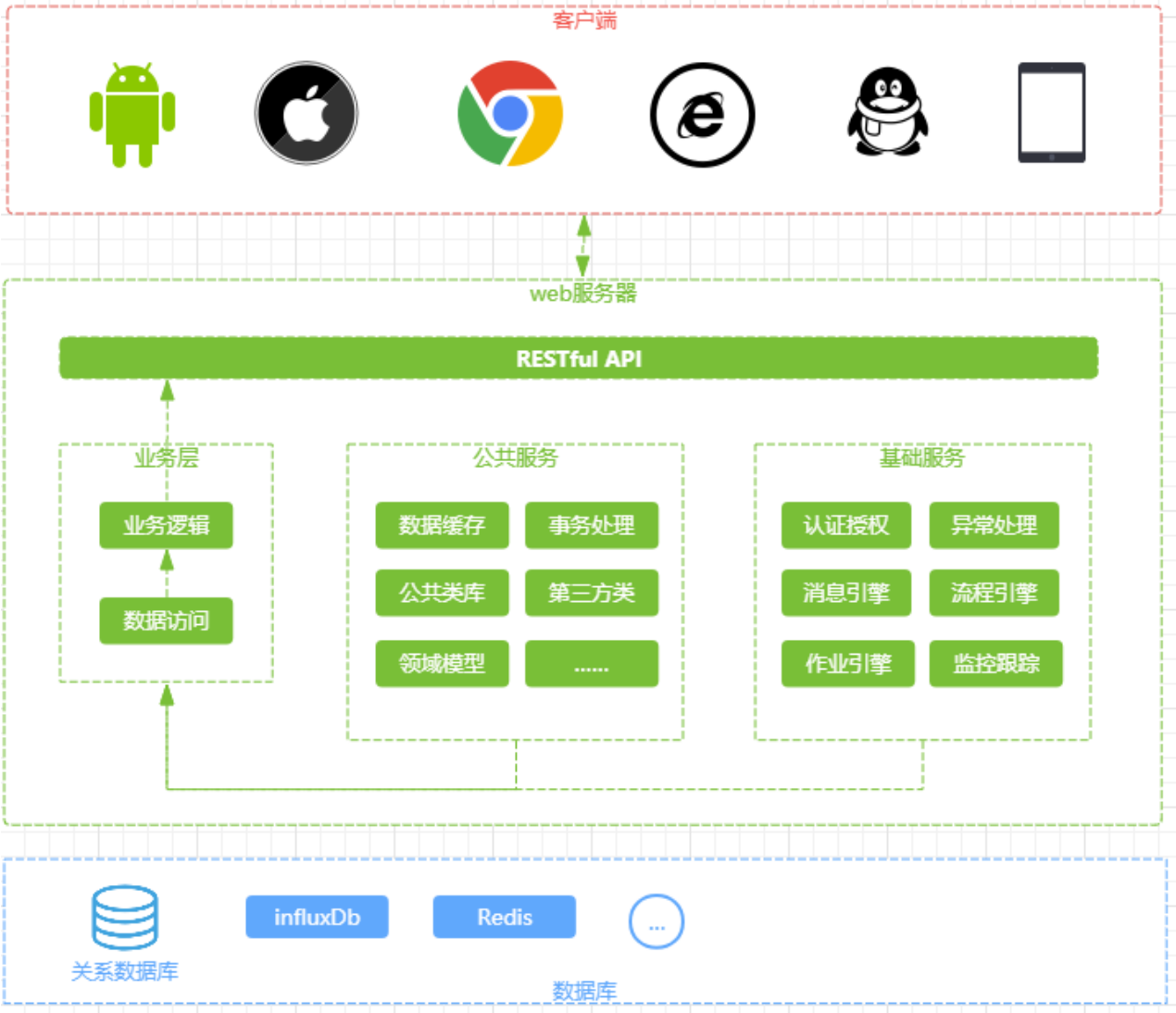
springboot因何而来

一、单体架构



单体架构的优点：

- 快速开发和验证想法，证明产品思路是否可行
- 投入资源和成本，包括人力和物力相对比较节约

单体架构的缺点：

随着业务的复杂度增加，单体的灵活度会逐渐下降，比如：

- IDE过载：随着代码量增加，代码整体编译效率下降。
- 规模化：无法满足团队规模化高效开发。
- 系统开发、测试、部署的冲突和效率低下等问题
- 只能关注一套技术栈

- 应用扩展性比较差
- 海量用户高并发访问数量有限

单体适用场景：

架构设计的三大原则告诉我们，架构需要的是**简单、适度、演化。**

微服务架构图

微服务架构图谱谷歌或Bing下，可以看到各种各样的架构图，源于业务和架构师自身的喜好或者粗细粒度，但是每个架构图的基本组件和架构分层都差别不大，只是有的细一些，有的粗一下。比如有客户端层，容器层(K8S)，API Gateway，微服务集群层，EventBus层是必须要有的，至于服务监控和服务跟踪、服务治理本身就是一个完整的系统，粒度就没有细了。基于这些概念，我把架构图稍微细化一下，这里省去服务监控跟踪和治理的部分，后续单独抽离出来分析。

网关

> 服务注册 服务发现 认证授权 熔断 限流 SSO 。。。

微服务集群

> 认证服务 OAuth2.0 用户服务 订单服务 消息服务等

微服务组件

> MQ Redis 系统监控

以上架构图主要分4层，每个层次遵循**架构分层的核心思想**：关注点分离，职责各异，边界清晰。

每个服务需要快速构建 那么就出现了微框架

微框架 到脚手架

SB快速构建微服务的应用

springboot和spring的关系

jar包之间的IOC增强

DI :依赖注入

DL:依赖查找 即：getbean

服务于框架的框架

SpringBoot框架体系结构分析

简介

SpringBoot是由Pivotal团队在2013年开始研发、2014年4月发布第一个版本的全新开源的轻量级框架。它基于Spring4.0设计，不仅继承了Spring框架原有的优秀特性，而且还通过简化配置来进一步简化了Spring应用的整个搭建和开发过程。另外SpringBoot通过集成大量的框架使得依赖包的版本冲突，以及引用的不稳定性等问题得到了很好的解决。

源码地址：https://github.com/spring-projects/spring-boot

官网参考文档：https://docs.spring.io/spring-boot/docs/current/reference/html/

约定配置属性列表：https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#common-application-properties

自动装配列表：https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-auto-configuration-classes.html#auto-configuration-classes

源码项目结构

spring-boot-project 项目

spring-boot 模块

`spring-boot` 模块，Spring Boot 的核心实现，大概在 4W 代码左右。提供了如下功能：

- 在 `org.springframework.boot.SpringApplication` 类，提供了大量的静态方法，可以很容易运行一个独立的 Spring 应用程序。

  > 是不是超级熟悉。

- 带有可选容器的嵌入式 Web 应用程序（Tomcat、Jetty、Undertow） 的支持。

  > 在 `org.springframework.boot.web` 包下实现。

- 边界的外部配置支持。

spring-boot-autoconfigure 模块

`spring-boot-actuator-autoconfigure` 模块，大概 4W代码左右。`spring-boot-autoconfigure` 可以根据类路径的内容，自动配置大部分常用应用程序。通过使用 `org.springframework.boot.autoconfigure.@EnableAutoConfiguration` 注解，会触发 Spring 上下文的自动配置。

> 这里的大部分，指的是常用的框架。例如说，Spring MVC、Quartz 等等。也就是说，如果 `spring-boot-actuator-autoconfigure` 模块，暂未提供的框架，需要我们自己去实现对应框架的自动装配。

这个模块的代码，必须要看，没得商量。

所以到此处为止，我们已经看到对我们来研究 Spring Boot 最最最中航要的两个模块：`spring-boot` 和 `spring-boot-autoconfigure`，一共是 9W 行代码左右。

spring-boot-actuator 模块

`spring-boot-actuator` 模块，大概 2W 行代码左右。正如其模块的英文 actuator ，它完全是一个用于暴露应用自身信息的模块：

- 提供了一个监控和管理生产环境的模块，可以使用 http、jmx、ssh、telnet 等管理和监控应用。
- 审计（Auditing）、 健康（health）、数据采集（metrics gathering）会自动加入到应用里面。

> 一般情况下，我们可以不看这块代码的代码。

spring-boot-actuator-autoconfigure 模块

`spring-boot-actuator-autoconfigure` 模块，大概 1W7 行代码左右。它提供了 `spring-boot-actuator` 的自动配置功能。

> 一般情况下，我们可以不看这块代码的代码。

spring-boot-starters 模块

`spring-boot-starters` 模块，它不存在任何的代码，而是提供我们常用框架的 Starter 模块。例如：

- `spring-boot-starter-web` 模块，提供了对 Spring MVC 的 Starter 模块。
- `spring-boot-starter-data-jpa` 模块，提供了对 Spring Data JPA 的 Starter 模块。

而每个 Starter 模块，里面只存在一个 `pom` 文件，这是为什么呢？简单来说，Spring Boot 可以根据项目中是否存在指定类，并且是否未生成对应的 Bean 对象，那么就自动创建 Bean 对象。因为有这样的机制，我们只需要使用 `pom` 文件，配置需要引入的框架，就可以实现该框架的使用所需要的类的自动装配。

spring-boot-cli 模块

`spring-boot-cli` 模块，大概 1W 行代码左右。它提供了 Spring 项目相关的命令行功能。它是 Spring Boot 的命令行界面。

- 它可以用来快速启动 Spring 。
- 它可以运行 Groovy 脚本，开发人员不需要编写很多样板代码，只需要关注业务逻辑。
- Spring Boot CLI 是创建基于Spring的应用程序的最快方法。

spring-boot-test 模块

`spring-boot-test` 模块，大概 1W 行代码左右。Spring Boot 提供测试方面的支持，例如说：

- SpringBootTestRandomPortEnvironmentPostProcessor 类，提供随机端口。
- `org.springframework.boot.test.mock.mockito` 包，提供 Mockito 的增强。

spring-boot-test-autoconfigure 模块

`spring-boot-test-autoconfigure` 模块，大概 1W 行代码不到。它提供了 `spring-boot-test` 的自动配置功能。

spring-boot-devtools 模块

`spring-boot-devtools` 模块，大概 8000 行代码左右。通过它，来使 Spring Boot 应用支持热部署，提高开发者的开发效率，无需手动重启 Spring Boot 应用。

spring-boot-tools 模块

`spring-boot-tools` 模块，大概 3W 行代码左右。它是 Spring Boot 提供的工具箱，所以在其内有多个子 Maven 项目。

注意哟，我们这里说的工具箱，并不是我们在 Java 里的工具类。困惑？我们来举个例子：`spring-boot-maven-plugin` 模块：提供 Maven 打包 Spring Boot 项目的插件。

关于 `spring-boot-tools` 模块的其它子模块，我们就暂时不多做介绍落。

3.11 其它

`spring-boot-project` 项目的其它子模块如下：

- `spring-boot-properties-migrator` 模块：500 行代码左右，帮助开发者从 Spring Boot 1 迁移到 Spring Boot 2 。
- `spring-boot-dependencies` 模块：无代码，只有所有依赖和插件的版本号信息。
- `spring-boot-parent` 模块：无代码，该模块是其他项目的 parent，该模块的父模块是 `spring-boot-dependencies` 。
- `spring-boot-docs` 模块：1000 行代码左右，貌似是提供 Spring Boot 文档里的一些示例。不太确定，也并不重要。

特性

SpringBoot所具备的特征有：
（1）可以创建独立的Spring应用程序，并且基于其Maven或Gradle插件，可以创建可执行的JARs和WARs；
（2）内嵌Tomcat或Jetty等Servlet容器；
（3）提供自动配置的“starter”项目对象模型（POMS）以简化Maven配置；
（4）尽可能自动配置Spring容器；
（5）提供准备好的特性，如指标、健康检查和外部化配置；
（6）绝对没有代码生成，不需要XML配置。

重要策略

> 开箱即用和约定优于配置。

开箱即用，Outofbox，是指在开发过程中，通过在MAVEN项目的pom文件中添加相关依赖包，然后使用对应注解来代替繁琐的XML配置文件以管理对象的生命周期。这个特点使得开发人员摆脱了复杂的配置工作以及依赖的管理工作，更加专注于业务逻辑。

约定优于配置，Convention over configuration，是一种由SpringBoot本身来配置目标结构，由开发者在结构中添加信息的软件设计范式。这一特点虽降低了部分灵活性，增加了BUG定位的复杂性，但减少了开发人员需要做出决定的数量，同时减少了大量的XML配置，并且可以将代码编译、测试和打包等工作自动化。

SpringBoot的关键组件架构设计

使用Spring Initializr创建一个项目，名称为hello

```
<groupId>com.naixue.springboot</groupId>
<artifactId>hello</artifactId>
```

修改版本为<java.version>8</java.version>

@Configuration的使用

新建HelloService类

```
public class HelloService {
    public String say() {
        String msg = "hello friend!";
        System.out.println(msg);
        return msg;
    }
}
```

新建HelloConfiguration

```
@Configuration
public class HelloConfiguration {
    @Bean
    public HelloService helloService(){
        return new HelloService();
    }
}
```

修改HelloApplication

```
@SpringBootApplication
public class HelloApplication {

  private static HelloService helloService;

  @Autowired
  public void setHelloService(HelloService helloService) {
    HelloApplication.helloService = helloService;
  }
```

```
10
11    public static void main(String[] args) {
12      SpringApplication.run(HelloApplication.class, args);
13          helloService.say();
14    }
15  }
```

运行项目

运行后成功打印出hello friend!

@EnableAutoConfiguration

1. 在HelloApplication上新增EnableAutoConfiguration注解
2. 在resource下新建META-INF文件夹
3. 在META-INF文件夹下创建spring.factories文件

```
1  org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.naix
   ue.springboot.hello.HelloConfiguration
```

运行项目

运行后成功打印出hello friend!

@Bean的使用

```
1  public class HelloConfiguration {
2      @Bean
3      public HelloService helloService(){
4          return new HelloService();
5      }
6  }
```

@ConditionalOnClass的使用

name中配置的class存在即往IOC容器中注入

对应的还有ConditionalOnMissClass 不存的时候注入

1. 创建一个friend项目

新建Friend类

```
1  public class Friend {
2      private String name;
3      private int age;
4
5      public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

2. hello项目添加friend的依赖

```
<dependency>
          <groupId>com.naixue.springboot</groupId>
          <artifactId>friend</artifactId>
          <version>0.0.1-SNAPSHOT</version>
</dependency>
```

修改版本为<java.version>8</java.version>

3. 添加@ConditionalOnClass注解

```
    @Bean
    @ConditionalOnClass(name = "com.naixue.springboot.friend.Friend")
    public HelloService helloService(){
        return new HelloService();
    }
```

运行项目

运行后成功打印出hello friend!

@ConditionalOnClass的通过装配的使用

1. 去除 @ConditionalOnClass(name = "com.naixue.springboot.friend.Friend")
2. META-INF下新增spring-autoconfigure-metadata.properties

```
com.naixue.springboot.hello.HelloConfiguration.ConditionalOnClass=com.naixue.springboot.friend.Friend
```

运行项目

运行后成功打印出hello friend!

如果修改Friend类名那么HelloService不会注入到IOC容器中

@ComponentScan注解讲解

会扫描@Component @Responsitory @Service @Controller

等同于xml中context:coponent-scan使得带有这些注解的类被IOC容器托管

新建example项目

创建

```
@Component
public class ExampleComponent {

}

@Repository
public class ExampleRepository {

}

@Service
public class ExampleService {

}

@Configuration
public class ExampleConfiguration {
}

@Controller
public class ExampleController {

}
```

新建一个自定义注解 NXRepository

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repository
public @interface NXRepository {
    String value() default "";
}
```

创建ExampleNXRepository

```
@NXRepository("nxRepository")
public class ExampleNXRepository {
}
```

讲解ComponentScan

```
@ComponentScan("com.naixue.springboot.example")
//@ComponentScan("com.naixue.springboot.example.service,com.naixue.springboot.example.controller")
//@ComponentScan("com.naixue.springboot.example.*")
//@SpringBootApplication(scanBasePackages = "com.naixue.springboot.example")
public class ExampleApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext configurableApplicationContext = SpringApplication.run(ExampleApplication.class, args);
        String[] beanDefinitionNames = configurableApplicationContext.getBeanDefinitionNames();
        for (String beanDefinitionName : beanDefinitionNames) {
            System.out.println(beanDefinitionName);
        }
    }
}
```

@Import注解

ImportAutoConfigurationImportSelector

AutoConfigurationImportSelector

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication {
}
```

```
18
19  @Target({ElementType.TYPE})
20  @Retention(RetentionPolicy.RUNTIME)
21  @Documented
22  @Inherited
23  @AutoConfigurationPackage
24  @Import({AutoConfigurationImportSelector.class})
25  public @interface EnableAutoConfiguration {
26      String ENABLED_OVERRIDE_PROPERTY =
    "spring.boot.enableautoconfiguration";
27
28      Class<?>[] exclude() default {};
29
30      String[] excludeName() default {};
31  }
```

@SpringBootApplication --> @EnableAutoConfiguration --> @Import --> AutoConfigurationImportSelector

@SpringBootApplication --> @EnableAutoConfiguration --> @AutoConfigurationPackage --> @Import({Registrar.class})
org.springframework.boot.autoconfigure.AutoConfigurationPackages.Registrar

> 根据上下文做动态加载bean，或者批量加载bean

SpringBoot的自动加载机制与原理

手写自定义自动装载注解

新建nx-auto-configuration项目

创建两个bean对象

```
1  public class AccountService {
2  }
3
4  public class UserService {
5  }
```

创建NXDefinitionRegistrar

```
public class NXDefinitionRegistrar implements
ImportBeanDefinitionRegistrar {

    @Override
    public void registerBeanDefinitions(AnnotationMetadata
annotationMetadata, BeanDefinitionRegistry beanDefinitionRegistry) {
        Class beanClass=AccountService.class;
        RootBeanDefinition beanDefinition=new
RootBeanDefinition(beanClass);
        String
beanName=StringUtils.uncapitalize(beanClass.getSimpleName());

 beanDefinitionRegistry.registerBeanDefinition(beanName,beanDefinition);
    }
}
```

创建NXImportSelector

```
public class NXImportSelector implements ImportSelector {

    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        Map<String,Object> attributes=

annotationMetadata.getAnnotationAttributes(EnableNXAutoConfiguration.class
.getName());
        //动态注入bean :判断逻辑实现动态配置

        //返回的是一个固定的UserService
        return new String[]{UserService.class.getName()};
    }
}
```

自定义注解EnableNXAutoConfiguration

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({NXImportSelector.class,NXDefinitionRegistrar.class}) //
public @interface EnableNXAutoConfiguration {

    //配置一些方法
    Class<?>[] exclude() default {};
}
```

运行验证

```java
@SpringBootApplication
@EnableNXAutoConfiguration
public class NxAutoConfigurationApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext
ca=SpringApplication.run(NxAutoConfigurationApplication.class,args);
        System.out.println(ca.getBean(UserService.class));
        System.out.println(ca.getBean(AccountService.class));
    }

}
```

源码分析自动装配

在 Spring Boot 场景下，基于约定大于配置的原则，实现 Spring 组件自动装配的目的。其中使用了

底层装配技术

Spring 模式注解装配
Spring @Enable 模块装配
Spring 条件装配装配
Spring 工厂加载机制

- 实现类: SpringFactoriesLoader
- 配置资源: META-INF/spring.factories

自动装配举例

参考 META-INF/spring.factories

实现方法

1. 激活自动装配 - @EnableAutoConfiguration
2. 实现自动装配 - XXXAutoConfiguration
3. 配置自动装配实现 - META-INF/spring.factories

自定义自动装配

NXAutoConfiguration
条件判断: user.name == "nx"
模式注解: @Configuration
@Enable 模块: @EnableNX -> NXImportSelector -> NXConfiguration - > NX

自动装载配置剖析

1. 打开注解SpringBootApplication，上面有ComponentScan注解用来扫描bean和排除bean

2. 看看上面的EnableAutoConfiguration注解，上面引用了@Import(AutoConfigurationImportSelector.class)
3. import引出`动态注入`

   ImportSelector

   ImportBeanDefinitionRegistrar

   通过CacheImportSelector演示一下ImportSelector
4. 演示CacheImportSelector实现自动装载配置
5. EnableAutoConfiguration注解的上面有个@AutoConfigurationPackage注解点进去，@Import(AutoConfigurationPackages.Registrar.class)
6. 我们再去看看AutoConfigurationImportSelector.class

   先看看类关系

   AutoConfigurationImportSelector implements DeferredImportSelector implements extends ImportSelector#selectImports()

   那么说明我们需要从selectImports方法入手去看

```
  @Override
  public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
      return NO_IMPORTS;
    }
    //自动装配入口 重点看getAutoConfigurationEntry方法
    AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
  }

  protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
      return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    //重点 getCandidateConfigurations
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    //移除重复
    configurations = removeDuplicates(configurations);
    //获取排除的集合
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    //检查
```

```
    checkExcludedClasses(configurations, exclusions);
    //移除需要排除的
    configurations.removeAll(exclusions);
    //过滤
    configurations = getConfigurationClassFilter().filter(configurations);
    //激活bean中的事件监听
    fireAutoConfigurationImportEvents(configurations, exclusions);
    //返回自动配置的数据对象
    return new AutoConfigurationEntry(configurations, exclusions);
  }

  protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    //SpringFactoriesLoader SPI 机制
    //去加载spring.factories文件
    //打开spring-boot-autoconfigure-2.4.0-sources.jar!/META-INF/spring.factories  看下EnableAutoConfiguration
    //去演示NXImportSelector 实际EnableAutoConfiguration的values就是需要自动装配加载的bean的类路径
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
        getBeanClassLoader());
    Assert.notEmpty(configurations, "No auto configuration classes found in META-INF/spring.factories. If you "
        + "are using a custom packaging, make sure that file is correct.");
    return configurations;
  }
```

7. 重点看看spring-boot-autoconfigure-2.1.6.RELEASE.jar!/META-INF/spring-autoconfigure-metadata.properties

> 搜索 Configuration、ConditionalOnClass、ConditionalOnBean发现bean加载过程的依赖条件

8. AutoConfigurationImportSelector中处理自动注入原数据的逻辑

```
    private AutoConfigurationMetadata getAutoConfigurationMetadata() {
      if (this.autoConfigurationMetadata == null) {
        this.autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
      }
      return this.autoConfigurationMetadata;
    }
```

跟踪源码 一直可以到 AbstractApplicationContext#refresh().invokeBeanFactoryPostProcessors(beanFactory);

SpringBoot SPI

SPI的全称是Service Provider Interface, 直译过来就是"服务提供接口"

涉及到的知识点

- SPI机制
- FactoryBean
- JDK动态代理

具体实现看META-INF下的文件

打开spring-boot-autoconfigure-2.4.0-sources.jar!/META-INF/spring.factories 看下EnableAutoConfiguration

key=values[]
key=value[]
key=values[]

spi的扩展
满足目录结构一致
文件名一致
key要存在并且符合当前的加载

SpringBoot手写starter

1. 创建项目nx-uid-starter

```
<groupId>com.naixue.springboot</groupId>
<artifactId>nx-uid-starter</artifactId>
```

2. 我们经常使用到生成主键ID，那么我们今天的做一个开箱即用的starter，基于推特的雪花算法。
3. 首先引入需要的jar

```
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>4.5.15</version>
    <optional>true</optional><!--表示可选-->
</dependency>

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

4. 演示下IdUtil.createSnowflake(workerId,datacenterId).nextId();
5. 我们开始定义参数配置类

```java
package com.naixue.springboot.nxuidstarter.autoconfiguration;


import
org.springframework.boot.context.properties.ConfigurationProperties;

import java.util.Map;

@ConfigurationProperties(prefix=UIDProperties.NX_UID_PREFIX)
public class UIDProperties {

    public static final String NX_UID_PREFIX ="naixue.springboot.uid";
    private Map<String,Long> info;

    public Map<String, Long> getInfo() {
        return info;
    }

    public void setInfo(Map<String, Long> info) {
        this.info = info;
    }
}
```

6. 定义处理器接口

```java
package com.naixue.springboot.nxuidstarter.processor;

import
com.naixue.springboot.nxuidstarter.autoconfiguration.UIDProperties;

public interface UIDProcessor {

    //定义一个格式化的方法
    String getUID(UIDProperties properties);
}
```

7. 定义UUIDProcessor

```java
package com.naixue.springboot.nxuidstarter.processor;


import
com.naixue.springboot.nxuidstarter.autoconfiguration.UIDProperties;

```

```
import java.util.UUID;

public class UUIDProcessor implements UIDProcessor{

    @Override
    public String getUID(UIDProperties properties) {
        UUID uuid = UUID.randomUUID();
        System.out.println("uuid="+uuid);
        return uuid.toString();
    }
}
```

8. 定义SnowflakeProcessor

```
package com.naixue.springboot.nxuidstarter.processor;

import cn.hutool.core.util.IdUtil;
import com.naixue.springboot.nxuidstarter.autoconfiguration.UIDProperties;

import java.util.Map;

public class SnowflakeProcessor implements UIDProcessor {

    @Override
    public String getUID(UIDProperties properties) {
        Map<String, Long> info = properties.getInfo();
        long id = IdUtil.createSnowflake((long) info.get("workerId"), (long) info.get("datacenterId")).nextId();
        System.out.println("Snowflake uid="+id);
        return id + "";
    }
}
```

9. 定义bean

```
package com.naixue.springboot.nxuidstarter.autoconfiguration;

import com.naixue.springboot.nxuidstarter.processor.SnowflakeProcessor;
import com.naixue.springboot.nxuidstarter.processor.UIDProcessor;
import com.naixue.springboot.nxuidstarter.processor.UUIDProcessor;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingClass;
import org.springframework.context.annotation.Bean;
```

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration
public class UIDConfiguration {

    @ConditionalOnMissingClass("cn.hutool.core.util.IdUtil")
    @Bean
    @Primary
    public UIDProcessor getUUID() {
        return new UUIDProcessor();
    }

    @ConditionalOnClass(name = "cn.hutool.core.util.IdUtil")
    @Bean
    public UIDProcessor getSnowflakeUUID() {
        return new SnowflakeProcessor();
    }
}
```

10. 定义自动装载类

```java
package com.naixue.springboot.nxuidstarter.autoconfiguration;

import com.naixue.springboot.nxuidstarter.NXUIDTemplate;
import com.naixue.springboot.nxuidstarter.processor.UIDProcessor;
import
org.springframework.boot.context.properties.EnableConfigurationPropert
ies;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Import(UIDConfiguration.class)
@EnableConfigurationProperties(UIDProperties.class)
@Configuration
public class UidAutoConfiguration {

    @Bean
    public NXUIDTemplate NXUIDTemplate(UIDProcessor uidProcessor,
UIDProperties uidProperties) {
        return new NXUIDTemplate(uidProcessor,uidProperties);
    }
}
```

11. 定义SPI文件

    META-INF/spring.factories

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

com.naixue.springboot.nxuidstarter.autoconfiguration.UidAutoConfiguration
```

12. 定义给使用调用的service

```
package com.naixue.springboot.nxuidstarter;

import com.naixue.springboot.nxuidstarter.autoconfiguration.UIDProperties;
import com.naixue.springboot.nxuidstarter.processor.UIDProcessor;

public class NXUIDTemplate {

    private UIDProcessor uidProcessor;

    private UIDProperties uidProperties;

    public NXUIDTemplate(UIDProcessor uidProcessor, UIDProperties uidProperties) {
        this.uidProcessor = uidProcessor;
        this.uidProperties = uidProperties;
    }

    public String getUID(){
        return uidProcessor.getUID(uidProperties);

    }
}
```

13. 使用方添加依赖

```
<dependency>
          <groupId>com.naixue.springboot</groupId>
          <artifactId>nx-uid-starter</artifactId>
          <version>0.0.1-SNAPSHOT</version>
</dependency>

      <!--<dependency>-->
          <!--<groupId>cn.hutool</groupId>-->
          <!--<artifactId>hutool-all</artifactId>-->
          <!--<version>4.5.15</version>-->
      <!--</dependency>-->

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

```
16        </dependency>
```

14. 使用

```
@SpringBootApplication
public class HelloApplication {
  private static NXUIDTemplate nxUIDTemplate;

  @Autowired
  public void setNxUIDTemplate(NXUIDTemplate nxUIDTemplate) {
    HelloApplication.nxUIDTemplate = nxUIDTemplate;
  }

  public static void main(String[] args) {
    SpringApplication.run(HelloApplication.class, args);
    nxUIDTemplate.getUID();
  }
}
```

SpringBootApplication运行原理剖析

剖析SpringApplication

SpringApplication 基本使用

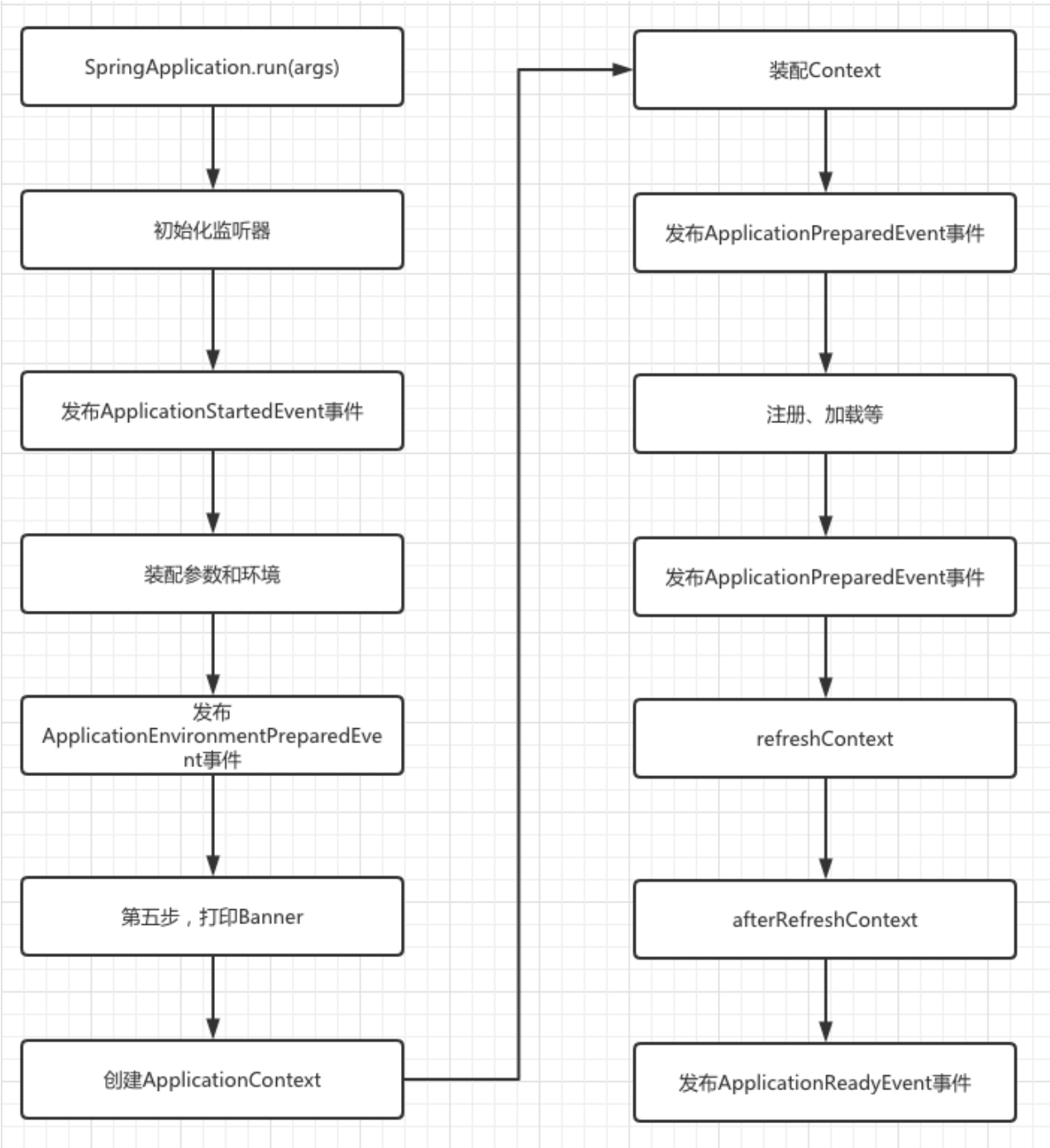
```
SpringApplication.run(DiveInSpringBootApplication.class, args)
```

通过 SpringApplication API 调整

```
SpringApplication springApplication = new
SpringApplication(DiveInSpringBootApplication.class);
springApplication.setBannerMode(Banner.Mode.CONSOLE);
springApplication.setWebApplicationType(WebApplicationType.NONE);
springApplication.setAdditionalProfiles("prod");
springApplication.setHeadless(true);
```

通过 SpringApplicationBuilder API 调整

```
new SpringApplicationBuilder(DiveInSpringBootApplication.class)
    .bannerMode(Banner.Mode.CONSOLE)
    .web(WebApplicationType.NONE)
    .profiles("prod")
    .headless(true)
    .run(args);
```



源码解读

```
public class SpringApplication {
  //setp1: 入口
    public static ConfigurableApplicationContext run(Class<?>[]
primarySources, String[] args) {
```

```
4       return new SpringApplication(primarySources).run(args);
5     }
6
7     //setp2:构造方法
8       public SpringApplication(ResourceLoader resourceLoader, Class<?>...
primarySources) {
9       this.resourceLoader = resourceLoader;
10      Assert.notNull(primarySources, "PrimarySources must not be null");
11      this.primarySources = new LinkedHashSet<>
(Arrays.asList(primarySources));
12      this.webApplicationType = WebApplicationType.deduceFromClasspath();
13      // 初始化 initializers 属性
14      // goto 3
15      setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
16      // 初始化 listeners 属性
17      setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
18      this.mainApplicationClass = deduceMainApplicationClass();
19    }
20
21      //setp3: 获取Spring工厂实例
22    private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
Class<?>[] parameterTypes, Object... args) {
23      ClassLoader classLoader = getClassLoader();
24      // Use names and ensure unique to protect against duplicates
25      // <1> 加载指定类型对应的，在 `META-INF/spring.factories` 里的类名的数组
26      Set<String> names = new LinkedHashSet<>(
27          SpringFactoriesLoader.loadFactoryNames(type, classLoader));
28      // <2> 创建对象们
29      List<T> instances = createSpringFactoriesInstances(type,
parameterTypes,
30          classLoader, args, names);
31      // <3> 排序对象们
32      AnnotationAwareOrderComparator.sort(instances);
33      return instances;
34    }
35
36    //  setp4:bean对象实例化
37    private <T> List<T> createSpringFactoriesInstances(Class<T> type,
Class<?>[] parameterTypes,
38        ClassLoader classLoader, Object[] args, Set<String> names) {
39      List<T> instances = new ArrayList<>(names.size()); // 数组大小，细节~
40      // 遍历 names 数组
41      for (String name : names) {
42        try {
43          // 获得 name 对应的类
44          Class<?> instanceClass = ClassUtils.forName(name, classLoader);
45          // 判断类是否实现自 type 类
```

```
        Assert.isAssignable(type, instanceClass);
        // 获得构造方法
        Constructor<?> constructor = instanceClass.getDeclaredConstructor(parameterTypes);
        // 创建对象
        T instance = (T) BeanUtils.instantiateClass(constructor, args);
        instances.add(instance);
      } catch (Throwable ex) {
        throw new IllegalArgumentException("Cannot instantiate " + type + " : " + name, ex);
      }
    }
    return instances;
  }

//  setp5:运行
public ConfigurableApplicationContext run(String... args) {
    // <1> 创建 StopWatch 对象，并启动。StopWatch 主要用于简单统计 run 启动过程的时长。
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    //
    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();
    // <2> 配置 headless 属性
    configureHeadlessProperty();
    // 获得 SpringApplicationRunListener 的数组，并启动监听
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
      // <3> 创建  ApplicationArguments 对象
      ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
      // <4> 加载属性配置。执行完成后，所有的 environment 的属性都会加载进来，包括 application.properties 和外部的属性配置。
      ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
      configureIgnoreBeanInfo(environment);
      // <5> 打印 Spring Banner
      Banner printedBanner = printBanner(environment);
      // <6> 创建 Spring 容器。
      context = createApplicationContext();
      // <7> 异常报告器
      exceptionReporters = getSpringFactoriesInstances(
          SpringBootExceptionReporter.class,
          new Class[] { ConfigurableApplicationContext.class }, context);
      // <8> 主要是调用所有初始化类的 initialize 方法
```

```
87        prepareContext(context, environment, listeners,
applicationArguments,
88            printedBanner);
89        // <9> 初始化 Spring 容器。
90        refreshContext(context);
91        // <10> 执行 Spring 容器的初始化的后置逻辑。默认实现为空。
92        afterRefresh(context, applicationArguments);
93        // <11> 停止 StopWatch 统计时长
94        stopWatch.stop();
95        // <12> 打印 Spring Boot 启动的时长日志。
96        if (this.logStartupInfo) {
97          new
StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog
(), stopWatch);
98        }
99        // <13> 通知 SpringApplicationRunListener 的数组，Spring 容器启动完成。
100       listeners.started(context);
101       // <14> 调用 ApplicationRunner 或者 CommandLineRunner 的运行方法。
102       callRunners(context, applicationArguments);
103     } catch (Throwable ex) {
104       // <14.1> 如果发生异常，则进行处理，并抛出 IllegalStateException 异常
105       handleRunFailure(context, ex, exceptionReporters, listeners);
106       throw new IllegalStateException(ex);
107     }
108
109     // <15> 通知 SpringApplicationRunListener 的数组，Spring 容器运行中。
110     try {
111       listeners.running(context);
112     } catch (Throwable ex) {
113       // <15.1> 如果发生异常，则进行处理，并抛出 IllegalStateException 异常
114       handleRunFailure(context, ex, exceptionReporters, null);
115       throw new IllegalStateException(ex);
116     }
117     return context;
118   }
119 }
```