服务器出问题，目前部分恢复

05 配置驱动：ShardingSphere 中的配置体系是如何设计的?

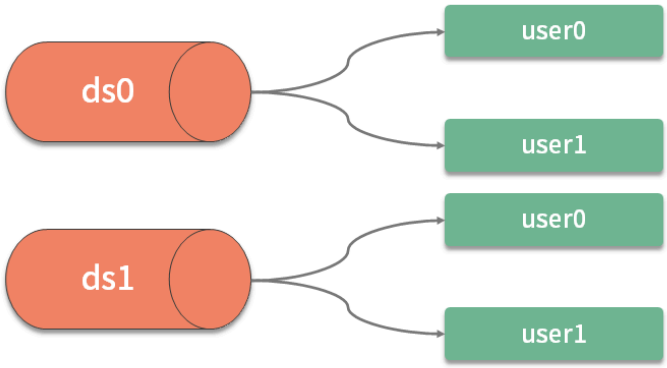
在上一课时中，我们介绍了在业务系统中应用 ShardingSphere 的几种方法。事实上，我们发现，除了掌握 Spring、Spring Boot、Mybatis 等常见框架的功能特性之外，使用 ShardingSphere 的主要工作在于根据业务需求完成各种分片操作相关配置项的设置。今天，我就来带领你剖析 ShardingSphere 中的配置体系到底是如何进行设计并实现的，这也是我们介绍 ShardingSphere 核心功能的前提。

什么是行表达式?

在引入配置体系的学习之前，我们先来介绍 ShardingSphere 框架为开发人员提供的一个辅助功能，这个功能就是行表达式。

行表达式是 ShardingSphere 中用于实现简化和统一配置信息的一种工具，在日常开发过程中应用得非常广泛。 它的使用方式非常直观，只需要在配置中使用 \({expression} 或 \)-> {expression} 表达式即可。

例如上一课时中使用的”ds\({0..1}.user\){0..1}“就是一个行表达式，用来设置可用的数据源或数据表名称。基于行表达式语法，\({begin..end} 表示的是一个从"begin"到"end"的范围区间，而多个 \){expression} 之间可以用”.“符号进行连接，代表多个表达式数值之间的一种笛卡尔积关系。这样，如果采用图形化的表现形式，”ds\({0..1}.user\){0..1}“表达式最终会解析成这样一种结果：



行表达式ds${0..1}.user${0..1}的解析效果图

当然，类似场景也可以使用枚举的方式来列举所有可能值。行表达式也提供了 \({[enum1, enum2,…, enumx]} 语法来表示枚举值，所以"ds\){0..1}.user\({0..1}"的效果等同于"ds\){[0,1]}.user${[0,1]}“。

同样，在上一课时中使用到的 ds${age % 2} 表达式，它表示根据 age 字段进行对 2 取模，从而自动计算目标数据源是 ds0 还是 ds1。所以，除了配置数据源和数据表名称之外，行表达式在 ShardingSphere 中另一个常见的应用场景就是配置各种分片算法，我们会在后续的示例中大量看到这种使用方法。

由于 \({expression} 与 Spring 本身的属性文件占位符冲突，而 Spring 又是目前主流的开发框架，因此在正式环境中建议你使用 \)->{expression} 来进行配置。

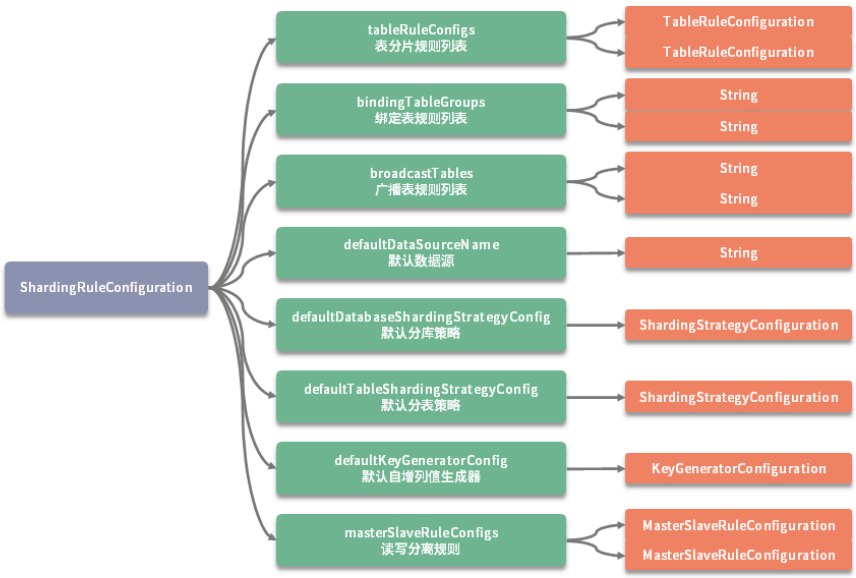
ShardingSphere 有哪些核心配置?

对于分库分表、读写分离操作而言，配置的主要任务是完成各种规则的创建和初始化。配置是整个 ShardingSphere 的核心，也是我们在日常开发过程中的主要工作。可以说，只要我们掌握了 ShardingSphere 的核心配置项，就相当于掌握了这个框架的使用方法。那么，ShardingSphere 有哪些核心配置呢？这里以分片引擎为例介绍最常用的几个配置项，而与读写分离、数据脱敏、编排治理相关的配置项我们会在介绍具体的应用场景时再做展开。

ShardingRuleConfiguration

我们在上一课时中已经了解了如何通过框架之间的集成方法来创建一个 DataSource，这个 DataSource 就是我们使用 ShardingSphere 的入口。我们也看到在创建 DataSource 的过程中使用到了一个 ShardingDataSourceFactory 类，这个工厂类的构造函数中需要传入一个 ShardingRuleConfiguration 对象。显然，从命名上看，这个 ShardingRuleConfiguration 就是用于分片规则的配置入口。

ShardingRuleConfiguration 中所需要配置的规则比较多，我们可以通过一张图例来进行简单说明，在这张图中，我们列举了每个配置项的名称、类型以及个数关系：



ShardingRuleConfiguration 中的配置项列表

这里引入了一些新的概念，包括绑定表、广播表等，这些概念在下一课时介绍到 ShardingSphere 的分库分表操作时都会详细展开，这里不做具体介绍。**事实上，对于 ShardingRuleConfiguration 而言，必须要设置的只有一个配置项，即 TableRuleConfiguration。**

TableRuleConfiguration

从命名上看，TableRuleConfiguration 是表分片规则配置，但事实上，这个类同时包含了对分库和分表两种场景的设置。TableRuleConfiguration 包含很多重要的配置项：

- actualDataNodes

actualDataNodes 代表真实的数据节点，由数据源名+表名组成，支持行表达式。例如，前面介绍的”ds\({0..1}.user\){0..1}“就是比较典型的一种配置方式。

- databaseShardingStrategyConfig

databaseShardingStrategyConfig 代表分库策略，如果不设置则使用默认分库策略，这里的默认分库策略就是 ShardingRuleConfiguration 中的 defaultDatabaseShardingStrategyConfig 配置。

- tableShardingStrategyConfig

和 databaseShardingStrategyConfig 一样，tableShardingStrategyConfig 代表分表策略，如果不设置也会使用默认分表策略，这里的默认分表策略同样来自 ShardingRuleConfiguration

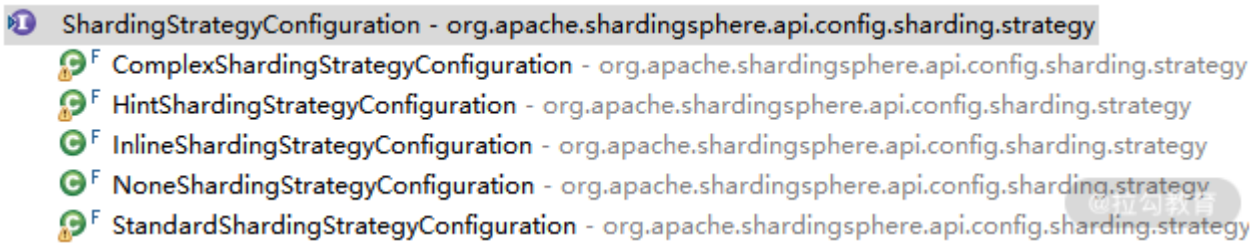
中的 defaultTableShardingStrategyConfig 配置。

- keyGeneratorConfig

keyGeneratorConfig 代表分布式环境下的自增列生成器配置，ShardingSphere 中集成了雪花算法等分布式 ID 的生成器实现。

ShardingStrategyConfiguration

我们注意到，databaseShardingStrategyConfig 和 tableShardingStrategyConfig 的类型都是一个 ShardingStrategyConfiguration 对象。在 ShardingSphere 中，ShardingStrategyConfiguration 实际上是一个空接口，存在一系列的实现类，其中的每个实现类都代表一种分片策略：



ShardingStrategyConfiguration 的类层结构图

在这些具体的分片策略中，通常需要指定一个分片列 shardingColumn 以及一个或多个分片算法 ShardingAlgorithm。当然也有例外，例如 HintShardingStrategyConfiguration 直接使用数据库的 Hint 机制实现强制路由，所以不需要分片列。我们会在《路由引擎：如何在路由过程中集成多种分片策略和分片算法？》中对这些策略的实现过程做详细的剖析。

KeyGeneratorConfiguration

可以想象，对于一个自增列而言，KeyGeneratorConfiguration 中首先需要指定一个列名 column。同时，因为 ShardingSphere 中内置了一批自增列的实现机制（例如雪花算法 SNOWFLAKE 以及通用唯一识别码 UUID），所以需要通过一个 type 配置项进行指定。最后，我们可以利用 Properties 配置项来指定自增值生成过程中所需要的相关属性配置。关于这一点，我们在上一课时中也看到了示例，即雪花算法中配置 workerId 为 33。

基于以上核心配置项，我们已经可以完成日常开发过程中常见的分库分表操作。当然，对于不同的开发人员，如何采用某一个特定的方式将这些配置项信息集成到业务代码中，也存在着不同的诉求。因此，ShardingSphere 中也提供了一系列的配置方式供开发人员进行选择。

ShardingSphere 提供了哪些配置方式?

从 Java 代码到配置文件，ShardingSphere 提供了 4 种配置方式，用于不同的使用场景，分别是：

- Java 代码配置

- Yaml 配置
- Spring 命名空间配置
- Spring Boot配置

我们来看一下这四种配置的具体方法。

Java 代码配置

Java 代码配置是使用 ShardingSphere 所提供的底层 API 来完成配置系统构建的原始方式。在上一课时中，我们已经看到了如何初始化 ShardingRuleConfiguration 和 TableRuleConfiguration 类，并通过 ShardingDataSourceFactory 创建目标 DataSource 的具体方式，这里不再展开。

在日常开发中，我们一般不会直接使用 Java 代码来完成 ShardingSphere 配置体系的构建。一方面，如果使用 Java 代码来实现配置，一旦有变动就需要重新编译代码并发布，不利于实现配置信息的动态化管理和系统的持续集成。另一方面，代码级别的配置方式也比较繁琐，不够直接且容易出错，维护性也不好。

当然，也可能有例外情况。一种情况是，如果我们需要和其他框架进行更加底层的集成或定制化开发时，往往只能采用 Java 代码才能达到理想的效果。同时，对于刚接触 ShardingSphere 的开发人员而言，基于框架提供的 API 进行开发更加有利于快速掌握框架提供的各种类之间的关联关系和类层结构。

Yaml 配置

Yaml 配置是 ShardingSphere 所推崇的一种配置方式。Yaml 的语法和其他高级语言类似，并且可以非常直观地描述多层列表和对象等数据形态，特别适合用来表示或编辑数据结构和各种配置文件。

在语法上，常见的”!!“表示实例化该类；以”-“开头的多行构成一个数组；以”:“表示键值对；以”#“表示注释。关于 Yaml 语法的更多介绍可以参考百度百科 https://baike.baidu.com/item/YAML。**请注意，Yaml 大小写敏感，并使用缩进表示层级关系。** 这里给出一个基于 ShardingSphere 实现读写分离场景下的配置案例：

```
dataSources:
  dsmaster: !!com.alibaba.druid.pool.DruidDataSource
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://119.3.52.175:3306/dsmaster
    username: root
    password: root
  dsslave0: !!com.alibaba.druid.pool.DruidDataSource
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://119.3.52.175:3306/dsslave0
    username: root
    password: root
  dsslave1: !!com.alibaba.druid.pool.DruidDataSource
    driverClassName: com.mysql.jdbc.Driver
```

```
    url: jdbc:mysql://119.3.52.175:3306/dsslave1
    username: root
    password: root
masterSlaveRule:
  name: health_ms
  masterDataSourceName: dsmaster
  slaveDataSourceNames: [dsslave0, dsslave1]
```

可以看到，这里配置了 dsmaster、dsslave0 和 dsslave1 这三个 DataSource，然后针对每个 DataSource 分别设置了它们的驱动信息。最后，基于这三个 DataSource 配置了一个 masterSlaveRule 规则，用于指定具体的主从架构。

在 ShardingSphere 中，我们可以把配置信息存放在一个 .yaml 配置文件中，并通过加载这个配置文件来完成配置信息的解析。这种机制为开发人员高效管理配置信息提供了更多的灵活性和可定制性。在今天内容的最后，我们会详细剖析这一机制的实现原理。

Spring 命名空间配置

我们可以通过自定义配置标签实现方案来扩展 Spring 的命名空间，从而在 Spring 中嵌入各种自定义的配置项。Spring 框架从 2.0 版本开始提供了基于 XML Schema 的风格来定义 Javabean 的扩展机制。通过 XML Schema 的定义，把一些原本需要通过复杂的 Javabean 组合定义的配置形式，用一种更加简单而可读的方式呈现出来。基于 Scheme 的 XML 由三部分构成，我们用一个示例说明：

```
<master-slave:load-balance-algorithm id="randomStrategy"/>
```

这段 XML 中，master-slave 是命名空间，从这个命名空间中可以明确地区分出所属的逻辑分类是用于实现读写分离；load-balance-algorithm 是一种元素，代表用于设置读写分离中的负载均衡算法；而 ID 就是负载均衡下的一个配置选项，它的值为随机策略 randomStrategy。

在 ShardingSphere 中，我们同样可以基于命名空间来实现完整的读写分离配置：

```
<beans
    ...
    http://shardingsphere.apache.org/schema/shardingsphere/masterslave
    http://shardingsphere.apache.org/schema/shardingsphere/masterslave/master
        <bean id=" dsmaster " class=" com.alibaba.druid.pool.DruidDataSource"
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/dsmaster"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>

    <bean id="dsslave0" class="com.alibaba.druid.pool.DruidDataSource" destro
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/dsslave0"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>
```

```
    <bean id="dsslave1" class="com.alibaba.druid.pool.DruidDataSource" destroy
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/dsslave1"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>
    <master-slave:load-balance-algorithm id="randomStrategy" type="RANDOM" />
    <master-slave:data-source id="masterSlaveDataSource" master-data-source-n

</beans>
```

在这段代码中，我们在 Spring 中引入了 master-slave 这个新的命名空间，并完成了负载均衡算法和三个主从 DataSource 的设置。

Spring Boot 配置

Spring Boot 已经成为 Java 领域最流行的开发框架，提供了约定优于配置的设计理念。通常，开发人员可以把配置项放在 application.properties 文件中。同时，为了便于对配置信息进行管理和维护，Spring Boot 也提供了 profile 的概念，可以基于 profile 来灵活组织面对不同环境或应用场景的配置信息。在采用 profile 时，配置文件的命名方式有一定的约定：

```
{application}-{profile}.properties
```

基于这种命名约定，如果我们根据面向的是传统的单库单表场景，还是主从架构的读写分离场景进行命名，就需要分别提供两个不同的 .properties 配置文件，如下面的代码所示：

```
application-traditional.properties
application-master-slave.properties
```

这两个文件名中的 traditional 和 master-slave 就是具体的 profile，现在在 application.properties 文件中就可以使用 spring.profiles.active 配置项来设置当前所使用的 profile：

```
#spring.profiles.active=traditional
spring.profiles.active=master-slave
```

基于 Spring Boot 的配置风格就是一组键值对，我们同样可以采用这种方式来实现前面介绍的读写分离配置：

```
spring.shardingsphere.datasource.names=dsmaster,dsslave0,dsslave1
spring.shardingsphere.datasource.dsmaster.type=com.alibaba.druid.pool.DruidDa
spring.shardingsphere.datasource.dsmaster.driver-class-name=com.mysql.jdbc.Dr
spring.shardingsphere.datasource.dsmaster.url=jdbc:mysql://localhost:3306/dsm
spring.shardingsphere.datasource.dsmaster.username=root
spring.shardingsphere.datasource.dsmaster.password=root
spring.shardingsphere.datasource.dsslave0.type=com.alibaba.druid.pool.DruidDa
```

```
spring.shardingsphere.datasource.dsslave0.driver-class-name=com.mysql.jdbc.Dr
spring.shardingsphere.datasource.dsslave0.url=jdbc:mysql://localhost:3306/dss
spring.shardingsphere.datasource.dsslave0.username=root
spring.shardingsphere.datasource.dsslave0.password=root
spring.shardingsphere.datasource.dsslave1.type=com.alibaba.druid.pool.DruidDa
spring.shardingsphere.datasource.dsslave1.driver-class-name=com.mysql.jdbc.Dr
spring.shardingsphere.datasource.dsslave1.url=jdbc:mysql://localhost:3306/dss
spring.shardingsphere.datasource.dsslave1.username=root
spring.shardingsphere.datasource.dsslave1.password=root
spring.shardingsphere.masterslave.load-balance-algorithm-type=random
spring.shardingsphere.masterslave.name=health_ms
spring.shardingsphere.masterslave.master-data-source-name=dsmaster
spring.shardingsphere.masterslave.slave-data-source-names=dsslave0,dsslave1
```

通过这些不同的配置方式，开发人员可以基于自己擅长的或开发框架所要求的方式，灵活完成各项配置工作。在本课程中的后续内容中，我们会组合使用 Yaml 配置和 Spring Boot 配置这两种配置方式来介绍 ShardingSphere 的具体使用方式。

ShardingSphere 的配置体系是如何实现的?

尽管在日常开发过程中很少使用，但在前面介绍的四种配置方式中，Java 代码配置的实现方式最容易理解，我们可以通过各个配置类的调用关系来梳理 ShardingSphere 提供的配置功能。所以，为了深入理解配置体系的实现原理，我们还是选择从 ShardingRuleConfiguration 类进行切入。

ShardingRuleConfiguration 配置体系

对于 ShardingSphere 而言，配置体系的作用本质上就是用来初始化 DataSource 等 JDBC 对象。例如，ShardingDataSourceFactory 就是基于传入的数据源 Map、ShardingRuleConfiguration 以及 Properties 来创建一个 ShardingDataSource 对象：

```
public final class ShardingDataSourceFactory {

    public static DataSource createDataSource(
            final Map<String, DataSource> dataSourceMap, final ShardingRuleCo
        return new ShardingDataSource(dataSourceMap, new ShardingRule(shardin
        }
}
```

在 ShardingSphere 中，所有规则配置类都实现了一个顶层接口 RuleConfiguration。RuleConfiguration 是一个空接口，ShardingRuleConfiguration 就是这个接口的实现类之一，专门用来处理分片引擎的应用场景。下面这段代码就是 ShardingRuleConfiguration 类的实现过程：

```
public final class ShardingRuleConfiguration implements RuleConfiguration {
    //表分片规则列表
    private Collection<TableRuleConfiguration> tableRuleConfigs = new LinkedL
```

```
    //绑定表规则列表
    private Collection<String> bindingTableGroups = new LinkedList<>();
    //广播表规则列表
    private Collection<String> broadcastTables = new LinkedList<>();
    //默认数据源
    private String defaultDataSourceName;
    //默认分库策略
    private ShardingStrategyConfiguration defaultDatabaseShardingStrategyConf
    //默认分表策略
    private ShardingStrategyConfiguration defaultTableShardingStrategyConfig;
    //默认自增列值生成器
    private KeyGeneratorConfiguration defaultKeyGeneratorConfig;
    //读写分离规则
    private Collection<MasterSlaveRuleConfiguration> masterSlaveRuleConfigs =
    //数据脱敏规则
    private EncryptRuleConfiguration encryptRuleConfig;
}
```

可以看到，ShardingRuleConfiguration 中包含的就是一系列的配置类定义，通过前面的内容介绍，我们已经明白了这些配置类的作用和使用方法。其中，核心的 TableRuleConfiguration 定义也比较简单，主要包含了逻辑表、真实数据节点以及分库策略和分表策略的定义：

```
public final class TableRuleConfiguration {
    //逻辑表
    private final String logicTable;
    //真实数据节点
    private final String actualDataNodes;
    //分库策略
    private ShardingStrategyConfiguration databaseShardingStrategyConfig;
    //分表策略
    private ShardingStrategyConfiguration tableShardingStrategyConfig;
    //自增列生成器
    private KeyGeneratorConfiguration keyGeneratorConfig;

    public TableRuleConfiguration(final String logicTable) {
        this(logicTable, null);
    }

    public TableRuleConfiguration(final String logicTable, final String actua
        Preconditions.checkArgument(!Strings.isNullOrEmpty(logicTable), "Logi
        this.logicTable = logicTable;
        this.actualDataNodes = actualDataNodes;
    }
}
```

因为篇幅有限，我们不对其他配置类的定义做具体展开。事实上，无论采用哪种配置方式，所有的配置项都是在这些核心配置类的基础之上进行封装和转换。基于 Spring 命名空间和 Spring Boot 配置的使用方式比较常见，这两种方式的实现原理都依赖于 ShardingSphere 与这两个框架的集成方式，我会在后续课时中做详细展开。而 Yaml 配置是 ShardingSphere 非常推崇的一种使用方式，为此，ShardingSphere 在内部对 Yaml 配置的应用场景有专门的处理。今天，我们就来详细解析一下针对 Yaml 配置的完整实现方案。

YamlShardingRuleConfiguration 配置体系

在 ShardingSphere 源码的 sharding-core-common 工程中，存在一个包结构 org.apache.shardingsphere.core.yaml.config，在这个包结构下包含着所有与 Yaml 配置相关的实现类。

与 RuleConfiguration 一样，ShardingSphere 同样提供了一个空的 YamlConfiguration 接口。这个接口的实现类非常多，但我们发现其中包含了唯一的一个抽象类 YamlRootRuleConfiguration，显然，这个类是 Yaml 配置体系中的基础类。 在这个 YamlRootRuleConfiguration 中，包含着数据源 Map 和 Properties：

```
public abstract class YamlRootRuleConfiguration implements YamlConfiguration

    private Map<String, DataSource> dataSources = new HashMap<>();

    private Properties props = new Properties();
}
```

在上面这段代码中，我们发现少了 ShardingRuleConfiguration 的对应类，其实，这个类的定义在 YamlRootRuleConfiguration 的子类 YamlRootShardingConfiguration 中，它的类名 YamlShardingRuleConfiguration 就是在 ShardingRuleConfiguration 上加了一个 Yaml 前缀，如下面这段代码所示：

```
public class YamlRootShardingConfiguration extends YamlRootRuleConfiguration

    private YamlShardingRuleConfiguration shardingRule;
}
```

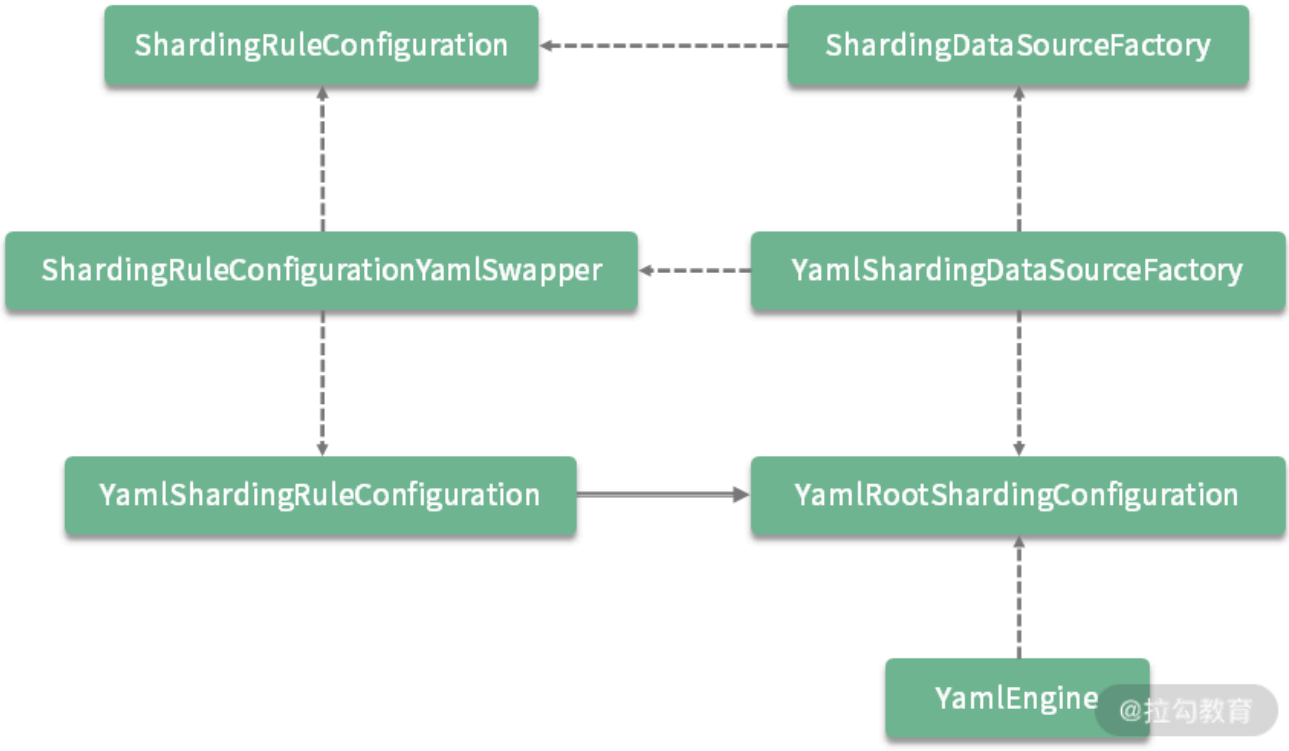
接下来，我们来到 YamlShardingRuleConfiguration 类，发现它所包含的变量与 ShardingRuleConfiguration 类中的变量存在一致对应关系，这些 Yaml 配置类都位于 org.apache.shardingsphere.core.yaml.config.sharding 包中：

```
public class YamlShardingRuleConfiguration implements YamlConfiguration {

    private Map<String, YamlTableRuleConfiguration> tables = new LinkedHashMa
    private Collection<String> bindingTables = new ArrayList<>();
    private Collection<String> broadcastTables = new ArrayList<>();
    private String defaultDataSourceName;
    private YamlShardingStrategyConfiguration defaultDatabaseStrategy;
    private YamlShardingStrategyConfiguration defaultTableStrategy;
    private YamlKeyGeneratorConfiguration defaultKeyGenerator;
    private Map<String, YamlMasterSlaveRuleConfiguration> masterSlaveRules =
    private YamlEncryptRuleConfiguration encryptRule;
}
```

那么这个 YamlShardingRuleConfiguration 是怎么构建出来的呢？这就要来到 YamlShardingDataSourceFactory 工厂类，这个工厂类实际上是对 ShardingDataSourceFactory 类的进一步封装，下面这段代码就演示了这一过程：

```
public final class YamlShardingDataSourceFactory {

    public static DataSource createDataSource(final File yamlFile) throws SQL
        YamlRootShardingConfiguration config = YamlEngine.unmarshal(yamlFile,
        return ShardingDataSourceFactory.createDataSource(config.getDataSourc
        }
        …
}
```

可以看到 createDataSource 方法的输入参数是一个 File 对象，我们通过这个 File 对象构建出 YamlRootShardingConfiguration 对象，然后再通过 YamlRootShardingConfiguration 对象获取了 ShardingRuleConfiguration 对象，并交由 ShardingDataSourceFactory 完成目标 DataSource 的构建。这里的调用关系有点复杂，我们来梳理整个过程的类层结构，如下图所示：



显然，这里引入了两个新的工具类，YamlEngine 和 YamlSwapper。我们来看一下它们在整个流程中起到的作用。

YamlEngine 和 YamlSwapper

YamlEngine 的作用是将各种形式的输入内容转换成一个 Yaml 对象，这些输入形式包括 File、字符串、byte[] 等。YamlEngine 包含了一批 unmarshal/marshal 方法来完成数据的转换。以 File 输入为例，unmarshal 方法通过加载 FileInputStream 来完成 Yaml 对象的构建：

```
    public static <T extends YamlConfiguration> T unmarshal(final File yamlFi
        try (
                FileInputStream fileInputStream = new FileInputStream(yamlFil
                InputStreamReader inputStreamReader = new InputStreamReader(f
        ) {
            return new Yaml(new Constructor(classType)).loadAs(inputStreamRea
        }
    }
```

当在 unmarshal 方法中传入想要的 classType 时，我们就可以获取这个 classType 对应的实例。在 YamlShardingDataSourceFactory 中我们传入了 YamlRootShardingConfiguration 类型，这样我们就将得到一个 YamlRootShardingConfiguration 的类实例 YamlShardingRuleConfiguration。

在得到 YamlShardingRuleConfiguration 之后，下一步需要实现将 YamlShardingRuleConfiguration 转换为 ShardingRuleConfiguration。为了完成这种具有对应关系的类地转换，ShardingSphere 还专门提供了一批转换器类，ShardingRuleConfigurationYamlSwapper 就是其中之一。ShardingRuleConfigurationYamlSwapper 实现了 YamlSwapper 接口：

```
public interface YamlSwapper<Y extends YamlConfiguration, T> {
    Y swap(T data);
    T swap(Y yamlConfiguration);
}
```

可以看到这里提供了一对方法完成两种数据结构之间的相互转换，ShardingRuleConfigurationYamlSwapper 中对这两个方法的实现过程也比较直接。以目标对象为 ShardingRuleConfiguration 的 swap 方法为例，代码结构基本上就是完成了 YamlShardingRuleConfiguration 与 ShardingRuleConfiguration 中对应字段的一对一转换：

```
        @Override
    public ShardingRuleConfiguration swap(final YamlShardingRuleConfiguration
        ShardingRuleConfiguration result = new ShardingRuleConfiguration();
        for (Entry<String, YamlTableRuleConfiguration> entry : yamlConfigurat
            YamlTableRuleConfiguration tableRuleConfig = entry.getValue();
            tableRuleConfig.setLogicTable(entry.getKey());
            result.getTableRuleConfigs().add(tableRuleConfigurationYamlSwappe
        }
        result.setDefaultDataSourceName(yamlConfiguration.getDefaultDataSourc
        result.getBindingTableGroups().addAll(yamlConfiguration.getBindingTab
        result.getBroadcastTables().addAll(yamlConfiguration.getBroadcastTabl
        if (null != yamlConfiguration.getDefaultDatabaseStrategy()) {
            result.setDefaultDatabaseShardingStrategyConfig(shardingStrategyC
        }
        if (null != yamlConfiguration.getDefaultTableStrategy()) {
            result.setDefaultTableShardingStrategyConfig(shardingStrategyConf
        }
        if (null != yamlConfiguration.getDefaultKeyGenerator()) {
            result.setDefaultKeyGeneratorConfig(keyGeneratorConfigurationYaml
        }
        Collection<MasterSlaveRuleConfiguration> masterSlaveRuleConfigs = new
```

```
        for (Entry<String, YamlMasterSlaveRuleConfiguration> entry : yamlConf
            YamlMasterSlaveRuleConfiguration each = entry.getValue();
            each.setName(entry.getKey());
            masterSlaveRuleConfigs.add(masterSlaveRuleConfigurationYamlSwappe
        }
        result.setMasterSlaveRuleConfigs(masterSlaveRuleConfigs);
        if (null != yamlConfiguration.getEncryptRule()) {
            result.setEncryptRuleConfig(encryptRuleConfigurationYamlSwapper.s
        }
        return result;
    }
```

这样，我们就从外部的 Yaml 文件中获取了一个 ShardingRuleConfiguration 对象，然后可以使用 ShardingDataSourceFactory 工厂类完成目标 DataSource 的创建过程。

小结

承接上一课时的内容，本课时我们对 ShardingSphere 中的配置体系进行了全面的介绍。事实上，在使用这个框架时，配置是开发人员最主要的工作内容。我们对 ShardingSphere 的核心配置项进行了梳理，然后给出了具体的四种配置方式，分别是 Java 代码配置、Yaml 配置、Spring 命名空间配置以及 Spring Boot 配置。最后，从实现原理上，我们也对 Yaml 配置这一特定的配置方式进行了深入的剖析。

这里给你留一道思考题：在 ShardingSphere中，配置体系相关的核心类之间存在什么样的关联关系?

从下一课时开始，我们就将基于 ShardingSphere 提供的配置体系，来逐步完成分库分表、读写分离以及分库分表+读写分离等操作的开发工作。

上一页 下一页