

二

服务器出问题，目前部分恢复

31 配置中心：如何基于配置中心实现配置信息的动态化管理？

ShardingSphere 在编排治理方面包括配置动态化、注册中心、数据库熔断禁用、调用链路等治理能力。

今天我们先来介绍最简单的配置中心，即如何基于配置中心从而实现配置信息的动态化管理。

ShardingSphere 中对配置中心的抽象过程

配置中心的核心接口 `ConfigCenter` 位于 `sharding-orchestration-config-api` 工程中，定义如下：

```
public interface ConfigCenter extends TypeBasedSPI {  
    //初始化配置中心  
    void init(ConfigCenterConfiguration config);  
    //获取配置项数据  
    String get(String key);  
    //直接获取配置项数据  
    String getDirectly(String key);  
    //是否存在配置项  
    boolean isExisted(String key);  
    //获取子配置项列表  
    List<String> getChildrenKeys(String key);  
    //持久化配置项  
    void persist(String key, String value);  
    //更新配置项  
    void update(String key, String value);  
    //持久化临时数据
```

```
void persistEphemeral(String key, String value);

//对配置项或路径进行监听

void watch(String key, DataChangedEventListener dataChangedEventListener)

//关闭配置中心

void close();

}
```

上述方法中，唯一值得展开的就是 watch 方法，该方法传入了一个代表事件监听器的 DataChangedEventListener 接口，如下所示：

```
public interface DataChangedEventListener {

    //当数据变动时进行触发

    void onChange(DataChangedEvent dataChangedEvent);

}
```

这里用到的 DataChangedEvent 类定义如下，可以看到事件的类型有三种，分别是 UPDATED、DELETED 和 IGNORED：

```
public final class DataChangedEvent {

    private final String key;

    private final String value;

    private final ChangedType changedType;

    public enum ChangedType {

        UPDATED, DELETED, IGNORED

    }

}
```

我们同样注意到 ConfigCenter 接口继承了 TypeBasedSPI 接口，所以集成了 SPI 机制。在 ShardingSphere 中，ConfigCenter 接口有两个实现类，分别基于 Apollo 的 ApolloConfigCenter 和基于 Zookeeper 的 CuratorZookeeperConfigCenter。

我们分别展开讲解一下。

ApolloConfigCenter

1. ApolloConfigCenter 的实现过程

我们先来看基于 Apollo 的 ApolloConfigCenter，它的 init 方法如下所示：

`@Override`

```
public void init(final ConfigCenterConfiguration config) {  
    //从配置对象中获取配置信息并设置系统属性  
  
    System.getProperties().setProperty("app.id", properties.getProperty("  
    System.getProperties().setProperty("env", properties.getProperty("env  
    System.getProperties().setProperty(ConfigConsts.APOLLO_CLUSTER_KEY, p  
    System.getProperties().setProperty(ConfigConsts.APOLLO_META_KEY, conf  
  
    //通过配置对象构建 ApolloConfig  
  
    apolloConfig = ConfigService.getConfig(config.getNamespace());  
  
}
```

上述 init 方法的作用是在设置系统属性的同时，构建一个 Config 对象。在 Apollo 中，基于这个 Config 对象就可以实现对配置项的操作，例如：

`@Override`

```
public String get(final String key) {  
    return apolloConfig.getProperty(key.replace("/", "."), "");  
  
}
```

`@Override`

```
public String getDirectly(final String key) {  
    return get(key);  
  
}
```

`@Override`

```
public boolean isExisted(final String key) {  
    return !Strings.isNullOrEmpty(get(key));  
  
}
```

注意这里的 getDirectly 方法和 get 方法的处理方式实际上是一致的。而对于 Apollo 而言，getChildrenKeys、persist、update 和 persistEphemeral 等方法都是无效的，因为不支持这

样的操作。但是对于常见的监听机制，Apollo 也提供了它的实现方案，可以通过对 Config 对象添加 ChangeListener 来实现监听效果，如下所示：

`@Override`

```
public void watch(final String key, final DataChangedEventListener dataChange

    //添加 Apollo 中的监听器

    apolloConfig.addChangeListener(new ConfigChangeListener() {

        @Override

        public void onChange(final ConfigChangeEvent changeEvent) {

            for (String key : changeEvent.changedKeys()) {

                //获取 Apollo 监听事件

                ConfigChange change = changeEvent.getChange(key);

                DataChangedEvent.ChangedType changedType = getChangedType

                if (DataChangedEvent.ChangedType.IGNORED != changedType)

                    //将 Apollo 中的监听事件转化为 ShardingSphere 中的监听事件

                    //通过 EventListener 触发事件

                    dataChangedEventListener.onChange(new DataChangedEven

            }

        }

    }, Sets.newHashSet(key));

}
```

上述代码的逻辑在于当事件被 Apollo 监听，并触发上述 watch 方法时，我们会将 Apollo 中的事件类型转化为 ShardingSphere 中自身的事件类型，并通过 DataChangedEventListener 进行传播和处理。

2.ShardingSphere 中的事件驱动架构

讲到 DataChangedEventListener，我们不得不对 ShardingSphere 中的事件驱动框架做一些展开。

显然，从命名上看，DataChangedEventListener 是一种事件监听器，用于监听各种 DataChangedEvent。

注意到 ShardingSphere 并没有提供 `DataChangedEventListener` 接口的任何实现类，而是大量采用了匿名方法进行事件的监听，一种典型的实现方式如下所示：

```
new DataChangedEventListener() {  
    @Override  
    public void onChange(final DataChangedEvent dataChangedEvent) {  
        ...  
        //通过 EventBus 发布事件  
        eventBus.post(createXXXEvent(dataChangedEvent));  
    }  
};
```

在通过 `DataChangedEventListener` 监听到某一个 `DataChangedEvent` 并进行传播时，ShardingSphere 的处理过程就是通过 `EventBus` 类的 `post` 方法将事件进行进一步转发。这里使用的 `EventBus` 同样来自 Google 的 Guava 框架，代表了一种事件总线的实现方式。

现在，事件已经可以通过 `EventBus` 进行发送了，那么这些被发送的事件是怎么被消费的呢？在 ShardingSphere 中，存在一个 `ShardingOrchestrationEventBus` 包装类，包装了对 `EventBus` 的使用过程。

这个包装过程非常简单，只是使用单例模式构建了一个 `EventBus` 对象而已，如下所示：

```
public final class ShardingOrchestrationEventBus {  
    private static final EventBus INSTANCE = new EventBus();  
    //使用单例模式构建单例对象  
    public static EventBus getInstance() {  
        return INSTANCE;  
    }  
}
```

如果我们想要订阅通过 `EventBus` 发送的事件，只要把自身注册到 `EventBus` 上即可，可以直接通过 `EventBus` 提供的 `register` 方法实现这一目标，如下所示：

```
ShardingOrchestrationEventBus.getInstance().register(this);
```

另一方面，在 Guava 的 EventBus 机制中，提供了 @Subscribe 注解用来标识对具体某一种事件的处理方法。一旦在某个方法上添加了 @Subscribe 注解，这个方法就可以自动用来处理所传入的事件。

所以，我们进一步总结事件订阅者的代码结构，可以得到如下所示的伪代码：

```
public class Subscriber {  
    public Subscriber(...) {  
        ...  
        //将自己注册到 EventBus 中  
        ShardingOrchestrationEventBus.getInstance().register(this);  
    }  
    @Subscribe  
    public void renew(DataSourceChangedEvent dataSourceChangedEvent){  
        //消费事件  
        ...  
    }  
}
```

可以想象，ShardingSphere 中势必存在一批符合上述代码结构的实现类，用于监听配置中心所产生的配置信息变更事件。以如下所示的 LogicSchema 类为例，我们可以看到它的实现过程就是很典型的一种事件订阅者：

```
@Getter  
public abstract class LogicSchema {  
    public LogicSchema(final String name, final Map<String, YamlDataSouc  
        ...  
        ShardingOrchestrationEventBus.getInstance().register(this);  
    }  
    @Subscribe  
    public final synchronized void renew(final DataSourceChangedEvent dataSou  
        if (!name.equals(dataSourceChangedEvent.getShardingSchemaName())) {  
            return;  
        }  
    }
```

```
//根据 DataSourceChangedEvent 更新 DataSource 的配置
```

```
backendDataSource.renew(DataSourceConverter.getDataSourceParameterMap  
  
}  
  
}
```

上述 LogicSchema 类会根据 DataSourceChangedEvent 中携带的配置信息来更新 DataSource 的配置，从而实现配置信息的动态化管理。

在介绍完 ApolloConfigCenter 之后，我们再来看一下 ShardingSphere 中另一种配置中心的实现方式，即 CuratorZookeeperConfigCenter。

CuratorZookeeperConfigCenter

1.Zookeeper 和 Curator 简介

CuratorZookeeperConfigCenter 使用 Zookeeper 作为配置中心的服务组件。针对如何使用 Zookeeper，业界也存在一些开源的客户端，而在 ShardingSphere 采用的是 Curator。

在介绍 CuratorZookeeperConfigCenter 之前，我们先来对 Zookeeper 和 Curator 做简要介绍。

- Zookeeper

对于 Zookeeper 而言，我们知道它有两个特性与分布式协调直接相关，一个是会话机制，一个是 Watcher 机制。

会话是客户端和服务端端的 TCP 连接，能够发送请求并接收监听器 Watcher 事件，而 Watcher 机制本质上就是分布式的回调。就类型而言，会话又可以分为**短暂性会话**和**持久性会话**两种，前者在会话断开的同时会自动删除会话对应的 ZNode，而后者则不会。ZNode 的客户端关注 ZNode 发生的变化，一旦发生变化则回传消息到客户端，然后客户端的消息处理函数得到调用。在 Zookeeper 中，任何读操作都能够设置 Watcher。

- Curator

在我们使用 Zookeeper 时，一般不使用它原生的 API，而是倾向于采用客户端集成框架，这其中最具代表性的就是 Curator。Curator 解决了很多 Zookeeper 客户端非常底层的细节开发工作，包括连接重试、反复注册 Watcher 和 NodeExistsException 异常等。

Curator 包含了几个包：其中 curator-framework 包提供了对 Zookeeper 底层 API 的一层封装；curator-client 包则提供一些客户端的操作，例如重试策略等；而 curator-recipes 包封装了一些高级特性，如选举、分布式锁、分布式计数器等。

在使用 Curator 时，首先需要创建一个 CuratorFramework 客户端对象，这一过程可以使用 CuratorFrameworkFactory 工厂类进行完成。对于 CuratorFrameworkFactory 而言，我们一方面需要指定与 Zookeeper 的链接 URL connectString、会话超时时间 sessionTimeoutMs、连接创建超时时间 connectionTimeoutMs，以及重试策略 retryPolicy；另一方面也可以根据需要设置安全认证信息。

一旦我们获取了 CuratorFramework 对象，就可以调用它的 start 方法启动客户端，然后通过 create/delete 来创建和删除节点，通过 getData/setData 方法获取，以及设置对应节点中的数据。当然，最为重要的是我们可以在节点上添加监听器。

接下来就让我们一起看一下 ShardingSphere 中如何使用 Curator 完成与 Zookeeper 的集成方法。

2. CuratorZookeeperConfigCenter 的实现过程

在 ShardingSphere 中，使用 CuratorFrameworkFactory 创建 CuratorFramework 客户端对象的过程如下所示：

```
private CuratorFramework buildCuratorClient(final ConfigCenterConfiguration c

    //构建 CuratorFrameworkFactory 并设置连接属性

    CuratorFrameworkFactory.Builder builder = CuratorFrameworkFactory.buide

        .connectString(config.getServerLists())

        .retryPolicy(new ExponentialBackoffRetry(config.getRetryInter

        .namespace(config.getNamespace());

    if (0 != config.getTimeToLiveSeconds()) {

        builder.sessionTimeoutMs(config.getTimeToLiveSeconds() * 1000);

    }

    if (0 != config.getOperationTimeoutMilliseconds()) {

        builder.connectionTimeoutMs(config.getOperationTimeoutMillisecond

    }

    //设置安全摘要信息

    if (!Strings.isNullOrEmpty(config.getDigest())) {

        builder.authorization("digest", config.getDigest().getBytes(Chars

            .aclProvider(new ACLProvider() {

                @Override

                public List<ACL> getDefaultAcl() {
```



```

        return ZooDefs.Ids.CREATOR_ALL_ACL;
    }

    @Override
    public List<ACL> getAclForPath(final String path) {
        return ZooDefs.Ids.CREATOR_ALL_ACL;
    }
});

}

return builder.build();
}

```


上述代码相对比较固化，我们可以直接在自己的应用程序中进行借鉴和参考。

然后我们来看它的 persist 方法，如下所示：

```

@Override
public void persist(final String key, final String value) {
    try {
        if (!isExisted(key)) {
            //创建持久化节点
            client.create().creatingParentsIfNeeded().withMode(CreateMode
        } else {
            update(key, value);
        }
    } catch (final Exception ex) {
        CuratorZookeeperExceptionHandler.handleException(ex);
    }
}

```



这里使用了 CreateMode.PERSISTENT 模式来创建接口，也就是说创建的是一种持久化节点。而另一个 persistEphemeral 方法中，则通过设置 CreateMode.EPHEMERAL 来创建临时节点。

@Override

```
public void watch(final String key, final DataChangedEventListener dataChange

    final String path = key + "/";

    if (!cache.containsKey(path)) {

        addCacheData(key);

    }

    TreeCache cache = caches.get(path);

    //添加 Zookeeper 监听器

    cache.getListenable().addListener(new TreeCacheListener() {

        @Override

        public void childEvent(final CuratorFramework client, final TreeC

            //获取 Zookeeper 监听事件

            ChildData data = event.getData();

            if (null == data || null == data.getPath()) {

                return;

            }

            //将 Zookeeper 中的监听事件转化为 ShardingSphere 中的监听事件

            //通过 EventListener 触发事件

            DataChangedEvent.ChangedType changedType = getChangedType(eve

            if (DataChangedEvent.ChangedType.IGNORED != changedType) {

                dataChangedEventListener.onChange(new DataChangedEvent(da

            }

        }

    });

}
```

可以看到，watch 方法的最终处理结果也是将 Zookeeper 中的监听事件转化为 ShardingSphere 中的监听事件，并通过 EventListener 触发事件。这个过程我们已经在介绍 ApolloConfigCenter 时做了展开。

从源码解析到日常开发

今天我们介绍的很多内容实际上也可以应用到日常开发过程中，包括如何基于 Apollo 以及 Zookeeper 这两款典型的配置中心实现工具，来进行配置信息的存储和监听。我们完全可以根据自身的需求，将应用场景和范围从配置中心扩大到各种需要进行动态化管理的业务数据，而基于这两款工具实现这一目标的实现细节，我们都可以直接进行参考和借鉴。

小结与预告

本课时关注于 ShardingSphere 中对配置中心的抽象和实现过程。配置中心的核心机制是需要实现配置信息的动态化加载，而 Apollo 和 Zookeeper 都提供了监听机制来实现这一目标。ShardingSphere 通过集成这两款主流的开源工具，以及 Guava 框架中的 EventBus 工具类实现了从事件监听到订阅消费的整个事件驱动架构。

这里给你留一道思考题：ShardingSphere 是如何将 Apollo 以及 Zookeeper 中的事件生成和监听机制抽象成一套统一的事件驱动架构的？欢迎你在留言区与大家讨论，我将逐一点评解答。

配置中心和注册中心在实现上存在一定的相似性，但又面向不同的应用场景。下一课时，我们将介绍 ShardingSphere 中的注册中心的实现机制和应用场景。

[上一页](#)

[下一页](#)