1 DataSourceAutoConfiguration 功能概况

- 初始化 `DataSourceProperties` 配置文件
- 初始化数据源
- 执行 `sql 文件`
- 为数据源注册一个 `DataSourcePoolMetadataProvider` 实例

2 源码分析

```
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
//2.1 从配置文件中映射 DataSource 的值
@EnableConfigurationProperties(DataSourceProperties.class)
//2.4/2.5
@Import({ DataSourcePoolMetadataProvidersConfiguration.class,
      DataSourceInitializationConfiguration.class })
public class DataSourceAutoConfiguration {

   //2.2
   @Configuration
   //判断是否引入 内置数据库：H2，DERBY，HSQL
   @Conditional(EmbeddedDatabaseCondition.class)
   //如果这是没有DataSource/XADataSource 对应的 BeanDefinition，就通过导入
EmbeddedDataSourceConfiguration.class 来，配置内置数据库对应的数据源！！
   @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
   @Import(EmbeddedDataSourceConfiguration.class)
   protected static class EmbeddedDatabaseConfiguration {}

   //2.3
   @Configuration
   //判断是否引入依赖的数据源：HikariDataSource、tomcat.jdbc.pool.DataSource、
BasicDataSource
   @Conditional(PooledDataSourceCondition.class)
   //如果这是没有DataSource/XADataSource 对应的 BeanDefinition，就通过以下属性的配置文件，配置数
据源！！
   //配置数据源的时候，如果没有指定一些数据库的参数，就会报错哦
```

```
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import({ DataSourceConfiguration.Hikari.class,
DataSourceConfiguration.Tomcat.class,
            DataSourceConfiguration.Dbcp2.class, DataSourceConfiguration.Generic.class,
            DataSourceJmxConfiguration.class })
    protected static class PooledDataSourceConfiguration {}

    //详细说明见 2.3
    static class PooledDataSourceCondition extends AnyNestedCondition {}
    //详细说明见 2.2
    static class EmbeddedDatabaseCondition extends SpringBootCondition {}
}
```

2.1 初始化 DataSourceProperties 配置文件

- 如果设置的不是内置数据库的话： 1) 必须配置的有：url,username,password 2) 数据库名不是必须的 3) driverClassName 不是必须的：可以从 url 中推导出 4) type 不是必须的：可以从上下文中推导出
- 如果设置的是内置数据库的话： 1) 必须配置的有：引入内置数据库依赖，如：H2 2) 其它所有的都可以不配置(有默认的配置)，如果随意配置，可能产生冲突

```
@ConfigurationProperties(prefix = "spring.datasource")
public class DataSourceProperties implements BeanClassLoaderAware, InitializingBean {
    private ClassLoader classLoader;
    //数据库名：如果使用内置数据库，默认为testdb
    private String name;
    //Whether to generate a random datasource name
    private boolean generateUniqueName;
    //如果generateUniqueName==true,则不使用name,而使用uniqueName来做数据库名
    private String uniqueName;
    //完整的数据库连接池名。默认从项目中检测出
    private Class<? extends DataSource> type;
    //JDBC driver的完整名，默认从URL中检测出相对应的driver
    private String driverClassName;
    //JDBC URL of the database
    private String url;

    //Login username of the database
    private String username;
    //Login password of the database
    private String password;
    //JNDI 数据源的位置：如果指定了，则数据库连接数据将会失效：
driverClassName,url,username,password
    private String jndiName;
    //初始化database使用的sql文件的模式，默认是EMBEDDED，如果是NONE就不会执行sql文件
    //如果设置的模式和检测出来的模式不匹配，也不会执行sql文件
    private DataSourceInitializationMode initializationMode =
DataSourceInitializationMode.EMBEDDED;

    //执行sql文件相关，schema-${platform}.sql,data-${platform}.sql
    //默认执行不带 platform 的 sql 文件 + 带 platform 的 sql 文件
    private String platform = "all";
    //具体的 schema 文件的位置，如果指定了这个就不会查找默认的sql文件了
```

```
    private List<String> schema;
    //执行schema使用数据库的用户名
    private String schemaUsername;
    //执行schema使用数据库的密码，如果schemaUsername和schemaPassword都不指定，就使用 **主数据源
** 作为执行目的数据库！
    private String schemaPassword;
    //同schema
    private List<String> data;
    private String dataUsername;
    private String dataPassword;

    //如果初始化database时报错，是否继续
    private boolean continueOnError = false;
    //Statement separator in SQL initialization scripts.
    private String separator = ";";
    //SQL scripts encoding.
    private Charset sqlScriptEncoding;
    //默认的内置数据库连接信息：
    //1 NONE(null, null, null)
    //2 H2(EmbeddedDatabaseType.H2,
"org.h2.Driver","jdbc:h2:mem:%s;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE")
    //3 DERBY(...)
    //4 HSQL(...)
    private EmbeddedDatabaseConnection embeddedDatabaseConnection =
EmbeddedDatabaseConnection.NONE;

    private Xa xa = new Xa();
    //从项目中匹配相应的内置数据库，查找是否引入了相应的依赖，如果引入了H2依赖，这里
embeddedDatabaseConnection就设置成 H2
    @Override
    public void afterPropertiesSet() throws Exception {
        this.embeddedDatabaseConnection = EmbeddedDatabaseConnection
                .get(this.classLoader);
    }

    // 通过 spring.datasource 属性 初始化一个DataSourceBuilder，用来方便的创建datasource，就是
一个封装的方法
    public DataSourceBuilder<?> initializeDataSourceBuilder() {
        return DataSourceBuilder.create(getClassLoader()).type(getType())
                .driverClassName(determineDriverClassName()).url(determineUrl())
                .username(determineUsername()).password(determinePassword());
    }
    //智能的获取DriverClassName
    public String  determineDriverClassName() {
        if (StringUtils.hasText(this.driverClassName)) {
            Assert.state(driverClassIsLoadable(),
                    () -> "Cannot load driver class: " + this.driverClassName);
            return this.driverClassName;
        }
        String driverClassName = null;
        if (StringUtils.hasText(this.url)) {
            driverClassName =
DatabaseDriver.fromJdbcUrl(this.url).getDriverClassName();
```

```
        }
        //如果走到这，还没识别出 driverClassName，且它为null，就去内置数据库中找匹配的
        //如果项目中没有引入 内置数据库依赖，那就会报错啦
        if (!StringUtils.hasText(driverClassName)) {
            driverClassName = this.embeddedDatabaseConnection.getDriverClassName();
        }
        if (!StringUtils.hasText(driverClassName)) {
            throw new DataSourceBeanCreationException(
                    "Failed to determine a suitable driver class", this,
                    this.embeddedDatabaseConnection);
        }
        return driverClassName;
    }
    private boolean driverClassIsLoadable() {
        try {
            ClassUtils.forName(this.driverClassName, null);
            return true;
        }
        catch (UnsupportedClassVersionError ex) {
            // Driver library has been compiled with a later JDK, propagate error
            throw ex;
        }
        catch (Throwable ex) {
            return false;
        }
    }
    //Determine the url to use based on this configuration and the environment.
    public String determineUrl() {
        if (StringUtils.hasText(this.url)) {
            return this.url;
        }
        String databaseName = determineDatabaseName();
        String url = (databaseName != null)
                ? this.embeddedDatabaseConnection.getUrl(databaseName) : null;
        if (!StringUtils.hasText(url)) {
            throw new DataSourceBeanCreationException(
                    "Failed to determine suitable jdbc url", this,
                    this.embeddedDatabaseConnection);
        }
        return url;
    }
    //Determine the name to used based on this configuration.
    public String determineDatabaseName() {
        if (this.generateUniqueName) {
            if (this.uniqueName == null) {
                this.uniqueName = UUID.randomUUID().toString();
            }
            return this.uniqueName;
        }
        if (StringUtils.hasLength(this.name)) {
            return this.name;
        }
        if (this.embeddedDatabaseConnection != EmbeddedDatabaseConnection.NONE) {
```

```
            return "testdb";
        }
        return null;
    }
    //
    public String determineUsername() {
        if (StringUtils.hasText(this.username)) {
            return this.username;
        }
        if (EmbeddedDatabaseConnection.isEmbedded(determineDriverClassName())) {
            return "sa";
        }
        return null;
    }

    public String determinePassword() {
        if (StringUtils.hasText(this.password)) {
            return this.password;
        }
        if (EmbeddedDatabaseConnection.isEmbedded(determineDriverClassName())) {
            return "";
        }
        return null;
    }
    //XA Specific datasource settings.
    public static class Xa {
        //XA datasource fully qualified name.
        private String dataSourceClassName;
        //Properties to pass to the XA data source.
        private Map<String, String> properties = new LinkedHashMap<>();
    }
}
```

2.2 引入 EmbeddedDatabaseConfiguration 配置类

```
//所有的 condition 类都会最终继承 SpringBootCondition，SpringBootCondition 是一个模板类，继承它后，我们只需要实现核心的 getMatchOutCome() 方法来自定义一个 Condition 类了。当这个类被 @Conditional 注解引入的时候，最终时候执行这个核心方法来判断是否匹配的
static class EmbeddedDatabaseCondition extends SpringBootCondition {
    private final SpringBootCondition pooledCondition = new
PooledDataSourceCondition();
    @Override
    public ConditionOutcome getMatchOutcome(ConditionContext context,
            AnnotatedTypeMetadata metadata) {
        ConditionMessage.Builder message = ConditionMessage
                .forCondition("EmbeddedDataSource");
        // anyMatches() 就是一个 SpringbootCondition 类中的模板方法，意思是：匹配任意一个 pooledCondition 中的条件
        // 这里 pooledCondition 中的条件其实是匹配非内置数据库的条件，这就很奇怪了，为什么不把 匹配非内置数据库的配置放在前面呢？
        if (anyMatches(context, metadata, this.pooledCondition)) {
            return ConditionOutcome
```

```
                    .noMatch(message.foundExactly("supported pooled data source"));
        }
        //这里查找了项目中有没有引入 H2，DERBY，HSQL 这3个class，如果没有引入，就返回 null，引入了返回 第一个 type
        EmbeddedDatabaseType type = EmbeddedDatabaseConnection
                .get(context.getClassLoader()).getType();
        if (type == null) {
            return ConditionOutcome
                    .noMatch(message.didNotFind("embedded database").atAll());
        }
        return ConditionOutcome.match(message.found("embedded database").items(type));
    }
}
```

2.3 引入 PooledDataSourceConfiguration 配置类

```
@Configuration
//满足其中的任意一个：1）有spring.datasource.type属性 2）满足PooledDataSourceAvailableCondition：项目中引入了数据源依赖
@Conditional(PooledDataSourceCondition.class)
@ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
//如果满足上面条件，就解析一下几个配置类（注意顺序，hikari优先）
@Import({ DataSourceConfiguration.Hikari.class, DataSourceConfiguration.Tomcat.class,
      DataSourceConfiguration.Dbcp2.class, DataSourceConfiguration.Generic.class,
      DataSourceJmxConfiguration.class })
protected static class PooledDataSourceConfiguration {}

 //继承了 AnyNestedCondition 的类，会对这个类中的所有内部类(不一定非得是静态内部类)上的注解做匹配，只要其中有一个匹配了，就匹配了
//说明：如果没有spring.datasource.type属性，就默认查看项目中有没有引入：hikari，tomcat，dbcp2。这样说明如果项目中exclude了这3个，那么就必须使用 spring.datasource.type来指定数据库连接池了
//type 属性优先级比较低，是在找不到，就通过 DataSourceConfiguration.Generic.class 类，来根据 type 属性配置
static class PooledDataSourceCondition extends AnyNestedCondition {
   PooledDataSourceCondition() {
      //因为 AnyNestedCondition 实现了 ConfigurationCondition，所以要设置 这个属性
      //这个属性在 sholudSkip() 方法中会用到，如果这个属性是 REGISTER_BEAN 的话，在生成 configClass 阶段就不会进行匹配过滤，要等到 loadBeanDefintion 的时候，在进行过滤
      //因为类中的静态内部类，都被 @ConditionalOnProperty 注解，这些注解都是在 configClass 阶段做匹配的，所以要设置为 PARSE_CONFIGURATION
      //如果这里设置为 REGISTER_BEAN，但是内部有应该在 configClass 阶段做匹配的，就不符合整体思想了（这样本应该在 configClass 阶段就做匹配的，延迟到了 loadBeanDefintion 阶段），就可能能出现莫名其妙的问题。
      //进一步思考：继承了 AnyNestedCondition 的子类中，不应该同时存在 configClass 阶段做匹配和在 loadBeanDefintion 阶段匹配的
      super(ConfigurationPhase.PARSE_CONFIGURATION);
   }
   //条件一：是否配置了 spring.datasource.type 属性
   @ConditionalOnProperty(prefix = "spring.datasource", name = "type")
   static class ExplicitType {
   }
   //条件二：项目中是否引入了数据源依赖(如，hikari)
```

```
    @Conditional(PooledDataSourceAvailableCondition.class)
    static class PooledDataSourceAvailable {
    }
}
static class PooledDataSourceAvailableCondition extends SpringBootCondition {
  @Override
  public ConditionOutcome getMatchOutcome(ConditionContext context,
        AnnotatedTypeMetadata metadata) {
     //这个类只是用来传递消息的
     ConditionMessage.Builder message = ConditionMessage
           .forCondition("PooledDataSource");
      //getDataSourceClassLoader(context)：内部做class.forName来找项目中的相关class，找到了就不为null啦，一般肯定能找到的，在org.springframework.boot:spring-boot-starter-jdbc中就已经引入了 hikariDatabase，而在 spring.boot:mybatis-spring-boot-starter中引入了 jdbc!
     if (getDataSourceClassLoader(context) != null) {
        return ConditionOutcome
              .match(message.foundExactly("supported DataSource"));
     }
     return ConditionOutcome
           .noMatch(message.didNotFind("supported DataSource").atAll());
  }
  private ClassLoader getDataSourceClassLoader(ConditionContext context) {
      //在DataSourceBuilder中有个关键的 findType方法来按：hikari，tomcat，dbcp2顺序查找，一查到就返回
     Class<?> dataSourceClass = DataSourceBuilder
           .findType(context.getClassLoader());
     return (dataSourceClass != null) ? dataSourceClass.getClassLoader() : null;
  }
}
```

2.4 导入 DataSourceInitializationConfiguration 配置类

- 见 Springboot 源码分析—— 自动运行 SQL 文件解析

2.5 导入 DataSourcePoolMetadataProvidersConfiguration 配置类

```
//为数据源注册一个DataSourcePoolMetadataProvider实例，这个实例主要用于获取内置数据源的一些状态
@Configuration
public class DataSourcePoolMetadataProvidersConfiguration {
    @Configuration
    @ConditionalOnClass(org.apache.tomcat.jdbc.pool.DataSource.class)
    static class TomcatDataSourcePoolMetadataProviderConfiguration {...}
    @Configuration
    @ConditionalOnClass(HikariDataSource.class)
    static class HikariPoolDataSourceMetadataProviderConfiguration {
        @Bean
        public DataSourcePoolMetadataProvider hikariPoolDataSourceMetadataProvider() {
            return (dataSource) -> {
                HikariDataSource hikariDataSource =
DataSourceUnwrapper.unwrap(dataSource,
```

```
                        HikariDataSource.class);
                if (hikariDataSource != null) {
                    //这里就返回了一个HikariDataSourcePoolMetadata实例，算是代理数据源吧
                    return new HikariDataSourcePoolMetadata(hikariDataSource);
                }
                return null;
            };
        }
    }
    @Configuration
    @ConditionalOnClass(BasicDataSource.class)
    static class CommonsDbcp2PoolDataSourceMetadataProviderConfiguration {...}
}
//函数式接口，可以这样声明
//DataSourcePoolMetadataProvider dpmp = dataSource->{...,return 一个
DataSourcePoolMetadata实例}
@FunctionalInterface
public interface DataSourcePoolMetadataProvider {
    //返回能够管理指定数据源的DataSourcePoolMetadata实例，如果无法处理给定的数据源，则返回空值。
    DataSourcePoolMetadata getDataSourcePoolMetadata(DataSource dataSource);
}
//每个都可能返回null，如果数据源没有提供相应的信息
public interface DataSourcePoolMetadata {
    //返回池的使用情况，值介于0和1之间(如果池不受限制，则返回-1)，1表示所有的都分配了
    Float getUsage();
    //返回从数据源分配的当前活动连接数
    Integer getActive();
    //返回可同时分配的最大活动连接数，无限制的话返回-1
    Integer getMax();
    //返回池中空闲连接的最小数目
    Integer getMin();
    //返回用于验证连接是否有效的查询
    String getValidationQuery();
    //此池创建的连接的默认自动提交状态
    Boolean getDefaultAutoCommit();
}
```

3 如何自定义数据源

- 每个数据源都有自己的属性，
- 通过 DataSourceBuilder 类来创建数据源实例
- 通过 `@ConfigurationProperties()` 注解，将配置文件中的属性，映射到相应的数据源实例中
- DataSourcePoolMetadataProvidersConfiguration 中只配置了默认的3个数据源，如果需要，需要自己定义它

```
@Bean(name = "druidDatasource")
@ConfigurationProperties("wang-kang.druid.datasource")
public DataSource druidDatasource() {
    return DataSourceBuilder.create().type(DruidDataSource.class).build();
}
```

4 参考

Spring-boot-AutoConfiguration-2.1.3