SpringBoot

最简单的方式，快速整合所有技术栈

目录

1. 快速入门
2. 基础功能
3. 进阶使用
4. 核心原理
5. 自定义starter

1. 快速入门

SpringBoot特性
场景启动器
依赖管理
自动配置机制

SpringBoot特性

- SpringBoot 帮我们简单、快速地创建一个独立的、生产级别的 Spring 应用；
- 大多数 SpringBoot 应用只需要编写少量配置即可快速整合 Spring 平台以及第三方技术
- 特性：
  - 快速创建独立 Spring 应用
  - 直接嵌入Tomcat、Jetty or Undertow
  - 提供可选的 starter，简化应用整合
  - 按需自动配置 Spring 以及 第三方库
  - 提供生产级特性：如 监控指标、健康检查、外部化配置等
  - 无代码生成、无xml； 都是基于自动配置技术
- 总结：
  - 简化开发，简化配置，简化整合，简化部署，简化监控，简化运维

SpringBoot特性 - 快速部署

```
<!--  SpringBoot应用打包插件-->
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
```

打包：mvn clean package

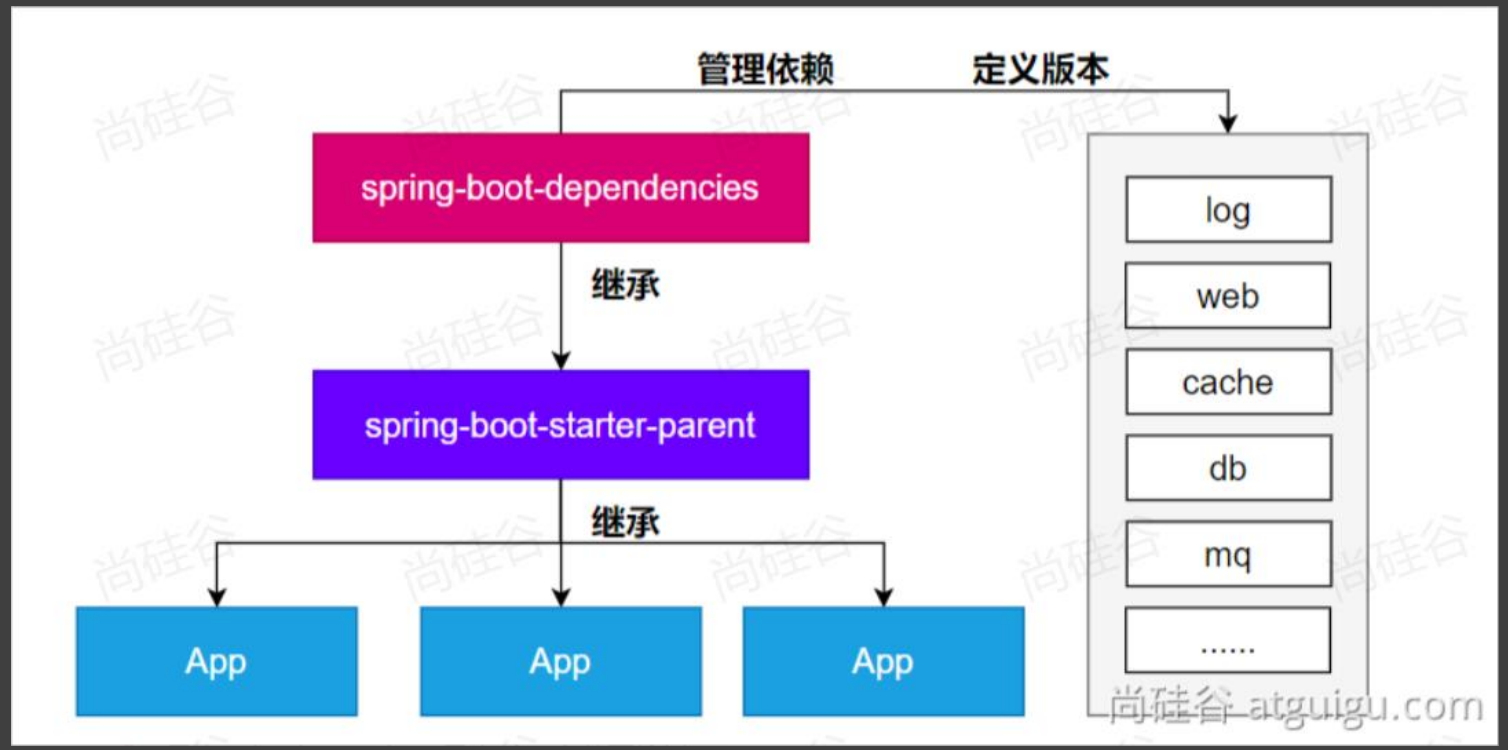
运行：java -jar demo.jar

场景启动器

- 场景启动器：导入相关的场景，拥有相关的功能。
- 默认支持的所有场景：
  - https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.build-systems.starters
  - 官方提供的场景：命名为：spring-boot-starter-*
  - 第三方提供场景：命名为：*-spring-boot-starter

依赖管理

- 思考：
  - 为什么 不用写版本号？ maven父子继承，父项目可以锁定版本
  - 哪些需要写版本号？
    - 父不管的都需要写版本
  - 如何修改默认版本号？
    - version 精确声明版本
    - 子覆盖父的属性设置
      - mysql.version



自动配置 - 初步理解

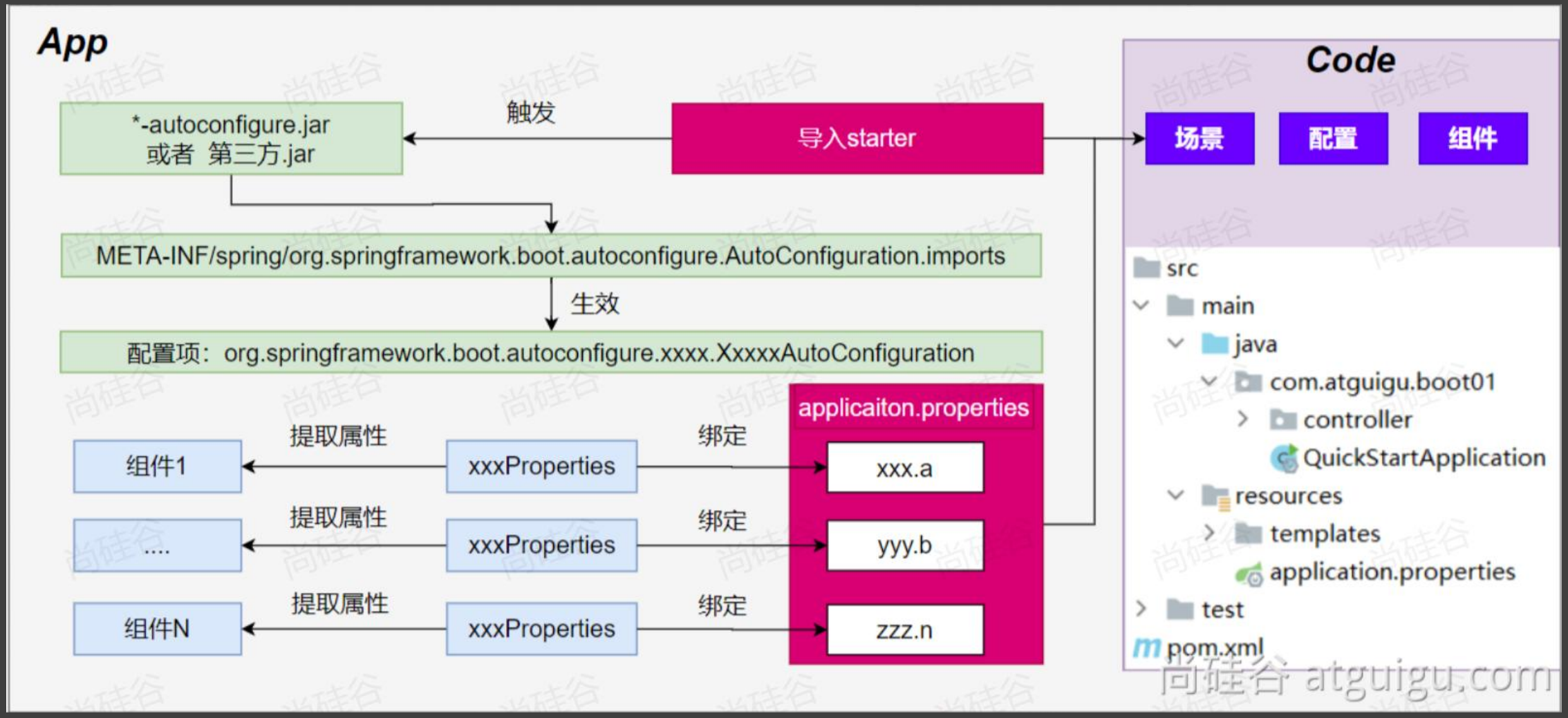
- 自动配置
  - 导入场景，容器中就会自动配置好这个场景的核心组件。
  - 如 Tomcat、 SpringMVC、 DataSource 等
  - 不喜欢的组件可以自行配置进行替换
- 默认的包扫描规则
  - SpringBoot只会扫描主程序所在的包及其下面的子包
- 配置默认值
  - 配置文件的所有配置项 是和 某个类的对象值进行一一绑定的。
  - 很多配置即使不写都有默认值，如：端口号，字符编码等
  - 默认能写的所有配置项： https://docs.spring.io/spring-boot/appendix/application-properties/index.html
- 按需加载自动配置
  - 导入的场景会导入全量自动配置包，但并不是都生效

约定大于配置

自动配置 - 完整流程

- 核心流程总结：
  - 1: 导入 starter，就会导入autoconfigure 包。
  - 2: autoconfigure 包里面 有一个文件 META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports,里面指定的所有启动要加载的自动配置类
  - 3: @EnableAutoConfiguration 会自动的把上面文件里面写的所有自动配置类都导入进来。xxxAutoConfiguration 是有条件注解进行按需加载
  - 4: xxxAutoConfiguration 给容器中导入一堆组件，组件都是从 xxxProperties 中提取属性值
  - 5: xxxProperties 又是和配置文件进行了绑定
- 效果：导入starter、修改配置文件，就能修改底层行为。

自动配置 - 完整流程


App
*-autoconfigure.jar
或者 第三方.jar
触发
导入starter
Code
场景
配置
组件
META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
生效
配置项：org.springframework.boot.autoconfigure.xxxx.XxxxxAutoConfiguration
applicaiton.properties
提取属性
绑定
组件1
xxxProperties
xxx.a
....
yyy.b
组件N
zzz.n
src
main
java
com.atguigu.boot01
controller
QuickStartApplication
resources
templates
application.properties
test
pom.xml

2. 基础功能

属性绑定
YAML文件
SpringApplication
日志系统

属性绑定

- 将容器中任意组件的属性值和配置文件的配置项的值进行绑定
  - 1、给容器中注册组件（@Component、@Bean）
  - 2、使用 @ConfigurationProperties 声明组件和配置文件的哪些配置项进行绑定

YAML 文件

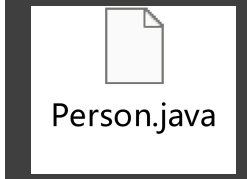
- 痛点：SpringBoot 集中化管理配置，application.properties
- 问题：配置多以后难阅读和修改，层级结构辨识度不高
- YAML: YAML Ain't Markup Language™
  - 设计目标，就是方便人类读写
  - 层次分明，更适合做配置文件
  - 使用.yaml或 .yml作为文件后缀

YAML 文件 - 基本语法

- 大小写敏感
- 键值对写法 k: v，使用空格分割k,v
- 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。换行
  - 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
- # 表示注释，从这个字符一直到行尾，都会被解析器忽略。
- Value支持的写法
  - 对象：键值对的集合，如：映射（map）/ 哈希（hash） / 字典（dictionary）
  - 数组：一组按次序排列的值，如：序列（sequence） / 列表（list）
  - 字面量：单个的、不可再分的值，如：字符串、数字、bool、日期

YAML 文件 – 数据表示

```java
@Data
class Person {
    private String name;
    private Integer age;
    private Date birthDay;
    private Boolean like;
    private Child child; //嵌套对象
     private List<Dog> dogs; //数组（里面是对象）
     private Map<String,Cat> cats; //表示Map
}
@Data
class Dog {
    private String name;
    private Integer age;
}
@Data
class Child {
    private String name;
    private Integer age;
    private Date birthDay;
    private List<String> text; //数组
}
@Data
class Cat {
    private String name;
    private Integer age;
}
```



properties文件表示

```
person.name=张三
person.age=18
person.birthDay=2010/10/12 12:12:12
person.like=true
person.child.name=李四
person.child.age=12
person.child.birthDay=2018/10/12
person.child.text[0]=abc
person.child.text[1]=def
person.dogs[0].name=小黑
person.dogs[0].age=3
person.dogs[1].name=小白
person.dogs[1].age=2
person.cats.c1.name=小蓝
person.cats.c1.age=3
person.cats.c2.name=小灰
person.cats.c2.age=2
```

yaml 文件表示

```yaml
person:
  name: 张三
  age: 18
  birthDay: 2010/10/10 12:12:12
  like: true
  child:
    name: 李四
    age: 20
    birthDay: 2018/10/10
    text: ["abc","def"]
  dogs:
    - name: 小黑
      age: 3
    - name: 小白
      age: 2
  cats:
    c1:
      name: 小蓝
      age: 3
    c2: {name: 小绿,age: 2}
```

SpringApplication

- 自定义 banner
  - 类路径添加banner.txt或设置spring.banner.location就可以定制 banner
  - https://www.bootschool.net/ascii
- 自定义 SpringApplication
  - new SpringApplication
  - new SpringApplicationBuilder

日志系统 - 简介

- 规范：项目开发不要写System.out.println()，用日志记录信息
- SpringBoot 默认使用 slf4j + logback

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	

日志系统 - 日志格式

- 默认输出格式：
  - 时间和日期：毫秒级精度
  - 日志级别：ERROR, WARN, INFO, DEBUG, or TRACE.
  - 进程 ID
  - ---：消息分割符
  - 线程名：使用[]包含
  - Logger 名：通常是产生日志的类名
  - 消息：日志记录的内容
- 注意：logback 没有FATAL级别，对应的是ERROR

日志系统 - 日志级别

由低到高： ALL,TRACE, DEBUG, INFO, WARN, ERROR,FATAL,OFF；

只会打印指定级别及以上级别的日志

- ALL：打印所有日志
- TRACE：追踪框架详细流程日志，一般不使用
- DEBUG：开发调试细节日志
- INFO：关键、感兴趣信息日志
- WARN：警告但不是错误的信息日志，比如：版本过时
- ERROR：业务错误日志，比如出现各种异常
- FATAL：致命错误日志，比如jvm系统崩溃
- OFF：关闭所有日志记录

```
logging:
  level:
    com.example.mypackage: DEBUG
    root: WARN
```

不指定级别的所有类，都使用 root 指定的级别作为默认级别

SpringBoot日志默认级别是 INFO

日志系统 - 日志分组

将相关的logger分组在一起，统一配置。SpringBoot 支持分组统一配置

```
logging.group.tomcat=org.apache.catalina,org.apache.coyote,org.apache.tomcat
logging.level.tomcat=trace
```

SpringBoot 预定义两个组

组名	范围
web	org.springframework.core.codec, org.springframework.http, org.springframework.web, org.springframework.boot.actuate.endpoint.web, org.springframework.boot.web.servlet.ServletContextInitializerBeans
sql	org.springframework.jdbc.core, org.hibernate.SQL, org.jooq.tools.LoggerListener

日志系统 - 文件输出

SpringBoot 默认只把日志写在控制台，如果想额外记录到文件，可以在application.properties中添加 logging.file.name 或 logging.file.path 配置项。

logging.file.name	logging.file.path	示例	效果
未指定	未指定		仅控制台输出
指定	未指定	my.log	写入指定文件。可以加路径
未指定	**指定**	/var/log	写入指定目录，文件名为spring.log
指定	**指定**		以logging.file.name为准

日志系统 - 文件归档与滚动切割

归档：每天的日志单独存到一个文档中。

切割：每个文件10MB，超过大小切割成另外一个文件。

默认**滚动切割**与归档规则如下：

配置项	描述
logging.**logback**.rollingpolicy.file-name-pattern	日志存档的文件名格式 默认值：${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz
logging.**logback**.rollingpolicy.clean-history-on-start	应用启动时是否清除以前存档；默认值：false
logging.**logback**.rollingpolicy.max-file-size	每个日志文件的最大大小；默认值：10MB
logging.**logback**.rollingpolicy.total-size-cap	日志文件被删除之前，可以容纳的最大大小（默认值：0B）。设置1GB则磁盘存储超过 1GB 日志后就会删除旧日志文件
logging.**logback**.rollingpolicy.max-history	日志文件保存的最大天数；默认值：7

日志系统 - 自定义配置

通常我们配置 application.properties 就够了。当然也可以自定义。比如：

日志系统	自定义
Logback	logback-spring.xml / logback.xml
Log4j2	log4j2-spring.xml / log4j2.xml
JDK (Java Util Logging)	logging.properties

日志系统 - 切换日志组合

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

日志系统 – 最佳实践

- 1、导入任何第三方框架，先排除它的日志包，因为Boot底层控制好了日志
- 2、修改 application.properties 配置文件，就可以调整日志的所有行为。如果不够，可以编写日志框架自己的配置文件放在类路径下就行，比如logback-spring.xml，log4j2-spring.xml
- 3、如需对接专业日志系统，也只需要把 logback 记录的日志灌倒 kafka之类的中间件，这和SpringBoot没关系，都是日志框架自己的配置，修改配置文件即可
- 4、业务中使用slf4j-api记录日志。不要再 sout 了

3. 进阶使用

Profiles环境隔离（掌握）
外部化配置（掌握）
单元测试进阶（熟悉）
可观测性（了解）

Profiles环境隔离 - 基础用法

- 环境隔离能力；快速切换开发、测试、生产环境
- 步骤：
  - 1. 标识环境：指定哪些组件、配置在哪个环境生效
    - @Profile 标记组件生效环境
  - 2. 切换环境：这个环境对应的所有组件和配置就应该生效
    - 激活环境：
      - 配置文件：spring.profiles.active=production,hsqldb
      - 命令行：java -jar demo.jar --spring.profiles.active=dev,hsqldb
    - 环境包含：
      - spring.profiles.include[0]=common
      - spring.profiles.include[1]=local
    - 生效的配置 = 默认环境配置 + 激活的环境 + 包含的环境配置
- 项目里面这么用
  - 基础的配置mybatis、log、xxx：写到包含环境中
  - 需要动态切换变化的 db、redis：写到激活的环境中

Profiles环境隔离 - 分组

- 创建 prod 组，指定包含 db 和 mq 配置
  - spring.profiles.group.prod[0]=db
  - spring.profiles.group.prod[1]=mq
- 使用 --spring.profiles.active=prod ， 激活prod，db，mq配置文件

Profiles环境隔离 - 配置文件

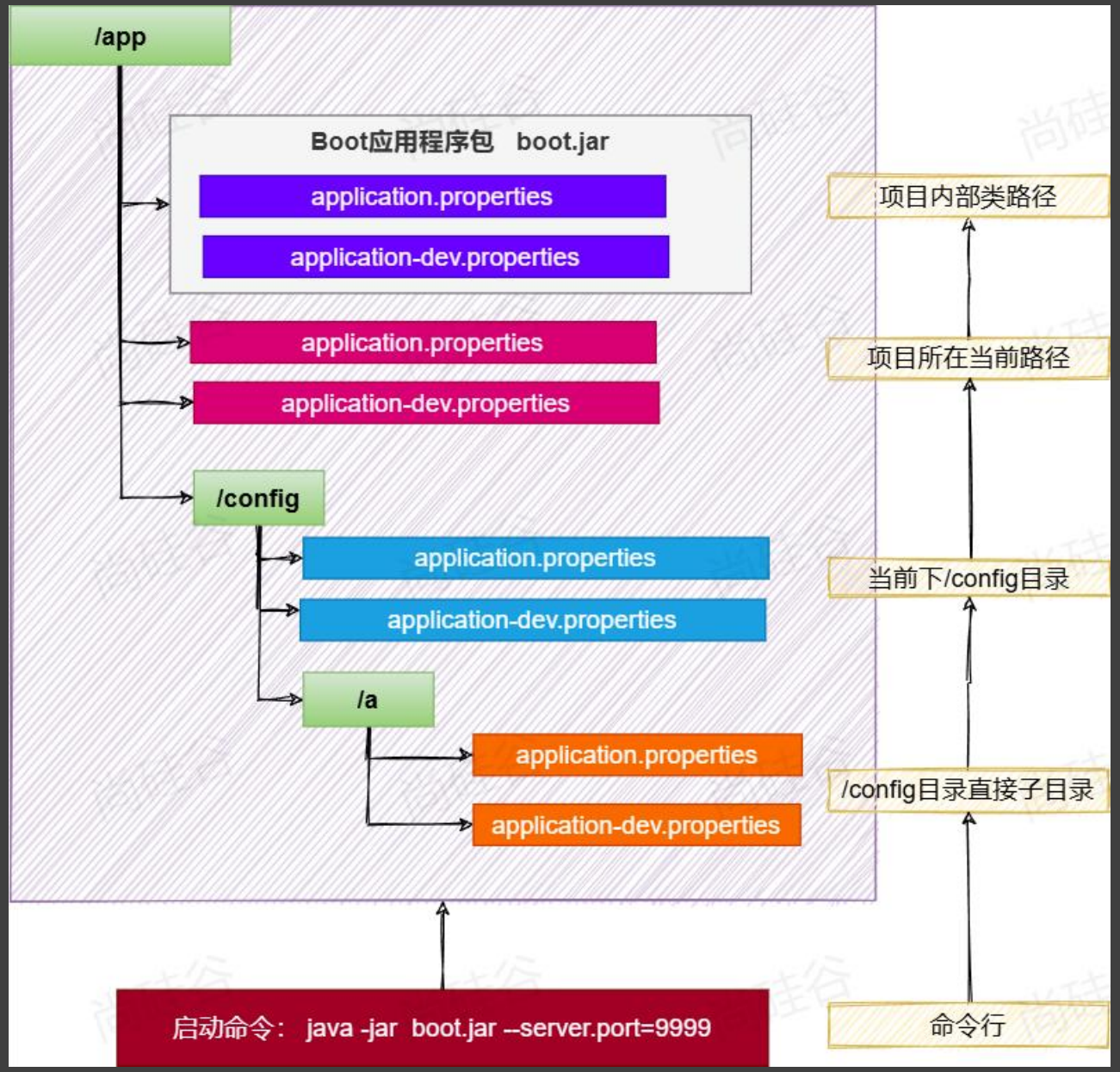
- application-{profile}.properties 可以作为指定环境的配置文件
- 激活这个环境，配置就会生效。最终生效的所有配置是
  - application.properties：主配置文件，任意时候都生效
  - application-{profile}.properties：指定环境配置文件，激活指定环境生效
- profile优先级 > application

外部化配置

外部配置优先于内部配置

属性占位符:
app.name=MyApp
app.description=${app.name} Hello

激活优先
外部优先



单元测试 - 测试注解

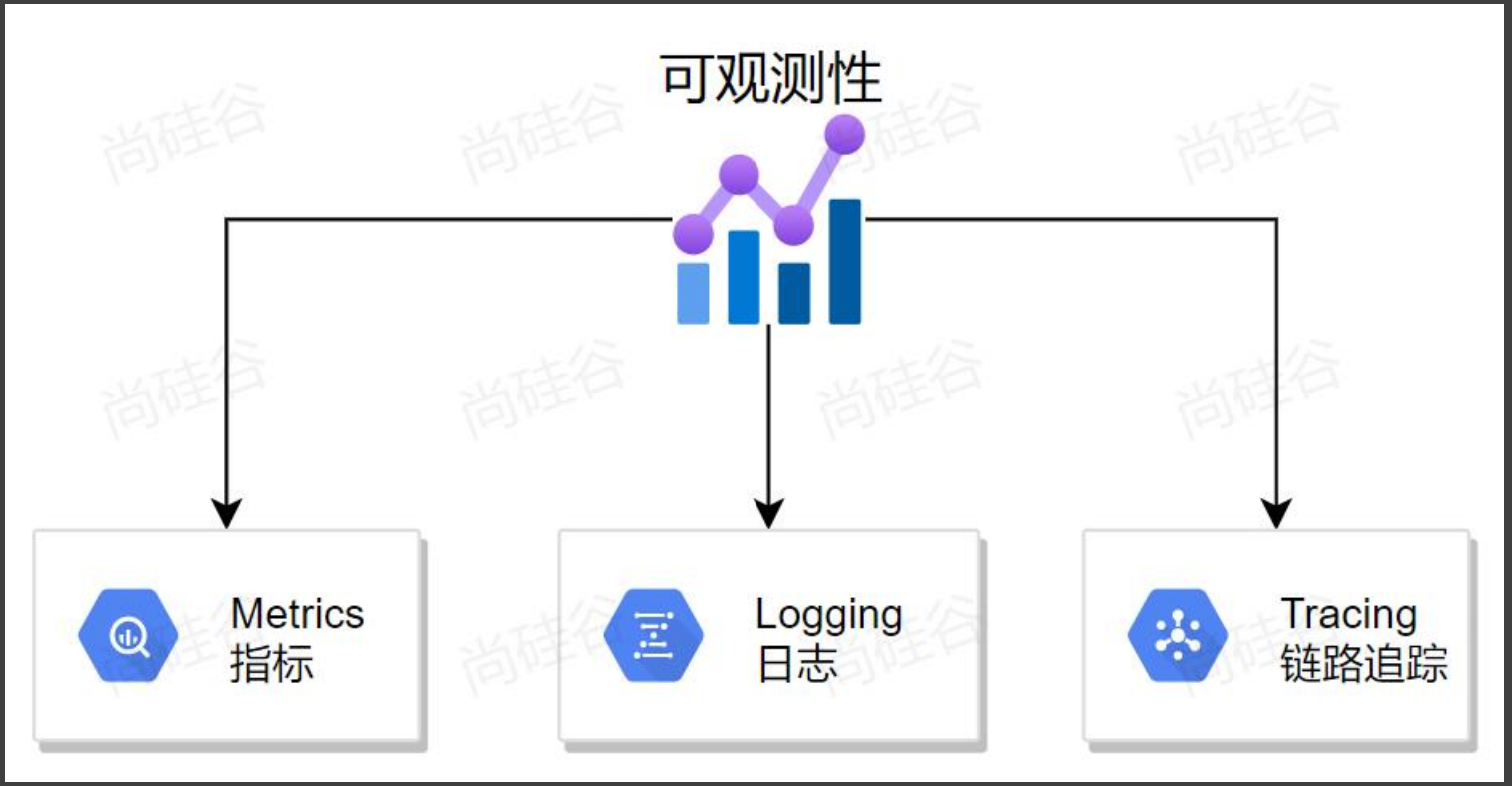
@Test :表示方法是测试方法。
@ParameterizedTest :表示方法是参数化测试，下方会有详细介绍
@RepeatedTest :表示方法可重复执行，下方会有详细介绍
@DisplayName :为测试类或者测试方法设置展示名称
@BeforeEach :表示在每个单元测试之前执行
@AfterEach :表示在每个单元测试之后执行
@BeforeAll :表示在所有单元测试之前执行
@AfterAll :表示在所有单元测试之后执行
@Tag :表示单元测试类别，类似于JUnit4中的@Categories
@Disabled :表示测试类或测试方法不执行，类似于JUnit4中的@Ignore
@Timeout :表示测试方法运行如果超过了指定时间将会返回错误
@ExtendWith :为测试类或测试方法提供扩展类引用

单元测试 - 断言机制

方法	说明
assertEquals	判断两个对象或两个原始类型是否相等
assertNotEquals	判断两个对象或两个原始类型是否不相等
assertSame	判断两个对象引用是否指向同一个对象
assertNotSame	判断两个对象引用是否指向不同的对象
assertTrue	判断给定的布尔值是否为 true
assertFalse	判断给定的布尔值是否为 false
assertNull	判断给定的对象引用是否为 null
assertNotNull	判断给定的对象引用是否不为 null
assertArrayEquals	数组断言
assertAll	组合断言
assertThrows	异常断言
assertTimeout	超时断言
fail	快速失败

可观测性

- 可观测性（Observability）指应用的运行数据，可以被线上进行观测、监控、预警等



可观测性

- SpringBoot 提供了 actuator 模块，可以快速暴露应用的所有指标
  - 导入： spring-boot-starter-actuator
- 访问 http://localhost:8080/actuator；
- 展示出所有可以用的监控端点

```yaml
management:
  endpoints:
    enabled-by-default: true
    web:
      exposure:
        include: '*'
```

可观测性 - Endpoints

端点名	描述
auditevents	暴露当前应用程序的审核事件信息。需要一个AuditEventRepository组件
beans	显示应用程序中所有Spring Bean的完整列表
caches	暴露可用的缓存
conditions	显示自动配置的所有条件信息，包括匹配或不匹配的原因
configprops	显示所有@ConfigurationProperties
env	暴露Spring的属性ConfigurableEnvironment
flyway	显示已应用的所有Flyway数据库迁移。需要一个或多个Flyway组件。
health	显示应用程序运行状况信息
httptrace	显示HTTP跟踪信息（默认情况下，最近100个HTTP请求-响应）。需要一个HttpTraceRepository组件
info	显示应用程序信息
integrationgraph	显示Spring integrationgraph 。需要依赖spring-integration-core
loggers	显示和修改应用程序中日志的配置
liquibase	显示已应用的所有Liquibase数据库迁移。需要一个或多个Liquibase组件。
metrics	显示当前应用程序的“指标”信息

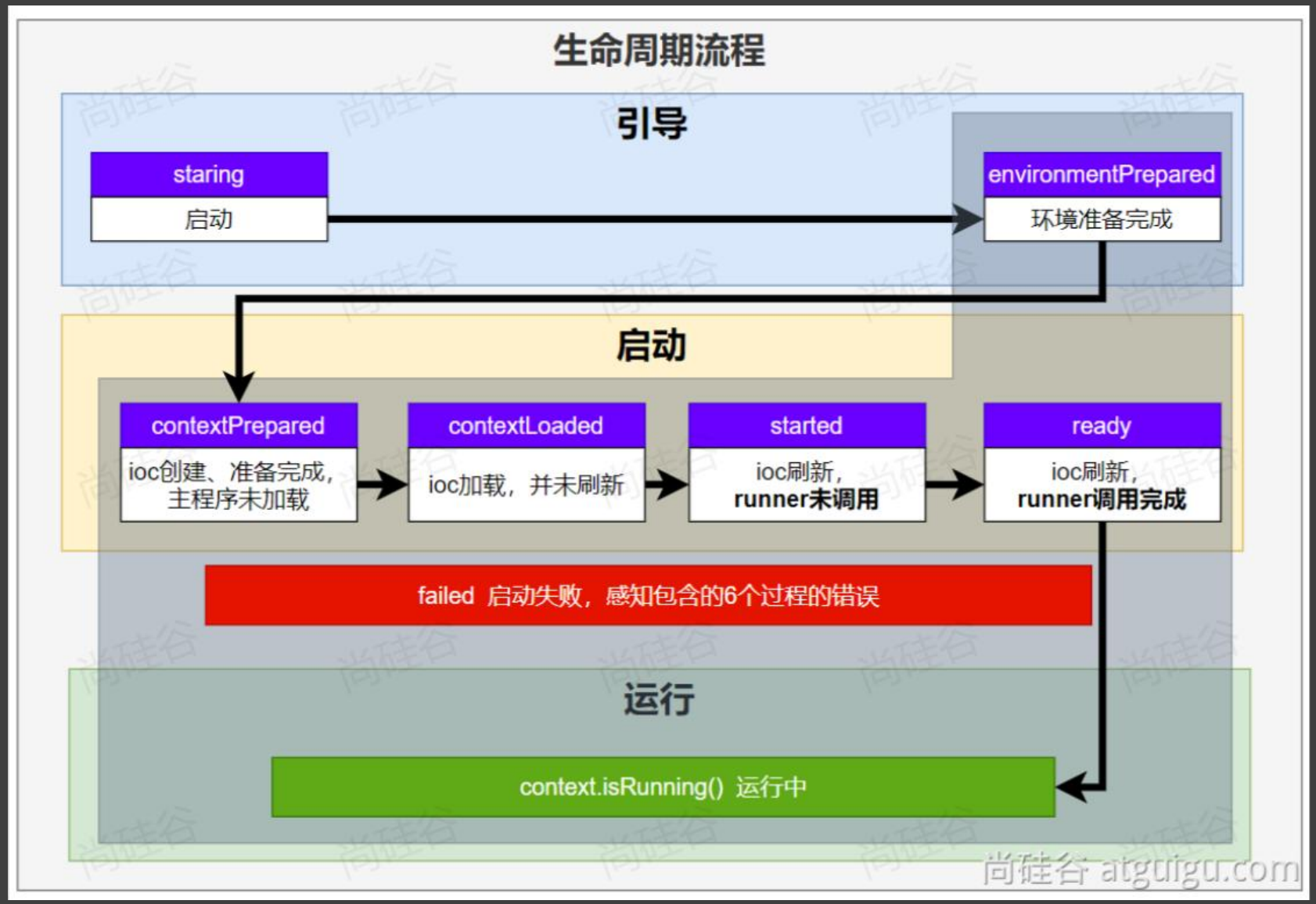
可观测性 - Endpoints

端点名	描述
mappings	显示所有@RequestMapping路径列表
scheduledtasks	显示应用程序中的计划任务
sessions	允许从Spring Session支持的会话存储中检索和删除用户会话。需要使用Spring Session的基于Servlet的Web应用程序
shutdown	使应用程序正常关闭。默认禁用
startup	显示由ApplicationStartup收集的启动步骤数据。需要使用SpringApplication进行配置BufferingApplicationStartup
threaddump	执行线程转储
heapdump	返回hprof堆转储文件
jolokia	通过HTTP暴露JMX bean（需要引入Jolokia，不适用于WebFlux）。需要引入依赖jolokia-core
logfile	返回日志文件的内容（如果已设置logging.file.name或logging.file.path属性）。支持使用HTTPRange标头来检索部分日志文件的内容
prometheus	以Prometheus服务器可以抓取的格式公开指标。需要依赖micrometer-registry-prometheus

4. 核心原理

生命周期监听（了解）
生命周期事件（了解）
事件驱动开发（了解）
自动配置原理（熟悉）

生命周期监听（了解）



- SpringApplicationRunListener
- META-INF/spring.factories

生命周期监听（了解）

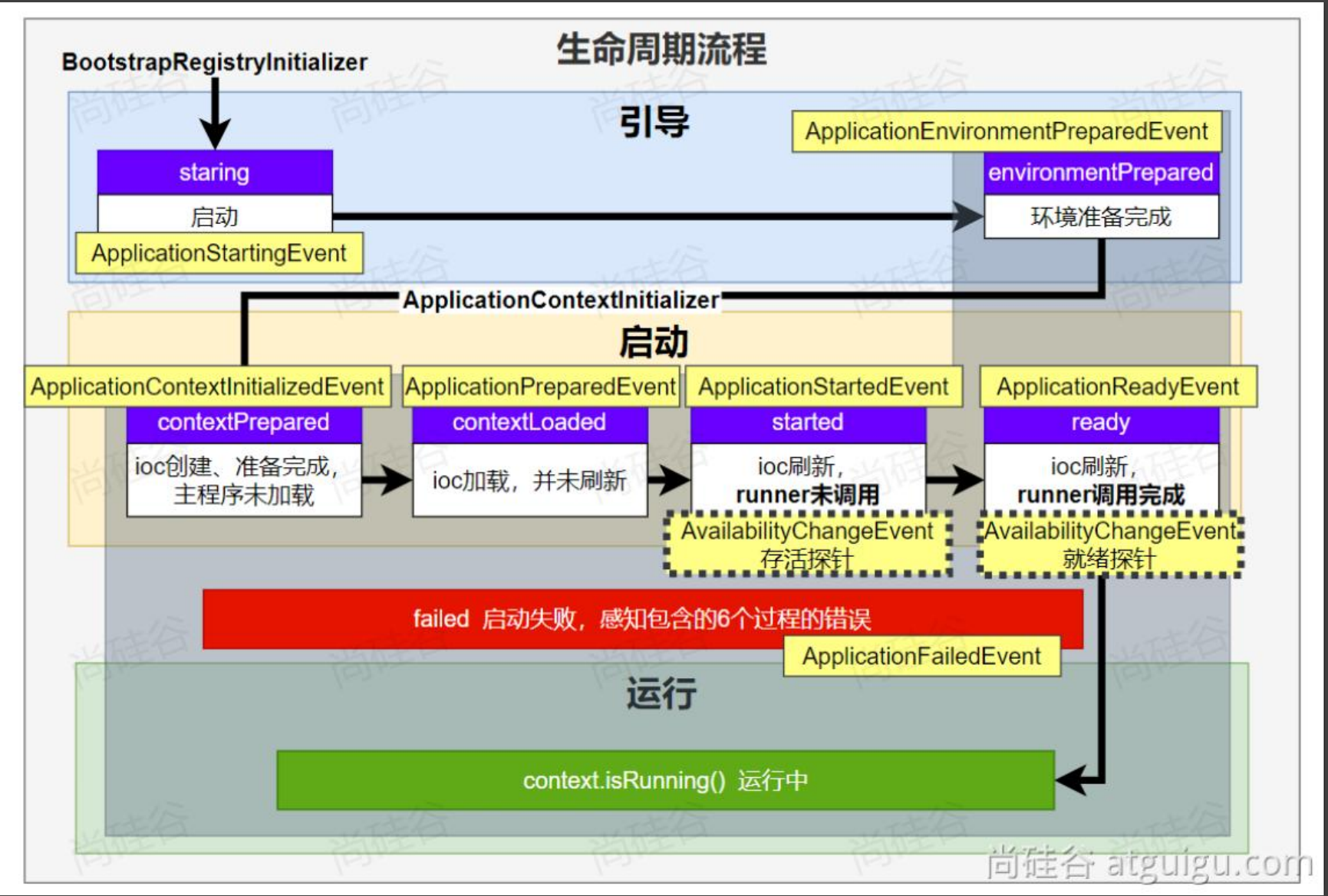
监听器	感知阶段	配置方式
BootstrapRegistryInitializer	特定阶段：引导初始化	1、META-INF/spring.factories 2、application.addBootstrapRegistryInitializer()
ApplicationContextInitializer	特定阶段：ioc容器初始化	1、META-INF/spring.factories 2、application.addInitializers()
ApplicationListener	全阶段	1、META-INF/spring.factories 2、SpringApplication.addListeners(…) 3、@Bean 或 @EventListener
SpringApplicationRunListener	全阶段	META-INF/spring.factories
ApplicationRunner	特定阶段：感知应用就绪	@Bean
CommandLineRunner	特定阶段：感知应用就绪	@Bean

最佳实践：

1、应用启动后做事：ApplicationRunner、CommandLineRunner

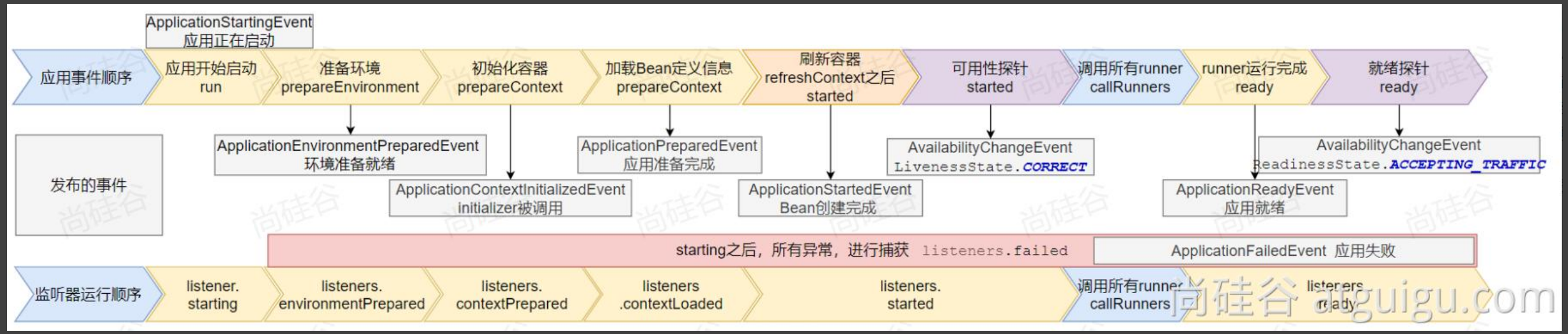
2、事件驱动开发：ApplicationListener

生命周期事件（了解）



系统九大事件
ApplicationStartingEvent
ApplicationEnvironmentPreparedEvent
ApplicationContextInitializedEvent
ApplicationPreparedEvent
ApplicationStartedEvent
AvailabilityChangeEvent
ApplicationReadyEvent
AvailabilityChangeEvent
ApplicationFailedEvent

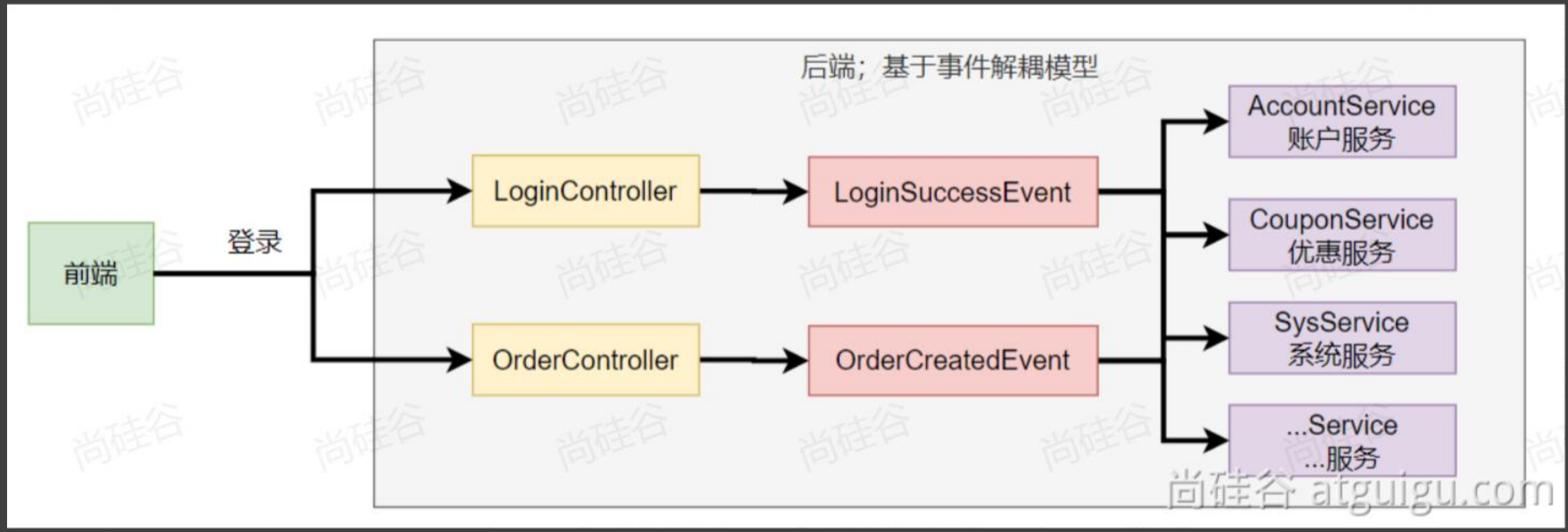
生命周期事件（了解）


ApplicationStartingEvent
应用正在启动
应用事件顺序
应用开始启动
run
准备环境
prepareEnvironment
初始化容器
prepareContext
加载Bean定义信息
prepareContext
刷新容器
refreshContext之后
started
可用性探针
started
调用所有runner
callRunners
runner运行完成
ready
就绪探针
ready
发布的事件
ApplicationEnvironmentPreparedEvent
环境准备就绪
ApplicationContextInitializedEvent
initializer被调用
ApplicationPreparedEvent
应用准备完成
ApplicationStartedEvent
Bean创建完成
AvailabilityChangeEvent
LivenessState.CORRECT
AvailabilityChangeEvent
ReadinessState.ACCEPTING_TRAFFIC
ApplicationReadyEvent
应用就绪
starting之后，所有异常，进行捕获 listeners.failed
ApplicationFailedEvent 应用失败
监听器运行顺序
listener.
starting
listeners.
environmentPrepared
listeners.
contextPrepared
listeners
.contextLoaded
listeners.
started
调用所有runner
callRunners
listeners
ready

事件驱动开发

- 应用启动过程生命周期事件感知（9大事件）
- 应用运行中事件感知（无数种）
- 事件驱动开发
  - 定义事件：
    - 任意事件：任意类可以作为事件类，建议命名 xxxEvent
    - 系统事件：继承 ApplicationEvent
  - 事件发布：
    - 组件实现 ApplicationEventPublisherAware
    - 自动注入 ApplicationEventPublisher
  - 事件监听：
    - 组件 + 方法标注@EventListener

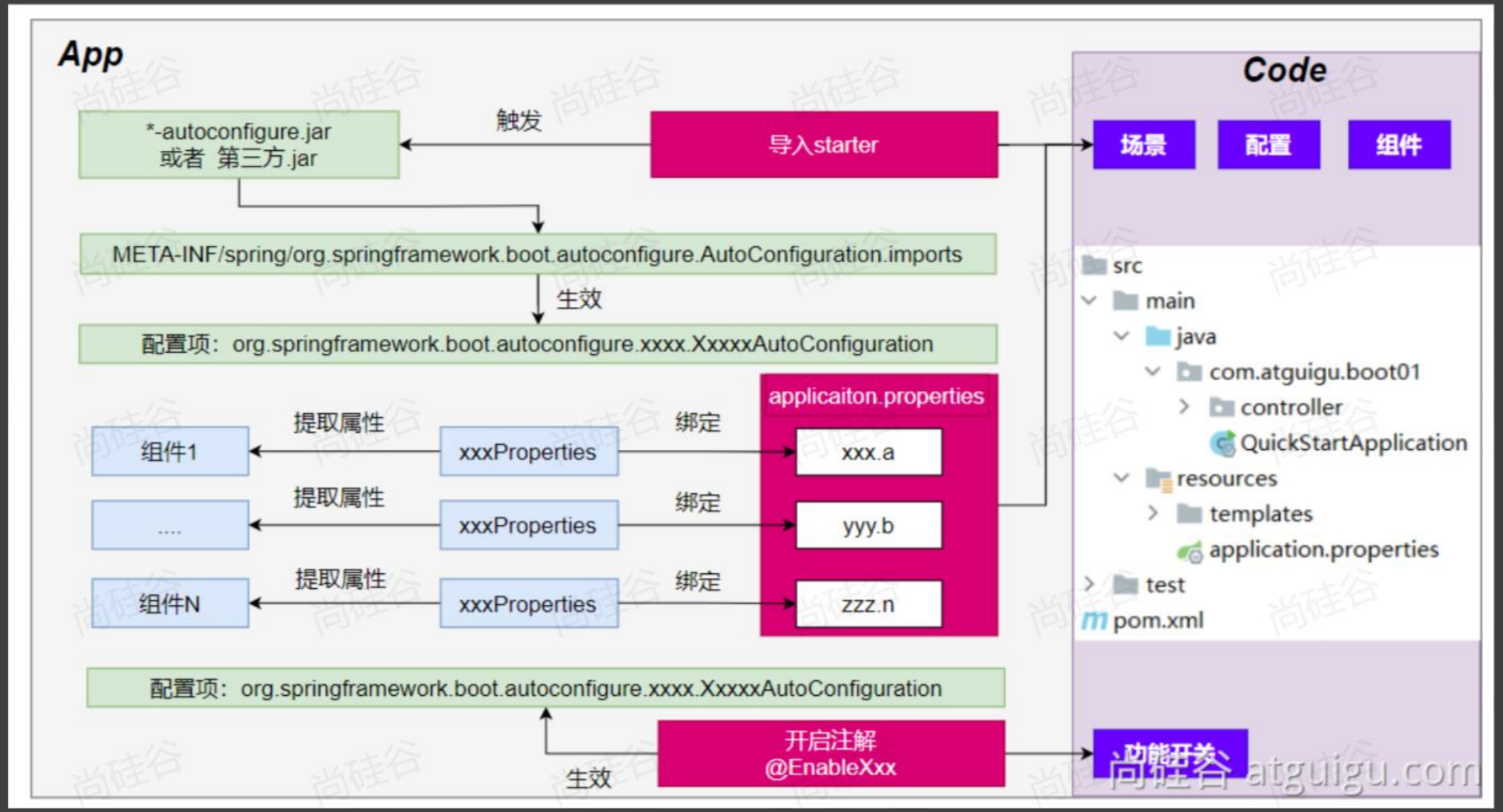
事件驱动开发


后端；基于事件解耦模型
前端
登录
LoginController
OrderController
LoginSuccessEvent
OrderCreatedEvent
AccountService
账户服务
CouponService
优惠服务
SysService
系统服务
...Service
...服务

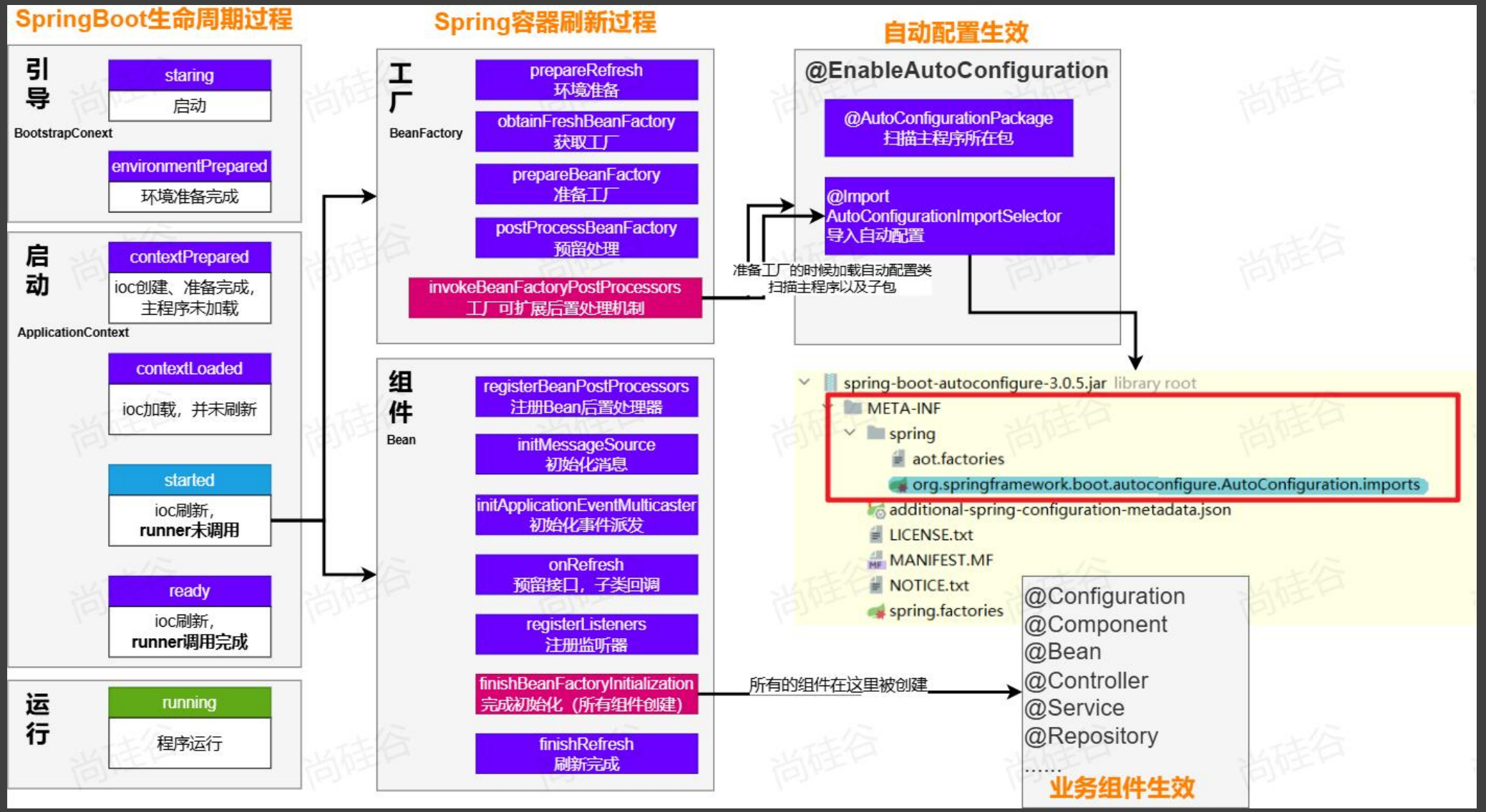
事件驱动开发

```
@Service
public class CouponService {

    @Order(1)
    @EventListener
    public void onEvent(LoginSuccessEvent loginSuccessEvent){
        System.out.println("===== CouponService ====感知到事件"+loginSuccessEvent);
        UserEntity source = (UserEntity) loginSuccessEvent.getSource();
        sendCoupon(source.getUsername());
    }

    public void sendCoupon(String username){
        System.out.println(username + " 随机得到了一张优惠券");
    }
}
```

自动配置原理


App
*-autoconfigure.jar
或者 第三方.jar
触发
导入starter
Code
场景
配置
组件
META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
生效
配置项：org.springframework.boot.autoconfigure.xxxx.XxxxxAutoConfiguration
applicaiton.properties
提取属性
绑定
组件1
xxxProperties
xxx.a
....
xxxProperties
yyy.b
组件N
xxxProperties
zzz.n
配置项：org.springframework.boot.autoconfigure.xxxx.XxxxxAutoConfiguration
开启注解
@EnableXxx
生效
功能开关
src
main
java
com.atguigu.boot01
controller
QuickStartApplication
resources
templates
application.properties
test
pom.xml

SpringBoot完整项目启动流程



5. 自定义starter

场景设计
基础抽取
@EnableXx机制
完全自动配置

场景设计

- 场景：抽取聊天机器人场景，它可以打招呼。
- 效果：任何项目导入此starter都具有打招呼功能，并且问候语中的人名需要可以在配置文件中修改

基础抽取

- 1. 创建自定义starter项目，引入spring-boot-starter基础依赖
- 2. 编写模块功能，引入模块所有需要的依赖。
- 3. 编写xxxAutoConfiguration自动配置类，帮其他项目导入这个模块需要的所有组件

@EnableXxx 机制

- 1. 编写自定义 @EnableXxx 注解
- 2. @EnableXxx 导入 自动配置类
- 3. 测试功能组件生效

完全自动配置

1、依赖 SpringBoot 的 SPI 机制

2、META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports 文件中编写好我们自动配置类的全类名即可

3、项目启动，自动加载我们的自动配置类

尚硅谷让天下没有难学的技术