



Spring Boot

内容概要

- 一、Spring Boot入门
- 二、Spring Boot配置
- 三、Spring Boot与日志
- 四、Spring Boot与Web开发
- 五、Spring Boot与Docker
- 六、Spring Boot与数据访问
- 七、Spring Boot启动配置原理
- 八、Spring Boot自定义starters
- 九、Spring Boot与缓存
- 十、Spring Boot与消息
- 十一、Spring Boot与检索
- 十二、Spring Boot与任务
- 十三、Spring Boot与安全
- 十四、Spring Boot与分布式
- 十五、Spring Boot与开发热部署
- 十六、Spring Boot与监控管理

一、Spring Boot入门

简介、HelloWorld、原理分析

一、简介

Spring Boot来简化Spring应用开发，**约定大于配置**，去繁从简，just run就能创建一个独立的，产品级别的应用

背景：

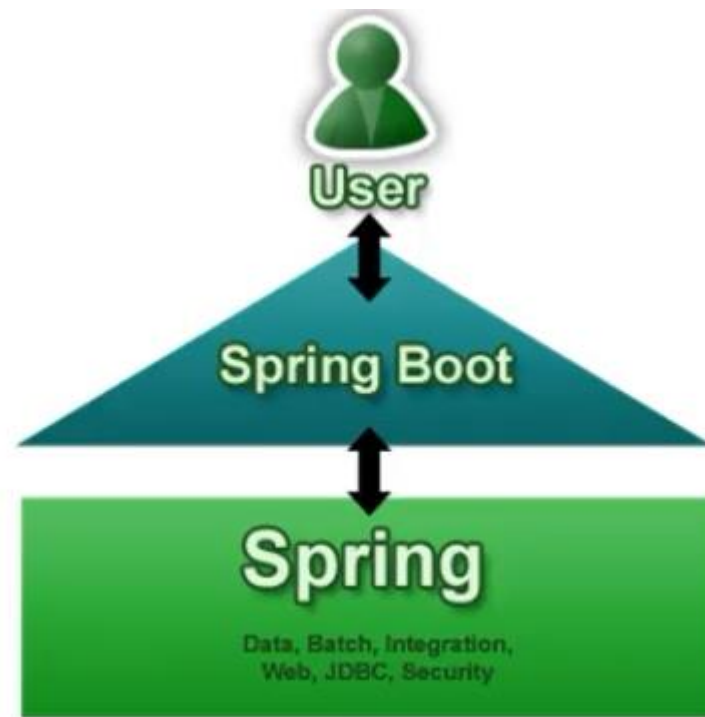
J2EE笨重的开发、繁多的配置、低下的开发效率、复杂的部署流程、第三方技术集成难度大。

解决：




“Spring全家桶”时代。

Spring Boot → J2EE一站式解决方案

Spring Cloud → 分布式整体解决方案

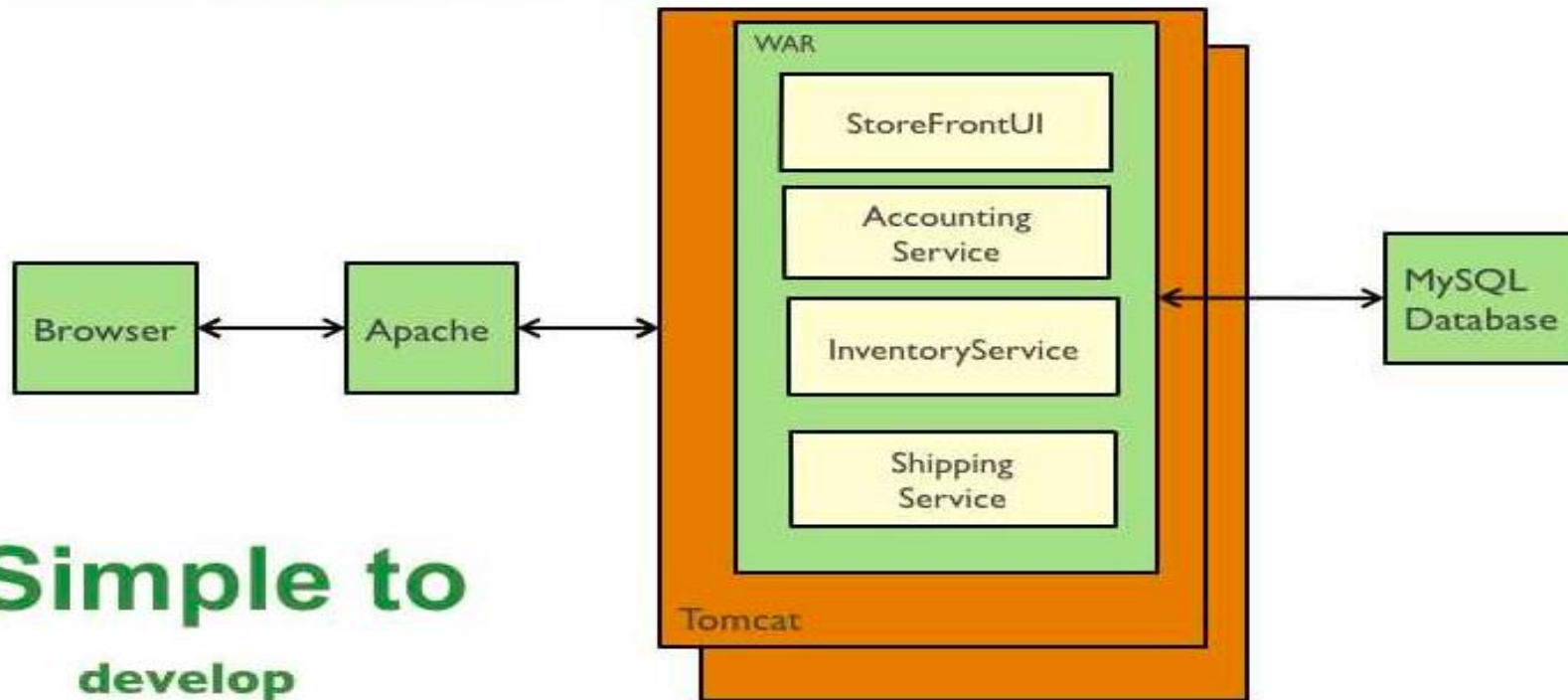


- 优点：

- 快速创建独立运行的Spring项目以及与主流框架集成
- 使用嵌入式的Servlet容器，应用无需打成WAR包 
- starters自动依赖与版本控制 
- 大量的自动配置，简化开发，也可修改默认值
- 无需配置XML，无代码生成，开箱即用 
- 准生产环境的运行时应用监控
- 与云计算的天然集成

单体应用

Traditional web application architecture



Simple to

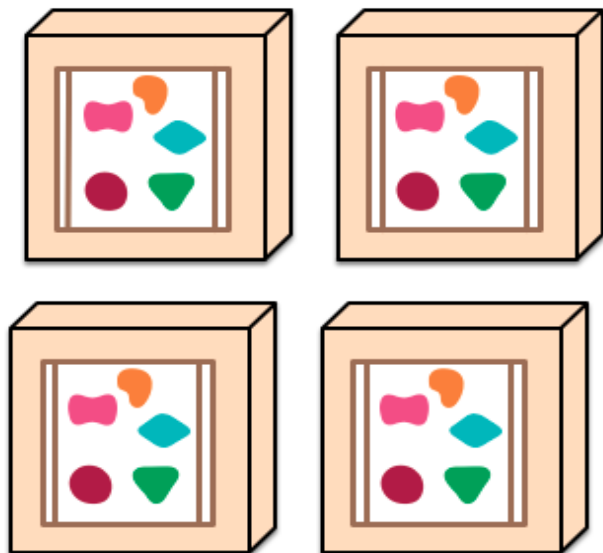
develop
test
deploy
scale

微服务

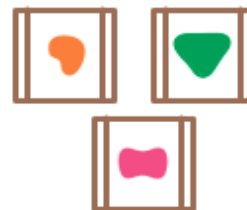
一个单体应用程序把它所有的功能放在一个单一进程中...



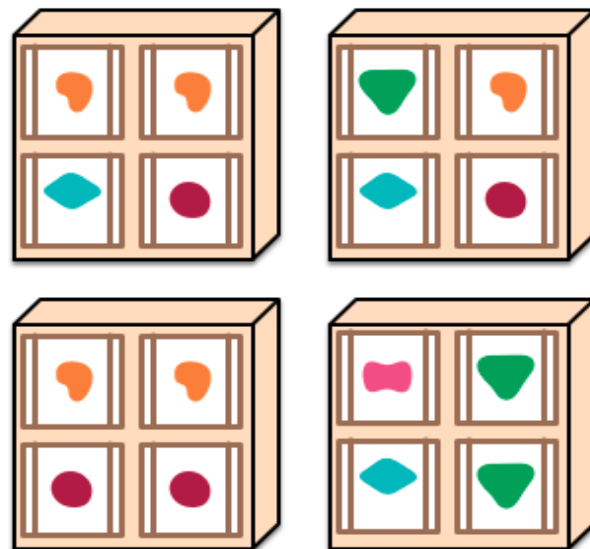
...并且通过在多个服务器上复制这个单体进行扩展



一个微服务架构把每个功能元素放进一个独立的服务中...



...并且通过跨服务器分发这些服务进行扩展，只在需要时才复制。



How Microservices



Josh Evans

@Ops_Engineering

Director of Operations Engineering at
Netflix



- 你必须掌握以下内容：
 - Spring框架的使用经验
 - 熟练使用Maven进行项目构建和依赖管理
 - 熟练使用Eclipse或者IDEA
- 环境约束
 - jdk1.8
 - maven3.x
 - IntelliJ IDEA 2017
 - Spring Boot 1.5.9.RELEASE

二、HelloWorld

- 1、创建maven项目
- 2、引入starters
- 3、创建主程序
- 4、启动运行

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.7.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

2

```
package hello;
```

3

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;
```

```
@Controller
```

```
@EnableAutoConfiguration
```

```
public class SampleController {
```

```
    @RequestMapping("/")
```

```
    @ResponseBody
```

```
    String home() {
```

```
        return "Hello World!";
```

```
    }
```

```
    public static void main(String[] args) throws Exception {
        SpringApplication.run(SampleController.class, args);
```

```
    }
```

```
}
```


三、HelloWorld探究

1、starters

- Spring Boot为我们提供了简化企业级开发绝大多数场景的starter pom（启动器），只要引入了相应场景的starter pom，相关技术的绝大部分配置将会消除（自动配置），从而简化我们开发。业务中我们就会使用到Spring Boot为我们自动配置的bean
- 参考 <https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#using-boot-starter>
- 这些starters几乎涵盖了javaee所有常用场景，Spring Boot对这些场景依赖的jar也做了严格的测试与版本控制。我们不必担心jar版本合适度问题。
- spring-boot-dependencies里面定义了jar包的版本

2、入口类和@SpringBootApplication

- 1、程序从main方法开始运行
- 2、使用SpringApplication.run()加载主程序类
- 3、主程序类需要标注@SpringBootApplication
- 4、@EnableAutoConfiguration是核心注解；
- 5、@Import导入所有的自动配置场景
- 6、@AutoConfigurationPackage定义默认的包扫描规则
- 7、程序启动扫描加载主程序类所在的包以及下面所有子包的组件；

```
@SpringBootApplication
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @Filter(type = FilterType.CUSTOM
    @Filter(type = FilterType.CUSTOM
public @interface SpringBootApplication
```

```
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

```
@Configuration
public @interface SpringBootConfiguration {
}
```

```
@Component
public @interface Configuration {
```

3、自动配置

- 1、xxxAutoConfiguration
 - Spring Boot中存在大量的这些类，这些类的作用就是帮我们进行自动配置
 - 他会将这个这个场景需要的所有组件都注册到容器中，并配置好
 - 他们在类路径下的 META-INF/spring.factories文件中
 - spring-boot-autoconfigure-1.5.9.RELEASE.jar中包含了所有场景的自动配置类代码
 - 这些自动配置类是Spring Boot进行自动配置的精髓

二、Spring Boot配置

配置文件、加载顺序、配置原理

一、配置文件

- Spring Boot使用一个全局的配置文件
 - application.properties
 - application.yml
- 配置文件放在src/main/resources目录或者类路径/config下
- .yml是YAML (YAML Ain't Markup Language) 语言的文件，以数据为中心，比json、xml等更适合做配置文件
 - <http://www.yaml.org/> 参考语法规范
- 全局配置文件的可以对一些默认配置值进行修改

二、YAML语法

1、YAML基本语法

- 使用缩进表示层级关系
- 缩进时不允许使用Tab键，只允许使用空格。
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
- 大小写敏感

2、YAML 支持的三种数据结构

- 对象：键值对的集合
- 数组：一组按次序排列的值
- 字面量：单个的、不可再分的值

- YAML常用写法

- 对象 (Map)

- 对象的一组键值对，使用冒号分隔。如：username: admin
 - 冒号后面跟**空格**来分键值；
 - {k: v}是行内写法

```
# YAML
hero:
  hp: 34
  sp: 8
  level: 4
orc:
  hp: 12
  sp: 0
  level: 2
```

```
# Java
{'hero': {'hp': 34, 'sp': 8, 'level': 4}, 'orc': {'hp': 12, 'sp': 0, 'level': 2}}
```

• 数组

- 一组连词线 (-) 开头的行，构成一个数组，[]为行内写法
- 数组，对象可以组合使用

```
# YAML
- name: PyYAML
  status: 4
  license: MIT
  language: Python
- name: PySyck
  status: 5
  license: BSD
  language: Python
```

```
# Java
[{'status': 4, 'language': 'Python', 'name': 'PyYAML', 'license': 'MIT'},
{'status': 5, 'license': 'BSD', 'name': 'PySyck', 'language': 'Python'}]
```

```
- Cat
- Dog
- Goldfish
```

①

```
- Cat
- Dog
- Goldfish
```

②

```
var obj = [ 'Cat', 'Dog', 'Goldfish' ];
```

①

```
var obj = [ [ 'Cat', 'Dog', 'Goldfish' ] ];
```

②

```
animal: [Cat, Dog]
```

③

```
var obj = { animal: [ 'Cat', 'Dog' ] };
```

③

- 复合结构。以上写法的任意组合都是可以
- 字面量
 - 数字、字符串、布尔、日期
 - 字符串
 - 默认不使用引号
 - 可以使用单引号或者双引号，单引号会转义特殊字符
 - 字符串可以写成多行，从第二行开始，必须有一个单空格缩进。换行符会被转为空格。
- 文档
 - 多个文档用 - - - 隔开

注意：

Spring Boot使用 snakeyaml 解析yaml文件；

<https://bitbucket.org/asomov/snakeyaml/wiki/Documentation#markdown-header-yaml-syntax> 参考语法


```
person:
  name: 'zhangsan \n'
  username: 张三
  age: 18
  pet:
    name: 小狗
    gender: male
  animal:
    - dog
    - cat
    - fish
  interests: [足球, 篮球]
  friends:
    -
      - zhangsan is my
        best friend
      - "lisi\n"
  childs:
    - name: xiaozhang
      age: 18
    - name: xiaoli
      pets:
        - a
        - b
    - {name: lisi, age: 18}
```

```
@Component
@ConfigurationProperties(prefix = "person")
public class Person {
    private String name;
    private String username;
    private Integer age;
    private Map<String, Object> pet;
    private List<String> animal;
    private List<String> interests;
    private List<Object> friends;
    private List<Map<String, Object>> childs;
```

Pets{name='zhangsan \n', username='张三', age=18, pet={name=小狗, gender=male}, animal=[dog, cat, fish], interests=[足球, 篮球], friends=[[zhangsan is my best friend, lisi]], childs=[{age=18, name=xiaozhang}, {pets={1=b, 0=a}, name=xiaoli}, {age=18, name=lisi}]}

三、配置文件值注入

- **@Value和@ConfigurationProperties为属性注值对比**

Feature	@ConfigurationProperties	@Value
Relaxed binding	Yes	No
Meta-data support	Yes	No
SpEL evaluation	No	Yes

- **属性名匹配规则 (Relaxed binding)**

- person.firstName : 使用标准方式
- person.first-name : 大写用-
- person.first_name : 大写用_
- PERSON_FIRST_NAME :
 - 推荐系统属性使用这种写法

- **@ConfigurationProperties**
 - 与@Bean结合为属性赋值
 - 与@PropertySource（只能用于properties文件）结合读取指定文件
- **@ConfigurationProperties Validation**
 - 支持JSR303进行配置文件值校验；

```
@ConfigurationProperties(prefix="connection")
@Validated
public class FooProperties {

    @NotNull
    private InetAddress remoteAddress;

    @Valid
    private final Security security = new Security();
}
```

- **@ImportResource读取外部配置文件**

四、配置文件占位符

- **RandomValuePropertySource**：配置文件中可以使用随机数

`${random.value}`、`${random.int}`、`${random.long}`

`${random.int(10)}`、`${random.int[1024,65536]}`

- **属性配置占位符**

```
app.name=MyApp  
app.description=${app.name} is a Spring Boot application
```

- 可以在配置文件中引用前面配置过的属性（优先级前面配置过的这里都能用）。
- `${app.name:默认值}`来指定找不到属性时的默认值

五、Profile

Profile是Spring对不同环境提供不同配置功能的支持，可以通过激活、指定参数等方式快速切换环境

1、多profile文件形式：

- 格式：application-{profile}.properties/yml：
 - application-dev.properties、application-prod.properties

2、多profile文档块模式：

3、激活方式：

- 命令行 `--spring.profiles.active=dev`
- 配置文件 `spring.profiles.active=dev`
- jvm参数 `-Dspring.profiles.active=dev`

```
spring:
  profiles:
    active: prod # profiles.active: 激活指定配置
---
spring:
  profiles: prod
server:
  port: 80
--- #三个短横线分割多个profile区（文档块）
spring:
  profiles: default # profiles: default表示未指定默认配置
server:
  port: 8080
```

六、配置文件加载位置

- spring boot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件
 - file:./config/
 - file:./
 - classpath:/config/
 - classpath:/
 - 以上是按照**优先级从高到低**的顺序，所有位置的文件都会被加载，**高优先级配置内容会覆盖低优先级配置内容**。
 - 我们也可以通过配置spring.config.location来改变默认配置

七、外部配置加载顺序

Spring Boot 支持多种外部配置方式

这些方式优先级如下：

<https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-external-config>

1. 命令行参数
2. 来自java:comp/env的JNDI属性
3. Java系统属性 (System.getProperties())
4. 操作系统环境变量
5. RandomValuePropertySource配置的random.*属性值
6. jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件
7. jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件
8. jar包外部的application.properties或application.yml(不带spring.profile)配置文件
9. jar包内部的application.properties或application.yml(不带spring.profile)配置文件
10. @Configuration注解类上的@PropertySource
11. 通过SpringApplication.setDefaultProperties指定的默认属性

八、自动配置原理

- 1、可以查看HttpEncodingAutoConfiguration
- 2、通用模式
 - xxxAutoConfiguration：自动配置类
 - xxxProperties：属性配置类
 - yml/properties文件中能配置的值就来源于[属性配置类]
- 3、几个重要注解
 - @Bean
 - @Conditional
- 4、--debug=true查看详细的自动配置报告

@Conditional扩展

@Conditional扩展注解	作用（判断是否满足当前指定条件）
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean；
@ConditionalOnMissingBean	容器中不存在指定Bean；
@ConditionalOnExpression	满足SpEL表达式指定
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean，或者这个Bean是首选Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定资源文件
@ConditionalOnWebApplication	当前是web环境
@ConditionalOnNotWebApplication	当前不是web环境
@ConditionalOnJndi	JNDI存在指定项

三、Spring Boot与日志

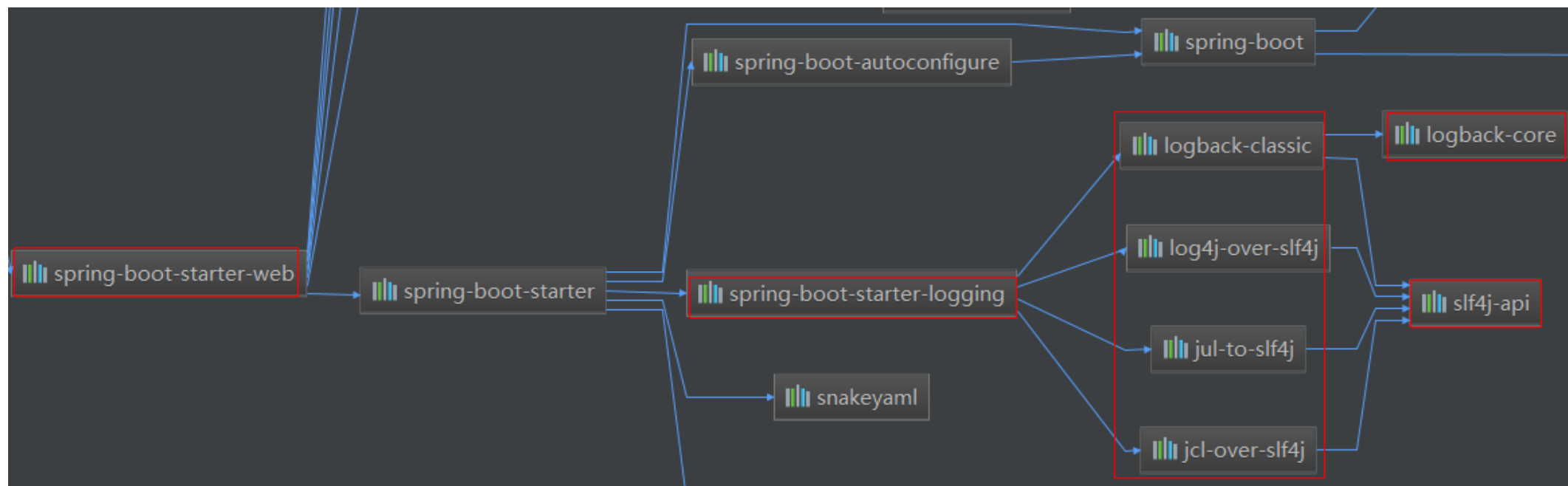
日志框架、日志配置

一、日志框架

市场上存在非常多的日志框架。JUL (java.util.logging) , JCL (Apache Commons Logging) , Log4j , Log4j2 , Logback、SLF4j、jboss-logging等。Spring Boot在框架内容部使用JCL , spring-boot-starter-logging采用了slf4j+logback的形式 , Spring Boot也能自动适配 (jul、log4j2、logback) 并简化配置

日志门面	日志实现
JCL (Jakarta Commons Logging)	Log4j JUL (java.util.logging)
SLF4j (Simple Logging Facade for Java)	Log4j2 Logback
jboss-logging	

Logging System	Customization
Logback	<code>logback-spring.xml</code> , <code>logback-spring.groovy</code> , <code>logback.xml</code> or <code>logback.groovy</code>
Log4j2	<code>log4j2-spring.xml</code> or <code>log4j2.xml</code>
JDK (Java Util Logging)	<code>logging.properties</code>



二、默认配置

- 1、全局常规设置（格式、路径、级别）
 - 2、指定日志配置文件位置
 - 3、切换日志框架
- 二选一

`spring-boot-starter-log4j2`

Starter for using Log4j2 for logging. An alternative to `spring-boot-starter-logging`

`spring-boot-starter-logging`

Starter for logging using Logback. Default logging starter

logging.file	logging.path	Example	Description
(none)	(none)		只在控制台输出
指定文件名	(none)	my.log	输出日志到my.log文件
(none)	指定目录	/var/log	输出到指定目录的 spring.log 文件中

四、Spring Boot与Web开发

Thymeleaf、web定制、容器定制

一、web自动配置规则

- 1、WebMvcAutoConfiguration
- 2、WebMvcProperties
- 3、ViewResolver自动配置
- 4、静态资源自动映射
- 5、Formatter与Converter自动配置
- 6、HttpMessageConverter自动配置
- 7、静态首页
- 8、favicon
- 9、错误处理

二、Thymeleaf模板引擎

Thymeleaf是一款用于渲染XML/XHTML/HTML5内容的模板引擎。类似JSP，Velocity，FreeMaker等，它也可以轻易的与Spring MVC等Web框架进行集成作为Web应用的模板引擎。与其它模板引擎相比，Thymeleaf最大的特点是能够直接在浏览器中打开并正确显示模板页面，而不需要启动整个Web应用

Spring Boot推荐使用Thymeleaf、Freemarker等后现代的模板引擎技术；一旦导入相关依赖，会自动配置ThymeleafAutoConfiguration、FreeMarkerAutoConfiguration。

1、整合Thymeleaf

- 1、导入starter-thymeleaf
- 2、template文件夹下创建模板文件
- 3、测试页面&取值
- 4、基本配置

2、基本语法

- 表达式：
 - #{...}：国际化消息
 - \${...}：变量取值
 - *{...}：当前对象/变量取值
 - @{...}：url表达式
 - ~{...}：片段引用
 - 内置对象/共用对象：
- 判断/遍历：
 - th:if
 - th:unless
 - th:each
 - th:switch、 th:case
- th:属性

三、定制web扩展配置

1、WebMvcConfigurerAdapter

Spring Boot提供了很多xxxConfigurerAdapter来定制配置

2、定制SpringMVC配置

3、@EnableWebMvc全面接管SpringMVC

4、注册view-controller、interceptor等

5、注册Interceptor

四、配置嵌入式Servlet容器

1、ConfigurableEmbeddedServletContainer

2、EmbeddedServletContainerCustomizer

3、注册Servlet、Filter、Listener

ServletRegistrationBean

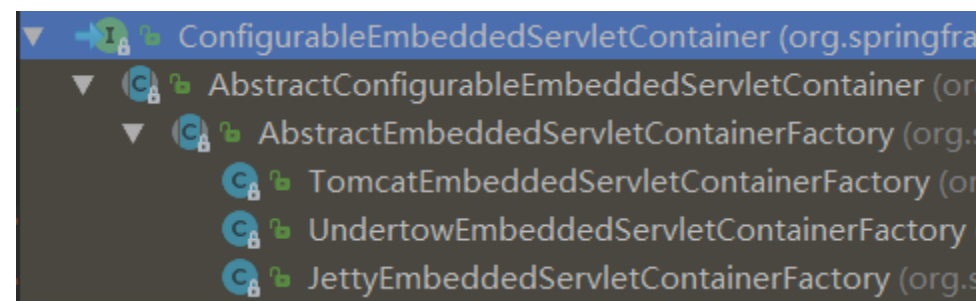
FilterRegistrationBean

ServletListenerRegistrationBean

4、使用其他Servlet容器

Jetty (长连接)

Undertow (不支持JSP)



```
public interface EmbeddedServletContainerCustomizer {  
  
    /**  
     * Customize the specified {@link ConfigurableEmbeddedServletContainer}.  
     * @param container the container to customize  
     */  
    void customize(ConfigurableEmbeddedServletContainer container);  
}
```

五、使用外部Servlet容器

1、SpringBootServletInitializer

- 重写configure

2、SpringApplicationBuilder

- builder.source(@SpringBootApplication类)

3、启动原理

- Servlet3.0标准ServletContainerInitializer扫描所有jar包中META-INF/services/javax.servlet.ServletContainerInitializer文件指定的类并加载
- 加载spring web包下的SpringServletContainerInitializer
- 扫描@HandleType(WebApplicationInitializer)
- 加载SpringBootServletInitializer并运行onStartup方法
- 加载@SpringBootApplication主类，启动容器等

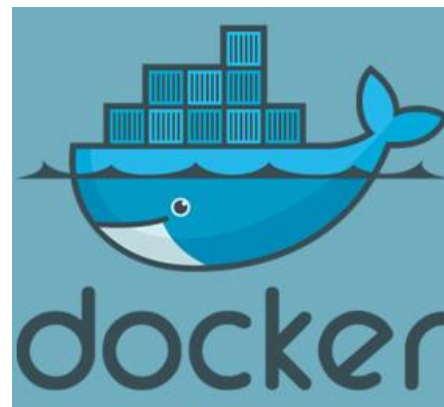
五、Spring Boot与Docker

Docker基本使用、Docker环境

一、何为Docker ?

Docker 是一个[开源的应用容器引擎](#)，基于 [Go 语言](#) 并遵从Apache2.0协议开源。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,更重要的是容器性能开销极低。

Docker支持将软件编译成一个镜像；然后在镜像中各种软件做好配置，将镜像发布出去，其他使用者可以直接使用这个镜像。运行中的这个镜像称为容器，容器启动是非常快速的。类似windows里面的ghost操作系统，安装好后什么都有了；



二、Docker核心概念

docker镜像(Images)：Docker 镜像是用于创建 Docker 容器的模板。

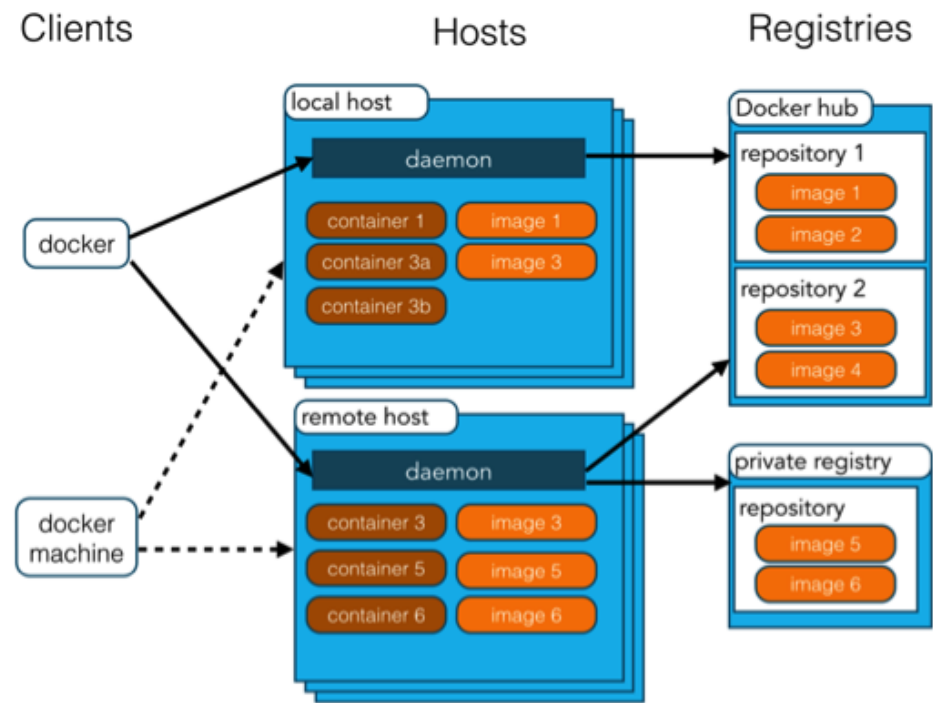
docker容器(Container)：容器是独立运行的一个或一组应用。

docker客户端(Client)：客户端通过命令行或者其他工具使用 Docker

API(https://docs.docker.com/reference/api/docker_remote_api)与 Docker 的守护进程通信

docker主机(Host)：一个物理或者虚拟的机器用于执行 Docker 守护进程和容器。

docker仓库(Registry)：Docker 仓库用来保存镜像，可以理解为代码控制中的代码仓库。Docker Hub(<https://hub.docker.com>) 提供了庞大的镜像集合供使用。



三、Docker安装

- 查看centos版本；

Docker 要求 CentOS 系统的内核版本高于 3.10

```
[root@atguigu ~]# uname -r  
3.10.0-327.el7.x86_64
```

- 升级软件包及内核；（选做）

```
[root@atguigu ~]# yum update
```

- 安装docker

```
[root@atguigu ~]# yum install docker
```

- 启动docker

```
[root@atguigu ~]# systemctl start docker
```

- 将docker服务设为开机启动

```
[root@atguigu ~]# systemctl enable docker
```

四、常用操作

1、镜像操作

操作	命令	说明
检索	<code>docker search 关键字</code> eg : <code>docker search redis</code>	我们经常去docker hub上检索镜像的详细信息，如镜像的TAG。
拉取	<code>docker pull 镜像名:tag</code>	:tag是可选的，tag表示标签，多为软件的版本，默认是latest
列表	<code>docker images</code>	查看所有本地镜像
删除	<code>docker rmi image-id</code>	删除指定的本地镜像

2、容器操作

操作	命令	说明
运行	<code>docker run --name container-name -d image-name</code> eg: <code>docker run --name myredis -d redis</code>	--name : 自定义容器名 -d : 后台运行 image-name: 指定镜像模板
列表	<code>docker ps</code> (查看运行中的容器) ;	加上-a ; 可以查看所有容器
停止	<code>docker stop container-name/container-id</code>	停止当前你运行的容器
启动	<code>docker start container-name/container-id</code>	启动容器
删除	<code>docker rm container-id</code>	删除指定容器
端口映射	-p 6379:6379 eg: <code>docker run -d -p 6379:6379 --name myredis docker.io/redis</code>	-p: 主机端口(映射到)容器内部的端口
容器日志	<code>docker logs container-name/container-id</code>	
更多命令	https://docs.docker.com/engine/reference/commandline/docker/	

五、环境搭建

- 1、安装mysql
- 2、安装redis
- 3、安装rabbitmq
- 4、安装elasticsearch

六、Spring Boot与数据访问

JBDC、MyBatis、Spring Data JPA

一、简介

对于数据访问层，无论是SQL还是NOSQL，Spring Boot默认采用整合Spring Data的方式进行统一处理，添加大量自动配置，屏蔽了很多设置。引入各种xxxTemplate，xxxRepository来简化我们对数据访问层的操作。对我们来说只需要进行简单的设置即可。我们将在数据访问章节测试使用SQL相关、NOSQL在缓存、消息、检索等章节测试。

- JDBC
- MyBatis
- JPA

<code>spring-boot-starter-jdbc</code>	Starter for using JDBC with the Tomcat JDBC connection pool
<code>spring-boot-starter-data-cassandra</code>	Starter for using Cassandra distributed database and Spring Data Cassandra
<code>spring-boot-starter-data-cassandra-reactive</code>	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive
<code>spring-boot-starter-data-couchbase</code>	Starter for using Couchbase document-oriented database and Spring Data Couchbase
<code>spring-boot-starter-data-elasticsearch</code>	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch
<code>spring-boot-starter-data-jpa</code>	Starter for using Spring Data JPA with Hibernate
<code>spring-boot-starter-data-ldap</code>	Starter for using Spring Data LDAP
<code>spring-boot-starter-data-mongodb</code>	Starter for using MongoDB document-oriented database and Spring Data MongoDB
<code>spring-boot-starter-data-mongodb-reactive</code>	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive
<code>spring-boot-starter-data-neo4j</code>	Starter for using Neo4j graph database and Spring Data Neo4j
<code>spring-boot-starter-data-redis</code>	Starter for using Redis key-value data store with Spring Data Redis and the Jedis client
<code>spring-boot-starter-data-redis-reactive</code>	Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client
<code>spring-boot-starter-data-rest</code>	Starter for exposing Spring Data repositories over REST using Spring Data REST
<code>spring-boot-starter-data-solr</code>	Starter for using the Apache Solr search platform with Spring Data Solr

二、整合基本JDBC与数据源

- 1、引入starter
 - spring-boot-starter-jdbc
- 2、配置application.yml
- 3、测试
- 4、高级配置：使用druid数据源
 - 引入druid
 - 配置属性
- 5、配置druid数据源监控

三、整合MyBatis

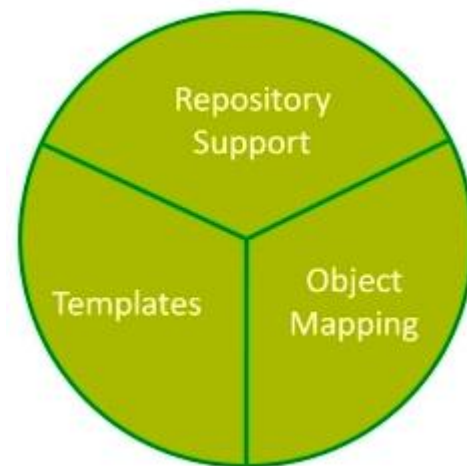
- 1、引入mybatis-starter
 - mybatis-spring-boot-starter
- 2、注解模式
- 3、配置文件模式
- 4、测试

四、Spring Data

简介：

Spring Data 项目的目的是为了简化构建基于 Spring 框架应用的数据访问技术，包括非关系数据库、Map-Reduce 框架、云数据服务等等；另外也包含对关系数据库的访问支持。

- Spring Data 包含多个子项目：
 - Spring Data Commons
 - Spring Data JPA
 - Spring Data KeyValue
 - Spring Data LDAP
 - Spring Data MongoDB
 - Spring Data Gemfire
 - Spring Data REST
 - Spring Data Redis
 - Spring Data for Apache Cassandra
 - Spring Data for Apache Solr
 - Spring Data Couchbase (community module)
 - Spring Data Elasticsearch (community module)
 - Spring Data Neo4j (community module)



1、SpringData特点

SpringData为我们提供使用统一的API来对数据访问层进行操作；这主要是Spring Data Commons项目来实现的。Spring Data Commons让我们在使用关系型或者非关系型数据访问技术时都基于Spring提供的统一标准，标准包含了CRUD（创建、获取、更新、删除）、查询、排序和分页的相关操作。

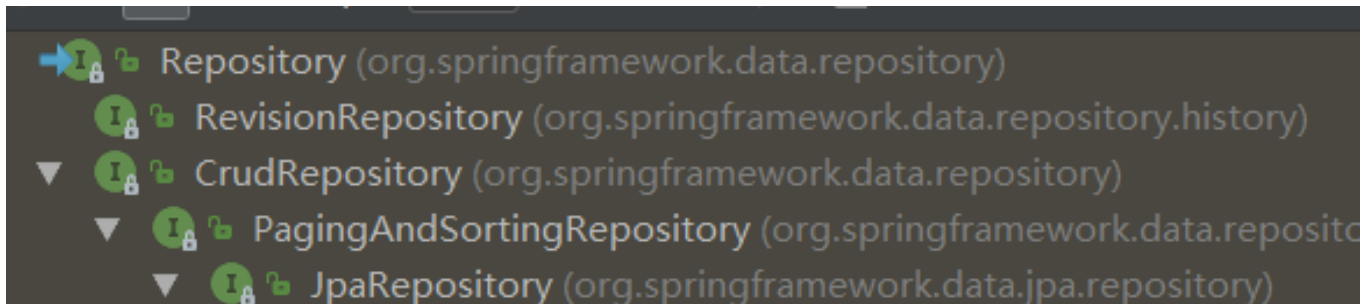
2、统一的Repository接口

Repository<T, ID extends Serializable>：统一接口

RevisionRepository<T, ID extends Serializable, N extends Number & Comparable<N>>：基于乐观锁机制

CrudRepository<T, ID extends Serializable>：基本CRUD操作

PagingAndSortingRepository<T, ID extends Serializable>：基本CRUD及分页



3、提供数据访问模板类 xxxTemplate ；

如：MongoTemplate、RedisTemplate等

4、JPA与Spring Data

1)、JpaRepository基本功能

编写接口继承JpaRepository既有crud及分页等基本功能

2)、定义符合规范的方法命名

在接口中只需要声明符合规范的方法，即拥有对应的功能

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	... where x.startDate between 1? and ?2

3)、@Query自定义查询，定制查询SQL

4)、Specifications查询 (Spring Data JPA支持JPA2.0的Criteria查询)

五、整合JPA

- 1、引入spring-boot-starter-data-jpa
- 2、配置文件打印SQL语句
- 3、创建Entity标注JPA注解
- 4、创建Repository接口继承JpaRepository
- 5、测试方法

七、Spring Boot启动配置原理

启动原理、运行流程、自动配置原理

一、启动原理

- SpringApplication.run(主程序类)
 - new SpringApplication(主程序类)
 - 判断是否web应用
 - 加载并保存所有ApplicationContextInitializer (META-INF/spring.factories) ,
 - 加载并保存所有ApplicationListener
 - 获取到主程序类
 - run()
 - 回调所有的SpringApplicationRunListener (META-INF/spring.factories) 的starting
 - 获取ApplicationArguments
 - 准备环境&回调所有监听器 (SpringApplicationRunListener) 的environmentPrepared
 - 打印banner信息
 - 创建ioc容器对象 (
 - AnnotationConfigEmbeddedWebApplicationContext (web环境容器)
 - AnnotationConfigApplicationContext (普通环境容器)

– run()

- 准备环境
 - 执行 **ApplicationContextInitializer**. initialize()
 - 监听器 **SpringApplicationRunListener** 回调 contextPrepared
 - 加载主配置类定义信息
 - 监听器 **SpringApplicationRunListener** 回调 contextLoaded
- 刷新启动IOC容器；
 - 扫描加载所有容器中的组件
 - 包括从 META-INF/spring.factories 中获取的所有 **EnableAutoConfiguration** 组件
- 回调容器中所有的 **ApplicationRunner**、**CommandLineRunner** 的 run 方法
- 监听器 **SpringApplicationRunListener** 回调 finished

二、自动配置

- Spring Boot启动扫描所有jar包的META-INF/spring.factories中配置的EnableAutoConfiguration组件
- spring-boot-autoconfigure.jar\META-INF\spring.factories有启动时需要加载的EnableAutoConfiguration组件配置
- 配置文件中使用debug=true可以观看到当前启用的自动配置的信息
- 自动配置会为容器中添加大量组件
- Spring Boot在做任何功能都需要从容器中获取这个功能的组件
- Spring Boot 总是遵循一个标准；容器中有我们自己配置的组件就用我们配置的，没有就用自动配置默认注册进来的组件；

八、Spring Boot自定义starters

starters原理、自定义starters

一、自定义starters

- 自动装配Bean ;
 - 自动装配使用配置类 (@Configuration) 结合Spring4 提供的条件判断注解 @Conditional及Spring Boot的派生注解如@ConditionalOnClass完成 ;
- 配置自动装配Bean ;
 - 将标注@Configuration的自动配置类 , 放在classpath下META-INF/spring.factories文件中 , 如 :

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
```

- 自动装配顺序
 - 在特定自动装配Class之前
 - @AutoConfigureBefore
 - 在特定自动装配Class之后
 - @AutoConfigureAfter
 - 指定顺序
 - @AutoConfigureOrder

- 启动器 (starter)

- 启动器模块是一个空 JAR 文件，仅提供辅助性依赖管理，这些依赖可能用于自动装配或者其他类库

- 命名规约：

- 推荐使用以下命名规约；

- 官方命名空间

- 前缀：“spring-boot-starter-”

- 模式：spring-boot-starter-模块名

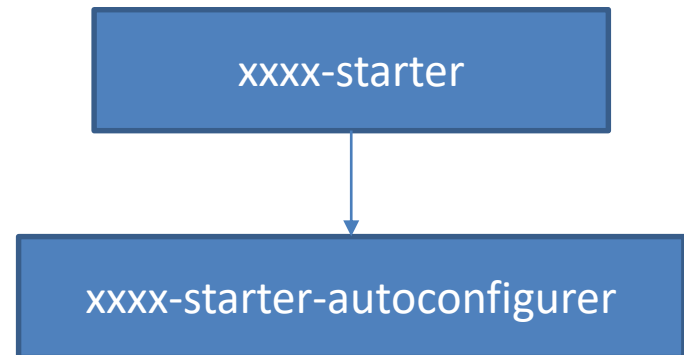
- 举例：spring-boot-starter-web、spring-boot-starter-actuator、spring-boot-starter-jdbc

- 自定义命名空间

- 后缀：“-spring-boot-starter”

- 模式：模块-spring-boot-starter

- 举例：mybatis-spring-boot-starter



九、Spring Boot与缓存

JSR-107、Spring缓存抽象、整合Redis

一、Spring缓存抽象

Spring从3.1开始定义了org.springframework.cache.**Cache**和org.springframework.cache.**CacheManager**接口来统一不同的缓存技术；并支持使用**JCache (JSR-107)**注解简化我们开发；

- Cache接口为缓存的组件规范定义，包含缓存的各种操作集合；
- Cache接口下Spring提供了各种xxxCache的实现；如RedisCache，EhCacheCache，ConcurrentMapCache等；
- 每次调用需要缓存功能的方法时，Spring会检查指定参数的指定的目标方法是否已经被调用过；如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法并缓存结果后返回给用户。下次调用直接从缓存中获取。
- 使用Spring缓存抽象时我们需要关注以下两点；
 - 1、确定方法需要被缓存以及他们的缓存策略
 - 2、从缓存中读取之前缓存存储的数据

二、几个重要概念&缓存注解

Cache	缓存接口，定义缓存操作。实现有：RedisCache、EhCacheCache、ConcurrentMapCache等
CacheManager	缓存管理器，管理各种缓存（Cache）组件
@Cacheable	主要针对方法配置，能够根据方法的请求参数对其结果进行缓存
@CacheEvict	清空缓存
@CachePut	保证方法被调用，又希望结果被缓存。
@EnableCaching	开启基于注解的缓存
keyGenerator	缓存数据时key生成策略
serialize	缓存数据时value序列化策略

@Cacheable/@CachePut/@CacheEvict 主要的参数

value	缓存的名称，在 spring 配置文件中定义，必须指定至少一个	例如： @Cacheable(value="mycache") 或者 @Cacheable(value={"cache1","cache2"})
key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如： @Cacheable(value="testcache",key="#userName")
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存/清除缓存	例如： @Cacheable(value="testcache",condition="#userName.length()>2")
allEntries (@CacheEvict)	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存	例如： @CacheEvict(value="testcache",allEntries=true)
beforeInvocation (@CacheEvict)	是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存，缺省情况下，如果方法执行抛出异常，则不会清空缓存	例如： @CacheEvict(value="testcache", beforeInvocation=true)

三、整合redis实现缓存

1. 引入spring-boot-starter-data-redis
2. application.yml配置redis连接地址
3. 配置缓存
 - @EnableCaching、
 - CachingConfigurerSupport、
4. 测试使用缓存
5. 切换为其他缓存&CompositeCacheManager

十、Spring Boot与消息

JMS、AMQP、RabbitMQ

一、概述

1. 在大多应用中，我们系统之间需要进行异步通信，即异步消息。

2. 异步消息中两个重要概念：

消息代理 (message broker) 和目的地 (destination)

当消息发送者发送消息以后，将由消息代理接管，消息代理保证消息传递到指定目的地。

3. 异步消息主要有两种形式的目的地

1. 队列 (queue)：点对点消息通信 (point-to-point)

2. 主题 (topic)：发布 (publish) / 订阅 (subscribe) 消息通信

4. 点对点式：

- 消息发送者发送消息，消息代理将其放入一个队列中，消息接收者从队列中获取消息内容，消息读取后被移出队列
- 消息只有唯一的发送者和接受者，但并不是说只能有一个接收者

5. 发布订阅式：

- 发送者（发布者）发送消息到主题，多个接收者（订阅者）监听（订阅）这个主题，那么就会在消息到达时同时收到消息

6. JMS (Java Message Service) java消息服务：

- 基于JVM消息代理的规范。ActiveMQ、HornetMQ是JMS实现

7. AMQP (Advanced Message Queuing Protocol)

- 高级消息队列协议，也是一个消息代理的规范，兼容JMS
- RabbitMQ是AMQP的实现

8. Spring支持

- **spring-jms提供了对JMS的支持**
- **spring-rabbit提供了对AMQP的支持**
- **需要ConnectionFactory的实现来连接消息代理**
- **提供JmsTemplate、RabbitTemplate来发送消息**
- **@JmsListener (JMS) 、 @RabbitListener (AMQP) 注解在方法上监听消息代理发布的消息**
- **@EnableJms、@EnableRabbit开启支持**

9. Spring Boot自动配置

- **JmsAutoConfiguration**
- **RabbitAutoConfiguration**

二、RabbitMQ简介

RabbitMQ简介：

RabbitMQ是一个由erlang开发的AMQP(Advanced Message Queue)的开源实现。

核心概念

Producer&Consumer

- producer指的是消息生产者，consumer消息的消费者。

Broker

- 它提供一种传输服务,它的角色就是维护一条从生产者到消费者的路线，保证数据能按照指定的方式进行传输，

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

- 消息交换机,它指定消息按什么规则,路由到哪个队列。
- Exchange有4种类型：direct(默认)，fanout, topic, 和headers，不同类型的Exchange转发消息的策略有所区别：

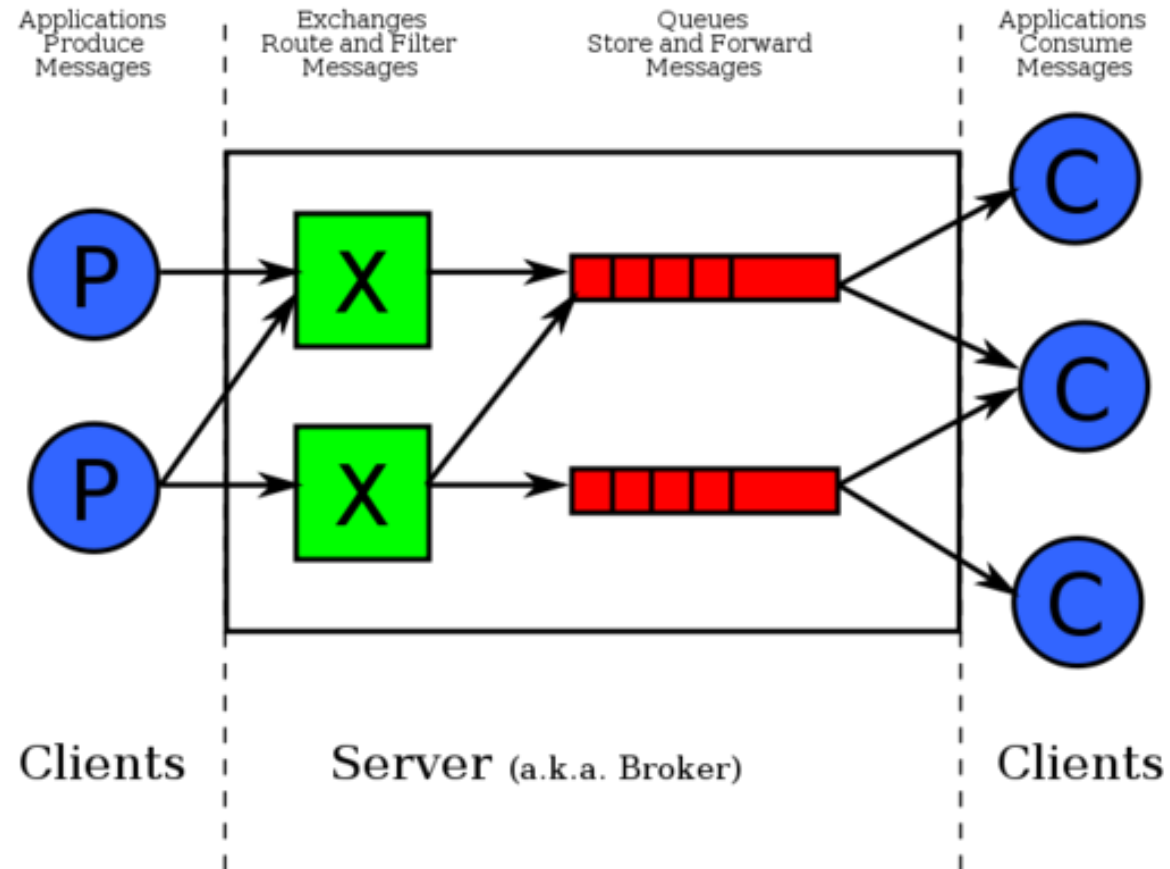
Binding

- 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 (vhosts)

- 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
- producer和consumer连接rabbit server需要指定一个vhost

三、RabbitMQ运行机制



四、RabbitMQ整合

1. 引入spring-boot-starter-amqp
2. application.yml配置
3. 测试RabbitMQ

十一、Spring Boot与检索

ElasticSearch

一、检索

我们的应用经常需要添加检索功能，更或者是大量日志检索分析等，Spring Boot 通过整合Spring Data Elasticsearch为我们提供了非常便捷的检索功能支持；

Elasticsearch是一个分布式搜索服务，提供Restful API，底层基于Lucene，采用多shard的方式保证数据安全，并且提供自动resharding的功能，github等大型的站点也是采用了Elasticsearch作为其搜索服务，

二、概念

- 以 **员工文档** 的形式存储为例：一个**文档**代表一个员工数据。存储数据到 Elasticsearch 的行为叫做 **索引**，但在索引一个文档之前，需要确定将文档存储在哪里。
- 一个 Elasticsearch 集群可以包含多个 **索引**，相应的每个索引可以包含多个 **类型**。这些不同的类型存储着多个 **文档**，每个文档又有多个 **属性**。
- 类似关系：
 - 索引-数据库
 - 类型-表
 - 文档-表中的记录
 - 属性-列

三、整合ElasticSearch测试

- 引入spring-boot-starter-data-elasticsearch
- 安装Spring Data 对应版本的ElasticSearch
- application.yml配置
- Spring Boot自动配置的
ElasticsearchRepository、Client
- 测试ElasticSearch

十二、Spring Boot与任务

异步任务、定时任务、邮件任务

一、异步任务

在Java应用中，绝大多数情况下都是通过同步的方式来实现交互处理的；但是在处理与第三方系统交互的时候，容易造成响应迟缓的情况，之前大部分都是使用多线程来完成此类任务，其实，在Spring 3.x之后，就已经内置了@Async来完美解决这个问题。

两个注解：

@EnableAysnc、@Aysnc

二、定时任务

项目开发中经常需要执行一些定时任务，比如需要在每天凌晨时候，分析一次前一天的日志信息。Spring为我们提供了异步执行任务调度的方式，提供 **TaskExecutor**、**TaskScheduler** 接口。

两个注解： @EnableScheduling、@Scheduled

cron表达式：

字段	允许值	允许的特殊字符	特殊字符	代表含义
秒	0-59	, - * /	,	枚举
分	0-59	, - * /	-	区间
小时	0-23	, - * /	*	任意
日期	1-31	, - * ? / L W C	/	步长
月份	1-12或JAN-DEC	, - * /	?	日/星期冲突匹配
星期	1-7或SUN-SAT	, - * ? / L C #	L	最后
年（可选）	空,1970-2099	, - * /	W	工作日
			C	和calendar联系后计算过的值
			#	星期，4#2，第2个星期三

三、邮件任务

- 邮件发送需要引入spring-boot-starter-mail
- Spring Boot 自动配置MailSenderAutoConfiguration
- 定义MailProperties内容，配置在application.yml中
- 自动装配JavaMailSender
- 测试邮件发送

十三、Spring Boot与安全

安全、Spring Security

一、安全

Spring Security是针对Spring项目的安全框架，也是Spring Boot底层安全模块默认的技术选型。他可以实现强大的web安全控制。对于安全控制，我们仅需引入**spring-boot-starter-security**模块，进行少量的配置，即可实现强大的安全管理。

几个类：

WebSecurityConfigurerAdapter：自定义Security策略

AuthenticationManagerBuilder：自定义认证策略

@EnableWebSecurity：开启WebSecurity模式

- 应用程序的两个主要区域是“认证”和“授权”（或者访问控制）。这两个主要区域是Spring Security 的两个目标。
- “认证”，是建立一个他声明的主体的过程（一个“主体”一般是指用户，设备或一些可以在你的应用程序中执行动作的其他系统）。
- “授权”指确定一个主体是否允许在你的应用程序执行一个动作的过程。为了抵达需要授权的店，主体的身份已经有认证过程建立。
- 这个概念是通用的而不只在Spring Security中。

二、Web&安全

1. CSRF (Cross-site request forgery) 跨站请求伪造
 - HttpSecurity启用csrf功能
2. 登陆/注销
 - HttpSecurity配置登陆、注销功能
3. remember me
 - 表单添加remember-me的checkbox
 - 配置启用remember-me功能
4. Thymeleaf提供的SpringSecurity标签支持
 - 需要引入thymeleaf-extras-springsecurity4
 - sec:authentication= "name" 获得当前用户的用户名
 - sec:authorize= "hasRole('ADMIN')" 当前用户必须拥有ADMIN权限时才会显示标签内容

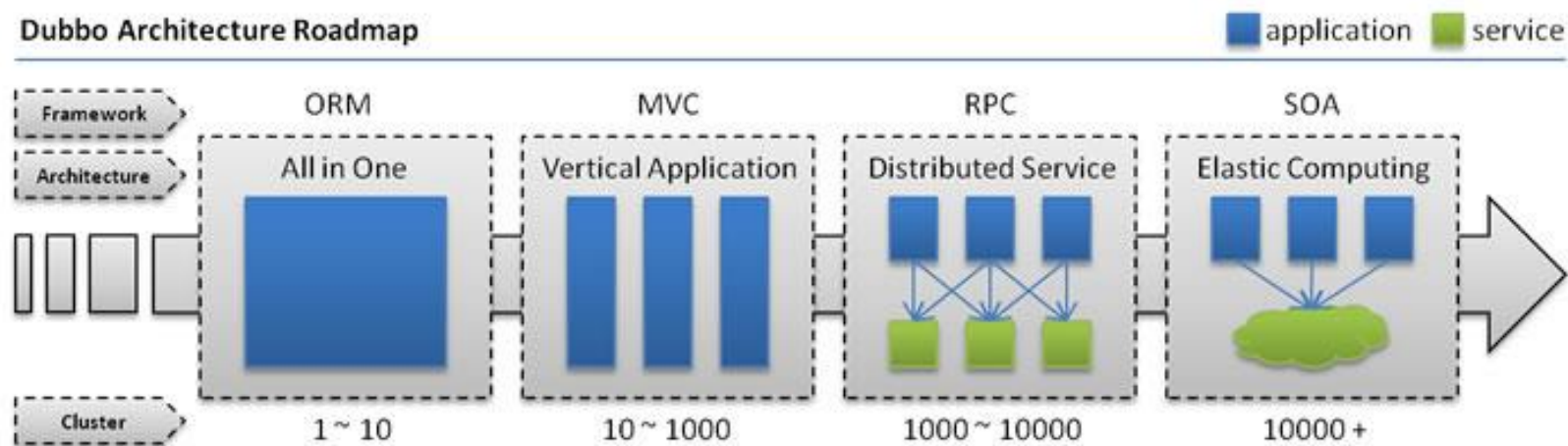
十四、Spring Boot与分布式

分步式、Spring Boot/Cloud、Dubbo/Zookeeper

一、分布式应用

在分布式系统中，国内常用zookeeper+dubbo组合，而Spring Boot推荐使用全栈的Spring，Spring Boot+Spring Cloud。

分布式系统：



- **单一应用架构**

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM)是关键。

- **垂直应用架构**

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。

- **分布式服务架构**

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。

- **流动计算架构**

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。

二、Zookeeper和Dubbo

- **ZooKeeper 服务注册中心**

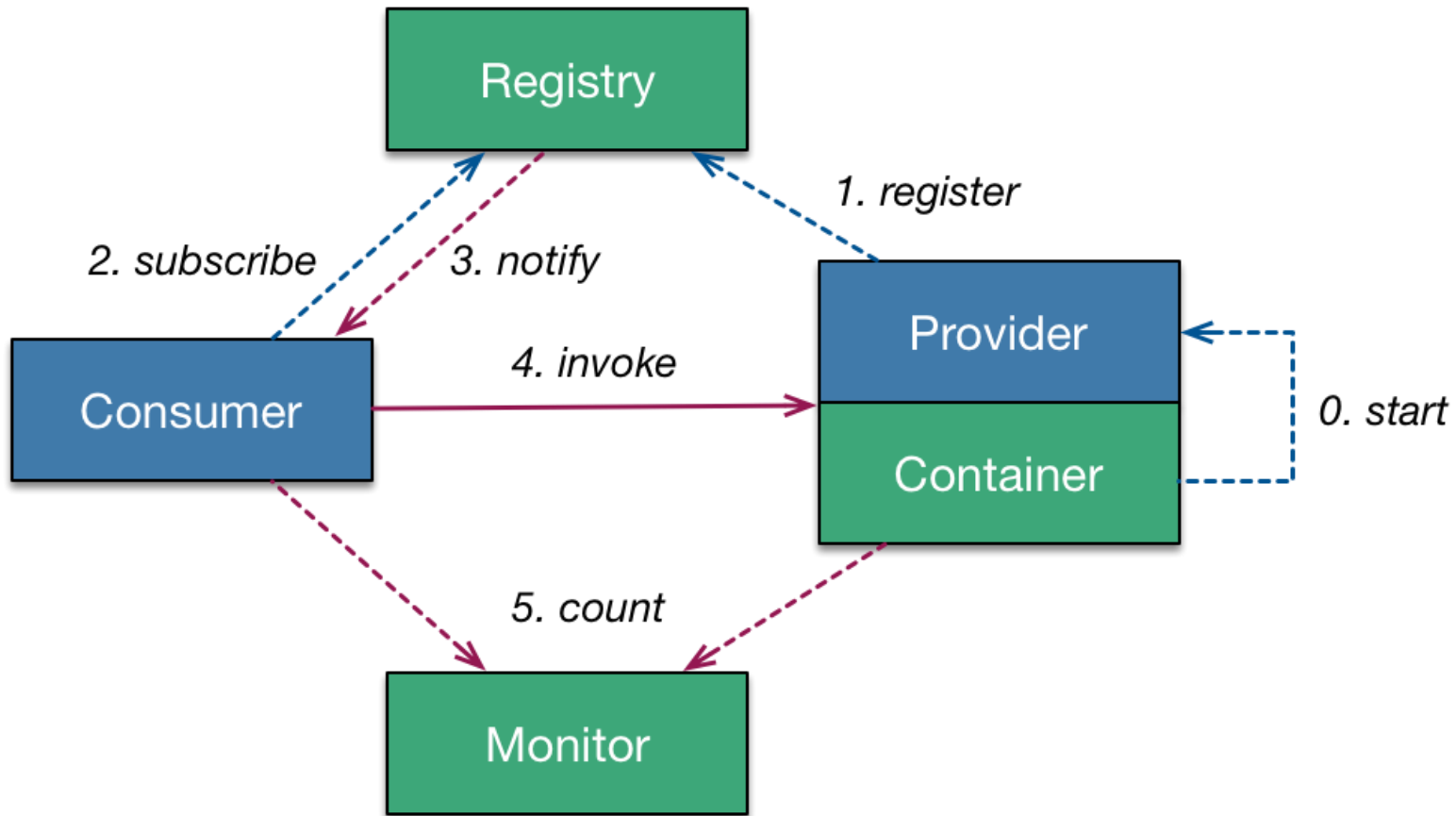
ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

- **Dubbo**

Dubbo是Alibaba开源的分布式服务框架，它最大的特点是按照分层的方式来架构，使用这种方式可以使各个层之间解耦合（或者最大限度地松耦合）。从服务模型的角度来看，Dubbo采用的是一种非常简单的模型，要么是提供方提供服务，要么是消费方消费服务，所以基于这一点可以抽象出服务提供方（Provider）和服务消费方（Consumer）两个角色。

Dubbo Architecture

-----> init -.-.-.-> async ———> sync



- 整合dubbo

- 引入spring-boot-starter-dubbo

- ```
<dependency>
```

- ```
  <groupId>com.gitee.reger</groupId>
```

- ```
 <artifactId>spring-boot-starter-dubbo</artifactId>
```

- ```
  <version>1.0.4</version>
```

- ```
</dependency>
```

- 配置服务提供者与消费者

- 测试

## 三、Spring Boot和Spring Cloud

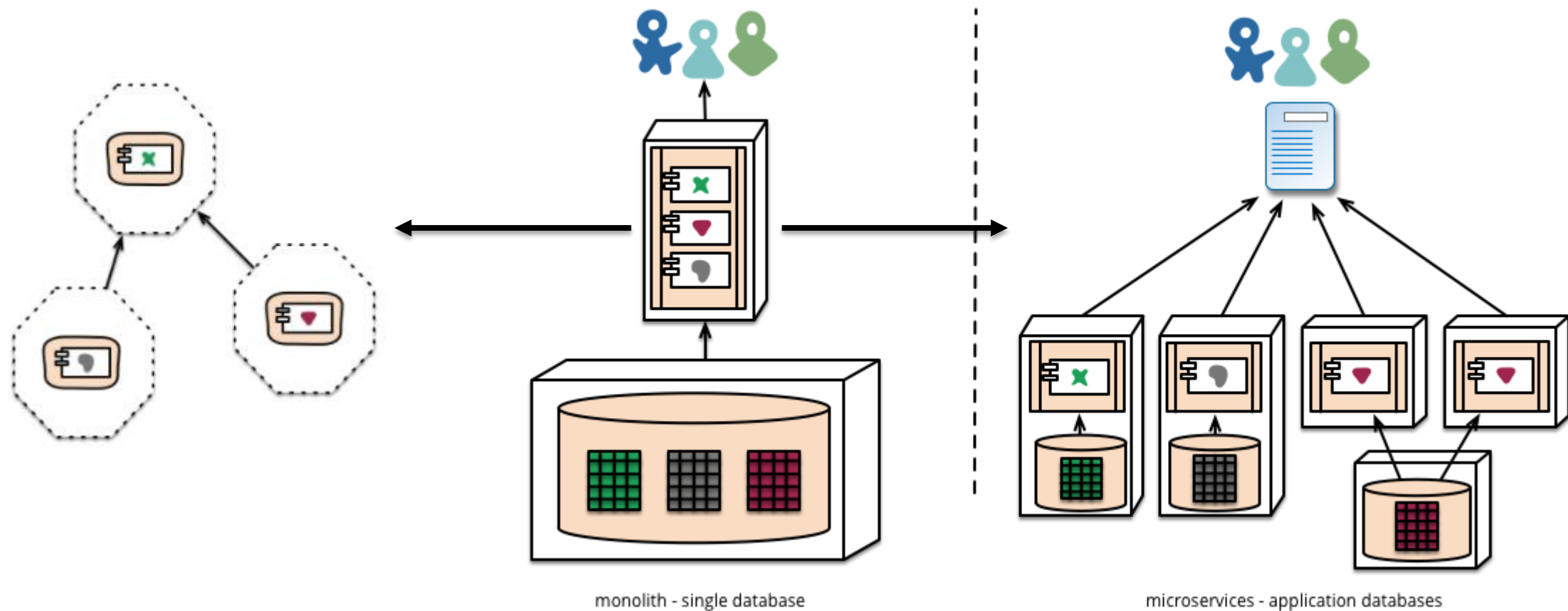
### Spring Cloud

Spring Cloud是一个分布式的整体解决方案。Spring Cloud 为开发者提供了**在分布式系统（配置管理，服务发现，熔断，路由，微代理，控制总线，一次性token，全局锁，leader选举，分布式session，集群状态）中快速构建的工具**，使用Spring Cloud的开发者可以快速的启动服务或构建应用、同时能够快速和云平台资源进行对接。

- **SpringCloud分布式开发五大常用组件**
  - 服务发现——Netflix Eureka
  - 客户端负载均衡——Netflix Ribbon
  - 断路器——Netflix Hystrix
  - 服务网关——Netflix Zuul
  - 分布式配置——Spring Cloud Config



# 微服务



- Spring Cloud 入门
  - 1、创建provider
  - 2、创建consumer
  - 3、引入Spring Cloud
  - 4、引入Eureka注册中心
  - 5、引入Ribbon进行客户端负载均衡
  - 6、引入Feign进行声明式HTTP远程调用

# 十五、Spring Boot与开发热部署

热部署



# 一、热部署

在开发中我们修改一个Java文件后想看到效果不得不重启应用，这导致大量时间花费，我们希望不重启应用的情况下，程序可以自动部署（热部署）。有以下四种情况，如何实现热部署。

- 1、模板引擎
  - 在Spring Boot中开发情况下禁用模板引擎的cache
  - 页面模板改变ctrl+F9可以重新编译当前页面并生效

## 2、Spring Loaded

Spring官方提供的热部署程序，实现修改类文件的热部署

- 下载Spring Loaded（项目地址<https://github.com/spring-projects/spring-loaded>）
- 添加运行时参数；  
-javaagent:C:/springloaded-1.2.5.RELEASE.jar -noverify

## 3、JRebel

- 收费的一个热部署软件
- 安装插件使用即可

## 4、Spring Boot Devtools ( 推荐 )

### – 引入依赖

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

### – IDEA必须做一些小调整

*IntelliJ IDEA和Eclipse不同，Eclipse设置了自动编译之后，修改类它会自动编译，而IDEA在非RUN或DEBUG情况下才会自动编译（前提是你已经设置了Auto-Compile）。*

- 设置自动编译（settings-compiler-make project automatically）
- ctrl+shift+alt+/（maintenance）
- 勾选compiler.automake.allow.when.app.running



# 十六、Spring Boot与监控管理

热部署

## 一、监控管理

通过引入spring-boot-starter-actuator，可以使用Spring Boot为我们提供的准生产环境下的应用监控和管理功能。我们可以通过HTTP，JMX，SSH协议来进行操作，自动得到审计、健康及指标信息等

- 步骤：
  - 引入spring-boot-starter-actuator
  - 通过http方式访问监控端点
  - 可进行shutdown（POST 提交，此端点默认关闭）

- 监控和管理端点

端点名	描述
actuator	所有Endpoint端点，需加入spring HATEOAS支持
autoconfig	所有自动配置信息
beans	所有Bean的信息
configprops	所有配置属性
dump	线程状态信息
env	当前环境信息
health	应用健康状况
info	当前应用信息
metrics	应用的各项指标
mappings	应用@RequestMapping映射路径
shutdown	关闭当前应用（默认关闭）
trace	追踪信息（最新的http请求）



## 二、定制端点信息

- 定制端点一般通过endpoints+端点名+属性名来设置。
- 修改端点id ( endpoints.beans.id=mybeans )
- 开启远程应用关闭功能 ( endpoints.shutdown.enabled=true )
- 关闭端点 ( endpoints.beans.enabled=false )
- 开启所需端点
  - endpoints.enabled=false
  - endpoints.beans.enabled=true
- 定制端点访问路径
  - management.context-path=/manage
- 关闭http端点
  - management.port=-1

