第16章

DataStore 相关的开发

本章主要介绍 DataStore 相关的开发，主要包括利用 DataBroker 实现对 DataStore 的操作和 Data Change 事件的实现。本章的实验是在第 15 章的基础上进行的，利用已实现的 RPC 来使用 DataBroker 实现对 DataStore 的操作，完成 Data Change 事件的实现。这两个功能是可独立实现的，并不存在必需的依赖关系。

本章首先在 16.1 节对 DataStore 相关开发过程进行简要说明，即利用已实现的 RPC 来使用 DataBroker 实现对 DataStore 操作的简要步骤。本节将介绍对 DataStore 的异步读写操作，创建包含 DataStore 的异步读写操作的类的实例并传递 DataBroker 参数，最后对项目进行测试验证。

本章在 16.2 节利用 DataBroker 实现对 DataStore 的操作，具体来说就是创建一个既可读又可写的读写事务操作 ReadWriteTransaction，然后利用从 MyprojectProvider 类中传递过来的 DataBroker 来实现对 DataStore 的读写操作。

本章在 16.3 节重点介绍 Data Change 事件的实现。这种消息传递方式属于多播异步事件，如果数据树中数据有变化，这个事件由数据代理并且传递到订阅者。具体的实现方式为先实现 DataChangeListener 接口以完成 onDataChange 函数，然后将数据树变动的监听注册到 MD-SAL，最后同样对项目进行测试验证。

最后在 16.4 节对全章进行总结。

16.1 DataStore 相关开发过程的简要说明

DataStore 相关的开发主要包括利用 DataBroker 实现对 DataStore 的操作和 Data Change 事件的实现，这两个功能是可独立实现的，并不存在必需的依赖关系。

16.1.1　使用 DataBroker 实现对 DataStore 的操作

总的来看，使用 DataBroker 实现对 DataStore 的操作可分为两个步骤：

1. 实现对 DataStore 的操作

从并发的角度来看，操作包含同步的操作和异步的操作；从操作的内容来看，操作包含读事务操作 ReadTransaction（对 DataStore 仅能实现只读操作）、写事务操作 WriteTransaction（对 DataStore 仅能实现写操作）、读写操作 ReadWriteTransaction（对 DataStore 能实现读操作和写操作）这 3 种方式，需要更深入了解的读者请参考本书第 12 章的 12.4 节中关于对 DataStore 操作的 3 种方式。以下是实现对 DataStore 的操作的要点：

（1）设定可供传递参数的私有 DataBroker 成员，设为 myDataBroker。

（2）创建修改或读取的节点，创建成员变量身份标识 Instance Identifier，设为 My_ID。

```
final InstanceIdentifier<树中的节点的类，如容器类> My_ID = \
        InstanceIdentifier.builder(树中的节点的类.class).build();
```

（3）基于 DataBroker 参数创建新的操作（读事务操作、写事务操作或读写事务操作）类成员，设为 myTx。以读写事务操作（ReadWriteTransaction）为例，读事务操作（ReadTransaction）、写事务操作（WriteTransaction）相应替换：

```
final ReadWriteTransaction myTx =  \
        this.myDataBroker.newReadWriteTransaction();
```

（4）实现对 DataStore 的读操作。

使用类型为读事务操作或读写事务操作的类成员变量（设为 myRWtx）对 DataStore 的树节点进行读操作：

```
ListenableFuture<Optional<MyContainer1>> myTx =  \
        myTx.read(LogicalDataStoreType, InstanceIdentifier);
```

其中参数 LogicalDataStoreType 是 DataStore 中存储的数据类型，可为 Configuration 或 Operational，写成 LogicalDatastoreType.CONFIGURATION 或 LogicalDatastoreType.OPERATIONAL 的方式；而参数 InstanceIdentifier 是待读节点的身份标识，即之前创建的成员变量身份标识 Instance Identifier（My_ID）。例如，读取树中标识为 MyID 节点的 Operational 数据类型可写成：

```
ListenableFuture<Optional<MyContainer1>> myTx =  \
        myTx.read(LogicalDatastoreType.OPERATIONAL, MyID);
```

（5）实现对 DataStore 的修改操作。

使用类型为写事务操作或读写事务操作的类成员变量（设为 myRWtx）。使用 YANG 定义映射生成的相关类中的 set 方法创建数据。使用变量成员 myRWtx 对 DataStore 的树节点进行修改操作。修改操作主要有 3 种修改树的操作：put、merge 和 delete。

① put：在树指定的节点上传数据，操作完成后原子树被此数据替换。

```
<T> void put(LogicalDatastoreType store, InstanceIdentifier<T> path, T data);
```

```
myTx.put(LogicalDatastoreType store, InstanceIdentifier<T> path, T data);
```

LogicalDatastoreType 是 DataStore 中存储的数据类型，可为 Configuration 或 Operational；InstanceIdentifier<T> path 是之前创建的成员变量身份标识 Instance Identifier（如 My_ID）；T data 是待上传的子树。读者可参考本章 16.2 节的示例，例如：

```
myRWtx.put(LogicalDatastoreType.OPERATIONAL, \
  MyContainer1_ID,  new MyContainer1Builder().\
  setMycontainer1Leaf1((byte)0).setMyContGrpLeaf1(88888)\
  .build());
```

② merge：在树指定的节点上传数据，操作完成后原子树不会被覆盖，与新上传的数据合并。

```
myTx.merge(LogicalDatastoreType store, InstanceIdentifier<T> path, T data);
```

③ delete：删除树指定节点的子树。

```
myTx.delete(LogicalDatastoreType store, InstanceIdentifier<T> path);
```

（6）使用方法 submit()完成事务，具体来说就是 submit()封装事务并提交它以进行处理。

提交事务通过下面的方法得到处理和提交：

```
return myTx.submit();
```

注意，方法 submit()的函数为：

```
CheckedFuture<Void,TransactionCommitFailedException> submit();
```

（7）可以通过阻塞或者异步调用以查看事务提交结果。

若需要通过阻塞或者异步调用以查看事务提交结果，则将上面的语句：

```
return myTx.submit();
```

替换为本节的语句。具体如下：

应用程序可使用 ListenableFuture 异步地监听事务提交状态。

```
Futures.addCallback( myTx.submit(), new FutureCallback<Void>() {
    public void onSuccess( Void result ) {
        LOG.debug("Transaction committed successfully.");
    }
    public void onFailure( Throwable t ) {
        LOG.error("Commit failed.",e);
    }
});
```

若应用程序需要阻塞直到提交成功，则可使用 checkedGet()方法以等待直到提交完成。

```
try {
    myTx.submit().checkedGet();
    } catch (TransactionCommitFailedException e) {
        LOG.error("Commit failed.",e);
}
```

2. 传递 DataBroker 参数

在创建包含实现对 DataStore 操作的类处（MyprojectProvider 类中）向此实例传递 DataBroker 参数。建议在 onSessionInitiated(ProviderContext session)函数中实现。

16.1.2　完成 Data Change 事件的实现

Data Change 事件是 OpenDaylight 消息传递方式的一种，这种消息传递方式属于多播异步事件，如果数据树中数据有变化，这个事件由数据代理并且传递到订阅者。具体的实现方式如下：

1. 创建一个实现 Data Change 事件的类

（1）类扩展监听接口（如 DataChangeListener）以完成 onDataChange 函数。

OpenDaylight 项目中有 3 种方式可对事件进行监听：DataChangeListener、DataTreeChangeListener 和 DOMDataTreeChangeListener。

① DataChangeListener 是最简单的方式，使用 Binding-aware 方式访问 DataStore，对整棵树进行监听，树中任何一个叶子节点的变化都会触发 DataChangeListener 事件。其对应的实现事件变化所激发处理的接口为 onDataChange 函数：

```
void onDataChanged(AsyncDataChangeEvent<InstanceIdentifier<?>, \
                   DataObject> data)
```

② DataTreeChangeListener 能定位到树中的树干，实现比 DataChangeListener 更精确的监听：

```
void onDataTreeChanged(Collection<DataTreeModification<Vlan>> changed)
```

③ DOMDataTreeChangeListener 通过 DOMDataBroker 访问 DataStore，使用 Binding Independent 类型，使用 QName 对数据树进行索引和数据的定位。

（2）类扩展项目的 Service 接口不同，如本章示例 MyprojectService 接口。

（3）设定可供传递参数的私有 DataBroker 成员，设为 myDataBroker。

（4）创建待监听节点，创建成员变量身份标识 Instance Identifier，设为 My_ID：

```
final InstanceIdentifier<树中的节点的类，如容器类> My_ID = \
        InstanceIdentifier.builder(树中的节点的类.class).build();
```

（5）实现事件变化所激发处理的函数，如使用 DataChangeListener 接口处理相应的数据变化处理的 onDataChange 函数：

```
public void onDataChanged(AsyncDataChangeEvent<InstanceIdentifier<?>, \
            DataObject> data) {
    // 获取变化的数据对象
    DataObject myDO = data.getUpdatedSubtree();
    if( myDO instanceof 子树类型){
    // 若变化的数据对象为子树类型的一个实例
        子树类型 myData = (子树类型)myDO;
}
```

DataTreeChangeListener 和 DOMDataTreeChangeListener 在各自的事件变化触发的函数中使用类似以上的方式实现功能。

2. 将数据树变动的监听注册到 MD-SAL

（1）创建实现 Data Change 事件的类（本书在 MyprojectProvider 类中创建实现 DataChange 事件的类）。

（2）向创建实现 Data Change 事件的类的实例传递 DataBroker 参数。

（3）添加数据变动 ListenerRegistration<DataChangeListener> 的监听成员变量 dataChangeListenerReg。

（4）在 onSessionInitiated(ProviderContext session)函数中注册相应的 P path（监听节点的身份标识 InstanceIdentifier）和 DataChangeScope triggeringScope（指定数据树变动通知执行的类），注册函数为：

```
registerDataChangeListener(LogicalDatastoreType store, P path, L listener,\
DataChangeScope triggeringScope, "Selector selector");
```

其中 LogicalDatastoreType 为 DataStore 中存储的数据类型，值为 Configuration 或 Operational；P path 为监听节点的身份标识 InstanceIdentifier，如 myID；DataChangeScope triggeringScope 指定数据树变动通知执行的类；Selector selector 为数据监听的范围，如 BASE 只监听当前节点变化事件，ONE 监听当前节点和其左右儿子节点变化事件，SUBTREE 监听当前节点和左右子树的所有变化事件。

（5）最后在 close 方法中加入对 dataChangeListenerReg 的注销。

16.2 利用 DataBroker 实现对 DataStore 的操作

我们将在 myproject 项目的子项目 impl 中使用 RPC 的方式来利用 DataBroker 实现对 DataStore 的操作。读者可将此操作放置于 MyRPCImpl 类任何指定的调用函数内，也可以放置到本项目的其他类甚至其他 MD-SAL 项目的类中的某函数内实现，只需引用的操作涉及相关类，然后在 MyprojectProvider 类中将 DataBroker 的值传递给此操作即可。

以下具体说明在 MyRPCImpl 类中使用 RPC 的方式来利用 DataBroker 实现对 DataStore 的操作。本书选择在 MyRPCImpl 类的 RPC 之一的 myRpc1 中实现，将创建一个既可读又可写的读写事务操作 ReadWriteTransaction，然后利用从 MyprojectProvider 类中传递过来的 DataBroker 来实现对 DataStore 的读写操作。若读者需要单独进行读写操作，则可参考本章 16.1 节中介绍的读事务操作 ReadTransaction 和写事务操作 WriteTransaction 分别对 DataStore 进行读操作或写操作。

16.2.1 实现对 DataStore 的异步读写操作

在 MyRPCImpl 类的 RPC 中实现对 DataStore 的异步读写操作。建议使用异步事务操作完成相应功能，这有利于在生产环境中使用。若读者需要使用同步事务操作功能，请参考本章 16.1 节的介绍进行替代。

MyRPCImpl 类的 RPC 之一的 myRpc1 的原代码为：

```
    @Override
    public Future<RpcResult<Void>> myRpc1() {
        LOG.info("RPC1: This is my rpc1.It has nothing in it.");
        return \
Futures.immediateFuture(RpcResultBuilder.<Void> success().build());
    }
```

首先，除已引入的类外，对 DataStore 的同步和异步读写操作所需的类再引入以下实现：

```
//引用 DataBroker
import org.opendaylight.controller.md.sal.binding.api.DataBroker;
/*
    引入读写事务操作所需的类 ReadWriteTransaction。若读者仅需求读事务操作，则需引用
ReadTransaction 类；若仅需要写事务操作，则仅需引用 WriteTransaction 类
*/
import org.opendaylight.controller.md.sal.binding.api.ReadWriteTransaction;
/*
    引入 DataStore 中存储的数据类型 LogicalDatastoreType，值为 Configuration 或 Operational
*/
import org.opendaylight.controller.md.sal.common.api.data.LogicalDatastoreType;
//引入对 DataStore 操作提交失败所需相关处理的类
import org.opendaylight.controller.md.sal.common.api.data.TransactionCommitFailedException;
//引入类 InstanceIdentifier，它是监听节点的身份标识
import org.opendaylight.yangtools.yang.binding.InstanceIdentifier;
//使用异步提交操作所需的类 AsyncFunction，若同步提交，则无须此类
import com.google.common.util.concurrent.AsyncFunction;
//引入读写事务操作所需的其他类
import com.google.common.base.Optional;
import com.google.common.util.concurrent.FutureCallback;
import com.google.common.util.concurrent.ListenableFuture;
```

接着，设定私有 DataBroker 成员 myDataBroker，以供之后的读写操作使用：

```
    private DataBroker myDataBroker;
```

由于私有 DataBroker 成员 myDataBroker 的值需要由 MyprojectProvider 类传递，因此建立构建函数（注意，读者也可创建一个将 DataBroker 值传递的函数以供 MyprojectProvider 类赋值）：

```
    public MyRPCImpl(DataBroker dataBroker){
        this.myDataBroker = dataBroker;
    }
```

随后指定读写事务操作的树节点。本实验对容器 my-container1 进行振作，因此根据需要创建容器的标志点 MyContainer1_ID：

```
    final InstanceIdentifier<MyContainer1> MyContainer1_ID = \
                        InstanceIdentifier.builder(MyContainer1.class).build();
```

接下来，在 myRpc1 方法中实现对 DataStore 的异步读写操作：

```
@Override
public Future<RpcResult<Void>> myRpc1() {
    LOG.info("RPC1: This is my rpc1.It has nothing in it.");
    /*
    完成 ReadWriteTransaction，此操作也可放置到本项目的其他类甚至其他项目的类中的某
    函数内实现，随后调用即可。
    若读者使用读事务操作，则将下面语句中的 ReadWriteTransaction 替换成 ReadTransaction；
    若读者使用写事务操作，则将下面语句中的 ReadWriteTransaction 替换成写事务操作
    WriteTransaction
    */
    final ReadWriteTransaction myRWtx =  \
        this.myDataBroker.newReadWriteTransaction();
    //读取 OPERATIONAL 类数据库中的 my-container1 节点
    ListenableFuture<Optional<MyContainer1>> myRWFuture =  \
        myRWtx.read(LogicalDatastoreType.OPERATIONAL, MyContainer1_ID);
    /*
    bResultOfCommitFuture 是写事务操作提交的条件，值为 true 时提交。读者可替代成自己
    设定的条件
    */
    final Boolean bResultOfCommitFuture = true;
    /*
    将 Optional< MyContainer1>类型的 ListenableFuture 转换成 Void 类型
    ListenableFuture
    */
    //使用异步完成写操作
    final ListenableFuture<Void> commitFuture = \
        Futures.transform(myRWFuture, \
            new AsyncFunction<Optional<MyContainer1>, Void>(){
        @Override
        public ListenableFuture<Void> apply(Optional<MyContainer1> myData) throws Exception {
            // 首先判断 myData 是否存在，并加入相应处理或返回语句
            if( myData.isPresent()){
                System.out.println("MyContainer1's Data Does Exist!");
            }else{
                return null;
            }
            // 分别针对情况提交操作或处理失败场景
            if( bResultOfCommitFuture == false){
            //处理失败场景
                return Futures.immediateFailedCheckedFuture(new \
                  TransactionCommitFailedException("", \
                  RpcResultBuilder.newWarning(ErrorType.APPLICATION, \
                    "in-use", "This happens when CommitFuture fails")));
            }else{
            /*
```

```
                    提交改动的 my-container1 数据，这里将 my-container1 的叶子节点 mycontainer1-leaf1
                    赋值为 0，将孙子叶子节点 my-cont-grp-leaf1 赋值为 88888。注意赋值可连续使用
                    .setXXX 完成
                    */
                        myRWtx.put(LogicalDatastoreType.OPERATIONAL, \
                      MyContainer1_ID,  new MyContainer1Builder().\
                      setMycontainer1Leaf1((byte)0).setMyContGrpLeaf1(88888)\
                      .build());
                        return myRWtx.submit();
                    }
            }       // ListenableFuture<Void> apply(…)结束
        }           // AsyncFunction<Optional<MyContainer1>, Void>()结束
        );          // Futures.transform 结束
        //添加 callback 函数，根据事务操作是否成功进行后续处理
        Futures.addCallback(commitFuture, new FutureCallback<Void>(){
            // 若更新 data store 成功，则进行以下操作
            @Override
            public void onSuccess(Void result) {
                LOG.info("ListenableFuture's Commit: Success!", result.toString());
                System.out.println("ListenableFuture's Commit: Success!");
            }       // onSuccess 语句结束
            // 若更新失败，则进行以下操作
            @Override
            public void onFailure(Throwable t) {
                // TODO Auto-generated method stub
                LOG.debug("ListenableFuture's Commit: Fails", t);
                System.out.println("ListenableFuture's Commit: Fails");
            }       // onFailure 语句结束
        }           // FutureCallback<Void>()语句结束
        );          // Futures.addCallback(…)语句结束
        // 以下原来的 my-rpc1 的返回语句不变
        return Futures.immediateFuture(RpcResultBuilder.<Void> \
                success().build());
    }
```

16.2.2　传递 DataBroker 参数

在 MyprojectProvider 类中，向类 MyRPCImpl 传递 DataBroker 参数。

首先引用函数所需的 DataBroker 类：

```
import org.opendaylight.controller.md.sal.binding.api.DataBroker;
```

然后在 MyprojectProvider 中创建 MyRPCImpl 类，并将 DataBroker 参数传递给 MyRPCImpl。我们只需要改动 onSessionInitiated(ProviderContext session)函数。原函数的代码为：

```
    public void onSessionInitiated(ProviderContext session) {
        LOG.info("MyprojectProvider Session Initiated");
```

```
        rpcRegistration = session.addRpcImplementation(MyprojectService.class, new MyRPCImpl());
    }
```

我们需要对 RPC 注册的 MyRPCImpl 实例进行改动。将以上代码改成：

```
    @Override
    public void onSessionInitiated(ProviderContext session) {
        LOG.info("MyprojectProvider Session Initiated");
        /*
        在之前的 MyRPCImpl 类中已经定义了构建函数 MyRPCImpl(DataBroker dataBroker)，
        以 MyprojectProvider 类在创建 MyRPCImpl 实例时传递 DataBroker 参数
        */
        //从 session 中获取 DataBroker 参数
        DataBroker myDataBroker = session.getSALService(DataBroker.class);
        //创建 MyRPCImpl 实例 myRPCImpl，并传递 DataBroker 参数
        MyRPCImpl myRPCImpl = new MyRPCImpl(myDataBroker);
        /* RPC 的注册需要使用带有 myDataBroker 参数的 MyRPCImpl 实例 myRPCImpl 来进行，
        使用以下语句来实现原来的 RPC 注册
        */
        rpcRegistration = \
            session.addRpcImplementation(MyprojectService.class, myRPCImpl);
    }
```

16.2.3　测试验证

进入项目目录后，输入以下命令，成功启动项目：

```
$ sudo karaf/target/assembly/bin/karaf
```

使用以下命令查看日志，可见 myproject 项目成功启动（见图 16-1）：

```
opendaylight-user@root> log:display | grep myproject
```

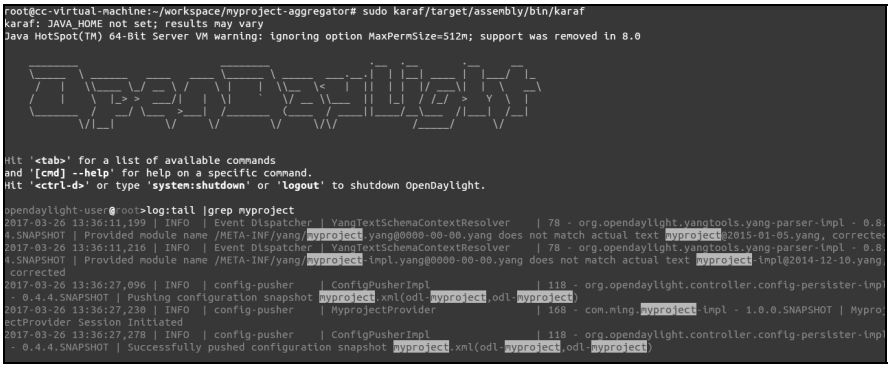

图 16-1　成功启动 myproject 项目

打开浏览器，访问 OpenDaylight 总控台的 YANG UI 界面，输入登录用户名/密码，可见项目成功加载，如图 16-2 所示。由于我们操作的是 OPERATIONAL 数据库，因此单击 operational 节点下面的 my-container1 节点。

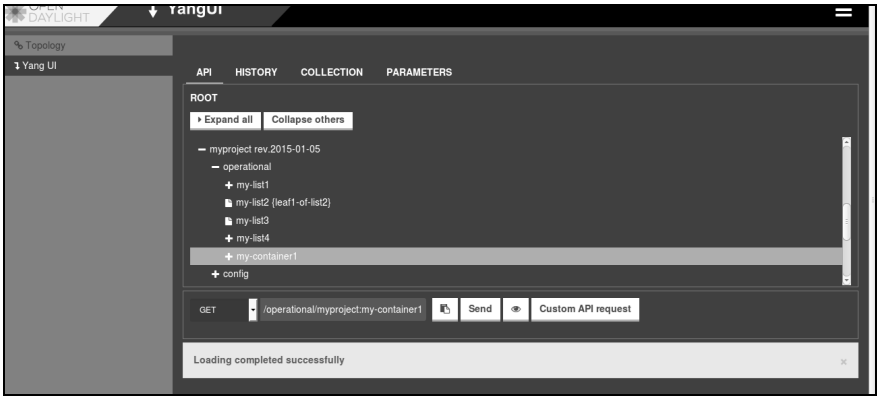

图 16-2　访问 OpenDaylight 总控台的 YANG UI 的 API 界面

此时选择 GET 参数后，单击带有眼睛的图标，可见待发送的 URL 地址及内容（这里内容为空），单击 Send 发送。由于未调用 rpc1（myRpc1 函数），因此此时 my-container1 的值为空。返回操作失败，提示可见数据为空，这符合实验的现状，如图 16-3 所示。

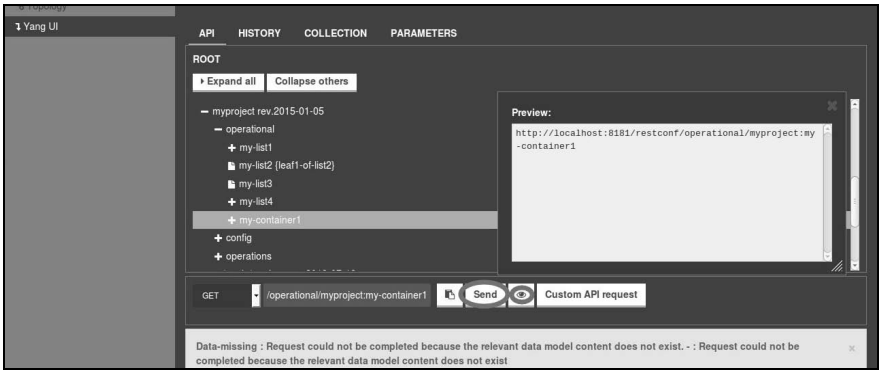

图 16-3　无法获取 my-container1 的值

单击 operations 节点下的 my-rpc1 节点，然后单击 Send 按钮，调用 rpc1（myRpc1 函数）赋值 my-container1。返回执行结果 Request sent successfully，如图 16-4 所示。

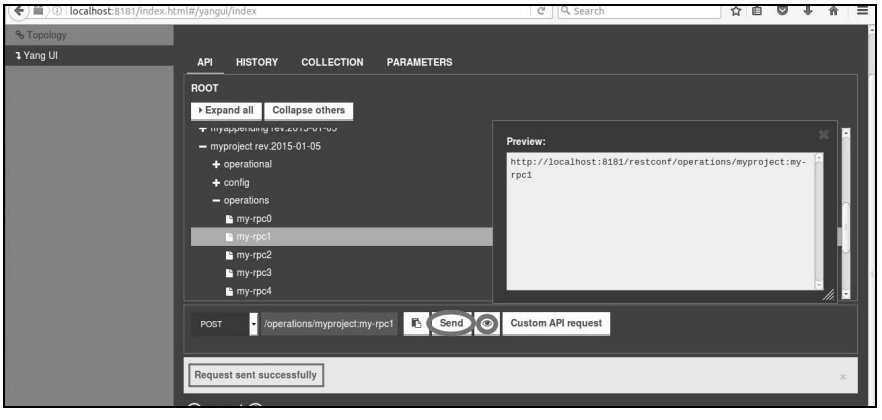

图 16-4　调用 rpc1（myRpc1 函数）赋值 my-container1

同时终端显示异步写操作事务成功完成，如图 16-5 所示。

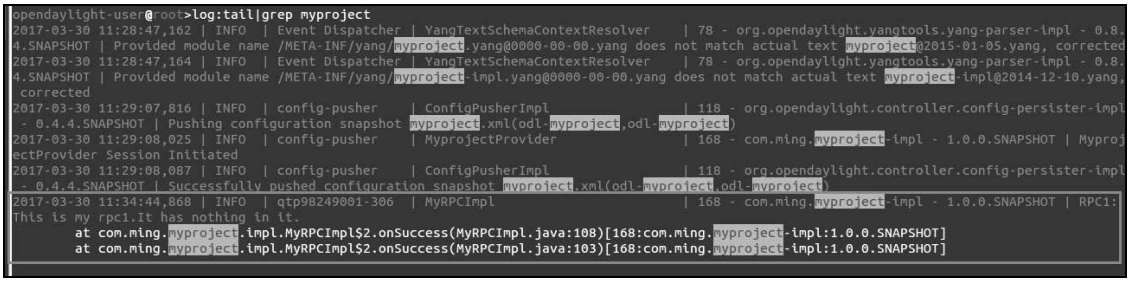

图 16-5　异步写操作事务成功完成

随后终端打印出程序所设定的打印值。返回后，单击 operational 节点下面的 my-container1 节点。选择 GET 参数后，单击 Send 发送。此时已调用 rpc1（myRpc1 函数），返回 my-container1 的值。可见 mycontainer1-leaf1 的值为 0，正如程序设定，利用 DataBroker 实现对 DataStore 的操作实验成功完成，如图 16-6 所示。

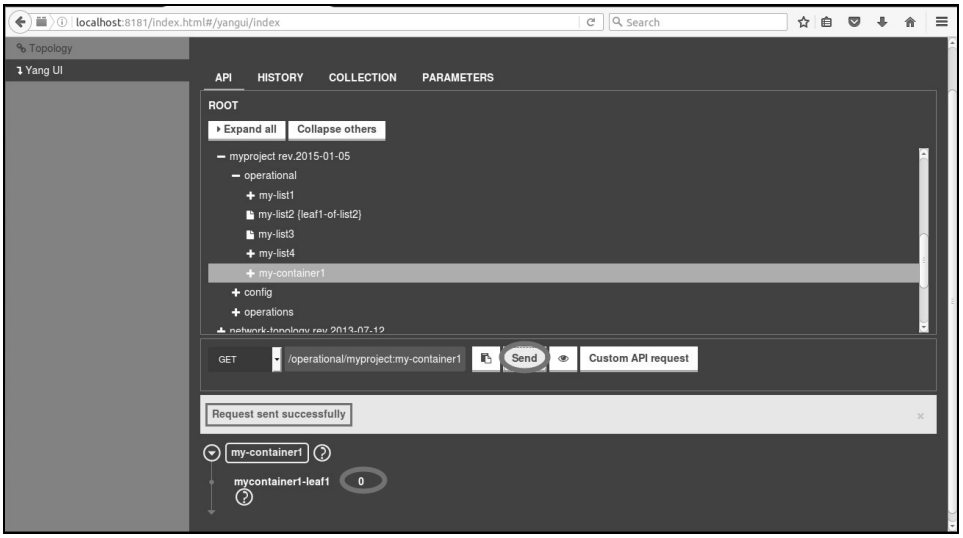

图 16-6　利用 DataBroker 实现对 DataStore 的操作成功完成

16.3　Data Change 事件的实现

16.3.1　实现 DataChangeListener 接口完成 onDataChange 函数

在 myproject 项目的子项目 impl 的 MyRPCImpl 类中实现 DataChangeListener 接口完成 onDataChange 函数。读者可将此操作放置在本项目的其他类甚至其他项目 MD-SAL 的类中的某函数内实现，只需引用的操作涉及相关类并在 MyprojectProvider 类的 onSessionInitiated(ProviderContext session)函数中注册相应的 P path（监听节点的身份标识 InstanceIdentifier）和 DataChangeScope triggeringScope（指定数据树变动通知执行的类）。

自动引用 onDataChange 函数所需的类 DataChangeListener 和 DataObject：

```
import org.opendaylight.controller.md.sal.binding.api.DataChangeListener;
import org.opendaylight.yangtools.yang.binding.DataObject;
```

需要在类 MyRPCImpl 中实现 DataChangeListener 接口，以使用 onDataChanged (AsyncDataChangeEvent <InstanceIdentifier<?>, DataObject> change)函数。

```
public class MyRPCImpl implements MyprojectService, DataChangeListener {
```

实现 DataChangeListener 接口后，eclipse 会自动添加 onDataChanged 函数。在以下代码中完成数据变化时实现的功能：

```
    @Override
    public void onDataChanged(AsyncDataChangeEvent<InstanceIdentifier<?>, \
            DataObject> change) {
        // 获取变化的数据对象
        DataObject myDO = change.getUpdatedSubtree();
        if( myDO instanceof MyContainer1){
        // 若变化的数据对象为容器 MyContainer1 的一个实例
            LOG.info("The Change is an instance of MyContainer1");
            // 获取容器 MyContainer1 的实例
            MyContainer1 myContainer1 = (MyContainer1)myDO;
            /*
            以下语句为数据变动时的处理，读者可编写自己的语句。
            本书这里设定打印 2 行语句。第一行为：
            MyContainer1's mycontainer1-leaf1 is  加上 mycontainer1-leaf1 的值
            第二行为：
            MyContainer1's my-container1-lflst1's first element is 加上 my-container1-lflst1 的值
            */
            System.out.println("MyContainer1's mycontainer1-leaf1 is " + \
                    myContainer1.getMycontainer1Leaf1().toString());
            List<Integer> myLfLst = myContainer1.getMyContainer1Lflst1();
            System.out.println("MyContainer1's my-container1-lflst1's first element is " + myLfLst.get(0));
        }else{
            LOG.info("The Change is NOT an instance of MyContainer1!");
            System.out.println("The Change is NOT an instance of MyContainer1!");
        }

    }
```

16.3.2　将数据树变动的监听注册到 MD-SAL

在 MyprojectProvider 类中，将数据变动 ListenerRegistration<DataChangeListener>的监听成员变量 dataChangeListenerReg 注册到 MD-SAL。

首先引用监听数据树变动监听所需的相关类：

```
import org.opendaylight.yangtools.concepts.ListenerRegistration;
import org.opendaylight.controller.md.sal.binding.api.DataChangeListener;
import org.opendaylight.controller.md.sal.common.api.data.AsyncDataBroker.DataChangeScope;
import org.opendaylight.controller.md.sal.common.api.data.LogicalDatastoreType;
```

然后在 MyprojectProvider 类中添加数据变动 ListenerRegistration<DataChangeListener>的监听成员变量 dataChangeListenerReg：

```
private ListenerRegistration<DataChangeListener> dataChangeListenerReg;
```

在 MyprojectProvider 类中的 onSessionInitiated(ProviderContext session) 函数中注册 dataChangeListenerReg。注册函数为：

```
registerDataChangeListener(LogicalDatastoreType store, P path, L listener, DataChangeScope
triggeringScope, "Selector selector");
```

其中 LogicalDatastoreType 为 DataStore 中存储的数据类型，值为 Configuration 或 Operational；P path 为监听节点的身份标识 InstanceIdentifier，使用在 MyRPCImpl 类中定义的指向容器 my-container1 的成员变量 myContainer1_ID；DataChangeScope triggeringScope 指定数据树变动通知执行的类，如 myRPCImpl 实例；Selector selector 为数据监听的范围，如 BASE 只监听当前节点变化事件，ONE 监听当前节点和其左右儿子节点变化事件，SUBTREE 监听当前节点和左右子树的所有变化事件，本实验选择监听当前节点和左右子树的所有变化事件，即将值设置为 DataChangeScope.SUBTREE。综上，在 onSessionInitiated(ProviderContext session)函数中注册 dataChangeListenerReg：

```
dataChangeListenerReg = myDataBroker.registerDataChangeListener
  (LogicalDatastoreType.CONFIGURATION, myRPCImpl.MyContainer1_ID, myRPCImpl,
  DataChangeScope.SUBTREE);
```

最后在 close 方法中加入对 dataChangeListenerReg 的注销。即在 close()函数中加入：

```
if(dataChangeListenerReg != null){
        dataChangeListenerReg.close();
}
```

16.3.3 测试验证

进入项目目录后，输入以下命令，成功启动项目：

```
$ sudo karaf/target/assembly/bin/karaf
```

使用以下命令查看日志，可见 myproject 项目成功启动：

```
opendaylight-user@root> log:display | grep myproject
```

打开浏览器，访问 OpenDaylight 总控台的 YANG UI 界面，输入登录用户名/密码，可见项目成功加载。监听变动的数据库为 CONFIG 数据库，单击 config 节点下面的 my-container1 节点，打开页面，如图 16-7 所示。

选择 GET 参数后，单击 Send 发送。显示当前 my-container1 的值，如图 16-8 所示。

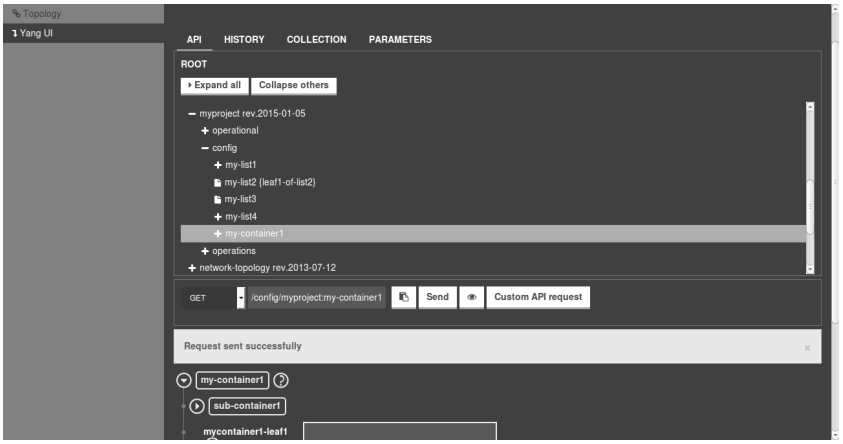

图 16-7　访问 OpenDaylight 总控台的 YANG UI 的 CONFIG 数据界面

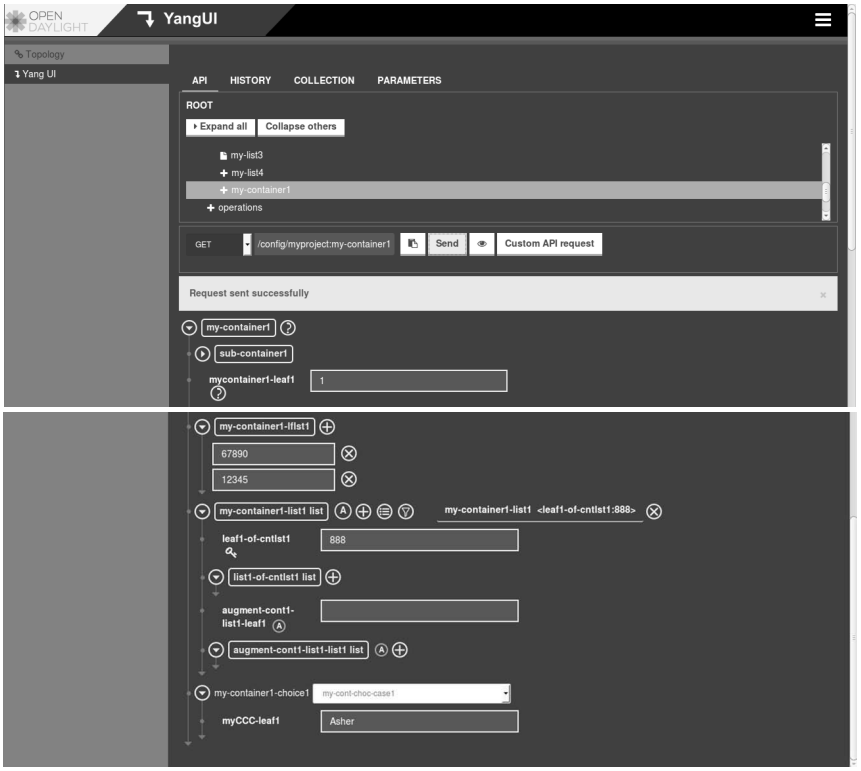

图 16-8　当前 my-container1 的值（续）

将容器 my-container1 的 leaf-list 子节点的第一个元素 my-container1-lflst1 的值改为 777777，单击 Send 按钮发送，如图 16-9 所示。

此时终端显示以下 2 行打印，如图 16-10 所示。

```
MyContainer1's mycontainer1-leaf1 is 1
MyContainer1's my-container1-lflst1's first element is 7777777
```

这个与 onDataChanged 中设定的功能相同，Data Change 事件成功执行。

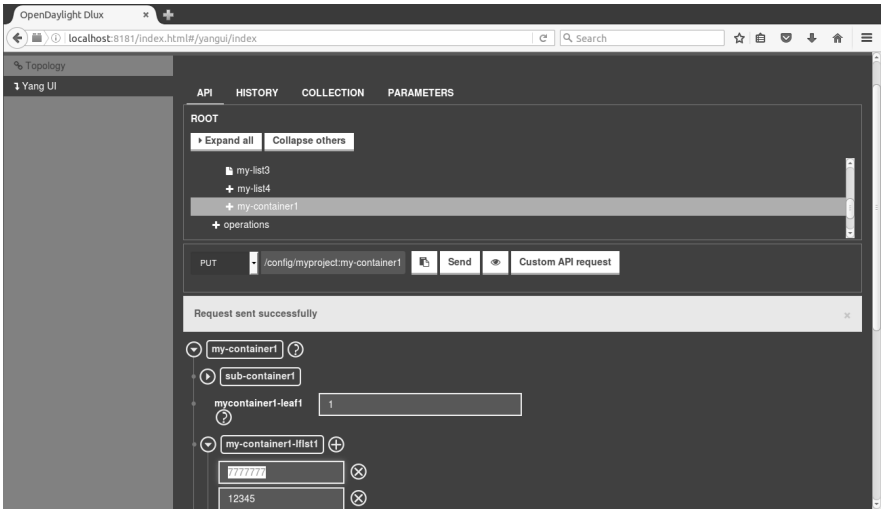

图 16-9 改动 my-container1 的值

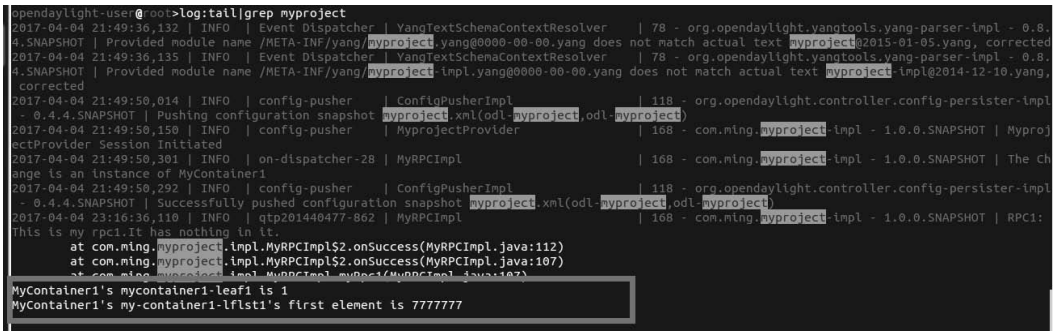

图 16-10 终端显示 Data Change 事件

16.4 本章总结

OpenDaylight 项目将所有数据都保存在 DataStore 中，并以树形结构进行存储。可通过对所存储的数据进行直接操作以获取更为底层的信息，或进行更底层的操作，同时也是验证读者自身项目的一个极好的方式。

OpenDaylight 项目的核心模块 MD-SAL 中通过数据代理 DataBroker 实现对 DataStore 中数据的操作。本章在 16.2 节重点介绍了利用 DataBroker 实现对 DataStore 的操作，这是 OpenDaylight 项目访问 DataStore 推荐的方式。

当 DataStore 中的数据发生变动时，会触发 Data Change 事件。OpenDaylight 项目中有 3 种方式可对事件进行监听：DataChangeListener、DataTreeChangeListener 和 DOMDataTreeChangeListener。其中 DataChangeListener 是最简单的方式，本章在 16.3 节中以 DataChangeListener 为例进行了介绍，其他两种方式读者可参考 16.1 节的内容自行实验完成。

本章尽量选用通用示例进行实验说明，读者可在本章的实验基础上简单地使用自己定义的类名/方法名/成员名对示例中相应的部分进行替换，快速地实现自己的项目。