# 敏捷开发：

是开发出核心的版本，后序不断的迭代完成升级， 敏捷开发的实现主要包括 SCRUM、XP（极限编程）、Crystal Methods、FDD（特性驱动开发）等等。其中 SCRUM 与 XP 最为流行。 同样是敏捷开发，XP 极限编程 更侧重于实践，并力求把实践做到极限。这一实践可以是测试先行，也可以是结对编程等，关键要看具体的应用场景。 SCRUM 则是一种开发流程框架，也可以说是一种套路。SCRUM 框架中包含三个角色，三个工件，四个会议，听起来很复杂，其目的是为了有效地完成每一次迭代周期的工作。在这里我们重点讨论的是 SCRUM

```
### 极限编程：
```

项目组针对客户端代表提出用户故事进行讨论，提出隐喻，项目中的在隐喻和用户故事基础上根据设定的优先级来交付计划，接下来迭代过程。迭代过程中不受开发过程干扰。 指定迭代计划----》站立会议---》代码共享编程---》最新版本。

## 持续集成：

持续集成指的是，频繁地（一天多次）将代码集成到主干。它的好处主要有两个： 快速发现错误。每完成一点更新，就集成到主干，可以快速发现错误，定位错误也比较容易。 防止分支大幅偏离主干。如果不是经常集成，主干又在不断更新，会导致以后集成的难度变大，甚至难以集成。Martin Fowler 说过，"持续集成并不能消除 Bug，而是让它们非常容易发现和改正 流程： 提交--测试--构建--测试（二）--部署---回滚

## 持续交付：

频繁的将软件的新版本，交付给质量团队或用户，以供评审，评审通过进入生产阶段。 持续交付可以看作持续集成的下一步。它强调的是，不管怎么更新，软件是随时随地可以交付的持续交付在持续集成的基础上，将集成后的代码部署到更贴近真实运行环境的「类生产环境」（production-like environments）中

## 持续部署：

持续部署（continuous deployment）是持续交付的下一步，指的是代码通过评审以后，自动部署到生产环境。 持续部署的目标是，代码在任何时刻都是可部署的，可以进入生产阶段。 持续部署的前提是能自动化完成测试、构建、部署等步骤

## 使用gitlab持续集成：

从 GitLab 8.0 开始，GitLab CI 就已经集成在 GitLab 中，我们只要在项目中添加一个 .gitlab-ci.yml 文件，然后添加一个 Runner，即可进行持续集成。 而且随着 GitLab 的升级，GitLab CI 变得越来越强大一次 Pipeline 其实相当于一次构建任务，里面可以包含多个流程，如安装依赖、运行测试、编译、部署测试服务器、部署生产服务器等流程。

```
    #### Stages:
```

表示构建阶段，说白了就是上面提到的流程。我们可以在一次 Pipeline 中定义多个 Stages： 一个stages接一个stages运行，只有所有的stages完成pipline的任务才会成功。如果任何一个 Stage 失败，那么后面的 Stages 不会执行，则任务失败。

```
   #### jobs:
```

表示构建工作。表示某个stage里面执行的工作，可以在stage中定义多个jobs.相同的stage中的jobs会并行执行，并且jobs都执行成功后才会成功，

## GitLab Runner:

一般来说，构建任务都会占用很多的系统资源 (譬如编译代码)，而 GitLab CI 又是 GitLab 的一部分，如果由 GitLab CI 来运行构建任务的话，在执行构建任务的时候，GitLab 的性能会大幅下降。 GitLab CI 最大的作用是管理各个项目的构建状态，因此，运行构建任务这种浪费资源的事情就交给 GitLab Runner 来做拉！ 因为 GitLab Runner 可以安装到不同的机器上，所以在构建任务运行期间并不会影响到 GitLab 的性能

```
##### 安装Flitlab Runner：
```

curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-ci-multi-runner/script.deb.sh | sudo bash  sudo apt-get update  sudo apt-get install gitlab-ci-multi-runner
**注册runner：** gitlab-ci-multi-runner register **命令：** gitlab-ci-multi-runner register：执行注册命令 Please enter the gitlab-ci coordinator URL：输入 ci 地址 Please enter the gitlab-ci token for this runner：输入 ci token Please enter the gitlab-ci description for this runner：输入 runner 名称 Please enter the gitlab-ci tags for this runner：设置 tag Whether to run untagged builds：这里选择 true ，代码上传后会能够直接执行 Whether to lock Runner to current project：直接回车，不用输入任何口令 Please enter the executor：选择 runner 类型，这里我们选择的是 shell **将gitlab-runner账户加入root组：** 安装完 GitLab Runner 后系统会增加一个 gitlab-runner 账户，我们将它加进 root 组： gpasswd -a gitlab-runner root 配置需要操作目录的权限，比如你的 runner 要在 gaming 目录下操作： chmod 775 gaming 由于我们的 shell 脚本中有执行 git pull 的命令，我们直接设置以 ssh 方式拉取代码： su gitlab-runner ssh-keygen -t rsa -C "你在 GitLab 上的邮箱地址" cd cd .ssh cat id_rsa.pub 复制 id_rsa.pub 中的秘钥到 GitLab： 通过 ssh 的方式将代码拉取到本地

```
        删除注册信息：

        gitlab-ci-multi-runner unregister --name "名称"
        查看注册列表：

        gitlab-ci-multi-runner list
```

## 测试集成效果

```
.gitlab-ci.yml:
这里面将配置依次分为五个阶段：
    安装依赖(install_deps)， 运行测试(test)，编译(build)，部署测试服务器(deploy_test)，部署生产服务器(deploy_production)
    设置 Job.only 后，只有当 develop 分支和 master 分支有提交的时候才会触发相关的 Jobs。

    节点说明：

    stages：定义构建阶段，这里只有一个阶段 deploy
    deploy：构建阶段 deploy 的详细配置也就是任务配置
    script：需要执行的 shell 脚本
    only：这里的 master 指在提交到 master 时执行
    tags：与注册 runner 时的 tag 匹配
    配置详情：
    stages:
        - install_deps
        - test
        - build
```

```
    - deploy_test
    - deploy_production

    cache:
    key: ${CI_BUILD_REF_NAME}
    paths:
        - node_modules/
        - dist/

    # 安装依赖
    install_deps:
    stage: install_deps
    only:
        - develop
        - master
    script:
        - npm install

    # 运行测试用例
    test:
    stage: test
    only:
        - develop
        - master
    script:
        - npm run test

    # 编译
    build:
    stage: build
    only:
        - develop
        - master
    script:
        - npm run clean
        - npm run build:client
        - npm run build:server

    # 部署测试服务器
    deploy_test:
    stage: deploy_test
    only:
        - develop
    script:
        - pm2 delete app || true
        - pm2 start app.js --name app

    # 部署生产服务器
    deploy_production:
    stage: deploy_production
    only:
        - master
    script:
        - bash scripts/deploy/deploy.sh
```

## druid:

Druid是一个关系型数据库连接池，它是阿里巴巴的一个开源项目。Druid支持所有JDBC兼容的数据库，包括Oracle、MySQL、Derby、PostgreSQL、SQL Server、H2等。Druid在监控、可扩展性、稳定性和性能方面具有明显的优势。通过Druid提供的监控功能，可以实时观察数据库连接池和SQL查询的工作情况。使用Druid连接池，在一定程度上可以提高数据库的访问性能。 Spring Boot的数据源配置的默认类型是org.apache.tomcat.jdbc.pool.DataSource，为了使用Druid连接池，可以将数据源类型更改为com.alibaba.druid.pool.DruidData-Source， 如代码清单4-2所示。其中，url、username、password是连接MySQL服务器的配置参数，其他一些参数设定Druid的工作方式。

```
        spring:
           datasource:
               type: com.alibaba.druid.pool.DruidDataSource
               driver-class-name: com.mysql.jdbc.Driver
               url: jdbc:mysql:// localhost:3306/test?characterEncoding=utf8
               username: root
               password: 12345678
               # 初始化大小，最小，最大
               initialSize: 5
               minIdle: 5
               maxActive: 20
               # 配置获取连接等待超时的时间
               maxWait: 60000
               # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
               timeBetweenEvictionRunsMillis: 60000
               # 配置一个连接在池中的最小生存时间，单位是毫秒
               minEvictableIdleTimeMillis: 300000
               validationQuery: SELECT 1 FROM DUAL
               testWhileIdle: true
               testOnBorrow: false
               testOnReturn: false
               # 打开PSCache，并且指定每个连接上PSCache的大小
               poolPreparedStatements: true
               maxPoolPreparedStatementPerConnectionSize: 20
               # 配置监控统计拦截的filters，去掉后监控界面sql将无法统计，'wall'用于防火
墙
               filters: stat,wall,log4j
               # 通过connectProperties属性来打开mergeSql功能；慢SQL记录
               connectionProperties:
druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000
               # 合并多个DruidDataSource的监控数据
               #useGlobalDataSourceStat=true
```

上面配置中的filters：stat表示已经可以使用监控过滤器，这时结合定义一个过滤器，就可以用来监控数据库的使用情况。 [插图]注意　在Spring Boot低版本的数据源配置中，是没有提供设定数据源类型这一功能的，这时如果要使用上面这种配置方式，就需要使用自定义的配置参数来实现。

```
    @Configuration
    public class DruidConfiguration {
        @Bean
        public ServletRegistrationBean statViewServle(){
            ServletRegistrationBean servletRegistrationBean = new ServletRegist
    rationBean(new StatViewServlet(),"/druid/*");
            // IP白名单
            servletRegistrationBean.addInitParameter("allow","192.168.1.218,127.
    0.0.1");
            // IP黑名单 (共同存在时，deny优先于allow)
```

```
            servletRegistrationBean.addInitParameter("deny","192.168.1.100");
            // 控制台管理用户
            servletRegistrationBean.addInitParameter("loginUsername","druid");

servletRegistrationBean.addInitParameter("loginPassword","12345678");
            // 是否能够重置数据
            servletRegistrationBean.addInitParameter("resetEnable","false");
            return servletRegistrationBean;
        }
        @Bean
        public FilterRegistrationBean statFilter(){
            FilterRegistrationBean filterRegistrationBean = new
FilterRegistrationBean
    (new WebStatFilter());
            // 添加过滤规则
            filterRegistrationBean.addUrlPatterns("/*");
            // 忽略过滤的格式

filterRegistrationBean.addInitParameter("exclusions","*.js,*.gif,*.jpg,*.
    png,*.css,*.ico,/druid/*");
            return filterRegistrationBean;
            }
    }
```

开启监控功能后，运行应用时，可以通过网址http://localhost/durid/index.html打开控制台，输入上面程序设定的用户名和密码，登录进去，可以打开如图4-2所示的监控后台。

在监控后台中，可以实时查看数据库连接池的情况，每一个被执行的SQL语句使用的次数和花费的时间、并发数等，以及一个URI请求的次数、时间和并发数等情况。这就为分析一个应用系统访问数据库的情况和性能提供了可靠、详细的原始数据，让我们能在一些基础的细节上修改和优化一个应用访问数据库的设计。

Durid在spring中的配置：

```
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
destroy-method="close">
        <property name="url" value="${jdbc_url}" />
        <property name="username" value="${jdbc_user}" />
        <property name="password" value="${jdbc_password}" />

        <property name="filters" value="stat" />

        <property name="maxActive" value="20" />
        <property name="initialSize" value="1" />
        <property name="maxWait" value="60000" />
        <property name="minIdle" value="1" />

        <property name="timeBetweenEvictionRunsMillis" value="60000" />
        <property name="minEvictableIdleTimeMillis" value="300000" />

        <property name="validationQuery" value="SELECT 'x'" />
        <property name="testWhileIdle" value="true" />
        <property name="testOnBorrow" value="false" />
        <property name="testOnReturn" value="false" />
        <property name="poolPreparedStatements" value="true" />
        <property name="maxPoolPreparedStatementPerConnectionSize" value="20" />
    </bean>
```

# itoken项目：

## itoken使用的是微服务架构：

itoken-config项目： 配置中心： itoken-dependencies:依赖配置 itoken-eureka：服务注册与发现 itoken-zipkin:服务链路追踪 itoken-zuul:服务路由网关 itoken-admin:监控中心

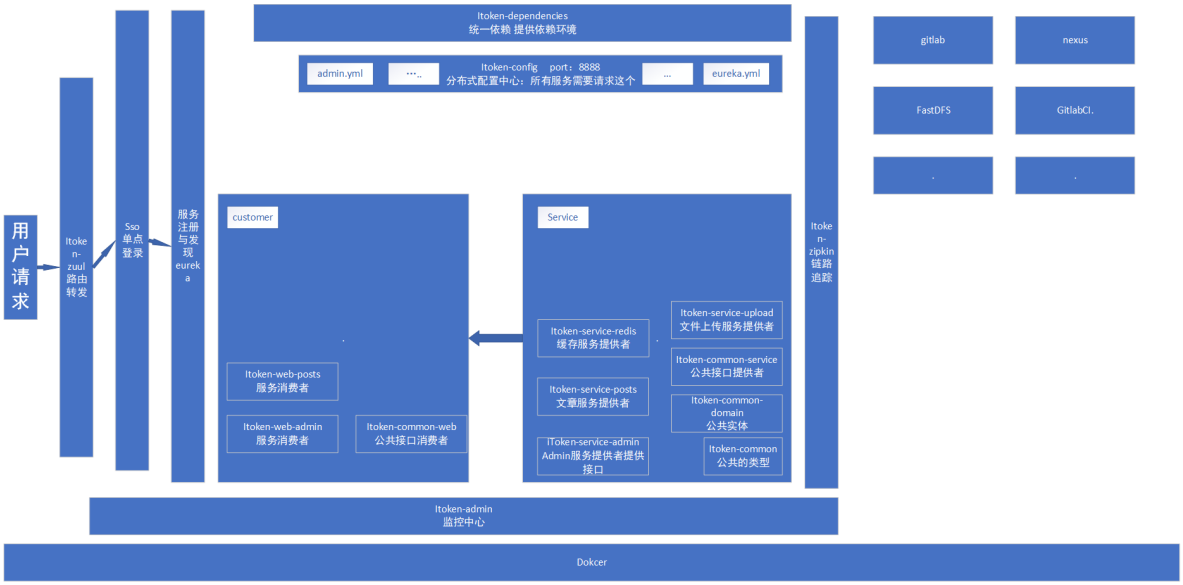
itoken-service-posts:文章服务提供者

itoken-service-admin:管理员服务提供者

itoken-web-posts:文章服务消费者

itoken-web-admin:管理员服务消费者

## 项目结构图




# nginx:

```
### 代理：
```

代理服务器，客户机在发送请求时，不会直接发送给目的主机，而是先发送给代理服务器，代理服务接受客户机请求之后，再向主机发出，并接收目的主机返回的数据，存放在代理服务器的硬盘中，再发送给客户机。

### 正向代理：

正向代理，架设在客户机与目标主机之间，只用于代理内部网络对 Internet 的连接请求，客户机必须指定代理服务器,并将本来要直接发送到 Web 服务器上的 Http 请求发送到代理服务器中。

### 反向代理：

反向代理服务器架设在服务器端，通过缓冲经常被请求的页面来缓解服务器的工作量，将客户机请求转发给内部网络上的目标服务器；并将从服务器上得到的结果返回给 Internet 上请求连接的客户端，此时代理服务器与目标主机一起对外表现为一个服务器。

## 虚拟主机：

虚拟主机是一种特殊的软硬件技术，它可以将网络上的每一台计算机分成多个虚拟主机，每个虚拟主机可以独立对外提供 www 服务，这样就可以实现一台主机对外提供多个 web 服务，每个虚拟主机之间是独立的，互不影响的。

通过 Nginx 可以实现虚拟主机的配置，Nginx 支持三种类型的虚拟主机配置

- 基于 IP 的虚拟主机
- 基于域名的虚拟主机
- 基于端口的虚拟主机

## nginx的使用：

Nginx 是一款高性能的 HTTP 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器。 由俄罗斯的程序设计师 Igor Sysoev 所开发，官方测试 Nginx 能够支支撑 5 万并发链接，并且 CPU、内存等资源消耗却非常低，运行非常稳定。 #### Nginx 的应用场景 HTTP 服务器：Nginx 是一个 HTTP 服务可以独立提供 HTTP 服务。可以做网页静态服务器。 虚拟主机：可以实现在一台服务器虚拟出多个网站。例如个人网站使用的虚拟主机。 反向代理，负载均衡：当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使用 Nginx 做反向代理。并且多台服务器可以平均分担负载，不会因为某台服务器负载高宕机而某台服务器闲置的情况。

**1.docker-compose.yml来使用nginx:**

```
version: '3.1'
services:
nginx:
    restart: always
    image: nginx
    container_name: nginx
    ports:
    - 81:80
    volumes:
    - ./conf/nginx.conf:/etc/nginx/nginx.conf
    - ./wwwroot:/usr/share/nginx/wwwroot
配置nginx的数据卷：
    在dokcer文件下创建：nginx的目录下创建：nginx.conf文件：
    在当前目录创建wwwroot目录
     这里直接书写为
     /usr/local/dokcer/nginx/conf/nginx.conf
    /usr/local/dokcer/nginx/wwwroot
    在nginx.conf目录配置nginx的虚拟主机：
```

**2.nginx中使用虚拟主机配置：**

需求 Nginx 对外提供 80 和 8080 两个端口监听服务 请求 80 端口则请求 html80 目录下的 html 请求 8080 端口则请求 html8080 目录下的 html

**创建目录及文件**

在 /usr/local/docker/nginx/wwwroot 目录下创建 html80 和 html8080 两个目录，并分辨创建两个 index.html 文件

```
# 配置虚拟主机
修改 /usr/local/docker/nginx/conf 目录下的 nginx.conf 配置文件：

worker_processes  1;

events {
    worker_connections  1024;
}

http {
```

```
            include       mime.types;
            default_type  application/octet-stream;

            sendfile        on;

            keepalive_timeout  65;
            # 配置虚拟主机 192.168.75.145
            server {
            # 监听的ip和端口，配置 192.168.75.145:80
                listen       80;
            # 虚拟主机名称这里配置ip地址
                server_name  192.168.75.145;
            # 所有的请求都以 / 开始，所有的请求都可以匹配此 location
                location / {
                # 使用 root 指令指定虚拟主机目录即网页存放目录
                # 比如访问 http://ip/index.html 将找到
/usr/local/docker/nginx/wwwroot/html80/index.html
                # 比如访问 http://ip/item/index.html 将找到
/usr/local/docker/nginx/wwwroot/html80/item/index.html

                    root   /usr/share/nginx/wwwroot/html80;
                # 指定欢迎页面，按从左到右顺序查找
                    index  index.html index.htm;
                }

            }
            # 配置虚拟主机 192.168.75.245
            server {
                listen       8080;
                server_name  192.168.75.145;

                location / {
                    root   /usr/share/nginx/wwwroot/html8080;
                    index  index.html index.htm;
                }
            }
        }
```

说明：这里的启动的端口必须和dockercompose中的nginx的启动端口一一对应：

例如：这里有两个分别为8080和80那么port应这样写

```
 ports:
    - 80:80
    - 8080:  8080
```

同时这里的location不能改变，改变的话也是改变对应的数据卷，其实就是这个文件映射到数据卷位置

这里不需要改变只需改变html8000这个就可以：

```
                location / {
                    root   /usr/share/nginx/wwwroot/html8080;
                    index  index.html index.htm;
                }
```

**基于域名的虚拟主机配置**

需求:两个域名指向同一台 Nginx 服务器，用户访问不同的域名显示不同的网页内容 两个域名是 admin.service.itoken.funtl.com 和 admin.web.itoken.funtl.com Nginx 服务器使用虚拟机 192.168.75.145 **配置 Windows Hosts 文件** 通过 host 文件指定 admin.service.itoken.funtl.com 和 admin.web.itoken.funtl.com 对应 192.168.75.145 虚拟机： 修改 window 的 hosts 文件：（C:\Windows\System32\drivers\etc)

```
    worker_processes  1;
            events {
                worker_connections  1024;
            }

            http {
                include       mime.types;
                default_type  application/octet-stream;

                sendfile        on;

                keepalive_timeout  65;

# 配置虚拟主机 192.168.75.145

•                server {

# 监听的ip和端口，配置 192.168.75.145:80

•                    listen       80;

# 虚拟主机名称这里配置ip地址

•                    server_name  www.kay.com;

# 所有的请求都以 / 开始，所有的请求都可以匹配此 location

•                    location / {

# 使用 root 指令指定虚拟主机目录即网页存放目录

# 比如访问 http://ip/index.html 将找到
/usr/local/docker/nginx/wwwroot/html80/index.html

# 比如访问 http://ip/item/index.html 将找到
/usr/local/docker/nginx/wwwroot/html80/item/index.html

•                        root   /usr/share/nginx/wwwroot/html80;

# 指定欢迎页面，按从左到右顺序查找

•                        index  index.html index.htm;
•                    }

•                }

# 配置虚拟主机 192.168.75.245

               server {
                   listen       8080;
                    server_name  192.168.75.145;

                    location / {
                        root   /usr/share/nginx/wwwroot/html8080;
                        index  index.html index.htm;
                    }
                }
```

```
            }
            通过 host 文件指定 admin.service.itoken.funtl.com 和
admin.web.itoken.funtl.com 对应 192.168.75.145 虚拟机：
            这样通过域名即可访问：

# 创建目录及文件

            在 /usr/local/docker/nginx/wwwroot 目录下创建 htmlservice 和 htmlweb 两
个目录，并分辨创建两个 index.html 文件

# 配置虚拟主机

            user  nginx;
            worker_processes  1;

            events {
                worker_connections  1024;
            }

            http {
                include       mime.types;
•               default_type  application/octet-stream;

•               sendfile        on;

•               keepalive_timeout  65;
•               server {
•                   listen       80;
•                   server_name  admin.service.itoken.funtl.com;
•                   location / {
•                       root   /usr/share/nginx/wwwroot/htmlservice;
•                       index  index.html index.htm;
•                   }

•               }

•               server {
•                   listen       80;
•                   server_name  admin.web.itoken.funtl.com;

•                   location / {
•                       root   /usr/share/nginx/wwwroot/htmlweb;
•                       index  index.html index.htm;
•                   }
•               }
•            }
```

3.使用nginx反向代理tomcat:

```
        （1） 启动两个tomcat：在dokcer-compose.yml 编辑：
            version: '3'
            services:
              tomcat1:
                image: tomcat
                container_name: tomcat1
                ports:
```

```
- 9090:8080

  tomcat2:
    image: tomcat
    container_name: tomcat2
    ports:

- 9091:8080
```

ocal/docker/nginx/conf 目录下的 nginx.conf 配置文件：

```
user  nginx;
worker_processes  1;

events {
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;

    keepalive_timeout  65;
```

配置一个代理即 tomcat1 服务器

```
    upstream tomcatServer1 {
        server 192.168.75.145:9090;
    }
```

配置一个代理即 tomcat2 服务器

```
    upstream tomcatServer2 {
        server 192.168.75.145:9091;
    }
```

配置一个虚拟主机

```
    server {
        listen 80;
        server_name admin.service.itoken.funtl.com;
        location / {
```

域名 admin.service.itoken.funtl.com 的请求全部转发到 tomcat_server1 即 tomcat1 服务上

可以直接书写tomcat的路径即可

```
            proxy_pass http://tomcatServer1;
```

欢迎页面，按照从左到右的顺序查找页面

```
            index index.jsp index.html index.htm;
        }
    }

    server {
```

```
                    listen 80;
                    server_name admin.web.itoken.funtl.com;

                    location / {

域名 admin.web.itoken.funtl.com 的请求全部转发到 tomcat_server2 即 tomcat2 服务上

                        proxy_pass http://tomcatServer2;
                        index index.jsp index.html index.htm;
                    }
                }
            }
        （3）启动nginx在docker-compose.yml中配置：
                version: '3'
                services:

nexus:

image: 'sonatype/nexus3'

restart: always

container_name: nexus

ports:

- '8081:8081'

volumes:

- '/usr/local/docker/nexus/data:/nexus-data'

                nignx:
                    restart: always
                    image: nginx
                    container_name: nginx
                    ports:

- '8088:8088'
- '9000:9000'
- '80:80'
- '/usr/local/docker/nginx/conf/nginx.conf:/etc/nginx/nginx.conf'
- '/usr/local/docker/nginx/wwwroot:/usr/share/nginx/wwwroot'
```

4.实战:

```
    在一个虚拟主机配置两个tomcat:
                1.vim /etc/profile
                    export CATALINA1_BASE="tomcat路径"
                    export CATALINA1_HOME="tomcat路径"
                    export Tomcat1Home=CATALINA1_BASE
                    export CATALINA2_BASE="tomcat2路径"
                    export CATALINA2_HOME="tomcat2路径"
                    export Tomcat2Home=CATALINA2_BASE
                2.在bin文件中修改 catalina.sh
                    在首行加入：
```

```
                    export CATALINA1_BASE=CATALINA1_BASE
                    export CATALINA1_HOME=CATALINA1_HOME
                3.修改host：
                    8005→9005
                    8009→9009
                    8080→9080
                    8443→9443

4. ./startup.sh启动即可
```

# Redis:

HA(High Available，高可用性群集)机集群系统简称，是保证业务连续性的有效解决方案，一般有两个或两个以上的节点，且分为活动节点及备用节点。 通常把正在执 行业务的称为活动节点，而作为活动节点的一个备份的则称为备用节点。当活动节点出现问题，导致正在运行的业务（任务）不能正常运行时， 备用节点此时就会侦测到，并立即接续活动节点来执行业务。从而实现业务的不中断或短暂中断。

Redis 一般以主/从方式部署（这里讨论的应用从实例主要用于备份，主实例提供读写）该方式要实现 HA 主要有如下几种方案：

- keepalived： 通过 keepalived 的虚拟 IP，提供主从的统一访问，在主出现问题时， 通过 keepalived 运行脚本将从提升为主，待主恢复后先同步后自动变为主，该方案的好处是主从切换后，应用程序不需要知道(因为访问的虚拟 IP 不变)，坏处是引入 keepalived 增加部署复杂性，在有些情况下会导致数据丢失
- zookeeper： 通过 zookeeper 来监控主从实例， 维护最新有效的 IP， 应用通过 zookeeper 取得 IP，对 Redis 进行访问，该方案需要编写大量的监控代码
- sentinel： 通过 Sentinel 监控主从实例，自动进行故障恢复，该方案有个缺陷：因为主从实例地址( IP & PORT )是不同的， 当故障发生进行主从切换后，应用程序无法知道新地址，故在 Jedis2.2.2 中新增了对 Sentinel 的支持， 应用通过 redis.clients.jedis.JedisSentinelPool.getResource() 取得的 Jedis 实例会及时更新到新的主实例地址做位缓存，session分离，应用排行榜。

## 搭建redis集群：

创建redis文件夹:

### 搭建一主两从环境创建docker-compose.yml

```
 version: '3.1'
        services:
        master:
            image: redis
            container_name: redis-master
            ports:

- 6379:6379

        slave1:
            image: redis
            container_name: redis-slave-1
            ports:

- 6380:6379
```

```
 redis-server --slaveof redis-master 6379

        slave2:
            image: redis
            container_name: redis-slave-2
            ports:

- 6381:6379
 redis-server --slaveof redis-master 6379
```

启动redis: docker-compose up -d

### 使用sentinel实现高可用监控：

创建sentinel文件在docker目录下： 创建docker-compose.yml文件：

```
            version: '3.1'
            services:
            sentinel1:
                image: redis
                container_name: redis-sentinel-1
                ports:
                - 26379:26379
                command: redis-sentinel /usr/local/etc/redis/sentinel.conf
                volumes:
                #数据卷
                - ./sentinel1.conf:/usr/local/etc/redis/sentinel.conf

            sentinel2:
                image: redis
                container_name: redis-sentinel-2
                ports:
                - 26380:26379
                command: redis-sentinel /usr/local/etc/redis/sentinel.conf
                volumes:
                - ./sentinel2.conf:/usr/local/etc/redis/sentinel.conf

            sentinel3:
                image: redis
                container_name: redis-sentinel-3
                ports:
                - 26381:26379
                command: redis-sentinel /usr/local/etc/redis/sentinel.conf
                volumes:
                - ./sentinel3.conf:/usr/local/etc/redis/sentinel.conf
        修改Sentinel文件：分别为 sentinel1.conf，sentinel2.conf，sentinel3.conf，配置文件内容相同
        配置内容如下：
            port 26379
            dir /tmp
            # 自定义集群名，其中 127.0.0.1 为 redis-master 的 ip，6379 为 redis-master 的端口，2 为最小投票数（因为有 3 台 Sentinel 所以可以设置成 2）
            sentinel monitor mymaster 192.168.147.132 6379 2
            sentinel down-after-milliseconds mymaster 30000
            sentinel parallel-syncs mymaster 1
            sentinel failover-timeout mymaster 180000
```

```
            sentinel deny-scripts-reconfig yes
        启动集群：
                docker-compose up -d

        查看集群是否生效：进入 Sentinel 容器，使用 Sentinel API 查看监控情况：
            docker exec -it redis-sentinel-1 /bin/bash
            redis-cli -p 26379
            sentinel master mymaster
            sentinel slaves mymaster
        生效后即可：
```

## 创建redis服务：

创建service-redis项目： 在配置中心创建service-redis-dev.yml:

```
  spring:
            application:
                name: itoken-service-redis
            boot:
                admin:
                client:
                    url: http://localhost:8084
            zipkin:
                base-url: http://localhost:9411
            redis:
                lettuce:
                pool:
                    max-active: 8
                    max-idle: 8
                    max-wait: -1ms
                    min-idle: 0
                sentinel:
                master: mymaster
                nodes: 192.168.147.132:6379
       server:
        port: 8502

        eureka:
        instance:
            hostname: localhost
        client:
            registerWithEureka: false
            fetchRegistry: false
            serviceUrl:
            defaultZone:
http://${eureka.instance.hostname}:${server.port}/eureka/

        management:
        endpoint:
            health:
            show-detail: always
        endpoints:
            web:
            exposure:
                include: health,info
    依次创建pom.xml:和service服务，service用来注入
```

```
@Autowired
        private RedisTemplate redisTemplate;
    RedisTemplate类的基本的使用：
        redisTemplate.opsForValue();//操作字符串，
        redisTemplate.opsForHash();//操作hash
        redisTemplate.opsForList();;//操作list
        redisTemplate.opsForSet();//操作set
        redisTemplate.opsForZSet();//操作有序set
```

```
         set void set(K key, V value);

        使用：redisTemplate.opsForValue().set("name","tom");
        结果：redisTemplate.opsForValue().get("name")  输出结果为tom
```

```
                set void set(K key, V value, long timeout, TimeUnit unit);

                使用：redisTemplate.opsForValue().set("name","tom",10,
TimeUnit.SECONDS);
                结果：redisTemplate.opsForValue().get("name")由于设置的是10秒失效，十秒之内查询有结果，十秒之后返回为null
```

```
                set void set(K key, V value, long offset);
                该方法是用 value 参数覆写(overwrite)给定 key 所储存的字符串值，从偏移量 offset 开始

                使用：template.opsForValue().set("key","hello world");
                        template.opsForValue().set("key","redis", 6);

 System.out.println("***************"+template.opsForValue().get("key"));
                结果：***************hello redis
```

```
                setIfAbsent Boolean setIfAbsent(K key, V value);

                使用：
System.out.println(template.opsForValue().setIfAbsent("multi1","multi1"));//fals
e  multi1之前已经存在

 System.out.println(template.opsForValue().setIfAbsent("multi111","multi111"));/
/true  multi111之前不存在
                结果：false
                true
```

```
                multiSet void multiSet(Map<? extends K, ? extends V> m);
                为多个键分别设置它们的值

                使用：Map<String,String> maps = new HashMap<String, String>();
                        maps.put("multi1","multi1");
                        maps.put("multi2","multi2");
                        maps.put("multi3","multi3");
                        template.opsForValue().multiSet(maps);
                        List<String> keys = new ArrayList<String>();
                        keys.add("multi1");
                        keys.add("multi2");
```

```
                        keys.add("multi3");

 System.out.println(template.opsForValue().multiGet(keys));
                结果：[multi1, multi2, multi3]
```

```
                multiSetIfAbsent Boolean multiSetIfAbsent(Map<? extends K, ?
extends V> m);
                为多个键分别设置它们的值，如果存在则返回false，不存在返回true

                使用：Map<String,String> maps = new HashMap<String, String>();
                        maps.put("multi11","multi11");
                        maps.put("multi22","multi22");
                        maps.put("multi33","multi33");
                        Map<String,String> maps2 = new HashMap<String, String>
();
                        maps2.put("multi1","multi1");
                        maps2.put("multi2","multi2");
                        maps2.put("multi3","multi3");

 System.out.println(template.opsForValue().multiSetIfAbsent(maps));

 System.out.println(template.opsForValue().multiSetIfAbsent(maps2));
                结果：true
                false
```

```
                get V get(Object key);

                使用：template.opsForValue().set("key","hello world");

 System.out.println("***************"+template.opsForValue().get("key"));
                结果：***************hello world
```

```
                getAndSet V getAndSet(K key, V value);
                设置键的字符串值并返回其旧值

                使用：template.opsForValue().set("getSetTest","test");

 System.out.println(template.opsForValue().getAndSet("getSetTest","test2"));
                结果：test
```

```
                multiGet List<V> multiGet(Collection<K> keys);
                为多个键分别取出它们的值

                使用：Map<String,String> maps = new HashMap<String, String>();
                        maps.put("multi1","multi1");
                        maps.put("multi2","multi2");
                        maps.put("multi3","multi3");
                        template.opsForValue().multiSet(maps);
                        List<String> keys = new ArrayList<String>();
                        keys.add("multi1");
                        keys.add("multi2");
                        keys.add("multi3");

 System.out.println(template.opsForValue().multiGet(keys));
                结果：[multi1, multi2, multi3]
```

increment Long increment(K key, long delta);
支持整数

使用：template.opsForValue().increment("increlong",1);

System.out.println("**********"+template.opsForValue().get("increlong"));
结果：**********1

increment Double increment(K key, double delta);
也支持浮点数

使用：template.opsForValue().increment("increlong",1.2);

System.out.println("**********"+template.opsForValue().get("increlong"));
结果：**********2.2

append Integer append(K key, String value);
如果key已经存在并且是一个字符串，则该命令将该值追加到字符串的末尾。如果键不存在，则它被创建并设置为空字符串，因此APPEND在这种特殊情况下将类似于SET。

使用：template.opsForValue().append("appendTest","Hello");

System.out.println(template.opsForValue().get("appendTest"));
template.opsForValue().append("appendTest","world");

System.out.println(template.opsForValue().get("appendTest"));
结果：Hello
Helloworld

get String get(K key, long start, long end);
截取key所对应的value字符串

使用：appendTest对应的value为Helloworld

System.out.println("**********"+template.opsForValue().get("appendTest",0,5));
结果：**********Hellow
使用：
System.out.println("**********"+template.opsForValue().get("appendTest",0,-1));
结果：**********Helloworld
使用：
System.out.println("**********"+template.opsForValue().get("appendTest",-3,-1));
结果：**********rld

size Long size(K key);
返回key所对应的value值得长度

使用：template.opsForValue().set("key","hello world");

System.out.println("**********"+template.opsForValue().size("key"));
结果：**********11

setBit Boolean setBit(K key, long offset, boolean value);
对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)

```
                key键对应的值value对应的ascii码,在offset的位置(从左向右数)变为value

                使用：template.opsForValue().set("bitTest","a");
                        // 'a' 的ASCII码是 97。转换为二进制是：01100001
                        // 'b' 的ASCII码是 98  转换为二进制是：01100010
                        // 'c' 的ASCII码是 99  转换为二进制是：01100011
                        //因为二进制只有0和1，在setbit中true为1，false为0，因此我要变为'b'的话第六位设置为1，第七位设置为0
                        template.opsForValue().setBit("bitTest",6, true);
                        template.opsForValue().setBit("bitTest",7, false);

 System.out.println(template.opsForValue().get("bitTest"));
                结果：b
                getBit Boolean getBit(K key, long offset);
                获取键对应值的ascii码的在offset处位值

                使用：
System.out.println(template.opsForValue().getBit("bitTest",7));
                结果：false
```

# 单点登录（sso:single sign on）：

指的是在多系统应用群中登录一个系统，使可在其他所有系统中得到授权而无需再次登录，包括单点登录与单点注销两部分 sso 需要一个独立的认证中心，只有认证中心能接受用户的用户名密码等安全信息，其他系统不提供登录入口，只接受认证中心的间接授权。间接授权通过令牌实现，sso 认证中心验证用户的用户名密码没问题，创建授权令牌， 在接下来的跳转过程中，授权令牌作为参数发送给各个子系统，子系统拿到令牌，即得到了授权，可以借此创建局部会话，局部会话登录方式与单系统的登录方式相同

用户访问系统 1 的受保护资源，系统1发现用户未登录，跳转至 sso 认证中心，并将自己的地址作为参数 sso 认证中心发现用户未登录，将用户引导至登录页面 用户输入用户名密码提交登录申请 sso 认证中心校验用户信息，创建用户与 sso 认证中心之间的会话，称为全局会话，同时创建授权令牌 sso 认证中心带着令牌跳转会最初的请求地址（系统1） 系统1拿到令牌，去 sso 认证中心校验令牌是否有效 sso 认证中心校验令牌，返回有效，注册系统 1 系统 1 使用该令牌创建与用户的会话，称为局部会话，返回受保护资源 系统2发现用户未登录，跳转至 sso 认证中心，并将自己的地址作为参数 sso 认证中心发现用户已登录，跳转回系统 2 的地址，并附上令牌 系统 2 拿到令牌，去 sso 认证中心校验令牌是否有效 sso 认证中心校验令牌，返回有效，注册系统 2 系统 2 使用该令牌创建与用户的局部会话，返回受保护资源 局部会话存在，全局会话一定存在 全局会话存在，局部会话不一定存在 全局会话销毁，局部会话必须销毁

单点登录自然也要单点注销，在一个子系统中注销，所有子系统的会话都将被销毁 实战部署：###SSO Client 拦截子系统未登录用户请求，跳转至 sso 认证中心 接收并存储 sso 认证中心发送的令牌 与 SSO Server 通信，校验令牌的有效性 建立局部会话 拦截用户注销请求，向 sso 认证中心发送注销请求 接收 sso 认证中心发出的注销请求，销毁局部会话 ### SSO Server 验证用户的登录信息 创建全局会话 创建授权令牌 与 SSO Client 通信发送令牌 校验 SSO Client 令牌有效性 系统注册 接收 SSO Client 注销请求，注销所有会话

## 实战SSO:

把sso登录的东西存到redis中，然后各个系统在redis服务中取的通过得到的token来获取是否进行登录， 其实就是把信息写入redis中，当下次使用的时候去redis中取结果即可以。

```
@RequestMapping(value = "login",method = RequestMethod.POST)
        public String login(@RequestParam(required = true)String loginCode,
@RequestParam(required = false) String url,
                            @RequestParam(required = true)String passWord,
                            HttpServletRequest request,
                            HttpServletResponse response,
                            RedirectAttributes redirectAttributes){
            TbSysUser tbSysUser = loginServcie.login(loginCod   e, passWord);

            // 登录失败
            if (tbSysUser == null) {
                redirectAttributes.addFlashAttribute("message", "用户名或密码错误，
请重新输入");
            }

            // 登录成功
            else {
                String token = UUID.randomUUID().toString();

                // 将 Token 放入缓存
                String result = redisService.put(token, loginCode, 60 * 60 *
24);
                if (StringUtils.isNotBlank(result) && "ok".equals(result)) {
                    //将token放入到cookie中
                    CookieUtils.setCookie(request, response, "token", token, 60
* 60 * 24);
                    if (StringUtils.isNotBlank(url)) {
                        return "redirect:" + url;
                    }
                }

                // 熔断处理
                else {
                    redirectAttributes.addFlashAttribute("message", "服务器异常，
请稍后再试");
                }
            }

            return "redirect:/login";
        }
```

```
## springboot拦截器的使用：

    ### 定义拦截器：
```

```
public class MyInterceptor implements HandlerInterceptor {
                @Override
                public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) {
                    return false;
                }

                @Override
                public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, ModelAndView modelAndView) {
```

```
                }

                @Override
                public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) {

                }
            }

################################

                 /**

- 实现redis的服务取数据，是否登录
- @Author kay三石
- @date:2019/6/28
                  public class LoginInterceptor implements HandlerInterceptor {
/**
  - 在这里需要用到一个redis服务
*/
@Autowired
private RedisService redisService;
  @Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) {
    return BaseInterceptorMethods.preHandleForLogin(request, response, handler,
"http://localhost:8601/" + request.getServletPath());
}
  @Override
public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) {
    String token = CookieUtils.getCookieValue(request,
WebConstants.SESSION_TOKEN);
    if (StringUtils.isNotBlank(token)) {
        String loginCode = redisService.get(token);
        if (StringUtils.isNotBlank(loginCode)) {
            BaseInterceptorMethods.postHandleForLogin(request, response,
handler, modelAndView, redisService.get(loginCode), "http://localhost:8601/" +
request.getServletPath());
        }
    }
}
  @Override
public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) {
  }
                }
    配置拦截器：
                @Configuration
                public class InterceptorConfig implements WebMvcConfigurer {
  @Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new MyInterceptor()).addPathPatterns("/**");
}
                }

#########################################################################
```

```
                /**
- 配置拦截器
- @Author kay三石
- @date:2019/6/28
                  public class InterceptorConfig  implements WebMvcConfigurer {
@Bean
public ConstantsInterceptor constantsInterceptor() {
    return new ConstantsInterceptor();
}
  @Override
public void addInterceptors(InterceptorRegistry registry) {
    // 常量拦截器
    registry.addInterceptor(constantsInterceptor())
            .addPathPatterns("/")
            .excludePathPatterns("/static/");
}
                }
```

```
    ## 使用tkmybatis进行对实体类和mapper的生成：

    ###  1.pom.xml中引入插件：
```

```
 <build>
 <plugins>
    <plugin>
        <groupId>org.mybatis.generator</groupId>
        <artifactId>mybatis-generator-maven-plugin</artifactId>
        <version>1.3.5</version>
        <configuration>
            <configurationFile>
{basedir}/src/main/resources/generator/generatorConfig.xml</configurationFile>
            <overwrite>true</overwrite>
            <verbose>true</verbose>
        </configuration>
        <dependencies>
            <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>{mysql.version}</version>
            </dependency>
            <dependency>
                <groupId>tk.mybatis</groupId>
                <artifactId>mapper</artifactId>
                <version>3.4.4</version>
            </dependency>
        </dependencies>
    </plugin>
   </plugins>
 </build>
```

```
   ### 2.在resource中创建generator文件并创建 generatorConfig.xml:
```

```
     <?xml version="1.0" encoding="UTF-8"?>

            <!DOCTYPE generatorConfiguration
                    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration
1.0//EN"
                    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

            <generatorConfiguration>
                <!-- 引入数据库连接配置 -->
                <properties resource="jdbc.properties"/>

                <context id="Mysql" targetRuntime="MyBatis3Simple"
defaultModelType="flat">
                    <property name="beginningDelimiter" value="`"/>
                    <property name="endingDelimiter" value="`"/>

                   <!-- 配置 tk.mybatis 插件 -->
                    <plugin type="tk.mybatis.mapper.generator.MapperPlugin">
                        <property name="mappers"
value="tk.mybatis.mapper.MyMapper"/>
                    </plugin>

                    <!-- 配置数据库连接 -->
                    <jdbcConnection
                            driverClass="${jdbc.driverClass}"
                            connectionURL="${jdbc.connectionURL}"
                            userId="${jdbc.username}"
                            password="${jdbc.password}">
                    </jdbcConnection>

                    <!-- 配置实体类存放路径 -->
                    <javaModelGenerator
targetPackage="com.kayleoi.itoken.common.domain" targetProject="src/main/java"/>

                    <!-- 配置 XML 存放路径 -->
                    <sqlMapGenerator targetPackage="mapper"
targetProject="src/main/resources"/>

                    <!-- 配置 DAO 存放路径 -->
                    <javaClientGenerator
                            targetPackage="com.kayleoi.itoken.common.mapper"
                            targetProject="src/main/java"
                            type="XMLMAPPER"/>

                    <!-- 配置需要生成的表，% 代表所有 -->
                    <!--itoken-service-admin-->

                    <table catalog="itoken-service-admin"
tableName="tb_sys_user">
                        <!-- mysql 配置 -->
                        <generatedKey column="user_code" sqlStatement="Mysql"
identity="false"/>
                    </table>

                    <!--itoken-service-posts-->

                    <table catalog="itoken-service-posts"
tableName="tb_sys_user">
```

```
                        <!-- mysql 配置 -->
                        <generatedKey column="post_guid" sqlStatement="Mysql"
identity="false"/>
                    </table>

                </context>
            </generatorConfiguration>
```

## 3.引入jdbc:

```
MySQL 8.x: com.mysql.cj.jdbc.Driver

因为在generatorConfiguration配置指定的数据库，所以这里不再进行数据库的操作

jdbc.driverClass=com.mysql.jdbc.Driver
jdbc.connectionURL=jdbc:mysql://localhost:3306?
useUnicode=true&characterEncoding=utf-8&useSSL=false
jdbc.username=root
jdbc.password=123
```

## 4.右侧maven插件中找到mybatis-generator:generate运行即可

# Swagger2使用接口文档引擎：

```
    ### 1.简介：
```

Swagger是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。 总体目标是使客户端和文件系统作为服务器以同样的速度来更新。 在Spring Boot项目中，可以将文件的方法、参数和模型紧密集成到服务器端的代码，允许API来始终保持同步。 Swagger作用主要包括两点：接口文档在线自动生成和功能测试。

```
   ### 2.使用形式：
```

```
pom.xml中引入：
            <!-- Swagger2 Begin -->
            <dependency>
                <groupId>io.springfox</groupId>
                <artifactId>springfox-swagger2</artifactId>
                <version>2.8.0</version>
            </dependency>
            <dependency>
                <groupId>io.springfox</groupId>
                <artifactId>springfox-swagger-ui</artifactId>
                <version>2.8.0</version>
            </dependency>
            <!-- Swagger2 End -->
```

**启用：** ** 在启动类上加入注解即可使用：@EnableSwagger2**

## 3.常用的类库：

Swagger 注解说明 Swagger 通过注解表明该接口会生成文档，包括接口名、请求方法、参数、返回信息的等等。

@Api：修饰整个类，描述 Controller 的作用 @ApiOperation：描述一个类的一个方法，或者说一个接口 @ApiParam：单个参数描述 @ApiModel：用对象来接收参数 @ApiProperty：用对象接收参数时，描述对象的一个字段 @ApiResponse：HTTP 响应其中 1 个描述 @ApiResponses：HTTP 响应整体描述 @ApiIgnore：使用该注解忽略这个API @ApiError：发生错误返回的信息 @ApiImplicitParam：一个请求参数 @ApiImplicitParams：多个请求参数

```
 /**

- @Author kay三石
- @date:2019/8/25
-
RequestHandlerSelectors.basePackage("com.funtl.itoken.service.admin.controller")
- 为 Controller 包路径，不然生成的文档扫描不到接口
              @Configuration
            public class Swagger2Config {
  @Bean
public Docket createRestApi(){
    return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()

 .apis(RequestHandlerSelectors.basePackage("com.kayleoi.itoken.service.admin"))
            .paths(PathSelectors.any())
            .build();
}
private ApiInfo apiInfo(){
    return new ApiInfoBuilder()
            .title("itoken API 文档")
            .description("itoken API 网关接口，http://www.kaynethy.com")
            .version("1.0.0")
            .build();
}

            }
```

# FastDFS:

FastDFS是一个开源的轻量级分布式文件系统，它对文件进行管理，功能包括：文件存储、文件同步、文件访问（文件上传、文件下载）等，解决了大容量存储和负载均衡的问题。 特别适合以文件为载体的在线服务，如相册网站、视频网站等等。 FastDFS为互联网量身定制，充分考虑了冗余备份、负载均衡、线性扩容等机制，并注重高可用、高性能等指标，使用FastDFS很容易搭建一套高性能的文件服务器集群提供文件上传、下载等服务。 FastDFS服务端主要有：跟踪器(tracker)群和存储节点(storage)，跟踪器做调度工作，在访问上起负载均衡的作用，跟踪器和存储节点都可以由一台或多台服务器构成。跟踪器和存储节点中的服务器均可以随时增加或下线而不会影响线上服务。其中跟踪器中的所有服务器都是对等的，可以根据服务器的压力情况随时增加或减少。 上传交互过程编辑

1. client询问tracker下载文件的storage，参数为文件标识（卷名和文件名）；
2. tracker返回一台可用的storage；
3. client直接和storage通讯完成文件下载。 需要说明的是，client为使用FastDFS服务的调用方，client也应该是一台服务器，它对tracker和storage的调用均为服务器间的调用。

## 安装方式：

### 需要以下的资源：

- ibfastcommon.tar.gz
- fastdfs-5.11.tar.gz
- nginx-1.13.6.tar.gz
- fastdfs-nginx-module_v1.16.tar.gz
- 在 Linux 服务器上创建 /usr/local/docker/fastdfs/environment 目录
- 说明： /usr/local/docker/fastdfs：用于存放 docker-compose.yml 配置文件及 FastDFS 的数据卷 /usr/local/docker/fastdfs/environment：用于存放 Dockerfile 镜像配置文件及 FastDFS 所需环境

### docker-compose.yml

```
version: '3.1'
 services:
 fastdfs:
     build: environment
     restart: always
     container_name: fastdfs
     volumes:

../storage:/fastdfs/storage
etwork_mode: host
```

```
#### Dockerfile
```

FROM ubuntu:xenial  MAINTAINER topsale@vip.qq.com

```
    #### 更新数据源
```

WORKDIR /etc/apt  RUN echo 'deb http://mirrors.aliyun.com/ubuntu/ xenial main restricted universe multiverse' > sources.list  RUN echo 'deb http://mirrors.aliyun.com/ubuntu/ xenial-security main restricted universe multiverse' >> sources.list  RUN echo 'deb http://mirrors.aliyun.com/ubuntu/ xenial-updates main restricted universe multiverse' >> sources.list  RUN echo 'deb http://mirrors.aliyun.com/ubuntu/ xenial-backports main restricted universe multiverse' >> sources.list  RUN apt-get update

```
    #### 安装依赖
```

RUN apt-get install make gcc libpcre3-dev zlib1g-dev --assume-yes

```
    #### 复制工具包
```

ADD fastdfs-5.11.tar.gz /usr/local/src  ADD fastdfs-nginx-module_v1.16.tar.gz /usr/local/src  ADD libfastcommon.tar.gz /usr/local/src  ADD nginx-1.13.6.tar.gz /usr/local/src

```
    #### 安装 libfastcommon
```

WORKDIR /usr/local/src/libfastcommon  RUN ./make.sh && ./make.sh install

```
    #### 安装 FastDFS
```

WORKDIR /usr/local/src/fastdfs-5.11  RUN ./make.sh && ./make.sh install

```
    #### 配置 FastDFS 跟踪器
```

ADD tracker.conf /etc/fdfs  RUN mkdir -p /fastdfs/tracker

```
    #### 配置 FastDFS 存储
```

ADD storage.conf /etc/fdfs  RUN mkdir -p /fastdfs/storage

```
    #### 配置 FastDFS 客户端
```

ADD client.conf /etc/fdfs

```
    #### 配置 fastdfs-nginx-module
```

ADD config /usr/local/src/fastdfs-nginx-module/src

```
    ### FastDFS 与 Nginx 集成
```

```
  WORKDIR /usr/local/src/nginx-1.13.6
        RUN ./configure --add-module=/usr/local/src/fastdfs-nginx-module/src
        RUN make && make install
        ADD mod_fastdfs.conf /etc/fdfs

        WORKDIR /usr/local/src/fastdfs-5.11/conf
        RUN cp http.conf mime.types /etc/fdfs/
```

```
    #### 配置 Nginx
```

```
ADD nginx.conf /usr/local/nginx/conf
COPY entrypoint.sh /usr/local/bin/
ENTRYPOINT ["/usr/local/bin/entrypoint.sh"]
WORKDIR /
EXPOSE 8888
CMD ["/bin/bash"]
entrypoint.sh
# !/bin/sh
/etc/init.d/fdfs_trackerd start
/etc/init.d/fdfs_storaged start
/usr/local/nginx/sbin/nginx -g 'daemon off;'
 注：Shell 创建后是无法直接使用的，需要赋予执行的权限，使用 chmod +x entrypoint.sh 命令
```

```
#### 各种配置文件说明
    ##### tracker.conf
```

FastDFS 跟踪器配置，容器中路径为：/etc/fdfs，修改为：

base_path=/fastdfs/tracker ##### storage.conf FastDFS 存储配置，容器中路径为：/etc/fdfs，修改为：

base_path=/fastdfs/storage store_path0=/fastdfs/storage tracker_server=192.168.75.128:22122 http.server_port=8888 ##### client.conf FastDFS 客户端配置，容器中路径为：/etc/fdfs，修改为：

base_path=/fastdfs/tracker tracker_server=192.168.75.128:22122 ##### config fastdfs-nginx-module 配置文件，容器中路径为：/usr/local/src/fastdfs-nginx-module/src，修改为：

修改前

```
            CORE_INCS="CORE_INCS /usr/local/include/fastdfs
/usr/local/include/fastcommon/"
            CORE_LIBS="CORE_LIBS -L/usr/local/lib -lfastcommon -lfdfsclient"
```

修改后

```
            CORE_INCS="CORE_INCS /usr/include/fastdfs /usr/include/fastcommon/"
            CORE_LIBS="CORE_LIBS -L/usr/lib -lfastcommon -lfdfsclient"
```

```
    ##### mod_fastdfs.conf
```

fastdfs-nginx-module 配置文件，容器中路径为：/usr/local/src/fastdfs-nginx-module/src，修改为：

connect_timeout=10 tracker_server=192.168.75.128:22122 url_have_group_name = true store_path0=/fastdfs/storage ##### nginx.conf Nginx 配置文件，容器中路径为：/usr/local/src/nginx-1.13.6/conf，修改为：

user root; worker_processes 1;

events { worker_connections 1024; }

http { include mime.types; default_type application/octet-stream;

sendfile on;

keepalive_timeout 65;

server { listen 8888; server_name localhost;

location ~/group([0-9])/M00 { ngx_fastdfs_module; }

error_page 500 502 503 504 /50x.html; location = /50x.html { root html; } } } #### 启动容器 docker-compose up -d #### 测试上传 ##### 交互式进入容器 docker exec -it fastdfs /bin/bash ##### 测试文件上传 /usr/bin/fdfs_upload_file /etc/fdfs/client.conf /usr/local/src/fastdfs-5.11/INSTALL ##### 服务器反馈上传地址 group1/M00/00/00/wKhLi1oHVMCAT2vrAAAeSwu9TgM3976771 ##### 测试 Nginx 访问 http://192.168.75.128:8888/group1/M00/00/00/wKhLi1oHVMCAT2vrAAAeSwu9TgM3976771

# MQ

## 什么是 MQ

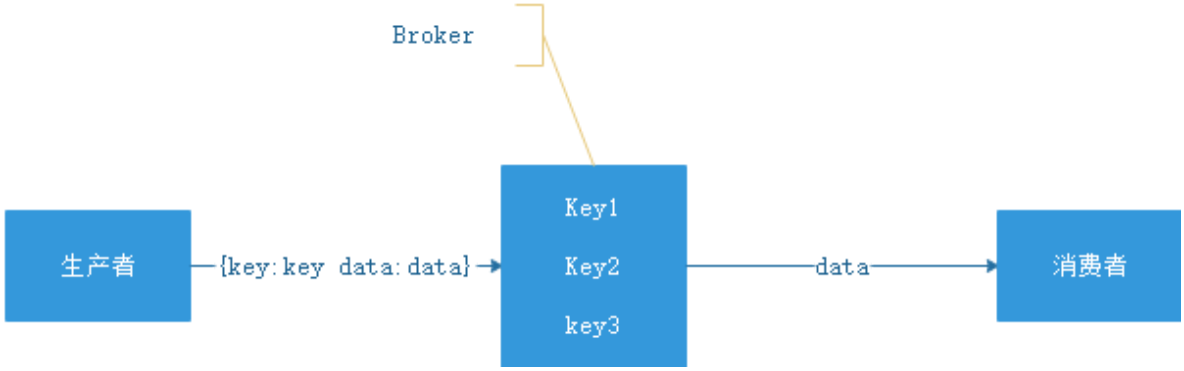
Message Queue（MQ），消息队列中间件。很多人都说：MQ 通过将消息的发送和接收分离来实现应用程序的异步和解偶，这个给人的直觉是——MQ 是异步的，用来解耦的，但是这个只是 MQ 的效果而不是目的。MQ 真正的目的是为了通讯，屏蔽底层复杂的通讯协议，定义了一套应用层的、更加简单的通讯协议。一个分布式系统中两个模块之间通讯要么是 HTTP，要么是自己开发的 TCP，但是这两种协议其实都是原始的协议。HTTP 协议很难实现两端通讯——模块 A 可以调用 B，B 也可以主动调用 A，如果要做到这个两端都要背上 WebServer，而且还不支持长连接（HTTP 2.0 的库根本找不到）。TCP 就更加原始了，粘包、心跳、私有的协议，想一想头皮就发麻。MQ 所要做的就是在这些协议之上构建一个简单的“协议”——生产者/消费者模型。MQ 带给我的“协议”不是具体的通讯协议，而是更高层次通讯模型。它定义了两个对象——发送数据的叫生产者；接收数据的叫消费者， 提供一个 SDK 让我们可以定义自己的生产者和消费者实现消息通讯而无视底层通讯协议

## 有 Broker 的 MQ

这个流派通常有一台服务器作为 Broker，所有的消息都通过它中转。生产者把消息发送给它就结束自己的任务了，Broker 则把消息主动推送给消费者（或者消费者主动轮询）

## 重-topic重 Topic

kafka、JMS（ActiveMQ）就属于这个流派，生产者会发送 key 和数据到 Broker，由 Broker 比较 key 之后决定给哪个消费者。这种模式是我们最常见的模式，是我们对 MQ 最多的印象。在这种模式下一个 topic 往往是一个比较大的概念，甚至一个系统中就可能只有一个 topic，topic 某种意义上就是 queue，生产者发送 key 相当于说：“hi，把数据放到 key 的队列中”

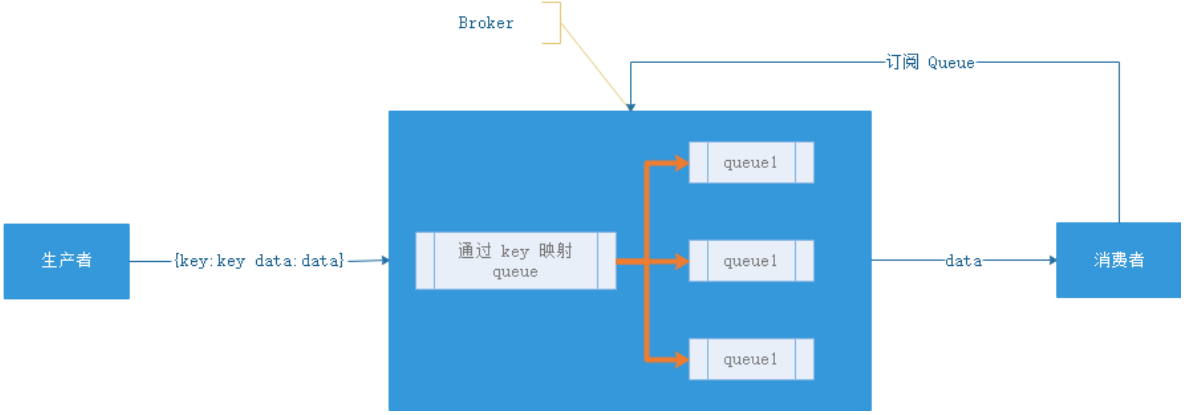



如上图所示，Broker 定义了三个队列，key1，key2，key3，生产者发送数据的时候会发送 key1 和 data，Broker 在推送数据的时候则推送 data（也可能把 key 带上）。

虽然架构一样但是 kafka 的性能要比 jms 的性能不知道高到多少倍，所以基本这种类型的 MQ 只有 kafka 一种备选方案。如果你需要一条暴力的数据流（在乎性能而非灵活性）那么 kafka 是最好的选择

## 轻 Topic

这种的代表是 RabbitMQ（或者说是 AMQP）。生产者发送 key 和数据，消费者定义订阅的队列，Broker 收到数据之后会通过一定的逻辑计算出 key 对应的队列，然后把数据交给队列



这种模式下解耦了 key 和 queue，在这种架构中 queue 是非常轻量级的（在 RabbitMQ 中它的上限取决于你的内存），消费者关心的只是自己的 queue；生产者不必关心数据最终给谁只要指定 key 就行了，中间的那层映射在 AMQP 中叫 exchange（交换机）。

AMQP 中有四种 exchange

- Direct exchange：key 就等于 queue
- Fanout exchange：无视 key，给所有的 queue 都来一份
- Topic exchange：key 可以用“宽字符”模糊匹配 queue
- Headers exchange：无视 key，通过查看消息的头部元数据来决定发给那个 queue（AMQP 头部元数据非常丰富而且可以自定义）

这种结构的架构给通讯带来了很大的灵活性，我们能想到的通讯方式都可以用这四种 exchange 表达出来。如果你需要一个企业数据总线（在乎灵活性）那么 RabbitMQ 绝对的值得一用

# #无 Broker 的 MQ

无 Broker 的 MQ 的代表是 ZeroMQ。该作者非常睿智，他非常敏锐的意识到——MQ 是更高级的 Socket，它是解决通讯问题的。所以 ZeroMQ 被设计成了一个“库”而不是一个中间件，这种实现也可以达到——没有 Broker 的目的


节点之间通讯的消息都是发送到彼此的队列中，每个节点都既是生产者又是消费者。ZeroMQ 做的事情就是封装出一套类似于 Socket 的 API 可以完成发送数据，读取数据

ZeroMQ 其实就是一个跨语言的、重量级的 Actor 模型邮箱库。你可以把自己的程序想象成一个 Actor，ZeroMQ 就是提供邮箱功能的库；ZeroMQ 可以实现同一台机器的 RPC 通讯也可以实现不同机器的 TCP、UDP 通讯，如果你需要一个强大的、灵活、野蛮的通讯能力，别犹豫 ZeroMQ

## Actor模型

### 什么是Actor模型

Actor 模式是一个解决分布式计算的数学模型，其中 Actor 是基础，它能回应接收到消息，能够自我决策，创建更多的 Actor，发送更多的消息，决定如何回应下一个接收到的消息。Actor 认为一切皆是 Actor，类似于面向对象认为一切皆 Object 一样。OO 的执行是顺序的，Actor 模型内在设计就是并行的

### Actor 是异步的

Actor 是计算实体，它回复接收到的消息，能够并行的：

- 发生有限的消息给其他 Actor
- 创建有限数目的新 Actor
- 指定一个消息到达时的行为

这些操作并没有顺序要求，它们能够并行地实施。由于没有对消息的时序做规定，Actor 模式是一种异步模型，发送到 Actor 不等待消息被接收而继续执行。Actor 之间不共享状态，如果想获取其他 Actor 的状态，只能通过消息请求的方式

Actor 在消息内部指定接收消息的 Actor 地址。Actor 可以用自己的地址发送消息，相当于自己接收到自己发送的消息，可以驱动自己的状态

### 所谓真正的 Actor 模型

Actor 可以被认为是在用户空间实现的并发实体，所以它应该是应用级别的线程。如果认同这个观点那么 **Actor 要满足的要求 = 操作系统对进程/线程** 提出的要求一样

### 内存结构

每个并发实体都是要有一个固定的数据结构，必须有一个容器可以保存当前所有的并发实体。这一点基本上很容易满足，Akka 中 Actor 就是一个类，所以它的结构就是这个类的数据结构，大小也就是这个类的大小。Akka 中的 Dispatcher 保存有所有 Actor 的列表

### 并发原语

操作系统的是通过临界区，锁来定义多线程共享数据模型的。在 Actor 中是通过消息来共享数据的。基于消息传递要求“数据只读”，你发送出去的数据再修改肯定就不对了。但是这一点在 Java 里面无论如何都是做不到的，你不修改变量的引用但是还可以修改变量里面的值，调用对象的方法。

### 调度

这是最重要的：没有调度，并发实体根本不能称之为并发实体。操作系统中 CPU 是由内核管理的，调度算法是基于时间片来调任务的，内核随时可以剥夺一个任务的 CPU 使用权这就是“抢占”。这一点非常重要，没有这个功能就意味着调度是不公平的。一个任务耗费大量 CPU 会把另个一任务给饿死。但是在用户空间（应用层）很难实现这一点，毕竟 CPU 是不受应用程序的控制的，没有把办法剥夺。抢占看似可有可无，但是没有它就没有“公平调度”，也就谈不上并发。（有任务撑死，有任务饿死）

### 所谓“公平调度”

比如写两个 Actor，使用无限循环输出字符串（while(true)）会疯狂的吃 CPU，如果是可抢占的公平调度，则 actor1 和 actor2 应该是比较有规律的交替（大家得到的 CPU 时间差不多）

## Java 中的 Akka

```
test1
test1
test1
...
test2
test2
test2
...
test1
...
```

## ErLang

```
test1
test2
test1
test2
test1
test2
test1
test2
...
```

ErLang 非常均匀的任务切换，实现了“可抢占的公平”

# RabbitMQ:

## RabbitMQ 的优点

- 基于 ErLang 语言开发具有高可用高并发的优点，适合辑群服务器
- 健壮、稳定、易用、跨平台、支持多种语言、文档齐全
- 有消息确认机制和持久化机制，可靠性高
- 开源

## RabbitMQ 的概念

### 生产者和消费者

- Producer：消息的生产者
- Consumer：消息的消费者

### Queue

- 消息队列，提供了 FIFO 的处理机制，具有缓存消息的能力。RabbitMQ 中，队列消息可以设置为持久化，临时或者自动删除。
- 设置为持久化的队列，Queue 中的消息会在 Server 本地硬盘存储一份，防止系统 Crash，数据丢失
- 设置为临时队列，Queue 中的数据在系统重启之后就会丢失
- 设置为自动删除的队列，当不存在用户连接到 Server，队列中的数据会被自动删除

### ExChange

Exchange 类似于数据通信网络中的交换机，提供消息路由策略。RabbitMQ 中，Producer 不是通过信道直接将消息发送给 Queue，而是先发送给 ExChange。一个 ExChange 可以和多个 Queue 进行绑定，Producer 在传递消息的时候，会传递一个 ROUTING_KEY，ExChange 会根据这个 ROUTING_KEY 按照特定的路由算法，将消息路由给指定的 Queue。和 Queue 一样，ExChange 也可设置为持久化，临时或者自动删除

#### ExChange 的 4 种类型

- direct（默认）：直接交换器，工作方式类似于单播，ExChange 会将消息发送完全匹配 ROUTING_KEY 的 Queue（key 就等于 queue）
- fanout：广播是式交换器，不管消息的 ROUTING_KEY 设置为什么，ExChange 都会将消息转发给所有绑定的 Queue（无视 key，给所有的 queue 都来一份）
- topic：主题交换器，工作方式类似于组播，ExChange 会将消息转发和 ROUTING_KEY 匹配模式相同的所有队列（key 可以用“宽字符”模糊匹配 queue），比如，ROUTING_KEY 为 `user.stock` 的 Message 会转发给绑定匹配模式为 `* .stock,user.stock`， `* . *` 和 `#.user.stock.#` 的队列。（* 表是匹配一个任意词组，# 表示匹配 0 个或多个词组）
- headers：消息体的 header 匹配，无视 key，通过查看消息的头部元数据来决定发给那个 queue（AMQP 头部元数据非常丰富而且可以自定义）

### Binding

所谓绑定就是将一个特定的 ExChange 和一个特定的 Queue 绑定起来。ExChange 和 Queue 的绑定可以是多对多的关系

### Virtual Host

在 RabbitMQ Server 上可以创建多个虚拟的 Message Broker，又叫做 Virtual Hosts (vhosts)。每一个 vhost 本质上是一个 mini-rabbitmq server，分别管理各自的 ExChange，和 bindings。vhost 相当于物理的 Server，可以为不同 app 提供边界隔离，使得应用安全的运行在不同的 vhost 实例上，相互之间不会干扰。Producer 和 Consumer 连接 rabbit server 需要指定一个 vhost

## RabbitMQ 的使用过程

- 客户端连接到消息队列服务器，打开一个 Channel。
- 客户端声明一个 ExChange，并设置相关属性。
- 客户端声明一个 Queue，并设置相关属性。
- 客户端使用 Routing Key，在 ExChange 和 Queue 之间建立好绑定关系。
- 客户端投递消息到 ExChange。
- ExChange 接收到消息后，就根据消息的 key 和已经设置的 binding，进行消息路由，将消息投递到一个或多个队列里

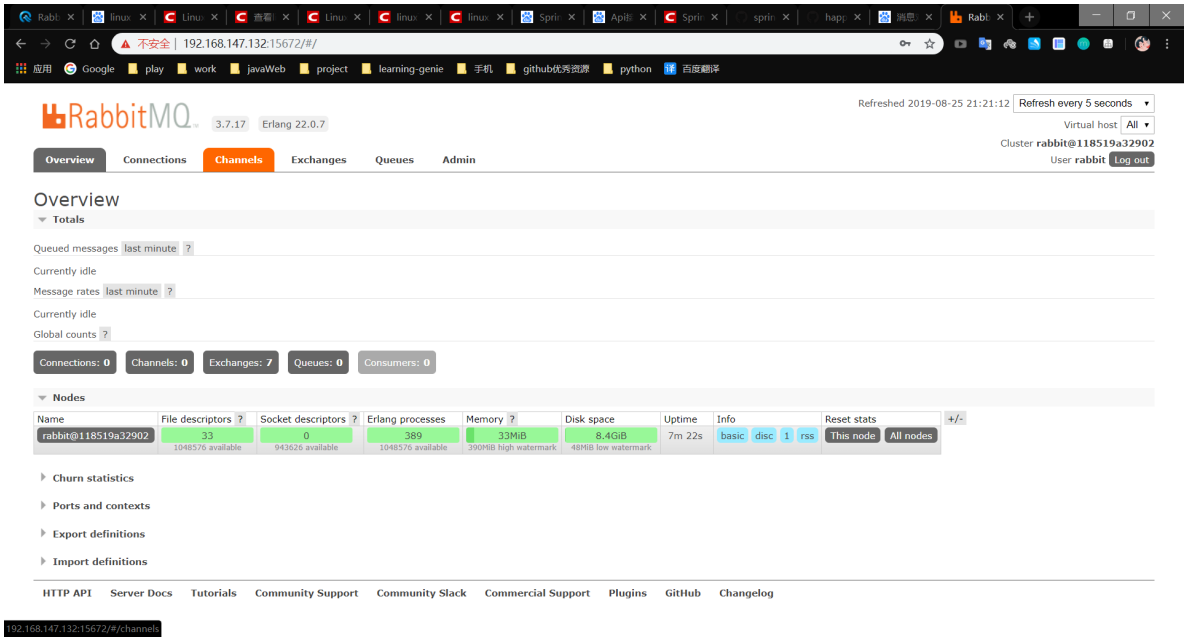
## RabbitMQ安装：

### docker-compose.yml

```
version: '3.1'
services:
  rabbitmq:
    restart: always
    image: rabbitmq:management
    container_name: rabbitmq
    ports:
      - 5672:5672
      - 15672:15672
    environment:
      TZ: Asia/Shanghai
      RABBITMQ_DEFAULT_USER: rabbit
      RABBITMQ_DEFAULT_PASS: 123456
    volumes:
      - ./data:/var/lib/rabbitmq
```

### 访问地址

http://ip:15672




# RabbitMQ 使用

## 生产者

创建一个名为 `spring-boot-amqp-provider` 的生产者项目

### application.yml

```
spring:
  application:
    name: spring-boot-amqp
  rabbitmq:
    host: 192.168.75.133
    port: 5672
    username: rabbit
    password: 123456
```

### 创建队列配置

```
package com.lusifer.spring.boot.amqp.config;

import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfiguration {
    @Bean
    public Queue queue() {
        return new Queue("helloRabbit");
    }
}
```

### 创建消息提供者

```
package com.lusifer.spring.boot.amqp.provider;

import org.springframework.amqp.core.AmqpTemplate;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class HelloRabbitProvider {
    @Autowired
    private AmqpTemplate amqpTemplate;

    public void send() {
        String context = "hello" + new Date();
        System.out.println("Provider: " + context);
        amqpTemplate.convertAndSend("helloRabbit", context);
    }
}
```

**创建测试用例**

```
package com.lusifer.spring.boot.amqp.test;

import com.lusifer.spring.boot.amqp.Application;
import com.lusifer.spring.boot.amqp.provider.HelloRabbitProvider;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class)
public class AmqpTest {
    @Autowired
    private HelloRabbitProvider helloRabbitProvider;

    @Test
    public void testSender() {
        for (int i = 0; i < 10; i++) {
            helloRabbitProvider.send();
        }
    }
}
```

## 消费者

创建一个名为 `spring-boot-amqp-consumer` 的消费者项目

**application.yml**

```
spring:
  application:
    name: spring-boot-amqp-consumer
  rabbitmq:
    host: 192.168.75.133
    port: 5672
    username: rabbit
    password: 123456
```

**使用.html#创建消息消费者)创建消息消费者**

```
package com.lusifer.spring.boot.amqp.consumer;

import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
@RabbitListener(queues = "helloRabbit")
public class HelloRabbitConsumer {
    @RabbitHandler
    public void process(String message) {
        System.out.println("Consumer: " + message);
    }
}
```

# Quartz

Quartz是OpenSymphony开源组织在Job scheduling领域又一个开源项目，它可以与J2EE与J2SE应用程序相结合也可以单独使用。Quartz可以用来创建简单或为运行十个，百个，甚至是好几万个Jobs这样复杂的程序。 Jobs可以做成标准的Java组件或 EJBs。Quartz的最新版本为Quartz 2.3.0。 Quartz是一个完全由java编写的开源作业调度框架。不要让作业调度这个术语吓着你。尽管Quartz框架整合了许多额外功能， 但就其简易形式看，你会发现它易用得简直让人受不了！。 简单地创建一个实现org.quartz.Job接口的java类。Job接口包含唯一的方法： public void execute(JobExecutionContext context)throws JobExecutionException; 在你的Job接口实现类里面，添加一些逻辑到execute()方法。一旦你配置好Job实现类并设定好调度时间表，Quartz将密切注意剩余时间。 当调度程序确定该是通知你的作业的时候，Quartz框架将调用你Job实现类（作业类）上的execute()方法并允许做它该做的事情。无需报告任何东西给调度器或调用任何特定的东西。仅仅执行任务和结束任务即可。如果配置你的作业在随后再次被调用，Quartz框架将在恰当的时间再次调用它

```
    #### 运行环境：
```

Quartz 可以运行嵌入在另一个独立式应用程序。 Quartz 可以在应用程序服务器(或 servlet 容器)内被实例化，并且参与 XA 事务。 Quartz 可以作为一个独立的程序运行(其自己的 Java 虚拟机内)，可以通过 RMI 使用。 Quartz 可以被实例化，作为独立的项目集群(负载平衡和故障转移功能)，用于作业的执行。

## 调度器：

Quartz框架的核心是调度器。调度器负责管理Quartz应用运行时环境。调度器不是靠自己做所有的工作，而是依赖框架内一些非常重要的部件。 Quartz不仅仅是线程和线程管理。为确保可伸缩性，Quartz采用了基于多线程的架构。启动时，框架初始化一套worker线程，这套线程被调度器用来执行预定的作业。这就是Quartz怎样能并发运行多个作业的原理。 Quartz依赖一套松耦合的线程池管理部件来管理线程环境。

## API:

Scheduler - 与调度程序交互的主要API。 Job - 由希望由调度程序执行的组件实现的接口。 JobDetail - 用于定义作业的实例。 Trigger（即触发器） - 定义执行给定作业的计划的组件。 JobBuilder - 用于定义/构建JobDetail实例，用于定义作业的实例。 TriggerBuilder - 用于定义/构建触发器实例。

## 什么是 cron 表达式?

cron 是 Linux 系统用来设置计划任务的，比如：每天晚上 12 点重启服务器。

### 格式

一个 cron 表达式具体表现就是一个字符串，这个字符串中包含 6~7 个字段，字段之间是由空格分割的，每个字段可以由任何允许的值以及允许的特殊字符所构成，下面表格列出了每个字段所允许的值和特殊字符

| 字段 | 允许值 | 允许的特殊字符 |
| --- | --- | --- |
| 秒 | 0-59 | `, - * /` |
| 分 | 0-59 | `, - * /` |
| 小时 | 0-23 | `, - * /` |
| 日期 | 1-31 | `, - * / L W C` |
| 月份 | 1-12 或者 JAN-DEC | `, - * /` |
| 星期 | 1-7 或者 SUN-SAT | `, - * / L C #` |
| 年（可选） | 留空, 1970-2099 | `, - * /` |

- `*` 字符被用来指定所有的值。如：`*` 在分钟的字段域里表示“每分钟”。
- `-` 字符被用来指定一个范围。如：`10-12` 在小时域意味着“10点、11点、12点”
- `,` 字符被用来指定另外的值。如：`MON,WED,FRI` 在星期域里表示“星期一、星期三、星期五”.
- `?` 字符只在日期域和星期域中使用。它被用来指定“非明确的值”。当你需要通过在这两个域中的一个来指定一些东西的时候，它是有用的。
- `L` 字符指定在月或者星期中的某天（最后一天）。即 “Last” 的缩写。但是在星期和月中 “ L ” 表示不同的意思，如：在月字段中 “L” 指月份的最后一天 “1月31日，2月28日”，如果在星期字段中则简单的表示为“7”或者“SAT”。如果在星期字段中在某个 value 值得后面，则表示 “某月的最后一个星期 value” ,如 “6L” 表示某月的最后一个星期五。
- `W` 字符只能用在月份字段中，该字段指定了离指定日期最近的那个星期日。
- `#` 字符只能用在星期字段，该字段指定了第几个星期 value 在某月中

每一个元素都可以显式地规定一个值（如 6），一个区间（如 9-12 ），一个列表（如 9，11，13 ）或一个通配符（如 *）。“月份中的日期”和“星期中的日期”这两个元素是互斥的，因此应该通过设置一个问号（?）来表明你不想设置的那个字段。

| 表达式 | 意义 |
| --- | --- |
| `0 0 12 * * ?` | 每天中午 12 点触发 |
| `0 15 10 ? * *` | 每天上午 10:15 触发 |
| `0 15 10 * * ?` | 每天上午 10:15 触发 |
| `0 15 10 * * ? *` | 每天上午 10:15 触发 |
| `0 15 10 * * ? 2005` | 2005 年的每天上午 10:15 触发 |
| `0 * 14 * * ?` | 在每天下午 2 点到下午 2:59 期间的每 1 分钟触发 |
| `0 0/5 14 * * ?` | 在每天下午 2 点到下午 2:55 期间的每 5 分钟触发 |
| `0 0/5 14,18 * * ?` | 在每天下午 2 点到 2:55 期间和下午 6 点到 6:55 期间的每 5 分钟触发 |
| `0 0-5 14 * * ?` | 在每天下午 2 点到下午 2:05 期间的每 1 分钟触发 |
| `0 10,44 14 ? 3 WED` | 每年三月的星期三的下午 2:10 和 2:44 触发 |
| `0 15 10 ? * MON-FRI` | 周一至周五的上午 10:15 触发 |
| `0 15 10 15 * ?` | 每月 15 日上午 10:15 触发 |
| `0 15 10 L * ?` | 每月最后一日的上午 10:15 触发 |
| `0 15 10 ? * 6L` | 每月的最后一个星期五上午 10:15 触发 |
| `0 15 10 ? * 6L 2002-2005` | 2002 年至 2005 年的每月的最后一个星期五上午 10:15 触发 |
| `0 15 10 ? * 6#3` | 每月的第三个星期五上午 10:15 触发 |

## Spring Boot 集成 Quartz

### 创建项目

创建一个名为 `hello-quartz` 的项目

### POM

```
<?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.funtl</groupId>
    <artifactId>hello-quartz</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>hello-quartz</name>

    <parent>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.4.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-quartz</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

主要增加了 `org.springframework.boot:spring-boot-starter-quartz` 依赖

**Application**

```
package com.funtl.hello.quatrz;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableScheduling;

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

使用 `@EnableScheduling` 注解来开启计划任务功能

**创建任务**

我们创建一个每 5 秒钟打印当前时间的任务来测试 Quartz

```
package com.funtl.hello.quatrz.tasks;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;

@Component
public class PrintCurrentTimeTask {
    @Scheduled(cron = "0/5 * * * * ? ")
    public void printCurrentTime() {
        System.out.println("Current Time is:" + new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date()));
    }
}
```

idea快捷键：ctrl+F9:bulid ctrl+shilft+space:自动补全 ctrl+space :自动完成， ctrl+alt+space:自动导报 ctrl+shilft+enter:在末位添加; ctrl+o重写 ctrl+i：实现方法

# 遇到的问题

## 1.在配置eureka中配置端口不生效导致很多服务未能够注册一直是在8080端口注册：

2019-06-21 15:13:10.753 ERROR [-,,,] 17908 --- [nfoReplicator-0] c.n.d.s.t.d.RedirectingEurekaHttpClient : Request execution error com.sun.jersey.api.client.ClientHandlerException: java.net.ConnectException: Connection refused: connect

```
    #### 解决办法：
```

新版本的security默认开启了csrf关掉即可。不关闭的话会令eureka的访问路径变为8080不是自己配置的port了： 这个也不行配置后 还有说是加入swager这版本配置： 去掉这这个配置文件还不能生效，那么久不是这问题了 我个人的解决办法，是把这个eureka这个服务给清除，然后从新打开idea这样就是可以了，我怀疑就是在配置中心的 这些问题，他访问时不能够访问到子节点，所以把这个项目从新bulider 然后再次把这些上传到github中，然后重新启动这样就可以访问到这个路径了。其实就是eureka没有找到这个配置中心中的他的本地的yml但是又一想这个已经是在本地的8888端口下访问的为什么 不能够访问呢，还是这个config工程的问题，但是从新restart后可以了。我深刻怀疑是idea中的bug.不过这个bug就此解决了。

## 2.在启动服务消费者的项目中itoken-service-admin时出现的错误是：Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

然而这时需要去配置中心去看结果，因为他是在配置中心拿的结果所以当检查配置中心的控制太打印是mybatis扫描包.xml报错2，这里又是不能够加载到这个xml所以配置改为 spring.mybatis.type-aliases-page:com.kayleoi.itoken.service.admin.domain spring.mybatis.mapper-localtions: classpath:*mapper/*.xml 这里就不会出现错误了，所以这里进行对其修改，然后就可以正常运行了。

## 3.启动admin项目时同样报错：和上面的错误一样到但是配置中心没有出现错误信息，所以这里应该如何修正呢，

## 4。org.mybatis.spring.MyBatisSystemException: nested exception is org.apache.ibatis.builder.BuilderExceptioncom.kayleoi.itoken.service.admin.test.service.AdminServiceTest#regist

这里的自动扫描的mapper：import tk.mybatis.spring.annotation.MapperScan;而不是org.mybatis..
@MapperScan(basePackages = "com.kayleoi.itoken.service.admin.mapper")

## 5.在启动服务admin使出错Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured：

```
<!--加入下面必须报错是Failed to configure a DataSource: 'url' attribute is not
specified and no embedded datasource could be configured：这里其实加入
        这个必须配置其其他的druid的属性配置-->
        <!--<dependency>-->
            <!--<groupId>com.alibaba</groupId>-->
            <!--<artifactId>druid-spring-boot-starter</artifactId>-->
        <!--</dependency>-->
        <!--<dependency>-->
            <!--<groupId>tk.mybatis</groupId>-->
            <!--<artifactId>mapper-spring-boot-starter</artifactId>-->
        <!--</dependency>-->
        <!--<dependency>-->
            <!--<groupId>com.github.pagehelper</groupId>-->
            <!--<artifactId>pagehelper-spring-boot-starter</artifactId>-->
        <!--</dependency>-->
```

## 6.com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '-service-admin..tb_sys_user' at line 1

## 7.使用@AutoWire不可以自动注入dao:

**需要使用@resources**