0.90 源码解析

◆package net.floodlightcontroller.core

public class FloodlightContext{}
这是一个上下文对象，可以被 floodlight 的 listener 所注册，之后检索与事件相关的上下文信息。

public enum HAListenerTypeMarker{}
这是一个空标记。 IListener 通过类型强制调用顺序。然而对于 IHAListeners 我们只有一个单一的顺序。因此我们使用这种类型作为占位符，以满足通用要求。

public interface IFloodlightProviderService **extends**
IFloodlightService, Runnable {
由核心包暴露的接口，它允许你与已连接的交换机进行交互。

public static final String *CONTEXT_PI_PAYLOAD*
存储在 floodlight 上下文中的一个数值，包含一个对 PACKET_IN 消息携带的数据解析后的表示。

public static enum Role
在 OF1.2、OVS 故障切换和负载平衡机制中所使用的控制器的角色。

public static final FloodlightContextStore<Ethernet> *bcStore*
一个 FloodlightContextStore 对象，该对象可以被用于获取 PACKET_IN 消息携带的数据。

public void addOFMessageListener(OFType type, IOFMessageListener listener);
增加一个 openflow 消息监听器。
@param type:想监听的 openflow 消息类型
@param listener: 需要的 IOFMessageListener 监听器。

public void removeOFMessageListener(OFType type, IOFMessageListener listener);
移除一个 openflow 消息监听器。
@param type:不再想监听的消息类型。
@param listener:不再需要的 IOFMessageListener 监听器。

public Map<OFType, List<IOFMessageListener>> getListeners();
返回一个当前所有 listener 的不可修改的列表。
@return listeners

public IOFSwitch getSwitch(**long** dpid);
如果具有给定 DPID 的交换机是被集群中的任何控制器所已知的，则此方法返回其所关联的 IOFSwitch 实例。因此返回的交换机不一定是已连接的或处于本地 MASTER 角色控制器下的。
多次用同一 DPID 调用该方法可能返回不同的 IOFSwitch 引用。调用者不得视 IOFSwitch 引用在交换机生命周期中为恒定的而储存或以其他方式依赖它。
@param dpid:想要查询交换机的 DPID。
@return 与 DPID 相关的 IOFSwitch 实例，如果在集群中没有已知的与 DPID 匹配的交换机则返回 null。

```
public Set<Long> getAllSwitchDpids();
```
返回所有已知交换机的一个 DPID 集合快照。
返回的集合是被调用者所拥有的：调用者可以随意修改它，加入没有反映在返回集合中的已知的交换机，如果版本进行了更新，则调用者需要调用 getAllSwitchDpids();
@return 所有已知交换机的 DPID 的集合。

```
public Map<Long,IOFSwitch> getAllSwitchMap();
```
返回一个快照。
FIXME:(in floodlight0.90)
@return

```
public Role getRole();
```
获取当前控制器的角色。

```
public RoleInfo getRoleInfo();
```
获取当前控制器的角色信息。

```
public Map<String,String> getControllerNodeIPs();
```
获取控制器当前 ID 到其 IP 地址的映射，返回一个当前映射的副本。
@see IHAListener

```
public void setRole(Role role, String changeDescription);
```
设置控制器的角色。
@param role:控制器的新角色。
@param changeDescription:进行角色变化的原因或其他信息。

```
public void addOFSwitchListener(IOFSwitchListener listener);
```
添加一个交换机监听器。
@param listener: IOFSwitchListener 监听器。

```
public void removeOFSwitchListener(IOFSwitchListener listener);
```
移除一个交换机监听器。
@param listener:不再需要的 IOFSwitchListener 监听器。

```
public void addHAListener(IHAListener listener);
```
添加一个 HA role 事件监听器。
@param listener: IHAListener 监听器。

```
public void removeHAListener(IHAListener listener);
```
移除一个 HA role 事件监听器。
@param listener:不再需要的 IHAListener 监听器。

```
public void addReadyForReconcileListener(IReadyForReconcileListener l);
```

添加一个 ready-for-flow-reconcile 事件监听器。
@param l:IReadyForReconcileListener 监听器。

```
public void terminate();
```
终止进程。

```
public boolean injectOfMessage(IOFSwitch sw, OFMessage msg);
```
把一个 OFMessage 重新注入包处理链中。
@param sw: 使用该消息的交换机。
@param msg:要注入的消息。
@return:如果注入成功则返回 TRUE，否则返回 FALSE。
@throws: NullPointerException 如果 switch 或 msg 为 null。

```
public boolean injectOfMessage(IOFSwitch sw, OFMessage msg,
          FloodlightContext bContext);
```
把一个OFMessage重新注入包处理链中。
@param sw: 使用该消息的交换机。
@param msg:要注入的消息。
@param bContext:如有需要时使用的floodlight上下文，可以为null。
@throws: NullPointerException 如果 switch 或 msg 为 null。

```
public void handleOutgoingMessage(IOFSwitch sw, OFMessage m,
         FloodlightContext bc);
```
进程产生的消息，来源于控制器的消息监听器。
@param sw:消息要送达的目的交换机。
@param m:消息。
@param bc: 任何所附的上下文对象。可以为null，此时一个新的上下文对象将被分配并传递给监听器。
@throws: NullPointerException 如果 switch 或 msg 为 null。

```
public BasicFactory getOFMessageFactory();
```
获取BasicFactory。
@return:一个openflow消息factory。

```
@Override
public void run();
```
运行控制器的主要的I/O循环。

```
public void addInfoProvider(String type, IInfoProvider provider);
```
添加特定类型的信息提供者。
@param type:
@param provider:

```
public void removeInfoProvider(String type, IInfoProvider provider);
```

移除特定类型的信息提供者。
@param type:
@param provider:

```
public Map<String, Object> getControllerInfo(String type);
```
返回一个特定类型的信息（REST服务）
@param type:
@return

```
public long getSystemStartTime();
```
返回以毫秒为单位的控制器启动时间.
@return

```
public void setAlwaysClearFlowsOnSwActivate(boolean value);
```
配置当交换机连接到控制器后控制器总是清除交换机上的流表。当交换机第一次重连后它将为真，以及当HA交换机切换为ACTIVE角色并重连控制器之后。

```
public Map<String, Long> getMemory();
```
获取控制器的内存信息。

```
public Long getUptime();
```
返回此控制器的正常运行时间。

```
public void addOFSwitchDriver(String desc, IOFSwitchDriver driver);
```
增加一个OFSwitch驱动。
@param desc:用以注册的驱动前缀。
@param driver: 一个用来处理IOFSwitch实例化的IOFSwitchDriver实例，具有给定的生产者说明驱动前缀。
@throws IllegalStateException:如果生产者说明前缀已被注册。
@throws NullPointerExeption:如果desc为null。
@throws NullPointerExeption:如果driver为null。

```
public void addSwitchEvent(long switchDPID, String reason, boolean flushNow);
```
在内存调试事件中记录一个交换机事件。
@param switchDPID:
@param reason:此事件发生的原因。
@param flushNow: 参见IDebugEventService中的debug-event flushing

```
public Set<String> getUplinkPortPrefixSet();
```
获取一个端口的前缀的集合，这将定义一个上行端口。
@return:端口前缀的集合。

```
//end of interface IFloodlightProviderService }
```

```
public interface IHAListener extends IListener<HAListenerTypeMarker> {

public void transitionToMaster();
```

如果控制器的初始角色是SLAVE并且现在转变为MASTER时它将被触发。模块需要从floodlight provider的startUp处读取初始的角色信息。

```
public void controllerNodeIPsChanged(
            Map<String, String> curControllerNodeIPs,
            Map<String, String> addedControllerNodeIPs,
            Map<String, String> removedControllerNodeIPs);
```

当控制器集群中的控制器节点IP地址发生改变时被调用，所有参数均为控制器ID到其IP地址的映射。
@param curControllerNodeIPs:当前的控制器ID和其IP地址映射。
@param addedControllerNodeIPs:自上次更新之后新增的IP。
@param removedControllerNodeIPs:自上次更新之后被移除的IP。

```
//end of interface IHAListener}

public interface IInfoProvider {

public Map<String, Object> getInfo(String type);
```

当REST API请求一个特定类型的信息时被调用。
@param type:

```
//end of interface IInfoProvider}

public interface IListener<T> {

public enum Command {
        CONTINUE, STOP
}

public String getName();
```

指定给此监听器的名称。
@return:

```
public boolean isCallbackOrderingPrereq(T type, String name);
```

检查名为name的模块是否是给这个模块执行回调命令的前提，换句话说，如果该方法对于给定的名称返回true，那么这个监听器会在该消息监听器后调用。
@param type:适用的对象类型。
@param name:模块名称。
@return:该模块是否为一先决条件。

```
public boolean isCallbackOrderingPostreq(T type, String name);
```

检查名为name的模块是否是这个模块执行回调命令的后继条件，换句话说，如果该方法对于给定的名

称返回true，那么这个监听器会在该消息监听器之前调用。
@param type:适用的对象类型。
@param name:模块名称。
@return:该模块是否为一后继条件。

```
//end of interface IListener<T>}
```

```
public class ImmutablePort{
```

一个不可变版本的OFPhysical端口。此外，它使用EnumSets而不是整数的Bitmaps来表示OFPortConfig，OFPortState和OFPortFeature。
端口名称用原始的大小写存储，但equals()和XXXX使用不区分大小写的端口名称来比较！
TODO：创建一个生成器，用于我们可以很容易地构造OFPhysicalPorts。

```
    private final short portNumber;
    private final byte[] hardwareAddress;
    private final String name;
    private final EnumSet<OFPortConfig> config;
    private final boolean portStateLinkDown;
    private final OFPortState stpState;
    private final EnumSet<OFPortFeatures> currentFeatures;
    private final EnumSet<OFPortFeatures> advertisedFeatures;
    private final EnumSet<OFPortFeatures> supportedFeatures;
    private final EnumSet<OFPortFeatures> peerFeatures;
```

```
public static class Builder{}
```

构建器类用来创建ImmutablePort实例。

```
private ImmutablePort(short portNumber, byte[] hardwareAddress,
                      String name, EnumSet<OFPortConfig> config,
                      boolean portStateLinkDown,
                      OFPortState portStateStp,
                      EnumSet<OFPortFeatures> currentFeatures,
                      EnumSet<OFPortFeatures> advertisedFeatures,
                      EnumSet<OFPortFeatures> supportedFeatures,
                      EnumSet<OFPortFeatures> peerFeatures)
```

类构造函数。

```
public static ImmutablePort fromOFPhysicalPort(OFPhysicalPort p)
```

将OFPhysicalPort转换为ImmutablePort。

```
public static ImmutablePort create(String name, Short portNumber)
```

创建一个ImmutablePort实例。

访问器：

`public short getPortNumber()`	获取端口号
`public byte[] getHardwareAddress()`	获取端口硬件地址
`public String getName()`	获取端口名称(for listener?)
`public Set<OFPortConfig> getConfig()`	获取端口配置信息
`public PortSpeed getCurrentPortSpeed()`	获取当前端口速率
`public OFPortState getStpState()`	获取当前端口的STP状态
`public Set<OFPortFeatures> getCurrentFeatures()`	见OPFortFeatures
`public Set<OFPortFeatures> getAdvertisedFeatures()`	见OPFortFeatures
`public Set<OFPortFeatures> getSupportedFeatures()`	见OPFortFeatures
`public Set<OFPortFeatures> getPeerFeatures()`	见OPFortFeatures

设置器：

`public Builder setPortNumber(short portNumber)`	设置端口号
`public Builder setHardwareAddress(byte[] hardwareAddress)`	设置端口硬件地址
`public Builder setName(String name)`	设置端口名称
`public Builder addConfig(OFPortConfig config)`	新增端口配置信息
`public Builder setPortStateLinkDown(boolean portStateLinkDown)`	设置端口状态(UP/DOWN)
`public Builder setStpState(OFPortState stpState)`	设置端口STP状态
`public Builder addCurrentFeature(OFPortFeatures currentFeature)`	见OPFortFeatures
`public Builder addAdvertisedFeature(OFPortFeatures advertisedFeature)`	见OPFortFeatures
`public Builder addSupportedFeature(OFPortFeatures supportedFeature)`	见OPFortFeatures
`public Builder addPeerFeature(OFPortFeatures peerFeature)`	见OPFortFeatures

转换器：

`public OFPhysicalPort toOFPhysicalPort()`	将当前端口转换为OFPhysicalPort
`public String toBriefString()`	提取当前端口中name和portNumber字段并格式化为一String返回
`public String toString()`	将当前端口中所有字段组合成一String返回。

判别器：

`public boolean isLinkDown()`	返回TRUE如果端口状态为down
`public boolean isEnabled()`	返回TRUE如果端口状态为up。
`public boolean equals(Object obj)`	比较两个ImmutablePort对象是否相同

```
public static List<ImmutablePort>
          immutablePortListOf(Collection<OFPhysicalPort> ports)
```
将参数中OFPhysicalPorts的集合转换为ImmutablePorts列表，在该集合中所有OFPhysicalPorts必须为非空和有效的。该方法不执行其它（端口名称/端口编号）的唯一性检查。
@param ports:
@return:一个ImmutablePort列表，该列表由调用者所有，返回的列表不是线程安全的。
@throws NullPointerException: 如有OFPhysicalPort或任何OFPhysicalPort的重要字段为空。
@throws IllegalArgumentException

```
public static List<OFPhysicalPort>
```
将参数中ImmutablePorts的集合转换为OFPhysicalPorts列表，在该集合中所有ImmutablePorts必须为非空和有效的。该方法不执行其它（端口名称/端口编号）的唯一性检查。
@param ports:
@return:一个OFPhysicalPorts列表，该列表由调用者所有，返回的列表不是线程安全的。
@throws NullPointerException: 如有ImmutablePorts或端口列表为空。
@throws IllegalArgumentException

```
//end of class ImmutablePort}

public interface IOFMessageFilterManagerService extends IFloodlightService{
```
过滤器管理服务。
```
String setupFilter(String sid, ConcurrentHashMap<String, String> f,
          int deltaInMilliSeconds);

//end of interface IOFMessageFilterManagerService }

public interface IOFMessageListener extends IListener<OFType>{
```
openflow消息监听器。
```
public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx);
```
Floodlight用来调用openflow消息监听器的方法。
@param sw:发送此openflow消息的交换机。
@param msg:消息。
@param cntx:一个Floodlight上下文对象，用于在监听器之间传递消息。
@return: 继续或停止执行的命令。

```
//end of interface IOFMessageListener }

public interface IOFSwitch {
    public static final String SWITCH_DESCRIPTION_FUTURE = "DescriptionFuture";
    public static final String SWITCH_SUPPORTS_NX_ROLE = "supportsNxRole";
    public static final String SWITCH_IS_CORE_SWITCH = "isCoreSwitch";
    public static final String PROP_FASTWILDCARDS = "FastWildcards";
    public static final String PROP_REQUIRES_L3_MATCH = "requiresL3Match";
```

```
    public static final String PROP_SUPPORTS_OFPP_TABLE = "supportsOfppTable";
    public static final String PROP_SUPPORTS_OFPP_FLOOD = "supportsOfppFlood";
    public static final String PROP_SUPPORTS_NETMASK_TBL = "supportsNetmaskTbl";

public enum OFPortType {
        NORMAL("normal"),         // normal port (default)
        TUNNEL("tunnel"),         // tunnel port
        UPLINK("uplink"),         // uplink port (on a virtual switch)
        MANAGEMENT("management"), // for in-band management
        TUNNEL_LOOPBACK("tunnel-loopback");
```

枚举OF端口类型。

```
OFPortType(String v)
```

构造方法。

```
public String toString()
@override
```

```
public static OFPortType fromString(String str)
```

返回values域和str相等的OFPortType值。如找不到匹配则返回NORMAL。

```
}
```

```
public static class PortChangeEvent{
```

描述一个openflow端口变化事件。

```
public PortChangeEvent(ImmutablePort port,PortChangeType type)
```

构造方法。

```
public String toString()
@override
}
```

```
public enum PortChangeType {ADD, OTHER_UPDATE, DELETE, UP, DOWN,}
```

枚举openflow端口变化的类型。

设置器：

`public void setFloodlightProvider(Controller controller);`	为此交换机实例设置IFloodlightProviderService，在交换机被实例化之后立即被调用。
`public void setThreadPoolService(IThreadPoolService threadPool);`	为此交换机实例设置IThreadPoolService，在交换机被实例化之后立即被调用。

public void setDebugCounterService(IDebugCounterService debugCounters) throws CounterException;	用于给每个交换机设置调试计数器服务，在交换机被实例化之后立即被调用。
public void setChannel(Channel channel);	为此交换机关联的实例设置Netty Channel。
public void setFeaturesReply(OFFeaturesReply featuresReply);	设置初始握手过程中由交换机返回的OFFeaturesReply消息
public OrderedCollection<PortChangeEvent> setPorts(Collection<ImmutablePort> ports);	用给定的端口替换该交换机上的端口。
public void setConnected(boolean connected);	设置是否对交换机进行连接
public void setHARole(Role role);	设置HA switch的角色，haRoleReplyReceived表示已从交换机收到回复信息（除错误回复外）。如果role为null，则交换机应该关闭此信道连接。
void setAttribute(String name, Object value);	为交换机的特定行为设置属性。
public void setSwitchProperties(OFDescriptionStatistics description);	基于交换机的自述设置SwitchProperties字段信息。
public void setTableFull(boolean isFull);	为交换机设置流表已满的标志
public void setAccessFlowPriority(short prio);	设置存取流优先级
public void setCoreFlowPriority(short prio);	设置核心流优先级

访问器：

public int getBuffers();	从features Reply报文中返回交换机功能信息。
public int getActions();	
public int getCapabilities();	
public byte getTables();	
public OFDescriptionStatistics getDescriptionStatistics();	获取一个对这台交换机描述性统计的副本
public Collection<ImmutablePort> getEnabledPorts();	获取当前活动端口的一个不可修改列表。
public Collection<Short> getEnabledPortNumbers();	获取当前活动端口端口号的一个不可修改列表
public ImmutablePort getPort(short portNumber);	通过端口号获取端口对象
public ImmutablePort getPort(String portName);	通过端口名称获取端口对象
public Collection<ImmutablePort> getPorts();	获取所有端口的列表
public long getId();	获取此交换机的DPID
public String getStringId();	获取此交换机ID的String版本
public SocketAddress getInetAddress();	获取此交换机的IP地址
public Map<Object, Object> getAttributes();	获取此交换机的属性
public Date getConnectedSince();	获取此交换机连接到控制器的时间
public int getNextTransactionId();	获取下一个可用的事务id

public Role getHARole();	获取控制器对于此交换机当前的角色信息。
Object getAttribute(String name);	获取交换机特定行为的属性集。
public Map<Short, Long> getPortBroadcastHits();	获取端口广播缓存命中。
public OFPortType getPortType(short port_num);	获取OFPort类型
public short getAccessFlowPriority();	获取存取流(?)优先级。
public short getCoreFlowPriority();	获取核心流(?)优先级。

判别器：

public boolean inputThrottled(OFMessage ofm);	当OFMessage进入流水线被调用，如果msg被丢弃则返回TRUE。
boolean isOverloaded();	如果交换机过载则返回TRUE，过载的定义是控制通路上承载过多的数据流量。
public boolean portEnabled(short portNumber);	该端口是否被启用。（configured down、link down和生成树端口阻塞的情况不在此列）
public boolean portEnabled(String portName);	该端口是否被启用。（configured down、link down和生成树端口阻塞的情况不在此列）
public boolean isConnected();	当交换机连接到控制器之后检查此交换机是否还是连接状态。
public boolean isActive();	检查该交换机是否处于活动状态。
boolean hasAttribute(String name);	检查给定属性是否存在于这台交换机上。
boolean attributeEquals(String name, Object other);	检查给定的属性是否存在，如果是，该属性是否为“other”。
public boolean updateBroadcastCache(Long entry, Short port);	更新广播缓存。TRUE如果有一个缓存命中，FALSE如果没有缓存命中。
public boolean isFastPort(short port_num);	检查此端口打开后是否会形成一个新的环路。
public boolean isWriteThrottleEnabled();	检查此交换机上的写入限制是否启用。
public boolean isDriverHandshakeComplete();	检查sub-handshake动作是否已完成。此方法只能在startDriverHandshake()之后被调用。

```
public void writeThrottled(OFMessage msg, FloodlightContext cntx) throws
IOException;
```
将OFMessage写到输出流，受交换机速率限制。该消息将被消息监听器交予floodlightProvider进行可能的过滤和处理。
@param msg:
@param cntx:
@throws IOException:

```
void writeThrottled(List<OFMessage> msglist, FloodlightContext bc) throws
IOException;
```
将消息写入到输出流的列表中，受速率限制。该消息将被被消息监听器交予floodlightProvider进行可能的过滤和处理。
@param msglist:

@param bc:
@throws IOException:

public void write(OFMessage m, FloodlightContext bc) throws IOException;
写入OFMessage到输出流，绕过速率限制。该消息将被被消息监听器交予floodlightProvider进行可能的过滤和处理。
@param m:
@param bc:
@throws IOException:

public void write(List<OFMessage> msglist, FloodlightContext bc) throws IOException;
写消息列表到输出流，绕过速率限制。该消息将被被消息监听器交予floodlightProvider进行可能的过滤和处理。
@param msglist:
@param bc:
@throws IOException:

public void disconnectOutputStream();
关闭输出流连接。(?)

public OrderedCollection<PortChangeEvent> processOFPortStatus(OFPortStatus ps);
用于添加或修改交换机端口。
被核心控制器代码调用以响应OFPortStatus消息。它一般不应该被其他Floodlight应用程序所调用。OFPPR_MODIFY和OFPPR_ADD将被视为是等同的。OpenFlow的规范并没有明确portName是否是对应portNumbers的权威标识符。我们视portName<->PortNumber的映射为固定的。如果他们发生了改变，将删除所有以前冲突的端口，并添加所有的新端口。
@param ps:端口状态消息。
@return:根据PortStatus消息“应用”到交换机的旧端口的变化的有序集合。单个PortStatus消息可能会导致多个更改。如果portName<->portNumber的映射已经改变，变化的有序集合根据PortStatus消息“应用”到交换机的旧端口。单PortStatus消息可能会导致多个更改。如果PORTNAME< - >端口号映射已经改变，迭代顺序确保删除旧冲突的事件发生在添加新端口事件之前。

public OrderedCollection<PortChangeEvent>
comparePorts(Collection<ImmutablePort> ports);
为交换机的新旧端口替换计算更改。
@param ports:要设置的新端口。
@return:发生在旧端口上的更改后的有序集合。如果portName<->portNumber的映射已经改变，变化的有序集合根据PortStatus消息“应用”到交换机的旧端口。单PortStatus消息可能会导致多个更改。如果PORTNAME< - >端口号映射已经改变，迭代顺序确保删除旧冲突的事件发生在添加新端口事件之前。

public Future<List<OFStatistics>> queryStatistics(OFStatisticsRequest request)
throws IOException;
返回可用于检索异步OFStatisticsReply消息的Future对象当其为可用时。

@param request: statistics request。
@return: 包装了OFStatisticsReply消息的Future对象。
@throws IOException:

```
public Future<OFFeaturesReply> querySwitchFeaturesReply()
          throws IOException;
```

返回可用于检索异步OFStatisticsReply消息的Future对象当其为可用时。
@param request: statistics request。
@return: 包装了OFStatisticsReply消息的Future对象。
@throws IOException:

```
void deliverOFFeaturesReply(OFMessage reply);
```

交付OFFeaturesReply回复。
@param reply:要交付的回复。

```
public void cancelFeaturesReply(int transactionId);
```

取消给特定事务ID的特性回复。
@param transactionId:事务ID。

```
public void deliverStatisticsReply(OFStatisticsReply reply);
```

交付StatisticsReply回复。
@param reply:要交付的回复。

```
public void cancelStatisticsReply(int transactionId);
```

取消给特定事务ID的StatisticsReply回复。
@param transactionId:事务ID。

```
public void cancelAllStatisticsReplies();
```

取消所有的StatisticsReply回复。

```
Object removeAttribute(String name);
```

移除交换机特定属性。
@param name:属性名
@return: 当前name对应的value，或null（不存在时）

```
public void clearAllFlowMods();
```

清除此交换机上的所以FlowMods。

```
public void sendStatsQuery(OFStatisticsRequest request, int xid,
                           IOFMessageListener caller) throws IOException;
```

发送流统计请求到该交换机。发送统计后此调用会返回。
@param request:流统计请求消息。
@param xid: 事务ID，必须使用getXid() API来获得。
@param caller: API的调用者，当接收来自交换机的答复时，该调用者的receive()回调会被调用。

@return: 该消息发送到交换机的事务ID。事务ID用于在响应和请求之间进行匹配。请注意，事务ID只此交换机范围内是唯一的。
@throws IOException

public void flush();
对当前线程中在此交换机队列中等待的所有流量进行flush操作。

public void startDriverHandshake();
启动此交换机Driver的sub-handshake。这可能是一个空操作，但在交换机准备就绪之后该方法必须至少被调用一次。此方法只能从I/O线程调用。
@throws SwitchDriverSubHandshakeAlreadyStarted:如果sub-handshake已经启动。

public void processDriverHandshakeMessage(OFMessage m);
通过给定的OFMessage到驱动程序作为这个驱动程序的sub-handshake的一部分，不能在握手过程已经结束后调用，此方法只能从I/O线程调用。
@param m: 该驱动程序要处理的消息。
@throws SwitchDriverSubHandshakeCompleted: 如果isDriverHandshake()在此方法调用之前返回false。
@throws SwitchDriverSubHandshakeNotStarted: 如果startDriverHandshake()还未被调用完成。

//end of interface IOFSwitch}

public interface IOFSwitchDriver {
public IOFSwitch getOFSwitchImpl(OFDescriptionStatistics description);
返回基于OFDescriptionStatitics的交换机制造商描述的IOFSwitch对象。
@param description: 来自于交换机实例的DescriptionStatistics。
@return: 一个IOFSwitch实例。如果该驱动程序发现给定描述的实现，否则返回null。

//end of interface IOFSwitchDriver}

public interface IOFSwitchListener {
交换机事件监听者。由一个并且总由这个单独的线程来进行调用。
public void switchAdded(**long** switchId);
当交换机在控制器集群中变为已知时被触发，即交换机被与集群中的某些控制器建立连接。
@param switchId:新交换机的数据路径标识（DPID）

public void switchRemoved(**long** switchId);
当交换机与控制器集群中断开连接时被触发。
@param switchId:新交换机的数据路径标识（DPID）

public void switchActivated(**long** switchId);
当交换机在本地控制器中变为活动状态时被触发，即交换机连接到本地控制器，并处于MASTER模式。
@param switchId the datapath Id of the switch

```
public void switchPortChanged(long switchId,
                              ImmutablePort port,
                              IOFSwitch.PortChangeType type);
```

当交换机上的一个已知端口发生改变时被触发。
使用此通知的用户需要注意：如果端口和类型的信息被直接使用，并且如果端口的集合已被查询过，该通知将只在该端口更改已经提交给IOFSwitch实例之后被发出。但是，如果用户之前调用过IOFSwitch.getPorts()或相关方法进行更新，则可能已经存在于由getPorts返回的信息。
@param switchId
@param port
@param type

```
public void switchChanged(long switchId);
```

在一个交换机连接控制器之后的任何非端口信息（例如属性、特征）出现后被触发。
TODO:当前未使用
@param switchId:新交换机的数据路径标识（DPID）

```
//end of interface IOFSwitchListener}

public interface IReadyForReconcileListener {
```

这个监听器是一个临时性的解决方案被用来在SLAVE到MASTER过渡的过程中进行流调节。所有已知的交换机被激活（或者是因为它们都重新连接或者是因为它们没有重新连接并已超时）的时候会被触发一次。
模块一般不应依赖此通知，除非有强烈的和令人信服的理由要这样做。普通模块应该处理的事件是一些已知的交换机未启用。

```
public void readyForReconcile();

//end of interface IReadyForReconcileListener}

public class Main {

public static void main(String[] args) throws FloodlightModuleException {
```

main方法，用来加载配置和模块。
@param args:
@throws FloodlightModuleException:

设置logger

```
FloodlightModuleLoader fml = new FloodlightModuleLoader();
IFloodlightModuleContext moduleContext =
fml.loadModulesFromConfig(settings.getModuleFile());
```

加载模块。

```
IRestApiService restApi = moduleContext.getServiceImpl(IRestApiService.class);
restApi.run();
```

启动REST服务。

```
IFloodlightProviderService controller =
                moduleContext.getServiceImpl(IFloodlightProviderService.class);
```

启动Floodlight主模块。

```
controller.run();
```

该调用块必须在main方法的最后一行。

```
    }

//end of class Main}

public class OFMessageFilterManager
       implements IOFMessageListener, IFloodlightModule,
IOFMessageFilterManagerService {
```

OpenFlow消息过滤管理器类。

……

```
}

public abstract class OFSwitchBase implements IOFSwitch {
```

这是一个OpenFlow交换机的内部表示形式。

……

```
protected class PortManager {}
```

管理该交换机的各个端口。

提供方法来查询和更新存储的端口，该类确保每个端口名称和端口号是唯一的。当该类检查到端口号<->端口名称的映射由于更新发生了变化时会更新端口。如果一个新的端口P的号码和端口与以前的映射不一致，该类将删除所有以前的端口的名称和新的端口号，然后添加新端口。

端口名称是被照原样保存的，但比较的时候不区分大小写。

该方法更改已存储的端口返回一个PortChangeEvents的列表，该列表表示已经应用到端口列表中的变化，以便IOFSwitchListeners可以通知有关变更。

备注：

我们保留端口的一些不同的表示形式，以便快速查找。

端口被存储在不可变的列表中。当一个端口被修改时，新的数据结构将被分配。

我们使用读写锁用于同步，所以多个reader是被允许的。

```
private void updatePortsWithNewPortsByNumber(
        Map<Short,ImmutablePort> newPortsByNumber) {}
```

设置内部数据结构存储此交换机由newPortsByNumber指定的端口。

调用者必须持有写锁。

```
@param newPortsByNumber:
@throws IllegaalStateException:
```

如果调用者不持有写锁。

```
private OrderedCollection<PortChangeEvent>
```

handlePortStatusDelete(ImmutablePort delPort) {}

处理给定的端口OFPortStatus删除消息。更新这个交换机的内部端口映射/列表，并返回由于删除所引起的PortChangeEvents。如果给定的端口已存在，它将被删除。如果给定端口的name<->number映射与此交换机上存储的端口是不一致的，该方法将删除所有具有给定端口号或名称的端口。
此方法将会递增error/warning计数器和日志。
@param delPort:从端口状态信息中获取到的应该被删除的端口。
@return: 应用到此交换机端口变化的有序集合。

```
public OrderedCollection<PortChangeEvent> handlePortStatusMessage(OFPortStatus ps)
{}
```

处理OFPortStatus信息，更新存储端口的内部数据结构，并返回OFChangeEvents列表。
此方法将会递增error/warning计数器和日志。

```
public OrderedCollection<PortChangeEvent>
        getSinglePortChanges(ImmutablePort newPort) {}
```

给定一个新的或修改后的端口给新端口。返回PortChangeEvents列表来“转化”这个交换机存储的包含或代表了新端口的当前端口。通过此交换机存储的端口不被更新。
此方法获取写锁并且其本身是线程安全的。
大部分调用者在调用此方法前将需要获取写锁。（如果调用者想要更新该存储在交换机上的端口）
@param newPort:新的或修改后的端口
@return:更改列表

```
public OrderedCollection<PortChangeEvent>
        comparePorts(Collection<ImmutablePort> newPorts) {}
```

比较此交换机上当前端口与新端口列表并返回将被应用到当前端口转换到新端口的更改。无内部数据结构被更新。
@param newPorts:新端口列表。
@return:新旧端口差异列表。

```
public OrderedCollection<PortChangeEvent>
        updatePorts(Collection<ImmutablePort> newPorts) {}
```

比较此交换机上当前端口与新端口列表并返回将被应用到当前端口转换到新端口的更改。无内部数据结构被更新。
@param newPorts:新端口列表。
@return:新旧端口差异列表。

```
private OrderedCollection<PortChangeEvent> compareAndUpdatePorts(
        Collection<ImmutablePort> newPorts,boolean doUpdate) {}
```

比较给定的新的端口列表和存储在该交换机实例的当前端口，并在PortChangeEvents表中返回差异。如果doUpdate标志为true，newPortList将替换此交换机当前列表（以及更新端口映射）
备注：
由于该方法可任选地修改当前端口，并且因为它不可能把读锁升级为写锁，所以我们需要首先为整个操作获取写锁。如果这成为一个问题并且compares()是常用到的，我们可以考虑用两种方法进行拆分，但这需要大量的代码重复。

@param newPorts:新端口列表。
@param doUpdate:如果为TRUE，新端口列表将会取代该交换机上的旧端口列表，如果为FALSE，将不会发生变更。
@throws NullPointerException:如果新端口列表为null
@throws IllegalArgumentException: 如果新端口列表中任一端口名称或端口号是重复的

```
public class RoleInfo {}
```
角色信息，包含角色名，角色变更说明和变更时间信息。

```
public class SwitchDriverSubHandshakeAlreadyStarted extends
             SwitchDriverSubHandshakeException {}
```
当IOFSwitch.startDriverHandshake()被不止一次调用时抛出。

```
public class SwitchDriverSubHandshakeCompleted
       extends SwitchDriverSubHandshakeException {}
```
表明一个消息已被传递给一个交换机驱动程序的子握手处理代码，但驱动程序已经完成了子握手。

```
public class SwitchDriverSubHandshakeException extends RuntimeException {}
```
基类抛出的异常通过交换机驱动子握手处理。

```
public class SwitchDriverSubHandshakeNotStarted extends
       SwitchDriverSubHandshakeException {}
```
一个交换机驱动的子握手尚未开始，但要求子握手操作已尝试时抛出。

```
public class SwitchDriverSubHandshakeStateException extends
       SwitchDriverSubHandshakeException {}
```
当交换机驱动的子握手状态机收到意想不到的OFMessage或处于无效状态时抛出。

```
public class SwitchSyncRepresentation {}
```
代表在BigSync存储的交换机。它工作得很好，我们只需要存储FeaturesReply和DescriptionStatistics。

◆**package** net.floodlightcontroller.core.annotations