

二

服务器出问题，目前部分恢复

32 注册中心：如何基于注册中心实现数据库访问熔断机制？

上一课时我们讨论了 ShardingSphere 中关于配置中心的相关内容。今天我们继续讨论编排治理模块的另一个核心功能，即注册中心。相较配置中心，注册中心在 ShardingSphere 中的应用更为广泛。

ShardingSphere 中的注册中心实现类

与配置中心一样，ShardingSphere 中的注册中心在代码结构上也包含三个独立的工程，即代表抽象接口的 API 工程，以及两个具体的实现 nacos 和 zookeeper-curator 工程。可以看到，这里同样使用上一课时中介绍的 Zookeeper 作为注册中心的一种实现方式，而另一种实现方式就是基于阿里巴巴的 Nacos。

我们先来看 ShardingSphere 中对注册中心的抽象，即如下所示的 RegistryCenter 接口：

```
public interface RegistryCenter extends TypeBasedSPI {  
    //根据配置信息初始化注册中心  
  
    void init(RegistryCenterConfiguration config);  
  
    //获取数据  
  
    String get(String key);  
  
    //直接获取数据  
  
    String getDirectly(String key);  
  
    //是否存在数据项  
  
    boolean isExisted(String key);  
  
    //获取子数据项列表  
  
    List<String> getChildrenKeys(String key);  
  
    //持久化数据项  
  
    void persist(String key, String value);  
  
    //更新数据项  
  
    void update(String key, String value);  
}
```

```
//持久化临时数据

void persistEphemeral(String key, String value);

//对数据项或路径进行监听

void watch(String key, DataChangedEventListener dataChangedEventListener)

//关闭注册中心

void close();

//对数据项初始化锁

void initLock(String key);

//对数据项获取锁

boolean tryLock();

//对数据项释放锁

void tryRelease();

}
```

我们发现，除了最后几个关于锁处理的方法，RegistryCenter 实际上与上一课时中介绍的 ConfigCenter 非常类似。从这点上，我们就不难想象为什么 Zookeeper 既可以用来做配置中心，也可以是实现注册中心的一种典型方案。沿着这个思路，我们就先来看一下 CuratorZookeeperRegistryCenter 这个基于 Zookeeper 的注册中心实现类。

1. CuratorZookeeperRegistryCenter

我们快速浏览整个 CuratorZookeeperRegistryCenter 类，发现通用接口方法的实现过程也与 CuratorZookeeperConfigCenter 中的完全一致。而对于新增的与锁相关的几个方法，实现方式也很简单，直接使用 Curator 所封装的 InterProcessMutex 即可，如下所示：

```
private InterProcessMutex leafLock;

@Override

public void initLock(final String key) {

    leafLock = new InterProcessMutex(client, key);

}

@Override

@SneakyThrows

public boolean tryLock() {
```

```
        return leafLock.acquire(5, TimeUnit.SECONDS);  
    }  
  
    @Override  
    @SneakyThrows  
    public void tryRelease() {  
        leafLock.release();  
    }
```

关于 CuratorZookeeperRegistryCenter 我们就介绍到这里，接下来我们来看注册中心的另一个实现类 NacosRegistryCenter。

2.NacosRegistryCenter

Nacos 框架同样提供了一个名为 ConfigService 的客户端组件，用于获取数据的 get、getDirectly，以及 isExisted 方法实际上都是使用了 ConfigService 的 getConfig 方法进行实现，我们也无须对其做过多的讨论。

NacosRegistryCenter 的 persist 方法实际上就是调用了它的 update 方法，而后者又基于 ConfigService 的 publishConfig 方法实现数据的更新，如下所示：

```
@Override  
public void persist(final String key, final String value) {  
    update(key, value);  
}  
  
@Override  
public void update(final String key, final String value) {  
    try {  
        String dataId = key.replace("/", ".");  
        String group = properties.getProperty("group", "SHARDING_SPHERE_D  
        configService.publishConfig(dataId, group, value);  
    } catch (final NacosException ex) {  
        log.debug("exception for: {}", ex.toString());  
    }
```

```
}
```

与 Zookeeper 不同，对于 Nacos 而言，getChildrenKeys、persistEphemeral、close、initLock、tryLock 和 tryRelease 方法都是无法实现或无须实现的。而对于 watch 方法，ConfigService 也提供了 addListener 方法完成监听器的使用，同样也是基于上一课时中介绍的 DataChangedEventListener 类完成事件的处理，如下所示：

```
@Override

public void watch(final String key, final DataChangedEventListener dataChange

    try {

        String dataId = key.replace("/", ".");

        String group = properties.getProperty("group", "SHARDING_SPHERE_D

        configService.addListener(dataId, group, new Listener() {

            @Override

            public Executor getExecutor() {

                return null;

            }

            @Override

            public void receiveConfigInfo(final String configInfo) {

                dataChangedEventListener.onChange(new DataChangedEvent(ke

            }

        });

    } catch (final NacosException ex) {

        log.debug("exception for: {}", ex.toString());

    }

}
```

至此，关于 ShardingSphere 中注册中心的两种实现方式我们也介绍完毕。请注意，注册中心本身只是一个工具，关键是看我们如何对其进行使用。

通过注册中心构建编排治理服务

让我们来到 sharding-orchestration-core 工程，找到 RegistryCenterServiceLoader，显然，这个类用于加载 RegistryCenter 的 SPI 实例。我们在该类的 load 方法中通过 SPI 机制创建了 RegistryCenter 实例并调用了它的 init 方法，如下所示：

```
public RegistryCenter load(final RegistryCenterConfiguration regCenterConfig)
    Preconditions.checkNotNull(regCenterConfig, "Registry center configur
    RegistryCenter result = newService(regCenterConfig.getType(), regCent
    result.init(regCenterConfig);
    return result;
}
```

然后我们跟踪代码的调用过程，发现使用 RegistryCenterServiceLoader 类的入口是在同一个包中的 ShardingOrchestrationFacade 类。这个类代码不多，但引出了很多新的类和概念。我们先来看一下它的变量定义：

```
//注册中心
private final RegistryCenter regCenter;

//配置服务
private final ConfigurationService configService;

//状态服务
private final StateService stateService;

//监听管理器
private final ShardingOrchestrationListenerManager listenerManager;
```

我们先来关注 ConfigurationService 这个新类，该类实际上是构建在 RegistryCenter 之上。

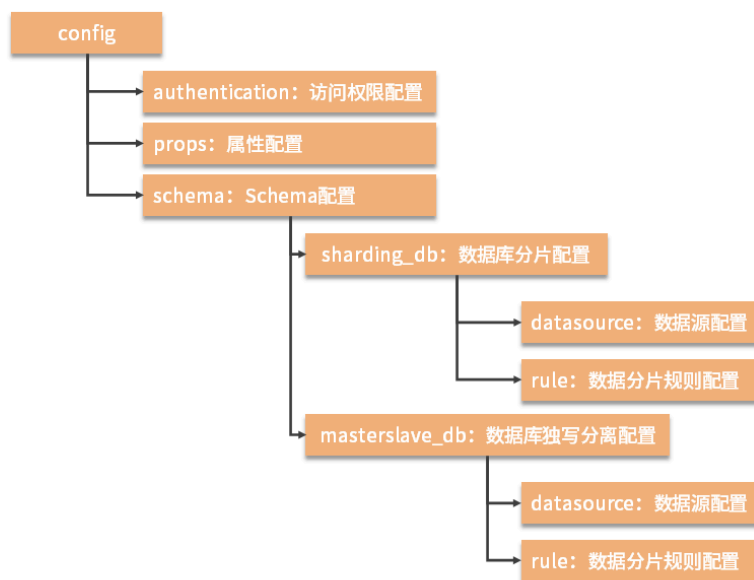
1.ConfigurationService

ConfigurationService 类对外提供了管理各种配置信息的入口。在该类中，除了保存着 RegistryCenter 之外，还存在一个 ConfigurationNode 类，该类定义了保存在注册中心中各种数据的配置项以及管理这些配置项的工具方法，具体的配置项如下所示：

```
private static final String ROOT = "config";
private static final String SCHEMA_NODE = "schema";
private static final String DATA_SOURCE_NODE = "datasource";
private static final String RULE_NODE = "rule";
```

```
private static final String AUTHENTICATION_NODE = "authentication";
private static final String PROPS_NODE = "props";
private final String name;
```

基于 ShardingSphere 中对这些配置项的管理方式，我们可以将这些配置项与具体的存储结构相对应，如下所示：



配置项存储结构示意图

有了配置项之后，我们就需要对其进行保存，ConfigurationService 的 persistConfiguration 方法完成了这一目的，如下所示：

```
public void persistConfiguration(final String shardingSchemaName, final Map<String, String> props,
                                final Authentication authentication, final boolean isOverwrite) {
    persistDataSourceConfiguration(shardingSchemaName, dataSourceConfigs, isOverwrite);
    persistRuleConfiguration(shardingSchemaName, ruleConfig, isOverwrite);
    persistAuthentication(authentication, isOverwrite);
    persistProperties(props, isOverwrite);
}
```

这里列举了四个 persist 方法，分别用于保存 DataSource、Rule、Authentication 以及 Properties。我们以 persistDataSourceConfiguration 方法为例来看它的实现过程：

```

private void persistDataSourceConfiguration(final String shardingSchemaName,

//判断是否覆盖现有配置

if (isOverwrite || !hasDataSourceConfiguration(shardingSchemaName))

    Preconditions.checkNotNull(dataSourceConfigurations, "dataSourceConfigurations cannot be null")

//构建 YamlDataSourceConfiguration

Map<String, YamlDataSourceConfiguration> yamlDataSourceConfigurations =
    new Function<DataSourceConfiguration, YamlDataSourceConfiguration>() {
        @Override
        public YamlDataSourceConfiguration apply(DataSourceConfiguration dataSourceConfiguration) {
            return new DataSourceConfigurationYamlSwapper().swap(dataSourceConfiguration);
        }
    };

//通过注册中心进行持久化

regCenter.persist(configNode.getDataSourcePath(shardingSchemaName, shardingSchemaName));
}
}

```

可以看到这里使用了 Guava 框架中的 `Maps.transformValues` 工具方法将输入的 `DataSourceConfiguration` 类转换成了 `YamlDataSourceConfiguration` 类，而转换的过程则借助于 `DataSourceConfigurationYamlSwapper` 类。关于 ShardingSphere 中的 `YamlSwapper` 接口以及各种实现类我们已经在《05 | 配置驱动：ShardingSphere 中的配置体系是如何设计的？》中进行了详细介绍，这里只需要明确，通过 `DataSourceConfigurationYamlSwapper` 能够把 Yaml 配置文件中的 `DataSource` 配置转化为 `YamlDataSourceConfiguration` 类。

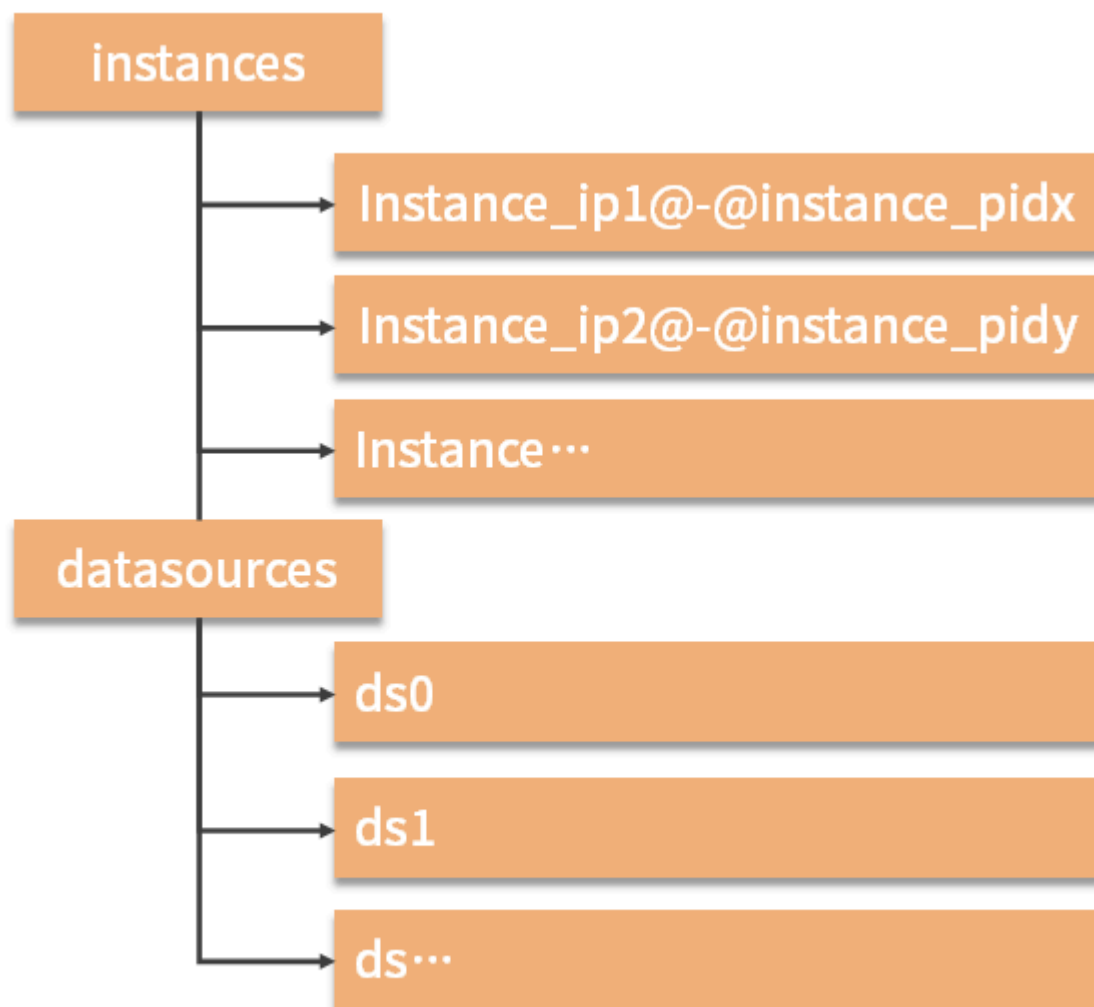
当获取了所需的 `YamlDataSourceConfiguration` 之后，我们就可以调用注册中心的 `persist` 方法完成数据的持久化，这就是 `persistDataSourceConfiguration` 方法中最后一句代码的作用。在这个过程中，我们同样需要把 `YamlDataSourceConfiguration` 数据结构转换为一个字符串，这部分工作是由 `YamlEngine` 来完成。关于 `YamlEngine` 的介绍我们也可以回顾《05 | 配置驱动：ShardingSphere 中的配置体系是如何设计的？》中的内容。

`ConfigurationService` 中其他方法的处理过程与 `persistDataSourceConfiguration` 方法本质上是一样的，只是所使用的数据类型和结构有所不同，这里不再赘述。

2.StateService

介绍完 `ConfigurationService` 类之后，我们来关注 `ShardingOrchestrationFacade` 类中的另一个核心变量 `StateService`。

从命名上讲，`StateService` 这个类名有点模糊，更合适的叫法应该是 `InstanceStateService`，用于管理数据库实例的状态，即创建数据库运行节点并区分不同数据库访问实例。存放在注册中心中的数据结构包括 `instances` 和 `datasources` 节点，存储结构如下所示：



数据库实例状态存储结构示意图

@拉勾教育

`StateService` 中保存着 `StateNode` 对象，`StateNode` 中的变量与上面的数据结构示例相对应，如下所示：

```
private static final String ROOT = "state";  
private static final String INSTANCES_NODE_PATH = "instances";  
private static final String DATA_SOURCES_NODE_PATH = "datasources";  
private final String name;
```

`StateService` 同时还保存着 `OrchestrationInstance` 对象，该对象用于根据你的 IP 地址、PID、一串 UUID 以及分隔符@构建 `instanceId`，如下所示：


```
instanceId = IpUtils.getIp() + DELIMITER + ManagementFactory.getRuntimeMXBean
```

需要注意的是，StateService 中对于 instances 和 datasources 的保存机制是不一样的：

```
//使用临时节点保存 Instance
```

```
public void persistInstanceOnline() {
    regCenter.persistEphemeral(stateNode.getInstanceNodeFullPath(instanceId));
}

//使用持久化节点保存DataSource

public void persistDataSourcesNode() {
    regCenter.persist(stateNode.getDataSourcesNodeFullRootPath(), "");
}
}
```

可以看到保存 Instance 用的是 RegistryCenter 中基于临时节点的 persistEphemeral 方法，而保存 DataSources 用的是基于持久化节点的 persist 方法，这样处理是有原因的。在可用性设计上，运行实例一般均可以标识为临时节点，当实例上线时注册，下线时自动清理。

3.ShardingOrchestrationListenerManager

我们接着来看 ShardingOrchestrationFacade 中的最后一个变量

ShardingOrchestrationListenerManager，从命名上看该类用于管理各种处理变更事件的监听器 Listener。而从前面的分析，我们不难看出系统中应该存在两大类的 Listener，一类用于监听配置信息的变更，一类用于监听实例状态的变更。

果然，在 ShardingOrchestrationListenerManager 中，我们进一步找到了两个 ListenerManager，即 ConfigurationChangedListenerManager 和 StateChangedListenerManager，如下所示：

```
public final class ShardingOrchestrationListenerManager {
    //配置变更监听管理器

    private final ConfigurationChangedListenerManager configurationChangedListenerManager;

    //状态变更监听管理器

    private final StateChangedListenerManager stateChangedListenerManager;


    public ShardingOrchestrationListenerManager(final String name, final RegistryCenter registryCenter,
        ConfigurationChangedListenerManager configurationChangedListenerManager = new ConfigurationChangedListenerManager(name, registryCenter);
        StateChangedListenerManager stateChangedListenerManager = new StateChangedListenerManager(name, registryCenter);
}
```

```
stateChangeListenerManager = new StateChangeListenerManager(name, r
}

public void initListeners() {
    configurationChangeListenerManager.initListeners();
    stateChangeListenerManager.initListeners();
}
}
```

我们创建了这两个 ListenerManager，并调用其 initListeners 方法进行了初始化。以 ConfigurationChangeListenerManager 为例，我们来看一下它内部的结构，如下所示：

```
public final class ConfigurationChangeListenerManager {
    private final SchemaChangeListener schemaChangeListener;
    private final PropertiesChangeListener propertiesChangeListener;
    private final AuthenticationChangeListener authenticationChangeListener
    public ConfigurationChangeListenerManager(final String name, final Regis
        schemaChangeListener = new SchemaChangeListener(name, regCenter, sh
        propertiesChangeListener = new PropertiesChangeListener(name, regCe
        authenticationChangeListener = new AuthenticationChangeListener(nam
    }
    public void initListeners() {
        schemaChangeListener.watch(ChangedType.UPDATED, ChangedType.DELETED)
        propertiesChangeListener.watch(ChangedType.UPDATED);
        authenticationChangeListener.watch(ChangedType.UPDATED);
    }
}
```



可以看到这里定义了 SchemaChangeListener、PropertiesChangeListener 和 AuthenticationChangeListener 这三个 Listener。显然，它们对应 ConfigurationService 中介绍的配置结构中的三大顶层配置项 schema、props 和 authentication。然后，对于这三种配置项，我们分别根据需要对具体某一个操作添加监视。从上面的代码中我们可以看到，对于 schema 配置项而言，当进行 UPDATE 和 DELETE 时，我们需要响应事件；而对于 props 和 authentication 配置项而言，则只需关注 UPDATE 操作。

因为这些具体的事件以及监听机制的处理方式大同小异，因此我们就以 SchemaChangeListener 为例进行进一步分析。SchemaChangeListener 继承自 PostShardingOrchestrationEventListener 抽象类，而后者又实现了 ShardingOrchestrationListener 接口，我们先来看这个接口的定义：

```
public interface ShardingOrchestrationListener {  
    //监听事件  
    void watch(ChangedType... watchedChangedTypes);  
}
```

PostShardingOrchestrationEventListener 实现了这个接口，其实现过程如下所示：

```
public abstract class PostShardingOrchestrationEventListener implements ShardingOrchestrationListener {  
    //创建 EventBus  
    private final EventBus eventBus = ShardingOrchestrationEventBus.getInstance();  
    private final RegistryCenter regCenter;  
    private final String watchKey;  
    @Override  
    public final void watch(final ChangedType... watchedChangedTypes) {  
        final Collection<ChangedType> watchedChangedTypeList = Arrays.asList(watchedChangedTypes);  
        regCenter.watch(watchKey, new DataChangedEventListener() {  
            @Override  
            public void onChange(final DataChangedEvent dataChangedEvent) {  
                if (watchedChangedTypeList.contains(dataChangedEvent.getChangedType())) {  
                    //通过 EventBus 发布事件  
                    eventBus.post(createShardingOrchestrationEvent(dataChangedEvent));  
                }  
            }  
        });  
    }  
    protected abstract ShardingOrchestrationEvent createShardingOrchestrationEvent(DataChangedEvent dataChangedEvent);  
}
```

上述代码的核心机制是通过 RegistryCenter 的 watch 方法为具体的事件添加事件处理程序，而这个事件处理过程就是通过 Guava 中的 EventBus 类的 post 方法将事件进行进一步转发。至于所需要转发的具体事件类型由抽象方法 createShardingOrchestrationEvent 来提供，PostShardingOrchestrationEventListener 的各个子类需要实现这个抽象方法。

我们来看 PostShardingOrchestrationEventListener 的子类 SchemaChangeListener 对事件创建过程的处理方法，这里以 createDataSourceChangedEvent 方法为例进行展开，这是一个比较典型的创建事件的方法：

```
private DataSourceChangedEvent createDataSourceChangedEvent(final String shardingSchemaName,
    Map<String, YamlDataSourceConfiguration> dataSourceConfigurations = (Preconditions.checkNotNull(dataSourceConfigurations) && !dataSourceConfigurations.isEmpty()) ? dataSourceConfigurations : Collections.emptyMap()
    //创建DataSourceChangedEvent
    return new DataSourceChangedEvent(shardingSchemaName, Maps.transformValues(dataSourceConfigurations, new DataSourceConfigurationYamlSwapper()));
}

@Override
public DataSourceConfiguration apply(final YamlDataSourceConfiguration input) {
    return new DataSourceConfigurationYamlSwapper().swap(input);
}

}

}));
}

}
```

可以看到，这里再次用到了前面提到的 YamlDataSourceConfiguration 以及 YamlEngine，不同的是这次的处理流程是从 YamlDataSourceConfiguration 到 DataSourceConfiguration。最终，我们构建了一个 DataSourceChangedEvent，包含了 shardingSchemaName 以及一个 dataSourceConfigurations 对象。

关于整个 Listener 机制，可以简单归纳为通过监听注册中心上相关数据项的操作情况来生成具体的事件，并对事件进行包装之后再进行转发。至于如何处理这些转发后的事件，取决于具体的应用场景，典型的一个应用场景就是控制数据访问的熔断，让我们一起来看一下。

注册中心的应用：数据访问熔断机制

ShardingOrchestrationFacade 是一个典型的外观类，通过分析代码的调用关系，我们发现该类的创建过程都发生在 sharding-jdbc-orchestration 工程的几个 DataSource 类中。我们先来到 AbstractOrchestrationDataSource 这个抽象类，该类的核心变量如下所示：

```
private final ShardingOrchestrationFacade shardingOrchestrationFacade;

//是否熔断
```

```
private boolean isCircuitBreak;

private final Map<String, DataSourceConfiguration> dataSourceConfigurations =
```

注意到这里还有一个 `isCircuitBreak` 变量，用来表示是否需要熔断，接下来我们会对熔断机制以及该变量的使用方法做详细展开。

我们继续来看 `AbstractOrchestrationDataSource` 的构造函数，如下所示：

```
public AbstractOrchestrationDataSource(final ShardingOrchestrationFacade shardingOrchestrationFacade) {
    this.shardingOrchestrationFacade = shardingOrchestrationFacade;
    //通过 EventBus 注册自己
    ShardingOrchestrationEventBus.getInstance().register(this);
}
```

可以看到这里用到了 Guava 中 `EventBus` 的 `register` 方法，这个方法用于对注册事件的订阅。在前面的内容中，我们留下了一个疑问，即**所创建的这些 `ShardingOrchestrationEvent` 是如何被处理的呢？**

答案就在这里进行了揭晓，即所有通过 `EventBus` 的 `post` 方法所发布的事件的最终消费者就是这个 `AbstractOrchestrationDataSource` 类以及它的各个子类。而在 `AbstractOrchestrationDataSource` 类中就存在了如下所示的 `renew` 方法，用于处理 `CircuitStateChangedEvent` 事件：

`@Subscribe`

```
public final synchronized void renew(final CircuitStateChangedEvent circuitStateChangedEvent) {
    isCircuitBreak = circuitStateChangedEvent.isCircuitBreak();
}
```

在这个方法上添加了 `@Subscribe` 注解，即一旦在系统中生成了 `CircuitStateChangedEvent` 事件，这个方法就可以自动响应这类事件。在这个处理方法中，我们看到它从 `CircuitStateChangedEvent` 事件中获取了是否熔断的信息并赋值给前面介绍的 `isCircuitBreak` 变量。

在 `AbstractOrchestrationDataSource` 的 `getConnection` 方法中调用了 `getDataSource` 抽象方法以获取特定的 `DataSource`，进而获取特定的 `Connection`，如下所示：

}

那么回到一个问题，即什么时候会触发熔断机制，也就是什么时候会发送这个 `CircuitStateChangedEvent` 事件呢？让我们跟踪这个事件的创建过程，来到了如下所示的 `InstanceStateChangeListener` 类：

}

从源码解析到日常开发

我们注意到 ShardingSphere 实现事件驱动架构时使用了 Guava 框架中的 EventBus 工具类，在日常开发过程中，我们也可以直接使用这个类来构建自定义的事件处理机制。

小结与预告

注册中心是 ShardingSphere 编排治理机制中的一个重要组成部分，但注册中心本身也只是一个工具，需要根据不同的业务场景来设计对应的应用方式。在 ShardingSphere 中，配置信息管理以及数据库实例管理就是典型的应用场景，我们基于这些场景详细分析了基于注册中心的事件驱动架构的设计和实现方法，并给出了基于数据访问熔断机制的案例分析。

这里给你留一道思考题：在 ShardingSphere 中，如何把服务实例的状态与注册中心整合在一起进行编排治理？

在下一课时中，我们将介绍 ShardingSphere 编排治理中的另一个重要主题，即服务访问的链路监控和跟踪机制。

[上一页](#)[下一页](#)