FlywayAutoConfiguration 启动执行分析和实战

spring-boot-autoconfigration 里面的 META-INF/spring.factories 定义了很多 autoConfiguration，现在挑选其中的一个 FlywayAutoConfiguration 分析里面的逻辑.

1,FlywayAutoConfiguration

在 FlywayAutoConfiguration .class 里面 有定义： 需要 3 个条件。。。

```
@ConditionalOnClass({Flyway.class}) //有 jar 包 时
@ConditionalOnBean({DataSource.class}) //有定义了一个 datasource 的 bean 并且注入到这里
@ConditionalOnProperty( //定义了属性 spring.flyway.enabled= true
prefix = "spring.flyway",
name = {"enabled"},
matchIfMissing = true
)
@AutoConfigureAfter({DataSourceAutoConfiguration.class, HibernateJpaAutoConfiguration.class})
public class FlywayAutoConfiguration {
}
```

2,继续分析 静态内部类 ： FlywayConfiguration

```
@Configuration
@ConditionalOnMissingBean({Flyway.class})
@EnableConfigurationProperties({DataSourceProperties.class, FlywayProperties.class})
public static class FlywayConfiguration {
	@Bean
	@ConditionalOnMissingBean
	public FlywayMigrationInitializer flywayInitializer(Flyway flyway) {
		return new FlywayMigrationInitializer(flyway, this.migrationStrategy);
	}
```

当没有定义 Flyway 这个 bean 时，就配置 FlywayConfiguration

```
public class FlywayMigrationInitializer implements InitializingBean, Ordered {
	private final Flyway flyway;
	private final FlywayMigrationStrategy migrationStrategy;
	private int order;

	public FlywayMigrationInitializer(Flyway flyway) {
		this(flyway, (FlywayMigrationStrategy)null);
	}

	public FlywayMigrationInitializer(Flyway flyway, FlywayMigrationStrategy migrationStrategy) {
		this.order = 0;
		Assert.notNull(flyway, "Flyway must not be null");
		this.flyway = flyway;
		this.migrationStrategy = migrationStrategy;
	}

	public void afterPropertiesSet() throws Exception {
		if (this.migrationStrategy != null) {
		this.migrationStrategy.migrate(this.flyway);
		} else {
		this.flyway.migrate();
		}

	}
```

定义了一个 bean： FlywayMigrationInitializer flywayInitializer

这个 bean 里面 调用生命周期方法 afterPropertiesSet ， 继续执行核心逻辑处理 this.flyway.migrate();

3,读取 classpath:db/migration 执行分析

```
public class FlywayProperties {
	private boolean enabled = true;
	private boolean checkLocation = true;
	private List<String> locations = new ArrayList(Collections.singletonList("classpath:db/migration"));
```

跟踪代码 ，发现 在 public Flyway flyway()这个方法 里面的 configureProperties(configuration, properties);读取这个路径

```
public Flyway flyway(FlywayProperties properties,
DataSourceProperties dataSourceProperties, ResourceLoader resourceLoader,
ObjectProvider<DataSource> dataSource,
@FlywayDataSource ObjectProvider<DataSource> flywayDataSource,
ObjectProvider<FlywayConfigurationCustomizer> fluentConfigurationCustomizers,
ObjectProvider<Callback> callbacks,
ObjectProvider<FlywayCallback> flywayCallbacks) {
      FluentConfiguration configuration = new FluentConfiguration();
      DataSource dataSourceToMigrate = configureDataSource(configuration,
      properties, dataSourceProperties, flywayDataSource.getIfAvailable(),
      dataSource.getIfAvailable());
      checkLocationExists(dataSourceToMigrate, properties, resourceLoader);
      configureProperties(configuration, properties);
```

继续分析 configureProperties 流程，通过 LocationResolver 来读取 locations ，源码如下：

```
private void configureProperties(FluentConfiguration configuration,
FlywayProperties properties) {
      PropertyMapper map = PropertyMapper.get().alwaysApplyingWhenNonNull();
      String[] locations = new LocationResolver(configuration.getDataSource())
      .resolveLocations(properties.getLocations()).toArray(new String[0]);
```

到这来 flyway 启动执行分析完毕，下面实战。

4， flyway 执行实战

总结下 FlywayAutoConfiguration 的条件注入

1， classpath 里面有 Flyway.class

2, 定义了一个 datasource 的 bean 并且注入到这里

3，定义了属性 spring.flyway.enabled= true

因此 pom 里面引入 flyway，datasource 相关的 jar，这里使用 mysql 作为例子

（1），引入 pom

```
<dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
      <groupId>org.flywaydb</groupId>
      <artifactId>flyway-core</artifactId>
</dependency>
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

（2） 定义 datasource 和 flyway 属性

```
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://192.168.80.79:3306/test?useSSL=false
spring.datasource.password=WTWFCY8mkwA&Wm1U

spring.flyway.locations=classpath:db/migration

spring.flyway.enabled=true
```

（3），定义 flyway 执行 sql 语句。

在/resources/db/migration 目录下面 创建我们的 flyway 脚本 ，注意 V 开头 版本号递增，2 个下划线

V1.00__init_table.sql

V1.02__insert_data.sql

（4）， 注入 datasource 到我们的 flyway app 里面

```
@SpringBootApplication
public class Application {
```



```
    // 在注入 DataSource 时，会自动执行 Flyway
    @Autowired
    DataSource dataSource;

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

}
```

执行结果：

```
2019-03-26 17:44:38.989 INFO 5632 --- [ main] o.f.c.internal.database.DatabaseFactory : Database: jdbc:mysql://192.168.80.79:3306/test (MySQL 5.7)
2019-03-26 17:44:39.045 INFO 5632 --- [ main] o.f.core.internal.command.DbValidate : Successfully validated 2 migrations (execution time 00:00.022s)
2019-03-26 17:44:39.114 INFO 5632 --- [ main] o.f.c.i.s.JdbcTableSchemaHistory : Creating Schema History table: `test`.`flyway_schema_history`
2019-03-26 17:44:39.280 INFO 5632 --- [ main] o.f.core.internal.command.DbMigrate : Current version of schema `test`: << Empty Schema >>
2019-03-26 17:44:39.293 INFO 5632 --- [ main] o.f.core.internal.command.DbMigrate : Migrating schema `test` to version 1.00 - init table
2019-03-26 17:44:39.411 INFO 5632 --- [ main] o.f.core.internal.command.DbMigrate : Migrating schema `test` to version 1.02 - insert data
2019-03-26 17:44:39.460 INFO 5632 --- [ main] o.f.core.internal.command.DbMigrate : Successfully applied 2 migrations to schema `test` (execution time 00:00.350s)
```

因此不用编写 太多的 flyway 处理逻辑，spring boot 的 FlywayAutoConfiguration 已经帮我们制定了一套框架，照着框架做就可以了 。。。