理等代码从业务逻辑代码中划分出来
  - 使得业务逻辑各部分之间的耦合度降低
  - 提高程序的可重用性
  - 提高了开发的效率
- AOP核心概念
  - 横切关注点：对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点
  - 切面 aspect：类是对物体特征的抽象，切面就是对横切关注点的抽象
  - 连接点 joinpoint：被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器
  - 切入点 pointcut：对连接点进行拦截的定义
  - 通知 advice：指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知五类
  - 目标对象 target：代理的目标对象
  - 织入 weave：将切面应用到目标对象并导致代理对象创建的过程

- AOP框架将自动生成AOP代理，即：代理对象的方法=增强处理+被代理对象的方法
- Spring 方面可以使用下面提到的五种通知工作
    - 前置通知: 在一个方法执行之前，执行通知。
    - 后置通知: 在一个方法执行之后，不考虑其结果，执行通知。
    - 返回后通知: 在一个方法执行之后，只有在方法成功完成时，才能执行通知。
    - 抛出异常后通知: 在一个方法执行之后，只有在方法退出抛出异常时，才能执行通知。
    - 环绕通知: 在建议方法调用之前和之后，执行通知

```
@Aspect
@Component
public class LogAspect {
    // 前置通知
    @Before("webLog()")
    public void deBefore(JoinPoint joinPoint) throws Throwable {

 System.out.println("ARGS : " + Arrays.toString(joinPoint.getArgs()));
    }

    // 切入点
    @Pointcut("execution(public * com.example.controller.*.*(..))")
    public void webLog(){}

    // 返回后通知
    @AfterReturning(returning = "ret", pointcut = "webLog()")
    public void doAfterReturning(Object ret) throws Throwable {
        System.out.println("方法的返回值 : " + ret);
    }
    // 后置异常通知
    @AfterThrowing("webLog()")
    public void throwss(JoinPoint jp){
        System.out.println("方法异常时执行.....");
    }
    // 后置最终通知,final增强，不管是抛出异常或者正常退出都会执行
    @After("webLog()")
    public void after(JoinPoint jp){
        System.out.println("方法最后执行.....");
    }
```

IoC 控制反转

- 控制反转是在系统运行中，动态的向某个对象提供它所需要的其他对象，这一点是通过依赖注入来实现的
- 反射允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring是通过反射来实例化对象，存入bean容器，再注入
- IoC好处：降低耦合；传统应用程序都是由程序员在类内部主动创建依赖对象，从而导致类与类之间高耦合。而IoC是将对象创建的控制权交由spring

Spring依赖注入三种方式

- 基于有参构造函数的依赖关系注入
  - argument constructor is called

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }  //有参构造
}
```

- 基于 Setter 的依赖项注入
  - no-argument constructor and setter is called

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    public SimpleMovieLister{}; //无参构造
    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

- 变量注入 Field Injection
  - 直接在变量上注解 @Autowired, 会有个警告提示 Field injection is not recommended
  - 创建Person（被注入对象）要实现的接口：

```
interface UserInject{
    void injectUser(User user);//这里必须是被注入对象依赖的对象
}
class Person implements UserInject{
    @Autowired
    private User user;
    public Person(){}
    @Override
    public void injectUser(User user){
        this.user = user;//实现注入方法，外部通过此方法给此对象注入User对象
    }
}
```

- 三种区别
  - Constructor Injection
    - 注入对象可以使用final修饰；防止NullPointerException；
    - 避免循环依赖，如果存在循环依赖，spring项目启动的时候就会报错；
    - 构造函数的代码臃肿

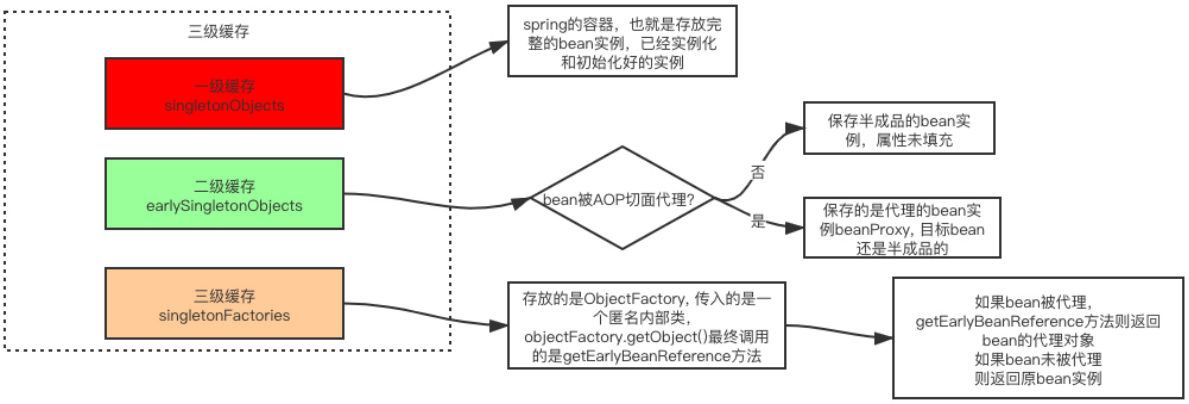
    - Setter Injection
        - 依赖注入中使用的依赖是可选的，选择依赖的意思是注入的依赖是可以为 NULL
        - 允许在类构造完成后重新注入；
        - 注入对象不能使用final修饰；
    - Field Injection
        - 注入方式简单，非常简洁
        - 注入对象不能用final修饰；可能会导致循环依赖，不会报错；
- 使用场景
    - 如果注入的属性是必选的属性，则通过构造器注入；
    - 如果注入的属性是可选的属性，则通过setter方法注入；
    - 至于field注入，不建议使用；

Spring循环依赖

- A依赖B的同时B也依赖了A

```
@Component
public class A {
    @Autowired
    private B b; // A中注入了B
}
@Component
public class B {
    @Autowired
    private A a;// B中也注入了A
}
```

- Spring实例化一个bean的时候，是分两步进行的，首先实例化目标bean，然后为其注入属性
- Spring在实例化一个bean的时候，是首先递归的实例化其所依赖的所有bean，直到某个bean没有依赖其他bean，此时就会将该实例返回，然后反递归的将获取到的bean设置为各个上层bean的属性的



- Spring通过三级缓存解决了循环依赖 Circular Dependency，其中一级缓存为单例池（singletonObjects）,二级缓存为早期曝光对象earlySingletonObjects，三级缓存为早期曝光对象工厂（singletonFactories）
    - 当A、B循环引用时，在A完成实例化后，就使用实例化后的对象去创建一个对象工厂，并添加到三级缓存中，如果A被AOP代理，那么通过这个工厂获取到的就是A代理后的对象，如果A没有被AOP代理，那么这个工厂获取到的就是A实例化的对象
    - 当A进行属性注入时，会去创建B，同时B又依赖了A，所以创建B的同时又会去调用getBean(a)来获取需要的依赖，此时的getBean(a)会从缓存中获取：先获取到三级缓存中的工厂，再调用对象工工厂的getObject获取对象，将它放到二级缓存并移除三级缓存并返回，B完成属性装配，一个完整的对象放到一级缓存singletonObjects中
- 为什么要三级缓存呢？二级缓存能解决循环依赖?
    - 能解决，但如果bean被代理则不行
    - 假如只有二级缓存：
        - ab循坏依赖，a被代理；a先被实例化，（事先spring无法知道是否有循环依赖）然后放入二级缓存时，有两种选择：
        - 1、提前创建好代理对象，将代理对象放入缓存（这样违背了spring对bean的设计原则，设计之初是让bean在生命周期最后一步完成代理而不是实例化后就马上被代理）
        - 2、将a放入缓存（这是错的，这样的话b从二级缓存获取到的是bean的实例而不是代理）
    - 所以引入第三级缓存；一开始a被实例化的时候，包装成ObjectFactory放入第三级缓存。if没有被别人循环引用注入，就不会提前生成代理；else，实时生成代理对象，并将代理对象放入第二级缓存
- setter和构造器注入谁产生的循环依赖系统没法解决？构造器
    - Spring文档建议的一种方式是使用setter注入。当依赖最终被使用时才进行注入。另一种解决构造器循环依赖的方式：通过@Lazy延迟加载。在注入依赖时，先注入代理对象，当首次使用时再创建对象完成注入

Springboot常用注解

- @SpringBootApplication
    - Spring Boot最核心的注解，用在 Spring Boot的主类上，标识这是一个 Spring Boot 应用，用来开启 Spring Boot 的各项能力
- @ComponentScan
    - 组件扫描。让spring Boot扫描到Configuration类并把它加入到程序上下文
- @Repository
    - 用于标注数据访问组件，即DAO组件。使用@Repository注解可以确保DAO或者repositories提供异常转译, 这个注解修饰的DAO或者repositories类会被ComponetScan发现并配置
- @Component: 泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注
- @Service: 一般用于修饰service层的组件
- @RequestMapping
    - RequestMapping是一个用来处理请求地址映射的注解；提供路由信息，负责URL到Controller中的具体函数的映射

    - GetMapping搭配@PathVariable或@RequestParam
    - @PathVariable 将url中{xx}绑定到输入参数
    - @RequestParam 用在方法的参数前面 age = 35 , `@RequestParam String a = request.getParameter("age")`
- @ResponseBody
    - 表示该方法的返回结果直接写入HTTP response body中
    - 一般在异步获取数据时使用，在使用@RequestMapping后，返回值通常解析为跳转路径，加上@responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中，将java对象转为json格式的数据返回给前端
- @Autowired
    - 对类成员变量、方法及构造函数进行标注，完成自动装配的工作；是spring的注解，默认使用的是byType的方式注入相应的Bean。有参数required=false意为如果找不到就不注入
- @Qualifier
    - 当有多个同一类型的Bean时，可以用@Qualifier("name")来指定。与@Autowired配合使用
- @CrossOrigin 跨域
    - 跨域，是指浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对JavaScript实施的安全限制
    - @CrossOrigin 利用spring的拦截器实现往response的header里添加 Access-Control-Allow-Origin等响应头信息

事务注解失效的情况

- 事务注解失效的情况
- 同一个类内部调用
    - 类里面的内部调用，使用的是实例对象本身去去调用方法B，非aop的cglib代理对象调用，方法B不会进入到切面
    - 把自身（这个类）注入到自身（这时候的自身就是spring代理过的），用注入的这个自身去调用本方法
- 抛出的异常被事务方法内部捕捉，没有抛到方法外,就没法回滚了

Springboot启动流程

- 注册bean到spring容器。通过不同的条件、方式来完成：
    - @SpringBootApplication由三个重要注解组成：
        - @ SpringBootConfiguration：用于定义配置类，可替换 xml 配置文件，被注解的类内部包含有一个或多个被 @Bean 注解的方法，并用于构建 Bean 定义，初始化 Spring 容器
        - @EnableAutoConfiguration启用自动配置，可以帮助 SpringBoot 应用将所有符合条件的 @Configuration 配置都加载到当前 SpringBoot 创建并使用的 IoC 容器
        - @ComponentScan：用于将一些标注了特定注解的 bean 定义批量采集注册到 Spring 的 IoC 容器之中，这些特定的注解大致包括：@Controller，@Entity，@Component，@Service，@Repository

- SpringApplication 的构造过程以及其 run() 方法的流程
  - 获取并创建 SpringApplicationRunListener 对象(是一个接口，可以监听springboot应用启动过程中的一些生命周期事件，并做一些处理)

```
public interface SpringApplicationRunListener {
    // 在run()方法开始执行时，该方法就立即被调用，可用于在初始化最早期时做一些工
作
    void starting();
    // 当environment构建完成，ApplicationContext创建之前，该方法被调用
    void environmentPrepared(ConfigurableEnvironment environment);
    // 当ApplicationContext构建完成时，该方法被调用
    void contextPrepared(ConfigurableApplicationContext context);
    // 在ApplicationContext完成加载，但没有被刷新前，该方法被调用
    void contextLoaded(ConfigurableApplicationContext context);
    // 在ApplicationContext刷新并启动后，CommandLineRunners和
ApplicationRunner未被调用前，该方法被调用
    void started(ConfigurableApplicationContext context);
    // 在run()方法执行完成前该方法被调用
    void running(ConfigurableApplicationContext context);
    // 当初始化失败后该方法被调用
    void failed(ConfigurableApplicationContext context, Throwable excep
tion);
```

  - 准备环境:环境就是管理配置应用和系统属性，如java版本、jvm名称、properties文件里的
  - 准备上下文：注册了一批PostProcessor
  - 刷新上下文：初始化BeanFactory, bean实例化

Springboot中的配置文件

- application.propertities 和 application.yml

```
server.port=8004 #服务端口
spring.application.name=service-cms  #服务名
# mysql数据库连接
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/guli?serverTimezone=GMT%2B8
spring.datasource.username=root
spring.datasource.password=123456
```

- 内容格式比较：
  - .properties文件，通过.来连接，通过=来赋值，结构上，没有分层的感觉，但比较直接。
  - .yml文件，通过：来分层，结构上，有比较明显的层次感，最后key赋值的：后需要留一个空格
- 执行顺序
  - 如果工程中同时存在application.properties文件和 application.yml文件，yml文件会先加载，而后加载的properties文件会覆盖yml文件
  - 所以建议工程中，只使用其中一种类型的文件即可

SpringBoot 的自动配置和原理

- SpringBoot在Spring Framework的基础上增加的自动配置(约定优于配置)特性能够让开发人员更少的关注底层而带来更快的开发速度
- Maven的目录结构
  - 默认有resources文件夹存放配置文件
  - 默认打包方式为jar
- 默认的配置文件：application.properties 或 application.yml 文件
  - Spring Boot启动的时候会通过@EnableAutoConfiguration注解找到META-INF/spring.factories配置文件中的所有自动配置类，并对其进行实例化、加载，而这些自动配置类都是以AutoConfiguration结尾来命名的
  - 自动配置类 xxxAutoConfiguration 负责使用xxxProperties 中属性进行自动配置，而xxxProperties(搭配@ConfigurationProperties注解)则负责将属性与配置文件application.properties的相关配置进行绑定，以便于用户通过配置文件修改默认的自动配置

Springboot启动时自动执行方法

- 将要执行的方法所在的类交给spring容器扫描(@Component),并且在要执行的方法上添加@PostConstruct注解或者静态代码块执行

```
@Component
public class Test2{
    //静态代码块会在依赖注入后自动执行,并优先执行
    static {
        System.out.println("--static--");
    }
    //@Postcontruct'在依赖注入完成后自动调用
    @PostConstruct
    public static void haha() {
        System.out.println("@Postcontruct'在依赖注入完成后自动调用");
    }
}
```

Spring如何创建对象

- 采用默认的无参构造创建实例

```
<bean id="user" class="ioc.pojo.User"></bean>
public void testUser() {

ApplicationContext context new ClassPathXmLApplication Context("classpathUser us
er =(User) context.getBean("user")
    System. out println("默认的无参构造创建:"+user);
}
```

- 采用静态工厂创建实例
  - 配置工厂类,XML配置

```
<bean id="user1" class="ioc.service.UserFactory" factory-method="getUser1">
</bean>
public void testUser(){
    ApplicationContext context new ClassPathXmLApplication Context("classpath
    User user1 =(User) context.getBean("user1")
    System.out.println("静态工厂创建:"+user1);
}
```

- 工厂
  - 静态工厂：工厂本身不用创建对象，通过静态方法调用，对象类=Factory.getInstance()
  - 实例工厂：工厂本身需要创建对象工厂类

```
public class UserFactory{
    public User getUser2(){
        return new User();
    }
}
<bean id="userFactory" class="ioc.service.UserFactory"></bean>
<bean id="user2" factory-bean="userFactory" factory-method="getUser2">
</bean>
public void testUser(){

ApplicationContext context new ClassPathXmLApplication Context("classpathUse
r user2 =(User) context.getBean("user2")
    System.out.println("实例工厂创建:"+user1);
}
```

BeanFactory和ApplicationContext

- BeanFactory
  - 是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系
  - BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化
- ApplicationContext
  - ApplicationContext继承BeanFactory接口，应用上下文，它是Spring的一各更高级的容器，提供了更多的有用的功能
  - ApplicationContext在启动的时候就把所有的Bean全部实例化了。它还可以为Bean配置lazy-init = true来让Bean延迟实例化
  - **懒加载机制**可以规定指定bean不在启动时立即创建，而是后续第一次用到时才创建，从而减轻在启动过程中对时间和内存的消耗

Bean生命周期

- 实例化bean对象(通过构造方法或者工厂方法)
- 设置对象属性 (依赖注入)
- 如果Bean实现了BeanNameAware接口，工厂调用 `setBeanName()` 方法传递Bean的ID
  - **BeanNameAware**接口是为了让自身Bean能够感知到，获取到自身在Spring容器中的id属性
- 如果Bean实现了BeanFactoryAware接口，工厂调用 `setBeanFactory()` 方法传入工厂自身
  - 同理其他的Aware接口也是为了能够感知到自身的一些属性
  - 实现了**ApplicationContextAware**接口的类，能够获取到ApplicationContext；
  - 实现了**BeanFactoryAware**接口的类，能够获取到BeanFactory对象
- 将Bean实例传递给Bean的前置处理器的 `postProcessBeforeInitialization()` 方法（这个机制使得在初始化对象之前对注册到容器中的BeanDefinition的存储信息进行修改。可以根据这个机制对Bean增加其它信息、修改Bean定义的某些属性值。）
  - BeanPostProcessor：bean的加工器，在bean的实例化过程中对bean做一些包装处理
- 调用Bean的初始化方法，如@PostConstruct, @Bean(initMethod=”myMethod()”
- 将Bean实例传递给Bean的后置处理器的postProcessAfterInitialization方法，在这一步进行代理
- 使用Bean
- 容器关闭 shutdown container 之前，调用Bean的销毁方法 `destroy()`

Spring bean的作用域 scope

- 两个__:
  - singleton：创建applicationContextC容器时自动创建一个bean的对象，每次获取到的对象都是同一个对象
  - prototype：每次调用都会创建一个新的bean
- 接下来是三个web环境下的，搭配springMVC使用
  - request：请求作用域，每次用到这个bean来处理HTTP请求的时候会创建一个bean实例。请求完成后销毁这个bean。request域下的bean 是存放在HttpServletRequest中，每个请求的HttpServletRequest均不同。threadLocal相关
  - session：会话作用域：session是服务器和浏览器的一次会话过程，是连续的多次请求。session结束后销毁，session中所有http请求共享同一个请求的bean实例
  - application全局作用域：是ServletContext级别的、整个web项目全局共享的。单例是作用在applicationContext（一个容器）当中的，一个项目可能有多个applicationContext

Spring 拓展接口

- aware系列，BeanNameAware、ApplicationContextAware

  BeanPostProcessor：方法会在bean的初始化前后被调用

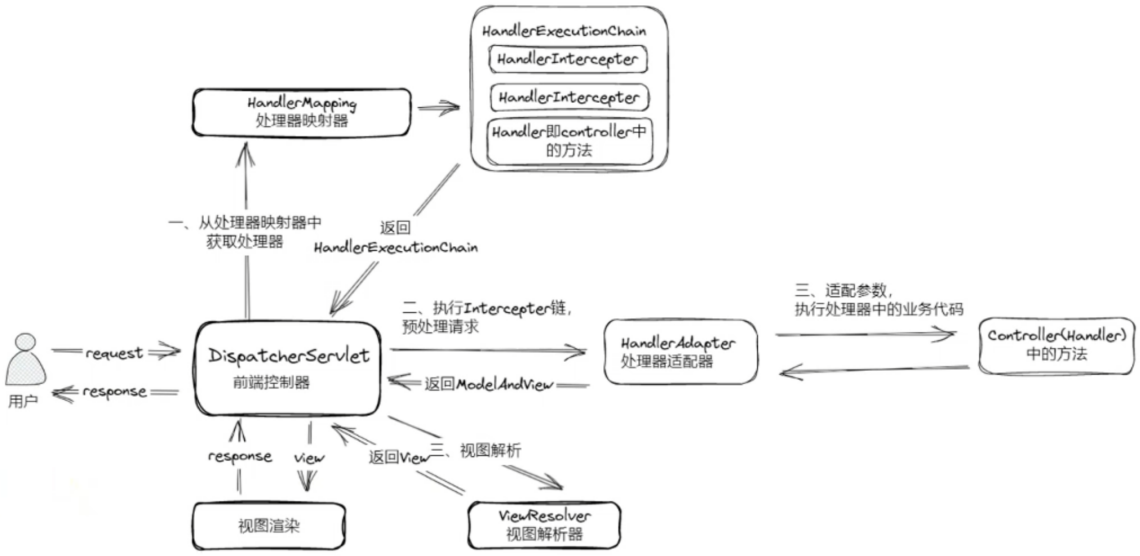
  FactoryBean: 是个接口，可以自定义Bean的创建过程

SpringBoot advantages

- spring boot starter
- auto configuration
- annotation
- default web server: tomcat
- flexible way to configure
- rest endpoints
- acuator

Spring MVC

- MVC 是一种使用 Model View Controller 模型-视图-控制器设计创建 Web 应用程序的模式
- Spring MVC是spring的一个独立的模块
    - 模型表示应用程序的数据模型, 视图表示应用程序的用户界面, 控制器表示应用程序的请求处理逻辑



- 关键组件
    - DispatcherServlet 前端控制器
        - 作用：接收请求，响应结果，相当于转发器，中央处理器。有了DispatcherServlet减少了其它组件之间的耦合度
    - HandlerMapping 处理器映射器
    - HandlerAdapter 处理器适配器
    - ViewResolver 视图解析器
- 处理流程
    - 接收请求：
        - servlet容器(tomcat和jetty等)接收到来自浏览器的request
        - 调用应用的Servlet的service()方法,最终会调到DispatcherServlet的doService()方法
    - 寻找处理器：

        - DispatcherServlet会从维护的HandlerMapping列表中寻找合适的controller (Handler Execution Chain处理器链)
        - handler即controller中的方法，对请求进行加工和拦截，”基于注解的应用会用RequestMappingHandlerMapping“
    - 调用拦截器：
        - 执行处理器链中的一系列拦截器，如果被拦截则不进行后续的处理
        - 调用被@RequestMapping注解后包装成的HttpMethod
    - 调用处理器：
        - 没有被拦截的请求，使用HandlerAdapter 处理request 中的请求参数，转换成controller接收的参数
        - 调用@RequestMapping方法的具体实现执行controller中的业务逻辑
    - 获取处理结果：
        - 将上一步的处理结果封装成ModelAndView返回给DispatcherServlet
    - 视图映射处理：
        - 调用ViewResolver将ModelAndView解析成View (JSONView,PDFView等)
    - 数据传给View:
        - 使用View的 render渲染方法将model放入response并响应给浏览器

Tomcat、servlet、spring之间关系

- Tomcat是springboot内置的Web应用服务器
    - 它的作用是解析客户端client发起的request，并封装HttpServletRequest，交由dispatchSetvlet处理
- Servlet是一个接口
- SpringMVC使用一个DispatcherServlet来接收所有的请求，并把它们分发到不同的controller中来做进一步处理

Tomcat处理请求原理、IO模型

- Tomcat用线程池处理请求，多路复用使一个线程处理多个连接，默认线程池maxThreads = 200
- Tomcat多路复用IO
    - NIO：这里的多路是指N个连接，每一个连接对应一个channel，或者说多路就是多个channel
    - 复用：是指多个连接复用了一个线程或者少量线程
- Tomcat打破双亲委派机制

5. Netty

6. 分布式

Elasticsearch

6.1 Elasticsearch 是什么

- Elasticsearch 是一个分布式、RESTful 风格的搜索和数据分析引擎，能够解决不断涌现出的各种用例
- Elastic Stack 是什么
  - Elasticsearch 是 Elastic Stack 的核心

6.2 安装 Elasticsearch

- Elasticsearch 是由 Java 开发的，现在版本中内置了 Java 环境，所以可以直接下载
  - bin 目录下是一些脚本文件，包括 Elasticsearch 的启动执行文件
  - config 目录下是一些配置文件
  - jdk 目录下是内置的 Java 运行环境
  - lib 目录下是一些 Java 类库文件
  - logs 目录下会生成一些日志文件
  - modules 目录下是一些 Elasticsearch 的模块。
  - plugins 目录下可以放一些 Elasticsearch 的插件
- 双击 bin 目录下的 elasticsearch.bat 文件就可以启动 Elasticsearch 服务
  - `http://localhost:9200` 进行查看

6.3 安装 Kibana

- 通过安装 Kibana，我们可以对 Elasticsearch 服务进行可视化操作，就像在 Linux 操作系统下安装一个图形化界面一样
- 双击 bin 目录下的 kibana.bat 文件，就可以启动 Kibana 服务，`[Kibana][http] http server running` 说明服务启动成功
  - `http://localhost:5601` 查看 Kibana 的图形化界面

6.4 Elasticsearch 的关键概念

- Elasticsearch 中的几个关键概念
  - MysQL -> ElasticSearch
  - Database(数据库) -> Index(索引)
  - Table(表) -> Type(类型)
  - Row(行) -> Document(文档)
  - Column(列) -> Field(字段)
  - Schema(方案) -> Mapping(映射)
  - Index(索引) -> Everthing Indexed by default(所有字段都被索引)
  - SQL(结构化查询语言) -> Query DSL(查询专用语言)

6.5 在 Java 中使用 Elasticsearch

- 在项目中添加 Elasticsearch 客户端依赖

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
    <version>7.6.2</version>
</dependency>
```

- 新建测试类 ElasticsearchTest

```
public class ElasticsearchTest {
    public static void main(String[] args) throws IOException {
        // RestHighLevelClient 为 Elasticsearch 提供的 REST 客户端
        // 可以通过 HTTP 的形式连接到 Elasticsearch 服务器，参数为主机名和端口号
        // 可以向 Elasticsearch 服务器端发送请求并获取响应
        RestHighLevelClient client = new RestHighLevelClient(
                RestClient.builder(
                        new HttpHost("localhost", 9200, "http")));
        // IndexRequest 用于向 Elasticsearch 服务器端添加一个索引，参数为索引关键字，比
如说“writer”
        // 还可以指定 id。通过 source 的方式可以向当前索引中添加文档数据源（键值对的形式）
        // IndexRequest 对象可以调用客户端的 index() 方法向 Elasticsearch 服务器添加索
引
        IndexRequest indexRequest = new IndexRequest("writer")
                .id("1")
                .source("name", "沉默王二",
                        "age", 18,
                        "memo", "一枚有趣的程序员");
        IndexResponse indexResponse = client.index(indexRequest,
RequestOptions.DEFAULT);

        // GetRequest 用于向 Elasticsearch 服务器端发送一个 get 请求，参数为索引关键字，
以及 id
        // 可以调用客户端的 get() 方法向 Elasticsearch 服务器获取索引
        GetRequest getRequest = new GetRequest("writer", "1");
        // getSourceAsString() 用于从响应中获取文档数据源（JSON 字符串的形式）
        GetResponse getResponse = client.get(getRequest,
RequestOptions.DEFAULT);
        String sourceAsString = getResponse.getSourceAsString();

        System.out.println(sourceAsString);
        client.close();
    }
}
```

6.6 ES 面试题

es 优缺点

- 优点
  - 面向文档，存储复杂结构对象 (对比面向行记录)
  - 倒排索引，搜索, 全文检索，每一个字段都可以检索
  - 分片机制，负载读请求，高可用
- 缺点
  - 创建好mapping后不能修改字段的类型

es 文档数据类型

- es的文档是以 **JSON** 进行序列化并保存. 文档通过其 *index*、type、_id 唯一确定。Id可以自己指定，也可以依托es生成
- text类型：全文索引
  - 会进行分词，分词后建立索引
  - 支持模糊查询，支持准确查询
  - 不支持聚合查询
- keyword类型
  - 不分词，直接建立索引
  - 支持模糊查询，支持准确查询
  - 支持聚合查询
- numeric 数字类型，这类数据类型都是以确切值索引的,long integer,float,etc
- date日期
- range范围
- array数组类型
- object对象类型（可嵌套），会扁平化，失去原有的映射关系：user{name, age} -> user.name / user.age

es 节点

- 节点就是ElasticSearch 实例，通常一个节点运行在一个隔离的容器或虚拟机中，一个节点存储多个分片
  - 主节点：创建或删除索引、节点分片的分配
  - 数据节点：存储数据、增删改查、聚合操作

es 分片

- 一个分片就是一个luence实例
- index包含多个shard，每个shard都是一个最小工作单元，承载部分数据，lucene实例，完整的建立索引和处理请求的能力

- 每个document只存在于某一个primary shard以及其对应的replica shard中。primary shard的数量在创建索引的时候就固定了，replica shard的数量可以随时修改
    - replica shard是primary shard的副本，负责容错，以及承担读请求负载
    - 写的时候只往 primary shard 写，然后 primary 同步到 replica 副本
    - primary shard的默认数量是5，每个primary的replica默认是1，故默认有10个shard
    - primary shard不能和自己的replica shard放在同一个节点上（否则节点宕机，primary shard和副本都丢失，起不到容错的作用），但是可以和其他primary shard的replica shard放在同一个节点上
- 当主分片异常时，副本可以提供数据的查询等操作

es Search和GET流程

- es的读取分为GET和Search两种操作
- GET根据 **索引index & 文档id** 从**正排索引**中获取内容；Search不指定id，根据关键字从**倒排索引**中获取内容
- GET属于基本document API；而Search是search API，底层是用http的get/post把请求给es
- Search:
    - 查询时不知道文档位于哪个分片，因此索引的所有分片都要参与搜索
    - 客户端发送请求到一个coordinate node协调节点
    - 协调节点将搜索请求转发到所有的分片（主分片、replica分片）
    - query phase：每个shard按照过滤、排序等条件进行分片粒度的文档id检索，将自己的搜索结果（其实就是一些doc id）返回给协调节点，由协调节点进行数据的合并、排序、分页等操作，产出最终结果
    - fetch phase：由协调节点根据doc id去各个节点上拉取实际的document数据，最终返回给客户端
- Get:
    - 客户端向集群中的某个节点发送读取请求，该节点就作为本次请求的协调节点；
    - 协调节点使用文档ID来确定文档属于某个分片，再通过集群状态中的内容路由表信息获知该分片的副本信息，此时它可以把请求转发到有副分片的任意节点读取数据。
    - 协调节点会将客户端请求轮询发送到集群的所有副本来实现负载均衡。
      收到读请求的节点将文档返回给协调节点，协调节点将文档返回给客户端

es 写流程

- 客户端向集群中的某个节点发送写请求，该节点就作为本次请求的协调节点
- 协调节点使用文档ID来确定文档属于某个分片，再通过集群状态中的内容路由表信息获知该分片的主分片位置，将请求转发到主分片所在节点
- 主分片节点上的主分片执行写操作。如果写入成功，则它将请求并行转发到副分片所在的节点，等待副分片写入成功。所有副分片写入成功后，主分片节点向协调节点报告成功，协调节点向客户端报告成功
- 底层：
    - 先写入内存buffer，同时写入translog日志（translog是追加写入，性能好）

    - 每个segment都是一个倒排索引
    - 默认每隔一秒，将buffer进行refresh，数据以segment file形式进入文件系统cache（此时数据可以被搜索到了）
    - translog 的作用：持久化数据。每隔一段时间或者translog过大，执行fsync
        - 所有在内存缓冲区的文档都被写入一个新的段
        - 缓冲区被清空
        - 一个提交点被写入硬盘
        - 文件系统缓存通过 fsync 被刷新（flush）
        - 老的 translog 被删除

ElasticSearch 是实时的么

- 不是，因为要先写到内存的一个 buffer 里然后过一段时间再进入os cache

Segment段合并

- 由于每秒会创建一个新的段，导致段数量太多。更重要的是，每个搜索请求都必须轮流检查每个段；所以段越多，搜索也就越慢
- Elasticsearch通过在后台进行段合并来解决这个问题。小的段被合并到大的段，然后这些大的段再被合并到更大的段
- 段合并的时候会将那些旧的已删除文档从文件系统中清除。被删除的文档（或被更新文档的旧版本）不会被拷贝到新的大段中

ES分页

- from+size: from指定查询的起始位置，size表示从起始位置开始的文档数量.当起始位置深的时候效率低。因为es是基于分片，如果from=90, size=10，则会从五个分片各取100个，然后对这500进行排序得到最后10个
- search after利用实时游标来解决实时滚动的问题。前一次查询的结果会返回一个唯一的字符串，下次查询带上这个字符串，进行下一页的查询
- scroll api:创建一个快照，有新的数据写入以后，无法被查到。每次查询后，输入上一次的 scroll_id。目前官方已经不推荐使用这个API，使用search_after即可

6.7 API网关基础

- 什么是API网关
    - **API网关是一个服务器，是系统的唯一入口。** 从面向对象设计的角度看，它与外观模式类似
    - API网关封装了系统内部架构，为每个客户端提供一个定制的API。它可能还具有其它职责，如身份验证、监控、负载均衡、缓存、协议转换、限流熔断、静态响应处理
    - **API网关方式的核心要点是，所有的客户端和消费端都通过统一的网关接入微服务**，在网关层处理所有的非业务功能。通常，网关也是提供REST/HTTP的访问API
- 网关的主要功能
    - 微服务网关作为微服务后端服务的统一入口，它可以统筹管理后端服务，主要分为数据平面和控制平面

- 数据平面主要功能是接入用户的HTTP请求和微服务被拆分后的聚合。使用微服务网关统一对外暴露后端服务的API和契约，路由和过滤功能正是网关的核心能力模块。另外，微服务网关可以实现拦截机制和专注跨横切面的功能，包括协议转换、安全认证、熔断限流、灰度发布、日志管理、流量监控等
- 控制平面主要功能是对后端服务做统一的管控和配置管理。例如，可以控制网关的弹性伸缩；可以统一下发配置；可以对网关服务添加标签；可以在微服务网关上通过配置Swagger功能统一将后端服务的API契约暴露给使用方，完成文档服务，提高工作效率和降低沟通成本

6.8 API网关选型

- 常用API网关
  - Nginx
    - Nginx是一个高性能的HTTP和反向代理服务器。**Nginx一方面可以做反向代理，另外一方面可以做静态资源服务器，接口使用Lua动态语言可以完成灵活的定制功能**
    - Nginx 在启动后，会有一个 Master 进程和多个 Worker 进程，Master 进程和 Worker 进程之间是通过进程间通信进行交互的，如图所示。Worker 工作进程的阻塞点是在像 select()、epoll_wait() 等这样的 I/O 多路复用函数调用处，以等待发生数据可读 / 写事件。Nginx 采用了异步非阻塞的方式来处理请求，也就是说，Nginx 是可以同时处理成千上万个请求的
  - Zuul
    - Zuul 是 Netflix 开源的一个API网关组件，它可以和 Eureka、Ribbon、Hystrix 等组件配合使用。社区活跃，融合于 SpringCloud 完整生态，是构建微服务体系前置网关服务的最佳选型之一
    - Zuul 的核心是一系列的过滤器，这些过滤器可以完成以下功能
      - 统一鉴权 + 动态路由 + 负载均衡 + 压力测试
      - 审查与监控：与边缘位置追踪有意义的数据和统计结果，从而带来精确的生产视图。
      - 多区域弹性：跨越 AWS Region 进行请求路由，旨在实现 ELB（Elastic Load Balancing，弹性负载均衡）使用的多样化，以及让系统的边缘更贴近系统的使用者
    - Zuul 目前有两个大的版本：**Zuul1 和 Zuul2**
    - Zuul1 是基于 Servlet 框架构建，如图所示，采用的是阻塞和多线程方式，即一个线程处理一次连接请求，这种方式在内部延迟严重、设备故障较多情况下会引起存活的连接增多和线程增加的情况发生
    - Netflix 发布的 Zuul2 有重大的更新，它运行在异步和无阻塞框架上，每个 CPU 核一个线程，处理所有的请求和响应，请求和响应的生命周期是通过事件和回调来处理的，这种方式减少了线程数量，因此开销较小
  - Spring Cloud GetWay
    - Spring Cloud Gateway 是Spring Cloud的一个全新的API网关项目，目的是为了替换掉Zuul1，它基于Spring5.0 + SpringBoot2.0 + WebFlux（基于高性能的Reactor模式响应式通信框架Netty，异步非阻塞模型）等技术开发，性能高于Zuul，官方测试，**Spring Cloud GateWay是Zuul的1.6倍**，旨在为微服务架构提供一种简单有效的统一的API路由管理方式
    - Spring Cloud Gateway可以与Spring Cloud Discovery Client（如Eureka）、Ribbon、Hystrix等组件配合使用，实现路由转发、负载均衡、熔断、鉴权、路径重写、日志监控等，并且Gateway还内置了限流过滤器，实现了限流的功能

    - Kong
        - Kong是一款基于OpenResty（Nginx + Lua模块）编写的高可用、易扩展的，由Mashape公司开源的API Gateway项目。Kong是基于NGINX和Apache Cassandra或PostgreSQL构建的，能提供易于使用的RESTful API来操作和配置API管理系统，所以它可以水平扩展多个Kong服务器，通过前置的负载均衡配置把请求均匀地分发到各个Server，来应对大批量的网络请求
        - Kong主要有三个组件：
            - Kong Server ： 基于Nginx的服务器，用来接收API请求
              Apache Cassandra/PostgreSQL ： 用来存储操作数据
              Kong dashboard：官方推荐UI管理工具，也可以使用 restfull 方式管理admin api
        - Kong采用插件机制进行功能定制，插件集（可以是0或N个）在API请求响应循环的生命周期中被执行。插件使用Lua编写，目前已有几个基础功能：HTTP基本认证、密钥认证、CORS（Cross-Origin Resource Sharing，跨域资源共享）、TCP、UDP、文件日志、API请求限流、请求转发以及Nginx监控
        - Kong网关具有以下的特性：
            - 可扩展性: 通过简单地添加更多的服务器，可以轻松地进行横向扩展，这意味着您的平台可以在一个较低负载的情况下处理任何请求；
            - 模块化: 可以通过添加新的插件进行扩展，这些插件可以通过RESTful Admin API轻松配置；
            - 在任何基础架构上运行: Kong网关可以在任何地方都能运行。您可以在云或内部网络环境中部署Kong，包括单个或多个数据中心设置，以及public，private 或invite-only APIs
        - Traefik
            - Træfɪk 是一个为了让部署微服务更加便捷而诞生的现代HTTP反向代理、负载均衡工具。 它支持多种后台 (Docker, Swarm, Kubernetes, Marathon, Mesos, Consul, Etcd, Zookeeper, BoltDB, Rest API, file...) 来自动化、动态的应用它的配置文件设置
            - 重要特性：
                - 它非常快，无需安装其他依赖，通过Go语言编写的单一可执行文件；
                - 多种后台支持：Docker, Swarm, Kubernetes, Marathon, Mesos, Consul, Etcd；
                - 支持支持Rest API、Websocket、HTTP/2、Docker镜像；
                - 监听后台变化进而自动化应用新的配置文件设置；
                - 配置文件热更新，无需重启进程；
                - 后端断路器、负载均衡、容错机制；
                - 清爽的前端页面，可监控服务指标
- API网关对比
    - 主要关注Kong、Traefik和Zuul即可
    - 从开源社区活跃度来看，无疑是Kong和Traefik较好；
    - 从成熟度来看，较好的是Kong、Tyk、Traefik；
    - 从性能来看，Kong要比其他几个领先一些；

- 从架构优势的扩展性来看，Kong、Tyk有丰富的插件，Ambassador也有插件但不多，而Zuul是完全需要自研，但Zuul由于与Spring Cloud深度集成，使用度也很高，近年来Istio服务网格的流行，Ambassador因为能够和Istio无缝集成也是相当大的优势

7. 消息队列

消息队列