OpenFlow 交换机规范(概要)

Version 1.3.0 （June 25, 2012）

1 介绍

本文档描述了对 OpenFlow 交换机的要求。此规范内容包括交换机的组件和基本功能,和一个远程控制器管理一个 OpenFlow 交换机的协议：OpenFlow。

2 交换机部件

OpenFlow 的交换机包括一个或多个流表和一个组表，执行分组查找和转发，和到一个外部控制器 OpenFlow 的信道（图 1）。该交换机与控制器进行通信，控制器通过 OpenFlow 协议来管理交换机。

控制器使用 OpenFlow 协议，可以添加、更新和删除流流表中的表项，主动或者被动响应数据包。 在交换机中的每个流表中包含的一组流表项;每个流表项包含匹配字段，计数器和一组指令，用来匹配数据包（见 5.2）。

匹配从第一个流表开始，并可能会继续匹配其它流表 （见 5.1）。 流表项匹配数据包是按照优先级的顺序，从每个表的第一个匹配项开始（见 5.3）。如果找到一个匹配项，那么与流表项相关的指令就会去执行。如果在流表中未找到匹配项,结果取决于漏表的流表项配置：（例如，数据包可能通过 OpenFlow 信道被转发到控制器、丢弃、或者可以继续到下一个的流表，见 5.4）。

与流表项相关联的指令包含行动或修改流水线处理（见 5.9）。 行动指令描述了数据包转发，数据包的修改和组表处理。流水线处理指令允许数据包被发送到后面的表进行进一步的处理，并允许信息以元数据的形式在表之间进行通信。当与一个匹配的流表项相关联的指令集没有指向下一个表的时候，表流水线停止处理，这时该数据包通常会被修改和转发（见 5.10）。

流表项可能把数据包转发到某个端口。 这通常是一个物理端口,但它也可能是由交换机定义的一个逻辑端口或通过本规范中定义的一个保留的端口（见 4.1）。 保留端口可以指定通用的转发行为，如发送到控制器、泛洪、或使用非 OpenFlow 的方法转发，如 “ 普通”交换机转发处理（见 4.5）；而交换机定义的逻辑端口，可以是指定的链路汇聚组、隧道或环回接口（见 4.4）。

与流表项相关的行动，也可直接把数据包发送到组，进行额外的处理（见 5.6）。 组表示一组泛洪的指令集，以及更复杂的转发（如多路径，快速重路由，链路聚合）。作为间接的通用层，组也使多个流表项转发到同一个标识者（例如 IP 转发到一个共同的下一跳）。 这种抽象的行为使相同的输出行动非常有效。

组表包含若干组表项，每个组表项包含一系列依赖于组类型的特定含义行动存储段（见

5.6.1)。一个或多个行动存储段里的行动会作用到发送到该组的数据包。

只要正确的匹配和指令含义保持不变，交换机设计者可以任意的实现内部结构。 例如，当一个流表项使用一个所有组来转发至多个端口，交换机设计人员可以在硬件转发表中用一个单一的位掩码去实现。 另一个例子是匹配；如果 OpenFlow 交换机是通过不同数量的硬件表进行物理实现的，那么流水线就会被暴露。

3 名词解释

本节介绍了关键的 OpenFlow 规范术语：

. 字节：一个 8 位位组。

. 数据包：以太网帧，包括报头和有效载荷。

. 端口：数据包进入和退出 OpenFlow 的流水线地方（见 4.1)。可以是一个物理端口，由交换机定义的一个逻辑端口，或由 OpenFlow 的协议定义的一个保留端口。

. 流水线：在一个 openflow 交换机中提供匹配、转发和数据包修改功能的相连流表的集合。

. 流表：流水线的一个阶段，包含若干流表项。

. 流表项：在流表中用于匹配和处理数据包的一个元素。它包含用于匹配数据包的匹配字段、匹配次序的优先级，跟踪数据包的计数器，以及应用的指令集。

. 匹配字段：用来匹配数据包的字段，包括包头，进入端口，元数据值。一个匹配字段可能会进行通配符匹配（匹配任何值）或者在某些情况下通过位掩码进行匹配。

. 元数据：一个可屏蔽寄存器的值，用于携带信息从一个表到下一个表。

. 指令：指令存在于流表项中，当数据包匹配流表项时，指令用来描述 OpenFlow 的处理方式。指令要么改变流水线处理，如指导包匹配另一个流表，要么包含一组添加到行动集的行动，或包含一组立即应用到数据包的行动。

. 行动：是一种操作，可将数据包转发到一个端口或修改数据包，如 TTL 字段减 1。行动可以是与流表项相关联的指令集的一部分，或者是与组表项相关联的行动存储段的一部分。我们可以将行动积累在数据包的行动集里，也可以立即将行动应用到该数据包。

行动集：与数据包相关的行动集合，在数据包被每个表处理的时候这些行动可以累加，在指令集指导数据包退出处理流水线的时候这些行动会被执行。

. 组：一系列的行动存储段和选择一个或者多个存储段应用到每个数据包的手段。

. 行动存储段：行动和相关参数的集合，为组而定义的。

. 标记：一个头部，可以通过压入行动插入到数据包或者通过弹出行动进行移除。

. 最外层的标记：一个数据包最开始出现的标记。

. 控制器：与 OpenFlow 交换机使用 OpenFlow 协议交互的实体。

. 计量： 一个交换机元件，可以测量和控制数据包的速度。当通过计量的数据包速率或字节速率超过预定义的阈值时，计量触发计量带。如果计量带丢弃该数据包，它则被称为一个限速器。

4 OpenFlow 端口

本节介绍了 OpenFlow 的端口的概念和 OpenFlow 支持的各类端口。

4.1 OpenFlow 端口

OpenFlow 的端口是 OpenFlow 处理机和网络其余部分之间传递数据包的网络接口。
OpenFlow 交换机之间通过 OpenFlow 端口在逻辑上相互连接。

OpenFlow 交换机提供一定数量的 OpenFlow 端口,开放给 OpenFlow 的处理机。OpenFlow 的端口组可能与交换机硬件中提供的网络端口组不完全相同，因为有些硬件网络接口可能被 OpenFlow 禁用，OpenFlow 交换机可能定义额外的端口。

OpenFlow 的数据包从入端口接收，经过 OpenFlow 的流水线处理（见 5.1），可将它们转发到一个输出端口。入端口是数据包的属性，它贯穿整个 OpenFlow 流水线，并代表数据包是从哪个 OpenFlow 交换机的端口上接收的。匹配数据包的时候会用到入端口（见 5.3）。OpenFlow 流水线可以决定数据包通过输出行动发送到输出端口（见 5.12），还定义了数据包怎样传回到网络中。

一个 OpenFlow 交换机必须支持三种类型的 OpenFlow 的端口：物理端口，逻辑端口和保留端口。

4.2 标准端口

OpenFlow 的标准端口为物理端口，逻辑端口，本地保留端口（其他保留的端口除外）。

标准端口可以被用作入口和出端口，它们可在组里使用（见 5.6），都有端口计数器（见 5.8）。

4.3 物理端口

OpenFlow 的物理端口为交换机定义的端口，对应于一个交换机的硬件接口。例如，以太网交换机上的物理端口与以太网接口一一对应。

在有些应用中，OpenFlow 交换机可以实现交换机的硬件虚拟化。在这些情况下，一个

OpenFlow 物理端口可以代表一个与交换机硬件接口对应的虚拟切片。

4.4 逻辑端口

OpenFlow 的逻辑端口为交换机定义的端口，并不直接对应一个交换机的硬件接口。逻辑端口是更高层次的抽象概念，可以是交换机中非 OpenFlow 方式的端口（如链路汇聚组，隧道，环回接口）。

逻辑端口可能包括数据包封装，可以映射到不同的物理端口。这些逻辑端口的处理动作相对于 openflow 处理机来说必须是透明的，而且这些端口必须像硬件端口一样与 openflow 处理机互通。

物理端口和逻辑端口之间的唯一区别是：一个逻辑端口的数据包可能有一个叫做隧道 ID 的额外的元数据字段与它相关联（见 7.2.3.7）；而当一个逻辑端口上接收到的分组被发送到控制器时，其逻辑端口和底层的物理端口都要报告给控制器（见 7.4.1）。

4.5 保留端口

OpenFlow 的保留端口由本规范定义。它们指定通用的转发动作，如发送到控制器，泛洪，或使用非 OpenFlow 的方法转发，如“正常”交换机处理。

一个交换机不需支持所有的保留端口，只支持那些标记为“Required”的保留端口。

. Required: ALL: 表示交换机可转发指定数据包到所有端口，它仅可用作输出端口。在这种情况下，数据包被复制后发送到所有的标准端口，不包括数据包的入端口和端口被配置为 OFPPC_NO_FWD。

. Required: CONTROLLER: 表示 OpenFlow 控制器的控制通道，它可以用作一个入端口或作为一个出端口。当用作一个出端口，数据包封装进输入包消息，并使用 OpenFlow 协议发送（见 7.4.1）。当用作一个入口端口，确认数据包来自控制器。

. Required: TABLE: 表示 openflow 流水线的开始（见 5.1）。这个端口仅在输出包消息的行动列表里的输出行为时候有效（见 7.3.7），此时交换机提交报文给第一流表使数据包可以通过 OpenFlow 流水线处理。

. Required: IN PORT:代表数据包的进入端口。当数据包通过它的入端口发送出去的话，只能用作输出端口。

. Required: ANY: 特别值，用在未指定端口的 OpenFlow 命令（端口通配符）。既不能作为入口端口，也不能作为一个输出端口。

. Optional: LOCAL: 表示交换机的本地网络堆栈和管理堆栈。可以用作一个入口端口或作为一个输出端口。远程实体通过本地端口与交换机和网络服务互通，而不是通过一个独立的控制网络。利用一组合适的默认流表项，本地端口被用来实现一个带内控制器连接。

. Optional: NORMAL: 代表传统的非 OpenFlow 流水线（见 5.1）。仅可用于为一个输出端口，使用普通的流水线处理数据包。如果交换机不能转发数据包从 OpenFlow 流水线到普通流水线，它必须表明它不支持这一行动。

. Optional: FLOOD: 表示使用普通流水线处理进行泛洪（见 5.1）。只作为一个输出端口，一般可以将数据包发往所有标准端口，但不能发往入端口或 OFPPS_BLOCKED 状态的端口。交换机也可以通过数据包的 VLAN ID 选择哪些端口泛洪。

OpenFlow-only 交换机不支持 NORMAL 端口和 FLOOD 端口，而 OpenFlow-hybrid 交换机均支持上述端口（见 5.1）。转发数据包到 FLOOD 端口依赖交换机的实现和配置，若使用一组 all 类型进行转发，则可以使控制器能更灵活地实现泛洪（见 5.6.1）。

5 OpenFlow 表

本节描述流表和组表的组件，以及匹配和行动处理的机制。

5.1 流水线处理

OpenFlow 兼容的交换机有两种类型： OpenFlow-only 和 OpenFlow-hybrid 。OpenFlow-only 交换机只支持 OpenFlow 操作，在这些交换机中的所有数据包都由 OpenFlow 流水线处理，否则不能被处理。

OpenFlow-hybrid 交换机支持 OpenFlow 的操作和普通的以太网交换操作，即传统的 L2 以太网交换、VLAN 隔离、L3 路由（IPv4 的路由，IPv6 路由）、ACL 和 QoS 处理。这种交换机必须提供一个 OpenFlow 外的分类机制，使流量路由到 OpenFlow 流水线或普通流水线。例如，某个交换机可以使用 VLAN 标记或数据包的输入端口，来决定是否使用一个流水线或其它流水线，或者它可引导所有数据包都到 OpenFlow 流水线进行处理。这种分类机制超出了本规范的范围。一个 OpenFlow-hybrid 交换机也允许数据包通过 NORMAL 和 FLOOD 保留端口从 OpenFlow 流水线转移到普通流水线处理（见 4.5）。

每个 OpenFlow 交换机的流水线包含多个流表，每个流表包含多个流表项。 OpenFlow 的流水线处理定义了数据包如何与那些流表进行交互（参见图 2）。 OpenFlow 交换机至少需要一个流表，可选更多的流表。只有单一流表的 OpenFlow 交换机是有效的，而且在这种情况下流水线处理进程可以大大简化。

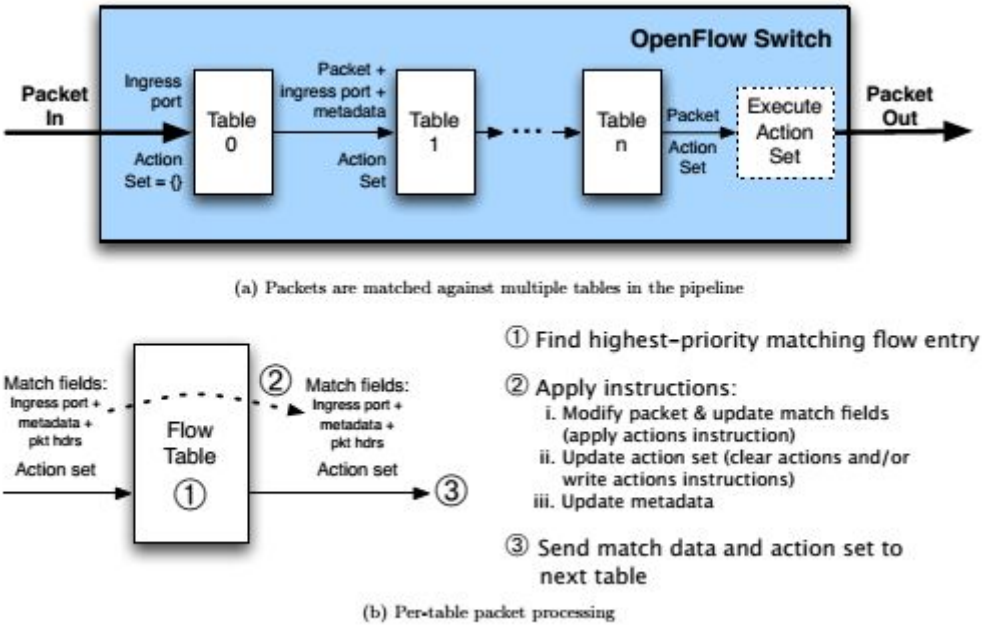

Figure 2: Packet flow through the processing pipeline.

Figure 2: 通过流水线处理的数据包流

OpenFlow 交换机的流表按顺序编号的，从 0 开始。流水线处理总是从第一流表开始：数据包首先与流表 0 的流表项匹配。其他流表根据第一个表的匹配结果来调用。

被一个流表处理时，数据包与流表中的流表项进行匹配，从而选择一个流表项（见 5.3）。如果匹配了流表项，则执行在该流表项里的指令；这些指令可能明确指导数据包传递到另一个流表（使用 Goto 指令，见 5.9），在那里同样的处理被重复执行。一个流表项只能指导数据包到大于自己表号的流表，换句话说流水线处理，只能前进而不能后退。显然，流水线的最后一个表项可以不包括 GOTO 指令。如果匹配的流表项并没有指导数据包到另一个流表，流水线处理将停止在该表中。当流水线处理停止，数据包被与之相关的行动集处理，通常是被转发（见 5.10）。

如果数据包在流表中没有匹配到流表项，这是一个漏表行为。漏表行为取决于表的配置（见 5.4）。一个流表中的漏表项可以指定如何处理无法匹配的数据包：可以选择丢弃，传递到另一个表，或利用输入包消息通过控制信道发送到控制器去（见 6.1.2）。

Openflow 流水线和各种 Openflow 操作用同样的方法处理特定类型的数据包和规范定义的相同类型的包，除非目前的规范或 Openflow 配置规定了不同的方法。例如，Openflow 定义的以太头部必须与 IEEE 规范一致，Openflow 使用的 TCP/IP 头部定义必须与 RFC 规范一致。另外，Openflow 交换机的包重排序必须与 IEEE 规范的要求一致，保证数据包能被流表项、组表和计量带同样的处理。

5.2 流表

一个流表中包含多个流表项。

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Table~1: Main components of a flow entry in a flow table.

每个流表项包含：

.匹配字段：对数据包匹配。包括入端口和数据包头，以及由前一个表指定的可选的元数据。

.优先级：流表项的匹配次序 。

.计数器：数据包匹配时更新计数。

.指令：修改行动集或流水线处理。

.超时：最大时间计数值或流在交换机中失效之前的剩余时间 。

.cookie：由控制器选择的不透明数据值。控制器用来过滤流统计数据、流修改和流删除。但处理数据包时不能使用。

流表项通过匹配字段和优先级决定：在一个流表中匹配字段和优先级共同确定唯一的流表项。所有字段通配(所有字段省略)和优先级等于 0 的流表项被称为 table-miss 流表项(见 5.4)。

5.3 匹配

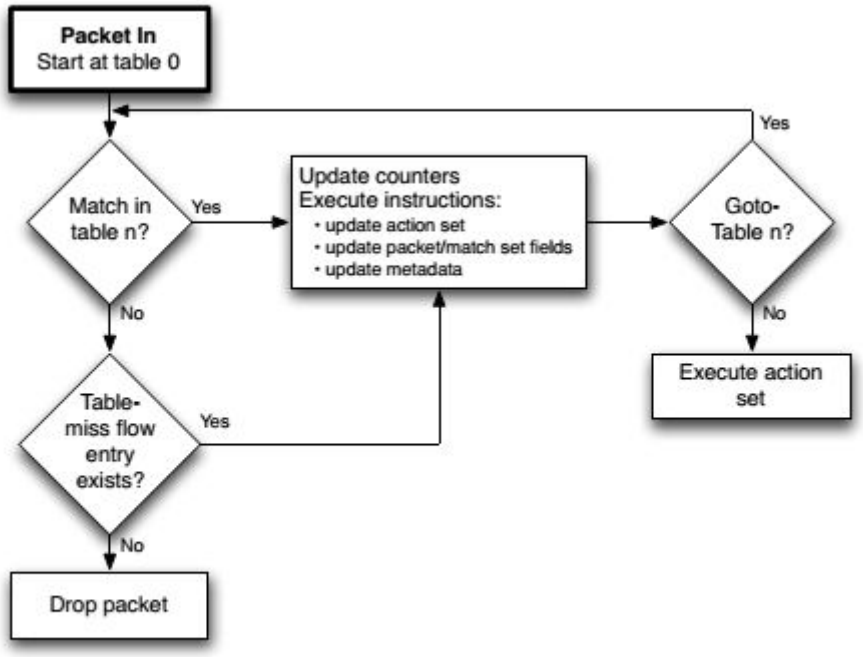

Figure 3: Flowchart detailing packet flow through an OpenFlow switch.

Figure 3:流程图详细描述了数据包流通过一个 OpenFlow 交换机。

当 OpenFlow 交换机接收一个数据包，就执行图 3 所示的功能。交换机开始对第一个流表进行查找，并基于流水线处理，也可能在其它流表中（见 5.1）执行表查找。

数据包匹配字段是从数据包中提取的。用于表查找的数据包匹配字段依赖于数据包类型，也就是包括各种数据包头部字段，如以太网源地址或 IPv4 目的地址（见 7.2.3）。除了通过数据包报头中进行匹配，也可以通过入端口和元数据字段进行匹配。元数据可以用来在一个交换机的不同表里面传递信息。数据包匹配字段表示数据包的当前状态，如果在前一个表中使用 Apply-Actions 改变了数据包的报头，那么这些变化也会在数据包匹配字段中反映。

如果数据包中用于查找的匹配字段值匹配了流表项定义的字段，就表示这个数据包匹配了此流表项。如果流表项字段的值是 ANY（字段省略），它就可以匹配包头部的所有可能的值。如果交换机支持对指定的匹配字段进行任意的的位掩码，这些掩码可以更精确地进行匹配。

数据包与表进行匹配，而且只有匹配数据包的最高优先级的表项必须被选择，此时与所选流表项相关的计数器也会被更新，所选流表项的指令集也会被执行。如果多个匹配的流表项具有相同的最高优先级，所选流表项则被确定为未定义表项。只有控制器记录器在流信息中没有设置 OFPFF_CHECK_OVERLAP 位并且增加了重复的表项的时候，这种情况才能出现。

如果交换机配置中包含 OFPC_FRAG_REASM 标志（见 7.3.2），则在流水线处理前 IP 碎片必须被重新组装。

当交换机接收到一个格式不正确或损坏的数据包，此版本的规范没有定义预期的行为。

5.4 Table-miss

每一个流表必须支持 table-miss 的流表项来处理漏表。table-miss 表项指定如何处理在流表中与其他流表项未匹配的数据包（见 5.1），比如把数据包发送给控制器、丢弃数据包或直接将包扔到后续的表。

table-miss 流表项也有它的匹配字段和优先级（见 5.2），它通配所有匹配字段（所有字段省略），并具有最低的优先级（0）。table-miss 流表项的匹配可能不属于正常范围内流表支持的匹配，例如精确匹配表可能不支持在其他流表项中使用通配符，但必须支持 table-miss 流表项通配所有字段。table-miss 流表项可能不具备正常流表项（见 7.3.5.5）相同的能力。table-miss 流表项必须至少支持利用 CONTROLLER 保留端口将数据包发送到控制器（见 4.5），和使用 Clear-Actions 指令（见 5.9）丢弃数据包。为了和早期版本规范一致，如果可能，在实现中鼓励支持引导包给后续的表。

table-miss 表项的行为在许多方面像任何其他流表项：默认情况下，在流表中不存在 table-miss 表项。控制器可以在任何时候添加或删除它（见 6.4），而且它可能会超时失效（见 5.5）。table-miss 流表项像期望的那样，利用它的匹配字段和优先级匹配表中的数据包。table-miss 流表项可以匹配流表中其他表项中不能匹配的数据。当数据包与 table-miss 表项匹配时，table-miss 表项指令就会执行（见 5.9）。如果该 table-miss 表项直接将数据包通过 CONTROLLER 保留端口发送到控制器（见 4.5），那么数据包输入的原因必须标识出 table-miss 表项（见 7.4.1）。

如果该 table-miss 表项不存在，默认情况下，流表项无法匹配的数据包将被放弃（丢弃）。交换机配置时，例如使用 OpenFlow 配置协议，可以覆盖此默认值，指定其他行为。

5.5 流表项删除

流表项可以通过两种方式在流表中删除，一个是通过控制器的请求，一个是利用交换机的流超时机制。

交换机的流超时机制独立于控制器，由交换机根据流表项的状态和配置来运行。每个流的表项具有一个和它相关的 idle_timeout 和 hard_timeout 值。如果 hard_timeout 值不为零，交换机必须注意的流表项的老化时间，因为交换机可能删除该项。一个非零 hard_timeout 字段可能在规定数秒后引起流表项被删除，无论有多少数据包与之匹配。如果给定非零 idle_timeout 的值，交换机必须注意与流的最后一个数据包到达的时间，可能后面需要删除这个流表项。当在规定数秒内没有匹配数据包时，一个非零 idle_timeout 字段将引起流表项删除。交换机必须实现流表项超时就会从流表中删除的功能。

控制器可以主动发送 DELETE 流表修改消息（OFPFC_DELETE，或 OFPFC_DELETE_STRICT - 见 6.4），从流表中删除流表项。

流表项被删除时，不管是控制器或流表项超时引起的，交换机必须检查流表项的 OFPFF_SEND_FLOW_REM 标志。如果该标志被设置，该交换机必须将流删除消息发送到控制器。每个流删除消息中包含一个完整的流表项描述、清除的原因（超时或删除），在清除时的流表项的持续时间，在清除时的流的统计数据。

5.6 组表

一个组表包括若干组表项。一个流表项指向一个组的能力使得 openflow 可以实现额外的转发方法（例如选择部分和所有）。

Group Identifier	Group Type	Counters	Action Buckets

Table~2: Main components of a group entry in the group table.

每个组表项（见表 2）由组编号确定，具体内容包含：

.组编号：一个 32 位的无符号整数，唯一标识该组

.组类型：确定组语义（参见 5.6.1 节）

.计数器：当数据包被组处理时更新

.行动存储段：有序的行动存储段，其中的每个行动存储段包含了一组要执行的行动和相关参数。

5.6.1 组类型

交换机只支持那些标记为“Required”的组类型，控制器可以查询交换机支持哪些“Optional”组类型。

. Required: all: 执行组中的所有存储段。这个组用于多播或广播的转发。数据包为每个存储段都有效地复制一份，然后被每个存储段处理。如果某个存储段中明确地指导数据包发往入端口，那么这个复制的包被丢弃。如果控制器记录器希望数据包从入端口转发出去，那么这个组必须包含一个额外的存储段,这个存储段包含到 OFPP_IN_PORT 保留端口的输出行动。

. Optional: select: 执行组中的一个存储段。基于交换机计算选择算法（如利用用户配置的元组/数组的哈希算法或简单的循环算法），数据包被组中的一个存储段处理。选择算法的所有配置和状态都在 OpenFlow 外部运行。选择算法实现可以使用等负荷分配，也可以选择根据存储段权重进行。当组中存储段所指定的端口出现故障时，交换机可对剩余部分（具有转发到有效端口行为的）限制选择存储段，而不是丢弃数据包。此行为可能会减少数据包在链路或交换机的中断。

.Required: indirect: 执行此组中定义的一个存储段。这个组只支持单一的存储段。允许多个流表项或者组指向一个共同的组编号，这样可以使转发更快，更高效的汇聚（例如：下一跳 IP 转发）。对于只有一个存储段的所有组，组类型应该是相同的。

. Optional: fast failover: 执行第一个有效的存储段。每一个行动存储段与控制其有

效性的一个指定端口/或组相关。组定义的存储段是有序的，首选选择与有效的端口或者组相关的第一个存储段。这个组类型可以使交换机改变转发，而无需通知控制器。如果没有有效的存储段，数据包将被丢弃。组类型必须实行有效机制（见 6.5）。

5.7 计量表

一个计量表包含若干计量表项，确定每个流的计数。每个流的计数可以使 OpenFlow 实现各种简单的 QoS 操作，如限速，并且可以结合每个端口队列（见 5.12）来实现复杂的 QoS 构架，如 DiffServ。

计量器可以测试数据包分配的速率，并可以控制数据包的速率。计量器直接连接到流表项（而不是被连接到端口的队列）。任意的流表项可以在它的指令集中定义一个计量器（见 5.9），计量器测量和控制和它有关的所有流表项的总速率。在同一个表中可以使用多个计量器，但必须使用专用的方式（流表项分离设置）。在连续的流表中，对于同样的数据包集合，可以使用多个计量器。

Meter Identifier	Meter Bands	Counters

Table~3: Main components of a meter entry in the meter table.

每个计量表项（见表 3）由其计量标识符来区分，其包含：

.计量器的标识符：一个 32 位的无符号整数唯一识别

.计量带：计量带的无序列表，其中每个计量带指定带速和处理数据包的方式

.计数器：计量器处理数据包时进行更新计数。

5.7.1 计量带

每个计量器可能有一个或多个计量带。每个带指定所用的速率和数据包处理的方式。单个计量带以当前测量的计量速率处理数据包。当测量速率超过最高配置速率的时候，计量器就启用计量带。若当前的速率比任何指定的计量带速率率低，就没有计量带需要工作。

Band Type	Rate	Counters	Type specific arguments

Table~4: Main components of a meter band in a meter entry.

每个计量带（见表 4）用速率来识别，包括：

.带类型：定义了数据包如何处理

.速率：用于计量器选择计量带，也就是计量带可以启用的最低速率

.计数器：当计量带处理数据包时更新计数

.类型的特定参数：带类型的可选参数

在本规范里带类型没有“Required”。控制器可以查询交换机支持的计量带类型“Optional”

. Optional: drop:丢弃数据包。可以用来定义速率限制带。

. Optional: dscp remark:增加数据包的 IP 头部 DSCP 字段丢弃的优先级。可用于定义一个简单的 DiffServ 策略。

5.8 计数器

每一个流表，流表项，端口，队列，组，组存储段，计量器和计量带都会修改计数器。OpenFlow-compliant 计数器可以在软件中实现，也可以通过查询硬件计数器获取计数进行有限范围的修改。表 5 中包含了 openflow 规范中定义的计数器集。交换机不要求支持所有的计数器，只有那些标记为“Required”是必须支持的。

持续时间指的是流表项，端口，组，队列或计量器在交换机中已安装的时间数值，而且必须精确到秒。Received Errors 字段是表 5 里定义的所有收到的和冲突的错误总和，也包括表里未列出的其它错误。

计数器都是无符号值，可以环回且没有溢出指示。如果交换机里没有指定值的计数器，则其值必须设置成字段的最大值（无符号数就是-1）。

5.9 指令

每个流表项中包含一组指令集，当一个数据包匹配表项时指令就会被执行。这些指令可导致数据包，行动组和/或流水线处理发生改变。

交换机不需要支持所有类型的指令，只需支持下面那些标记为“Required Instruction”。控制器可查询交换机支持的“Optional Instruction”。

. Optional Instruction: Meter meter_id :将包转给指定的计量器。计量的结果可能会丢弃这个数据包（依赖于计量器的配置和状态）。

. Optional Instruction: Apply-Actions action(s): 立即执行指定的行动，而不改变行动集。在两个表之间传递或者执行同类型的多个行动的时候，这个指令可用来修改数据包。这些行动被指定为一个行动列表（见 5.11）。

. Optional Instruction: Clear-Actions:立即清除行动集中的所有行动。

. Required Instruction: Write-Actions action(s): 将指定的行动添加到当前的行动集中。 如果行动存在于当前集合中，则进行覆盖，否则进行追加。

. Optional Instruction: Write-Metadata metadata / mask ：在元数据字段写入掩码的元数据数值。掩码指的是元数据寄存器应进行修改的比特。(new_metadata=old_metadata&~mask|value&mask)。

Counter	Bits	
Per Flow Table		
Reference Count (active entries)	32	*Required*
Packet Lookups	64	*Optional*
Packet Matches	64	*Optional*
Per Flow Entry		
Received Packets	64	*Optional*
Received Bytes	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Port		
Received Packets	64	*Required*
Transmitted Packets	64	*Required*
Received Bytes	64	*Optional*
Transmitted Bytes	64	*Optional*
Receive Drops	64	*Optional*
Transmit Drops	64	*Optional*
Receive Errors	64	*Optional*
Transmit Errors	64	*Optional*
Receive Frame Alignment Errors	64	*Optional*
Receive Overrun Errors	64	*Optional*
Receive CRC Errors	64	*Optional*
Collisions	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Queue		
Transmit Packets	64	*Required*
Transmit Bytes	64	*Optional*
Transmit Overrun Errors	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group		
Reference Count (flow entries)	32	*Optional*
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group Bucket		
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Per Meter		
Flow Count	32	*Optional*
Input Packet Count	64	*Optional*
Input Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Meter Band		
In Band Packet Count	64	*Optional*
In Band Byte Count	64	*Optional*

Table~5: List of counters.

. Required Instruction: Goto-Table next-table-id: 指示流水线处理的下一张表。表 ID 必须大于当前表 ID。流水线最后一张表的流表项不能含有这个指令（见 5.1）。Openflow 交换机若只有一个流表则不需要实现这个指令。

流表项所属的指令集中每个类型的单个指令都有个最大值。指令就是按照上述列表中指定的顺序来执行。实际上，有限定的是：Meter 指令在 Apply_Actions 指令前执行，Clear_Actions 指令在 Write_Actions 指令前执行，Goto_Table 最后执行。

如果流表项不能执行相关指令，交换机必须拒绝这个流表项。这种情况，交换机必须返回一个不支持的流错误信息（见 6.4）。流表不一定支持每个匹配，每个指令或每个行动。

5.10 行动集

行动集是与每个数据包相关的，默认情况下是空的。一个流表项可以使用 Write-Action 指令或者与特殊匹配有关的 Clear-Action 指令来修改行动集。行动集在表间被传递。当一个流表项的指令集没有包含 Goto-Table 指令时，流水线处理就停止了，然后数据包的行动集执行其行动。

行动集中每个类型的行动有一个最大值。 set_field 行动用字段类型来标识，因此行动集对每个字段类型的 set_field 行动（即，多个字段可被设置）有个最大值。当同一个类型需要多个行动时，例如，压入多个 MPLS 标签或弹出多个 MPLS 标签，应使用 Apply_Actions 指令（见 5.11）。

行动集中所有的行动，不管它们以什么顺序添加到行动集中，行动的顺序均按照下列顺序执行。如果行动集包含组行动，那么组行动存储段中的行动也按照下列顺序执行。当然，交换机也可以支持通过 Apply-Actions 指令任意修改行动执行顺序。

1. copy TTL inwards: 向数据包内复制 TTL 的行动
2. pop: 从数据包弹出所有标记的行动
3. push-MPLS: 向数据包压入 MPLS 标记的行动
4. push-PBB: 向数据包压入 PBB 标记的行动
5. push-VLAN: 向数据包压入 VLAN 标记的行动
6. copy TTL outwards: 向数据包外复制 TTL 的行动
7. decrement TTL: 将数据包的 TTL 字段减 1
8. set: 数据包使用所有的 set_field 行动
9. qos: 使用所有的 QOS 行动，如对数据包排队
10. group: 如果指定了组行动，那么按顺序执行组行动存储段里的行动。
11. output: 如果没有指定组行动，数据包就会按照 output 行动中指定的端口转发。

行动集中的 Output 行动是最后执行的。如果在一个行动集里组行动和输出行动都存在，则组行动优先。如果两者均不存在，数据包将被丢弃。 如果交换机支持的话，组的执行将返回；组存储段可指定另外一个组，在这种情况下，行动将在组配置中指定的所有组中执行。

5.11 行动列表

Apply-Actions 指令和 Packet-out 消息包含一个行动列表。行动列表的含义与 Openflow1.0 规范的相同。行动列表中的行动按照列表中的次序执行，并立即作用到数据包。

列表中的行动从第一个行动开始执行，行动都是按序执行的。行动的结果是累积的，比如行动列表中有两个 push VLAN 行动，数据包就会被加上两个 VLAN 头部。如果行动列表有一个输出行动，一个当前状态下的包复制后就转发给所需端口。如果列表包含组行动，相关组存储段就会处理当前状态下的复制包。

一个 Apply_Actions 指令执行完一个行动列表后，流水线继续处理已修改的数据包（见 5.1）。数据包的行动集本身在行动列表执行的时候没有改变。

5.12 行动

交换机不要求支持所有类型的行动，只需支持标记为“Required Action”。控制器也可查询交换机所支持的“Optional Action”。

Required Action: Output. 数据包输出到指定 Openflow 端口（见 4.1）。Openflow 交换机必须支持转发到物理端口，交换机定义的逻辑端口和所需的保留端口 （见 4.5）。

Optional Action: Set-Queue.设置数据包的队列 ID。当数据包使用输出行动转发到一个端口，队列 ID 决定数据包安排到端口所属的哪个队列并转发。转发行为受队列配置控制，并用来提供 Qos 支持（见 7.2.2）。

Required Action: Drop. 没有明确的行动来表现丢弃。相反，那些行动集中没有输出行动的数据包应该被丢弃。当流水线处理时或执行 Clear_Actions 指令后，空指令集或空指令行动存储段会导致丢弃这个结果。

Required Action: Group. 通过指定的组处理数据包，准确的解释依靠组类型。

Optional Action: Push-Tag/Pop-Tag. 交换机可具有压入/弹出表 6 所示标记的能力。为了和已有网络更好结合，建议支持压入/弹出 VLAN 标记的能力。

最新的压入标记应插入到最外侧有效位置作为最外侧的标记。当压入一个新 VLAN 标记，应作为最外侧标记来插入，位于以太头部后面，其它标记前面。同样的，当压入一个新 MPLS 标记，也应作为最外侧标记来插入，位于以太头部后面，其它标记前面。

当多个压入行动添加到数据包行动集，按照行动集定义的规则依次作用到数据包，开始时 MPLS，接着是 PBB，后面是 VLAN（见 5.10）。当一个行动列表中有多个压入行动，按照列表次序（见 5.11）作用到数据包。

注意：5.12 节所涉及的信息都是默认字段值。

Action	Associated Data	Description
Push VLAN header	Ethertype	Push a new VLAN header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8100 and 0x88a8 should be used.
Pop VLAN header	-	Pop the outer-most VLAN header from the packet.
Push MPLS header	Ethertype	Push a new MPLS shim header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8847 and 0x8848 should be used.
Table 6 – Continued on next page		

Action	Associated Data	Description
Pop MPLS header	Ethertype	Pop the outer-most MPLS tag or shim header from the packet. The Ethertype is used as the Ethertype for the resulting packet (Ethertype for the MPLS payload).
Push PBB header	Ethertype	Push a new PBB service instance header (I-TAG TCI) onto the packet (see 7.2.5). The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x88E7 should be used.
Pop PBB header	-	Pop the outer-most PBB service instance header (I-TAG TCI) from the packet (see 7.2.5).

Table˜6: Push/pop tag actions.

Optional Action: Set-Field. 不同 Set-Field 行动由它们的字段类型来标识，并对数据包中头部字段分别修改数值。 当要求不很严格时，若支持使用 Set-Field 行动进行重写头部各字段将非常有益于 Openflow 的实现。为了和已有网络更好结合，建议支持 VLAN 修改行动。Set-Field 行动应一直作用到头部可能的最外侧（例如，“set VLAN ID”行动一直设置 VLAN 标记的最外侧 ID），除非该字段类型指定其它值。

Optional Action: Change-TTL. 不同 Change-TTL 行动修改数据包中的 IPV4 TTL、IPV6 Hop Limit 或 MPLS TTL。当要求不很严格时，表 7 所示的行动非常有益于 Openflow 中路由功能的实现。

Action	Associated Data	Description
Set MPLS TTL	8 bits: New MPLS TTL	Replace the existing MPLS TTL. Only applies to packets with an existing MPLS shim header.
Decrement MPLS TTL	-	Decrement the MPLS TTL. Only applies to packets with an existing MPLS shim header.
Set IP TTL	8 bits: New IP TTL	Replace the existing IPv4 TTL or IPv6 Hop Limit and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Decrement IP TTL	-	Decrement the IPv4 TTL or IPv6 Hop Limit field and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Copy TTL outwards	-	Copy the TTL from next-to-outermost to outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or IP-to-MPLS.
Copy TTL inwards	-	Copy the TTL from outermost to next-to-outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or MPLS-to-IP.

Table˜7: Change-TTL actions.

Openflow 交换机检查出携带无效 IP TTL 或 MPLS TTL 的数据包并拒绝接收。不是每个数据包都需要检查 TTL 是否有效，但是每次数据包完成 TTL 减 1 行动后应在最短时间内检查。交换机可能会改变其异步配置（见 6.1.1)），利用输入包消息通过控制信道发送携带无效 TTL 的数据包给控制器（见 6.1.2)。

5.12.1 字段压入的默认值

当执行压入行动时，表 8 中所有字段指定的值应从已有外部头复制给新建外部头。表 8 中所列的新字段与已有字段不一致的应设置为 0。Openflow set_field 行动不能修改的字段用初始化成合适的协议数值。

New Fields		Existing Field(s)
VLAN ID	←	VLAN ID
VLAN priority	←	VLAN priority
MPLS label	←	MPLS label
MPLS traffic class	←	MPLS traffic class
MPLS TTL	←	{ MPLS TTL IP TTL
PBB I-SID	←	PBB I-SID
PBB I-PCP	←	VLAN PCP
PBB C-DA	←	ETH DST
PBB C-SA	←	ETH SRC

Table˜8: Existing fields that may be copied into new fields on a push action.

在压入操作之后，可通过指定的“set”行动对新建头部的某些字段进行设置。

6、openflow 通道

Openflow 通道是每个交换机连接控制器的接口，通过这个接口，控制器配置和管理交换机，接收来自交换机的事件，将包从交换机转发出去。

尽管所有 openflow 通道消息必须遵守 openflow 协议格式， 但在数据通路和 openflow 通道之间，接口是具体实施者。openflow 通道通常使用 TLS 加密，但也可能直接在 TCP 上运行。

6.1 openflow 协议概述

Openflow 协议支持三种消息：控制器到交换机消息、异步消息和对称消息，每个都有多个子消息类型。controller-to-switch 消息由控制器发起，用来直接管理检查交换机的状态；异步消息由交换机发起，用于控制器更新网络事件和交换机状态变化；对称消息由交换机或者控制器发起，而且无需请求。下面介绍 Openflow 使用的消息类型。

6.1.1 Controller-to-Switch

Controller-to-Switch 是由控制器发起，不强求交换机作出回应。

Features:控制器通过发送 feature 请求查询交换机的身份以及基本功能,交换机必须响应，回答其身份和基本能力。通常在 openflow 通道建立后运行。

Configuration：控制器用来设置、查询交换机的配置参数。交换机仅需要回应来自控制器的查询消息。

Modify-State：mmodify-sate 消息由控制器发出，用来管理交换机的状态。它们的首要目标是增加、删除、修改 openflow 表中的流表项/组表项，以及设置交换机的端口属性。

Read-state：控制器用 Read-state 消息收集交换机的各种消息，例如当前配置、统计数据和性能等。

Packet-out：控制器用这个 Packet-out 发送数据包到交换机特定的端口。并且转发通过 Packet-in 消息收到的数据包。Packet-out 消息必须包括一个完整的数据包或者一个指明交换机中存储数据包缓冲区的 ID。这个消息必须包含一个动作列表，并按指定顺序应用这些动作。若动作列表为空则丢弃该包。

Barrier: 控制器使用 Barrier 请求/回复消息确保消息依赖已经遇到或者收到操作完成的通知。

Role-Request:控制器使用 Role-Request 消息，用来设置 openflow 通道角色，或者查询这个角色。当交换机连接多个控制器时这个消息则非常有用（见 6.3.4）。

Asynchronous-Configuration：控制器使用 Asynchronous-Configuration 消息来设置一个附加过滤器，滤出想在 openflow 通道中接受的异步消息，或者用来查询这些过滤器。这个消息在交换机连接多个控制器时非常有用（见 6.3.4），通常在 openflow 通道建立后运行。

6.1.2 Asynchronous

交换机发送 Asynchronous 消息时无需控制器的请求，通知控制器数据包到达、交换机状态改变或错误。四个主要的异步消息描述如下。

Packet-in:将对包的控制转移给控制器。由于全部数据包使用流表项或者漏表项转发到“CONTROLLER”保留端口，一个 packet-in 事件就会发给控制器（见 5.12）。其他处理，例如 TTL 检查,也可能产生 packet-in 事件将数据包发给控制器。

缓冲数据包可以配置 packet-in 事件。由流表项或者组存储段中的输出行动产生的 packet-in，可以在输出行动中各自分别指定（见 7.2.5），其他的 packet-in 可在交换机中配置（见 7.3.2）。如果这个 packet-in 事件配置给缓冲数据包，并且交换机中有足够的缓存，在交换机准备转发这个数据包时，这个 packet-in 事件仅仅包含数据包头部的一部分和控制器使用的缓冲 ID。不支持内部缓冲的交换机，packet-in 事件配置为不缓冲数据包，或内部缓冲耗尽时，将整个包作为事件的一部分发送给控制器。缓冲的数据包通常是利用控制器发来的 packet-out 消息进行处理，或者一段时间后自动过期。

如果数据包已缓冲，包含在 packet-in 中的原始包字节数就可以配置，默认的是 128 字节。由流表项或者组存储段中的输出行动产生的 packet-in，可以在输出行动中各自分别指定（见 7.2.5），其他的可在交换机中配置（见 7.3.2）。

（注：交换机缓存足够，包被临时放在缓存中，包的部分内容（默认 128 字节）和在交换机缓存中的序号也一同发给控制器；如果交换机缓存不足以存储包，则将整个包作为消息的附带内容发给控制器）

Flow-Removed：通知控制器一个流表项从流表中移除。流表项只有设置了 OFPFF_SEND_FLOW_REM 标志，Flow-Removed 消息才会发送。此消息是由于控制器的流删除请求产生，或者交换机流处理超时产生，即某个流有效时间超出范围了。

Port-status：通知控制器某个端口发生变化。交换机需要在端口配置或状态改变时发送 port-status 消息给控制器。包括端口配置的改变等事件，例如，端口被用户关闭，端口状态被用户改变，连接断开等。

Error:交换机用 Error 消息通知控制器出现问题。

6.1.3 Symmetric（对称）

Symmetric(对称)消息向每个方向发送时无需询问。包括 Hello，Echo，Experimenter 三种消息。

Hello: Hello 消息在连接一旦建立，就在交换机和控制器之间传递。

Echo:从交换机或者控制器发出的 Echo 应答/请求消息,必须得到一个返回的 Echo 响应消息。他们主要用来验证 controller-switch 连接是否活跃，也可能来测量延迟率和带宽。

Experimenter:Experimenter 消息为交换机在其消息类型中提供附加功能提供了一个标准的方法。这是将来 openflow 修订时的一个功能中转区。

6.2 Message Handing(消息处理)

Openflow 协议提供可靠的消息传递和处理，但是不自动提供确认和保证消息的有序化处理。这一节所说的 openflow 消息处理行为是由可靠传输的主辅链接上完成的。

Message Delivery:消息可靠传输，除非 Openflow 通道完全失败，控制器不了解交换机的任何状态（如交换机可能已经进入了“失败独立模式”）。

Message Processing:交换机必须完全处理从控制器接收的每个消息，并可能生成一个回复。如果交换机不能完全处理来自控制器的消息，那么它必须返回一个 error 消息。对于 pack-out 消息，消息被完全处理之后也不能保证包实际上退出了交换机。由于交换机的堵塞、QoS 策略、或者数据包发送到一个阻塞或无效的端口，交换机处理之后数据包可能被丢弃。

此外，交换机须发送所有 OpenFlow 状态改变时产生的异步消息，例如流删除，端口状态或者 packet-in 消息,这样控制器就可以与交换机的实际情况同步。基于异步配置(见 6.1.1)，这些消息可能被过滤，而且，在引起这些变化之前，触发 Openflow 状态改变的环境可能会被过滤掉。例如，数据端口接收的用来发送到控制器的数据包，由于交换机的阻塞或 QoS 策略和 packet-in 消息的缺失，可能被丢弃。这些丢失在数据包对控制器有明显的输出行动时会发生，也可能在数据包在表项里匹配失败产生，尽管表的默认行动是发给控制器。（这个错误会被发送到交换机）。发给控制器的数据包建议采用 QoS 行动和速率限制进行监管，以防止控制器链接的拒绝服务，此内容超出本规范范围。

控制器可以自由忽略他们接收的消息，但是必须响应应答消息来防止交换机中断连接。

Message Ordering:排序可以通过使用 barrier 消息来保证。在缺少屏障消息时 ，交换机可以任意重新排序消息，来达到最佳性能。因此，交换机不应该依赖一个特定的处理顺序。特别的是，流表项插入到流表时，其顺序不同于交换机接收到的流模式。消息跨越一个屏障消息就不需要重新排序，只有前面所有的消息都处理后屏障消息才被处理。具体的说：

1. 屏障消息前面的消息必须先被处理，包括发送任何产生的错误和回复。
2. 屏障必须被处理，而后发送一个响应。
3. 屏障消息后面的消息可继续处理。

如果从控制器接收到的 2 个消息是相互依赖的，那他们必须通过屏蔽消息来分离。

6.3 Openflow 通道连接

Openflow通道用来在控制器和交换机之间交换Openflow消息。控制器管理多个Openflow通道，每个通道连接到一个不同的交换机。一个交换机可能有一个通道连到控制器，或者，有多个通道连接到不同的控制器（见 6.3.4）。

控制器可以通过一个或者多个网络来远程管理交换机。这种方法是不在当前规范之内的，所使用的可能是一个独立的专用网络，或者由交换机管理的网络（带内控制器连接）。唯一的要求就是应提供 TCP/IP 连接。

Openflow 通道实际上作为一个交换机和控制器之间的单一的网络连接，使用 TLS 或 TCP 协议（见 6.3.3）。另外，为了具有并行性，通道可以由多个网络连接组成。交换机通过初始化到控制器的连接来创建一个到控制器的通道（见 6.3.1）。一些交换机实现时可能允许一个控制器连接到交换机，这种情况下，交换机通常应保证安全的连接，以防止没有授权的连接。

6.3.1 连接建立

交换机必须能够与一个用户可配置 IP 地址（否则是固定的）的控制器建立连接，连接使用用户指定的一个端口或是默认端口。如果交换机与控制器的 IP 地址进行配置连接，那么交换机启动了一个标准的 TLS 或 TCP 连接。Openflow 通道的流量不通过 Openflow 流水线。因此，交换机必须在流表检查之前就必须区分输入流量。

当 Openflow 连接初次建立之时，连接的每一方必须立即发送携带版本字段的 OFPT_HELLO 消息，版本代表发送者支持的最高的 OpenFlow 协议版本。这个 Hello 消息可能含有一些可选内容来帮助建立连接（见 7.5.1）。一旦接收到消息，接收方须立即计算出使用的协议版本。如果发送和接收的 Hello 消息都包含 OFPHET_VERSIONBITMAP 的 hello 元素，并且如果这些位图有一些共同的位设置，那双方协商的协议是最高版本的。否则，协商版本是接收到的版本字段中较小版本数字。

如果接收方支持协商版本，那么连接可行。否则，接收方必须回应一个 OFPT_ERROR 消息，此消息带有 OFPET_HELLO_FAILED 的类型字段，OFPHFC_INCOMPATIBLE 的字段以及一个解释数据状态的随机 ASCII 字符串，然后终止连接。

在交换机和控制器交换过 OFPT_HELLO 消息并且有共同的版本数字之后，连接设置就完成了，标准的 Openflow 消息能够在连接之上交换。首先，控制器发送一个 OFPT_FEATURES_REQUEST 消息来取得交换机的数据路径 ID（见 7.3.1）.

6.3.2 连接中断

当交换机与所有的控制器丢失连接，会导致 echo 请求超时，TLS（安全传输协议）握手超时，或者其他的连接失败，交换机根据当前运行状态和配置，必须立即进入“失败安全模式”或者“失败独立模式”。在失败安全模式下，交换机行为唯一的改变就是丢弃发向控制器的包和消息。流表项应在“失败安全模式”下依据定时设置继续会发生超时现象。在“失败独立模式”下，交换机使用 OFPP_NORMAL 保留端口处理所有的包。换机话说，就是交换机作

为传统以太网的交换机和路由器。“失败独立模式”一般只在 Hybrid 交换机上存在（见 5.1）。

当再一次连接控制器时，已有的流表项将保留。如果有必要的话，控制器可删除所有的流表项。

交换机第一次启动时，它会运行在“失败安全模式”或者“失败独立模式”，直到它成功的连接到控制器。启动时流表项的默认设置内容不在本 Openflow 协议之内。

6.3.3 加密

交换机和控制器可通过 TLS 连接（安全传输层协议）通信。交换机启动时初始化 TLS 连接来连接控制器，TLS 连接默认 TCP 端口是 6633。交换机和控制器通过交换由位置专一密钥签名的证书，来相互鉴权。每个交换机必须是用户可配置的，并携带控制器鉴权的证书（控制器证书）和另一个证书向控制器鉴权（交换机证书）。

交换机和控制器可选择用普通 TCP 来互通，此 TCP 连接默认位于 TCP 端口 6633，由交换机发起和初始化。当使用默认的普通 TCP 连接，推荐使用其他安全措施来防止窃听，伪装控制器以及其他 Openflow 通道上的攻击。

6.3.4 多控制器

交换机可能与一个或多个控制器建立通信。多控制器模式可靠性更好，如果一个控制器连接失败，交换机还是能够继续运行在 Openflow 模式中的。控制器的 hand-over（切换）机制由控制器自己管理，这能够使其从失败中快速恢复或者使其负载平衡。控制器通过现有规范之外的模式来调节对交换机的管理。多控制器的目的是帮助同步控制器的传输。多控制器的功能基于控制器容错和负载平衡，而不是基于 Openflow 协议之外的虚拟化。

当 Openflow 操作启动，交换机必须连接到所有与其配置相关的控制器，并且尽力保持与他们的连接。许多控制器发送 controller-to-swith 命令到交换机，有关此命令的回复消息或错误消息必须被返回通过相关的控制器连接。异步消息可能发送给多个控制器，为每一个 Openflow 通道复制一个消息，当控制器允许时就发出去。

控制器默认的身份是 OFPCR_ROLE_EQUAL。以这种身份，控制器能够完全访问交换机，这对其他控制器是一样的。默认地，控制器接收交换机所有的异步消息（比如包输入消息，流删除）。控制器发送 controller-to-swith 命令来改变交换机的状态。交换机与众控制器之间互不干涉，也不进行资源共享。

控制器可要求更换身份到 OFPCR_ROLE_SLAVE。这样，控制器以只读方式接入到交换机。控制器默认不接收交换机的异步消息，除了端口状态消息。控制器拒绝执行发送数据或改变交换机状态的 controller-to-swith 命令，例如，OFPT_PACKET_OUT, OFPT_FLOW_MOD, OFPT_GROUP_MOD, OFPT_PORT_MOD, OFPT_TABLE_MOD 请求，以及 OFPMP_TABLE_FEATURES 非空体复合请求必须拒绝。如果控制器发送以上命令中的一个，那么交换机必须返回一个携带 OFPET_BAD_REQUEST 类型字段和 OFPBRC_IS_SLAVE 代码段的 OFPT_ERROR 消息。其他 controller-to-swith 消息，例如 OFPT_ROLE_REQUEST, OFPT_SET_ASYNC and OFPT_MULTIPART_REQUEST 这些查询数据的消息，应该正常进行处理。

控制器也可更换身份为 OFPCR_ROLE_MASTER。这个角色与 OFPCR_ROLE_EQUAL 相似的，可完全访问交换机，不同的是，控制器要确保它的角色是唯一的。当控制器换到此身份时，交换机让 OFPCR_ROLE_MASTER 控制器改变为 OFPCR_ROLE_SLAVE，但是不会影响到

OFPCR_ROLE_EQUAL 控制器。当交换执行身份切换操作时，没有消息生成发送发到正进行身份切换的控制器（大多数情况下控制器不再可到达）。

每个控制器会发送 OFPT_ROLE_REQUEST 消息给交换机让其知道自己的身份，交换机必须记住每个连接控制器的身份。控制器可能随时改变身份，只要在当前消息中提供 generation_id。

身份请求消息提供了一种轻量级机制来帮助控制器管理选举进程,控制器互相协调来配置自己身份。交换机不能直接的改变控制器的状态，这是由另一个控制器发出的消息作用的结果。任何的从控制器或者对等控制器（equal controller）可选择自己的主控制器。交换机可能同时连接到多个对等控制器，或多个从控制器，但最多连接一个主控制器。主控制器和对等控制器能完全改变交换机的状态，控制器没有分割交换机的机制。如果主控制器需要作为唯一能改变交换机的控制器，其它控制器应该全是从控制器，而不能有对等控制器。

控制器可控制在 Openflow 通道中传输的异步消息的类型，并改变上述的默认值。通过异步配置消息（6.1.1）列出需要使能或过滤的消息类型的所有原因。通过这个特性，不同的控制器可以接收不同的通知，主控制器可以选择性的禁用其不关心的通知，从控制器可以启动它想监控的通知。

为了检测主从控制器切换时产生的无序消息，OFPT_ROLE_REQUEST 消息包含了一个 64 位序列字段，generation_id，来标识主控身份。作为主控选举机制的一部分，控制器（或者第三方）协调 generation_id 的分配。generation_id 是单调递增的计数器：每次主控关系改变时，会分配一个新的（更大的）generation_id，例如，每当指定一个新的主控制器，generation_id 会增加。

当接收到身份是 OFPCR_ROLE_MASTER 和 OFPCR_ROLE_SLAVE 控制器发来的 OFPT_ROLE_REQUEST 消息，交换机必须将其包含的 generation_id 消息与之前最大的 generation_id 比较，若较小的话，则丢弃。 交换机必须用错误消息回应，错误消息的类型是 OFPET_ROLE_REQUEST_FAILED，代码是 OFPRRFC_STALE。

下面的伪代码描述了交换机处理 generation_id 的行为。

交换机的启动：

```
generation_is_defined = false;
```

接收到身份是 OFPCR_ROLE_MASTER 和 OFPCR_ROLE_SLAVE 控制器发来的 OFPT_ROLE_REQUEST 消息，并携带一个给定的 generation_id，代码 GEN_ID_X:

```
if (generation_is_defined AND
distance(GEN_ID_X, cached_generation_id) < 0) {
<discard OFPT_ROLE_REQUEST message>;
<send an error message with code OFPRRFC_STALE>;
} else {
cached_generation_id = GEN_ID_X;
generation_is_defined = true;
<process the message normally>;
}
```

distance()作为计算卷序列号距离的操作定义如下：

```
distance(a, b) := (int64_t)(a - b);
```

I.e.distance（）表示序列号之间无符号的差异，被解读为一个有符号的二进制补码值。

当 a 大于 b 且满足小于“序列长度的一半”，则为正值，若 a 小于 b 则为负值。

如果消息请求身份是对等身份，交换机则忽略 generation_id，因为 generation_id 是用来协调主从身份转换的。

6.3.5 辅助连接

默认情况下，Openflow 通道是一个独立的网络连接。通道可能由一个主连接和多个辅连接组成。辅助连接由交换机创建，有利于改善交换机的性能和充分利用交换机的并行性。

交换机到控制器的每个连接，都由 Datapath ID 和 Auxiliary ID 来确认。主连接必须设置 Auxiliary ID 为 0，而辅连接须有一个非零的 Auxiliary ID 和相同的 Datapath ID。辅助连接必须使用和主连接相同的 IP 地址，但是可以使用不同的传输方式，例如，TLS，TCP，DTLS 或者 UDP，这取决于交换机的配置。辅助连接应该有和主连接相同的目标 IP 地址和相同的传输目的端口，除非交换机配置了指定值。控制器必须识别带有非零 Auxiliary ID 的连接，并绑定到具有相同 Datapath ID 的主连接。

交换机不能在主连接完成建立之前启动辅连接（见 6.3.1），因为只有确保主连接存在，辅连接才会被设置和维持（见 6.3.1）。辅连接的建立和主连接是相同的。如果交换机发现到控制器的主连接断了，则必须立即关闭所有到那个控制器的辅连接，这样使控制器正确的解决 Datapath ID 的冲突。

交换机和控制器必须接收所有连接的全部消息类型和子类型：主连接和辅连接不会限制一个指定的消息类型和子类型。但是，在不同的连接上不同的消息类型处理性能可能会不同。交换机可能以不同的优先级来处理辅连接，例如优先级高的就优先处理其消息。使用 OpenFlow 配置协议的交换机，可能随机的赋予辅连接优先级。

Openflow 请求的回应必须用相同的连接返回。连接之间是不同步的，不同连接的消息可能以任意的顺序来处理。一个屏障消息仅用于使用它的连接（见 6.2）。使用 DTLS 或 UDP 的辅连接可能丢失或重排消息，Openflow 在辅连接上不提供排序或传输保证。如果消息必须要按序处理，那么他们则通过相同的连接发送，此连接不重新对包排序，并且使用屏蔽消息。

在整个运行期间，控制器自由的使用各种交换机连接来发送消息，但是为了最大化交换机的性能，提出以下建议：

1. 非包出消息（流模型，统计请求）的控制器消息应通过主连接传送。
2. 含有包入消息的包出消息，应从包入的连接发送出去。
3. 其他的包出消息通过辅连接发送，通过某种机制保证同一个流的数据包映射到相同的连接。
4. 如果需要的辅连接找不到，控制器应该利用主连接。

交换机可自由的使用不同的控制器连接来发送消息，尽管有下面的指导建议：

1. 所有非包出的 Openflow 消息应通过主连接传送。
2. 所有包出消息通过不同的辅连接发送，但通过某种机制保证同一个流的数据包映射到相同的连接。

建立在不可靠传输（UDP,DTLS）之上的辅连接有额外的限制和规则，这些限制和规则不适用于 TCP，TLS。不可靠的辅连接支持的消息类型有如下：OFPT_HELLO, OFPT_ERROR, OFPT_ECHO_REQUEST, OFPT_ECHO_REPLY, OFPT_FEATURES_REQUEST, OFPT_FEATURES_REPLY, OFPT_PACKET_IN, OFPT_PACKET_OUT 和 OFPT_EXPERIMENTER, 其他类型的不被规范支持。

在不可靠的辅连接上，HELLO 消息在连接启动时被发送用来设置连接（见 6.3.1）。如果 Openflow 设备在接收到一个 HELLO 消息之前，接收来自不可靠辅连接的另一个消息，则设备

要么假定连接已合理设置，并且使用来自消息的版本号，要么返回一个携带 OFPET_BAD_REQUEST 类型和 FPBRC_BAD_VERSION 代码的消息。如果 Openflow 设备从辅连接之上，接收到一个携带 OFPET_BAD_REQUEST 类型和 FPBRC_BAD_VERSION 代码的消息，那么它必须发送一个新的 Hello 消息或者终止这个不可靠的辅连接（连接可能稍后重试）。如果辅连接上的消息在所选时间（低于 5 秒）之内没有被接收，则设备必须发送一个新的 Hello 消息或者终止不可靠的辅连接。如果在发送一个 Feature Request 消息之后，控制器在所选时间之内没有收到一个 Feature Reply 消息，则设备必须发送一个新的 Feature Request 消息或者终止不可靠的辅连接。如果在接收到消息之后，设备在 30 秒内的所选时间没有接收任何消息，那么设备就终止不可靠的辅连接。如果设备接收到一个已中止的不可靠辅连接消息，设备必须假定他是一个新的连接。

使用不可靠的辅连接的 Openflow 设备，应尽可能遵循 RFC 5405 的规范。

6．4 流表修改信息

流表修改信息有以下类型：

```
enum ofp_flow_mod_command {
OFPFC_ADD = 0, /* 新流表. */
OFPFC_MODIFY = 1, /*修改所有匹配表. */
OFPFC_MODIFY_STRICT = 2, /*严格匹配通配符和修改条目优先级. */
OFPFC_DELETE = 3, /*删除所有匹配表. */
OFPFC_DELETE_STRICT = 4, /*严格匹配通配符和删除条目优先级. */
};
```

携带 OFPFF_CHECK_OVERLAP 标志设置的添加请求(OFPFC_ADD),交换机必须首先检查请求表中任何重叠流表项。如果一个数据包同时匹配两个，而且两个项有相同的优先级，两个流表项就重叠。如果一个重叠冲突出现在已有流表项和添加请求之间，交换机必须拒绝添加，并且回馈一个 OFPET_FLOW_MOD_FAILED 类型和 OFPFMFC_OVERLAP 代码的 ofp_error_msg。

对于没有重叠的添加请求，或者他们没有重叠的检查，交换机必须在请求表中插入流表项。如果在请求表中已经有了含有完全相同的匹配字段和优先级的一个流表项，则已有的流表项，包括其生存期，必须从表中删除，新的流表项将被添加。如果设置 OFPFF_RESET_COUNTS 标志，流表项计数器必须清零，否则他们应该从替换的流表项中复制。作为添加请求的一部分，流表项删除不会生成 flow-removed 消息；如果控制器想要一个 flow-removed 消息，它应当在添加一个新流表项之前对旧的流表项发送一个明确的删除请求。

关于修改请求 (OFPFC_MODIFY 或 OFPFC_MODIFY_STRICT)，如果匹配项出现在表中，表项中的指令字段更新为请求的值，然而它的 cookie ，idle_timeout, hard_timeout, 标记，计数器，和生存字段不发生改变。如果 OFPFF_RESET_COUNTS 标记被设定，流表项计数器必须被清零。关于修改请求，如果当前请求表中没有流表项匹配这个请求，则没有错误记录，也没有流表修改发生。

对于删除请求(OFPFC_DELETE 或 OFPFC_DELETE_STRICT)，如果一个匹配项出现在表中，

它必须被删除，如果有 OFPFF_SEND_FLOW_REM 标记集，它必须发生一个流移除信息。对于删除请求，如果当前在请求表中没有流表项来匹配请求，则没有错误记录，也没有流表修改发生。

修改和删除流 flow- mod 命令有非严格的版本(OFPFC_MODIFY and OFPFC_DELETE)和严格的版本(OFPFC_MODIFY_STRICT or OFPFC_DELETE_STRICT)，在严格的版本中，匹配字段集,所有匹配字段,包括他们的掩码、优先级，是和表项严格匹配的,只有一个相同的流表项被修改或删除。比如，发送一个移除表项的信息，如果没有字段能够匹配，则 OFPFC_DELETE 命令就要从表中删除所有的流表项，而 OFPFC_DELETE_STRICT 命令只会删除一个应用在指定优先级数据包上的流表项。

关于非严格修改和删除命令，与流模式描述相匹配的所有流表项都会被修改和删除。在非严格版本，当流表项正好匹配或者比流模式命令描述的更多，一个匹配就会产生。在流模式中失配字段变为通配的，字段掩码是有效的，比如优先级等其他的流模式字段则被忽略。比如，如果一个 OFPFC_DELETE 命令去删除目标端口为 80 的所有流表项，那么通配所有匹配字段的流表项将不会被删除。然而，通配所有字段的一个 OFPFC_DELETE 命令将会删除一个匹配端口 80 的表项。同样的解释混合通配符和精确匹配字段也适用于单个和汇聚流数据请求。

目标组或输出端口可以有选择地过滤删除命令。如果输出端口字段包含一个值除了 OFPP_ANY。匹配时它引入一个约束，这个约束是,每个匹配流表项必须包含一个针对指定端口的输出行动。这个约束只对直接与流表项关联的行动进行限制。换句话说,交换机不能通过点到组的行动集递归,这可能已经匹配输出操作。out_group,如果不同于 OFPG_ANY,对组引入了类似的限制行动。这些字段被 OFPFC_ADD,OFPFC_MODIFY 和 OFPFC_MODIFY_STRICT 消息忽略。

删除和修改命令也能够通过 cookie 值过滤。如果 cookie-mask 字段包含一个值而不是 0. 这种约束就是流模式 cookie 字段中 cookie_mask 指定的位和流条目的 cookie 值必须相同。换句话说(flow_entry.cookie & flow_mod.cookie_mask) == (flow_mod.cookie & flow_mod.cookie_mask)。

删除命令针对 table_id 可使用 OFPTT_ALL 值来表示匹配的流表项将被从所有流表中删除。

如果流表修改消息指定一个无效的 table_id。交换机必须发送一个带有 OFPET_FLOW_MOD_FAILED 类型和 OFPFMFC_BAD_TABLE_ID 代码的 ofp_error_msg 消息。

如果流修改消息在添加和修改请求中对 table_id 指定了 OFPTT_ALL，交换机必须发送相同的错误消息。

当向请求表中添加进入的流表项时，交换机不能在请求表中找到空间。交换机必须发送一个带有 OFPET_FLOW_MOD_FAILED 类型和 OFPFMFC_TABLE_FULL 代码的 ofp_error_msg 消息。

如果流模式消息中请求指令不明白，交换机必须发送一个带有 OFPET_BAD_INSTRUCTION 类型和 OFPBIC_UNKNOW_INTS 代码的 ofp_error_msg 消息。如果流模式消息中请求指令是不受支持的，交换机必须发送一个带有 OFPET_BAD_INSTRUCTION 类型和 OFPBIC_UNSUP_INTS 代码的 ofp_error_msg 消息。

如果请求指令包含 Goto-Table 和 next-table-id，指向一个无效的表，交换机必须返回一个带有 OFPET_BAD_INSTRUCTION 类型和 OFPBIC_BAD_TABLE_ID 代码的 ofp_error_msg 消息。

如果请求指令包含 Write-Metadata，而且元数据值或元数据掩码的值是不支持的，交换机必须返回一个带有 OFPET_BAD_INSTRUCTION 类型和 OFPBIC_UNSUP_METADATA 或 OFBIC_UNSUP_METADATA_MASK 代码的 ofp_error_msg 消息。。

如果流模式消息指定的一个字段,匹配时在表中是不支持的，交换机必须返回一个带有 OFPET_BAD_MATCH 类型和 OFPBIC_BAD_FIELD 代码的 ofp_error_msg 消息。如果流模式消息指

定的一个字段,匹配时不止一次，交换机必须返回一个带有 OFPET_BAD_MATCH 类型和 OFPBIC_DUP_FIELD 代码的 ofp_error_msg 消息。如果匹配交换机流消息指定字段,但未能指定相关的先决条件,例如指定一个 IPv4 地址而没有匹配 EtherType 到 0x800,交换机必须返回一个携带 OFPET_BAD_MATCH 类型和 OFPBMC_BAD_PREREQ 代码的 ofp_error_msg。

流模式特定的数据链路或网络地址一个随机位掩码匹配,如果交换机不能支持,交换机必须返回一个带有 OFPET_BAD_MATCH 类型和 OFPBMC_BAD_DL_ADDR_MASK 或 OFPBMC_BAD_NW_ADDR_MASK 的 ofp_error_msg 消息。如果 bitmasks 中指定不支持数据链接和网络地址就应使用 OFPBMC_BAD_DL_ADDR_MASK。如果流模式指定另一个字段的一个随机位掩码匹配不能支持,交换机必须返回一个带有 OFPET_BAD_MATCH 类型和 OFPBMC_BAD_MASK 代码的 ofp_error_msg 消息。

如果流模式指定值不能匹配，例如，一个大于 4095 的 VLAN ID，与一个非保留值或两个高位比特设置一个的 DSCP 值,交换机必须返回一个带有 OFPET_BAD_MATCH 类型和 OFPBMC_BAD_VALUE 代码的 ofp_error_msg 消息。

如果任意行动作用到交换机的一个不再有效端口，交换机必须返回一个带有 OFPET_BAD_ACTION 类型和 OFPBAC_BAD_OUT_PORT 代码的 ofp_error_msg 消息。如果有关的端口也许在将来会是有效的，例如当一个 linecard 添加到传统交换机,或动态添加一个端口到软交换机,交换机要么丢弃发送到该端口的数据包,要么立即返回一个 OFPBAC_BAD_OUT_PORT 错误并拒绝此流模式。

如果流模式消息中的一个行动涉及到一个交换机没有定义的组，或是一个保留组比如 OFPG_ALL,交换机必须返回一个带有 OFPET_BAD_ACTION 类型和 OFPBAC_BAD_OUT_GROUP 代码的 ofp_error_msg 消息。

如果流模式消息的行动有一个无效的值,例如 Set VLAN ID 的行动值大于 4095,或用无效 Ethertype 进行 Push 行动,交换机必须返回一个带有 OFPET_BAD_ACTION 类型和 OFPBAC_BAD_ARGUMENT 代码的 ofp_error_msg 消息。

如果一个流模式消息中的行动执行一个与匹配不一致的操作,比如,一个弹出 VLAN 行动,而匹配没有指定 VLAN，或使用通配 Ethertype 的匹配来设置 IPv4 地址，交换机可以选择拒绝此流模式并立即返回一个带有 OFPET_BAD_ACTION 类型和 OFPBAC_MATCH_INCONSISTENT 代码的 ofp_error_msg 消息。任何不一致的行动对匹配数据包的影响没有定义，强烈建议控制器避免产生可能引起不一致行动的表项组合。

如果一个行动列表有一系列的行动,而交换机不能按其指定次序支持,交换机必须返回一个带有 OFPET_BAD_ACTION 类型和 OFPBAC_UNSUPPORTED_ORDER 代码的 ofp_error_msg 消息。

在流模式消息处理期间发生任何其它的错误，交换机可能会返回一个带有 OFPET_FLOW_MOD_FAILED 类型和 OFPFMC_UNKNOWN 代码的 ofp_error_msg 消息。

6.5 组表修改信息

组表修改信息有以下类型：

```
/* 组命令 */
enum ofp_group_mod_command {
OFPGC_ADD = 0, /*新组. */
OFPGC_MODIFY = 1, /* 修改所有匹配组. */
OFPGC_DELETE = 2, /* 删除所有匹配组. */
};
```

组可能含有零个或零个以上的储存段。没有储存段的组将不会更改与包有关的行动集。如果交换机支持的话，一个含有储存段的组他们本身也可转向其他的组。

行动集对每个储存段要验证，使用的规则就是与流模式相同的规则（6.4 节），并进行组指定的校验。如果在一个储存段中一个行动是无效的或不支持，交换机应当返回一个 ofp_error_msg 和 OFPET_BAD_ACTION 类型，以及相应的错误代码。（见 6.4）

对于 add 请求(OFPGC_ADD)，如果组表中已经有了指定标识符的组表项， 交换机就会拒绝添加组表项。并且发送一个带有 OFPET_GROUP_MOD_FAILED 类型和 OFPGMFC_GROUP_EXISTS 代码的 ofp_error_msg 消息。

对于修改请求(OFPGC_MODIFY)，如果组表中已经有了指定标识符的组表项，那么包括它的类型和行动储存段，必须被移除，然后新组表项被添加。如果指定标识符的组中此组表项不存在，交换机必须拒绝此流模式并发送一个带有 OFPET_GROUP_MOD_FAILED 类和 OFPGMFC_UNKNOWN_GROUP 代码的 ofp_error_msg 消息。

如果一个指定的组类型是无效的，（比如：指定组类型中没有定义的像**权重**这样的字段）交换机必须拒绝添加组表项，并且发送一个 ofp_error_msg 伴随 OFPET_GROUP_MOD_FAILED 类型和 OFPGMFC_INVALID_GROUP 代码。

如果交换机在所选组中不支持非均等负荷共享（存储段的权重不等于 1），必须拒绝添加这个组表项，必须发送一个带有 OFPET_GROUP_MOD_FAILED 类和 OFPGMFC_WEIGHT_UNSUPPORTED 代码的 ofp_error_msg 消息。

如果由于缺少空间，交换机不能添加到来的组表项，交换机必须发送一个带有 OFPET_GROUP_MOD_FAILED 类和 OFPGMFC_OUT_OF_GROUPS 代码的 ofp_error_msg 消息。

如果由于组储存段数量限制的约束(硬件或其他原因)，一个交换机不能添加进来的组表项，交换机必须发送一个带有 OFPET_GROUP_MOD_FAILED 类和 OFPGMFC_OUT_OF_BUCKETS 代码的 ofp_error_msg 消息。

如果由于不支持推荐的活动配置，交换机不能添加组，交换机必须发送一个带有 OFPET_GROUP_MOD_FAILED 类和 OFPGMFC_WATCH_UNSUPPORTED 代码的 ofp_error_msg 消息。这包括为不支持活动性的组指定 watch_port 或 watch_group，或对 watch_port 中指定不支持活动性的端口，或对 watch_group 中指定不支持活动性的组。

对于删除请求(OFPGC_DELETE)，如果所标识组的组表里没有组表项，就不会记录错误，也不会发生组表修改。否则，组被删除，包含在组行动中所有组流表项也会被删除。组类型不需要在删除请求中指定。删除不同于没有储存段的添加或修改，以后试图添加组标识符不会造成组已存在的错误。如果希望使用它有效地删除保留在流表项里的组，组可以通过发送一个无指定储存段的修改来删除。

用单个消息删除所有组的话，需要指定 OFPG_ALL 作为组值。

如果交换机支持的话，组可以串联起来，至少有一组指向另一组，或有更复杂的结构。例如一个快速重路由组可能有两个储存段，每一个指向选择的组。如果一个交换机不支持串联组的组，它必须发送一个带有 OFPET_GROUP_MOD_FAILED 类型和 OFPGMFC_CHAINING_UNSUPPORTED 代码的 ofp_error_msg 消息。

一个交换机可支持检测组成链时不能形成循环：如果组将创建一个转发循环，交换机必须拒绝组并且必须发送一个带有 OFPET_GROUP_MOD_FAILED 类型和 OFPGMFC_LOOP 代码的 ofp_error_msg 消息。如果交换机不支持这样的检查，则转发行为就是未定义的。

一个交换机可支持检测被其它组转发的组没有删除：如果交换机由于组关联另一个组而

不能删除，必须拒绝删除组的表项，必须发送一个带有 OFPET_GROUP_MOD_FAILED 类型和 OFPGMFC_CHAINED_GROUP 代码的 ofp_error_msg 消息。如果交换机不支持这样的检查,则转发行为就是未定义的。

若支持组的快速故障切换需要在线监测,以确定要处理的指定储存段。其它组类型不需要实现在线监视,但可以选择应用它。如果一个交换机不能实现在线检查组中的任意储存段,它必须拒绝组并返回一个错误。确定活跃的规则包括:

- 一个端口被认为是活的如果它端口状态设置为 OFPPS_LIVE 标志。端口活性可由交换机 OpenFlow 部分之外的代码管理，代码由 OpenFlow 规范之外来定义，如生成树或 KeepAlive 机制。如果交换机某个端口活跃机制认为端口不活跃，此端口就必须不能认为活跃(OFPPS_LIVE 标志必须未设置),或者端口配置位 OFPPC_PORT_DOWN 表示端口已关闭,或者端口状态 OFPPS_LINK_DOWN 显示链接已拆除。
- 如果 watch_port 不是 OFPP_ANY 而且端口监测是活跃的，或者 watch_group 不是 OFPG_ANY 而且组监测是活跃的，则存储段就认为活跃。
- 如果至少一个储存段是活跃的则一个组就认为是活跃的。

通过监控不同端口的状态控制器可以推断组的活性状态。

6.6 测量修改信息

测量修正信息有以下类型：

```
/*测量指令 */
enum ofp_meter_mod_command {
OFPMC_ADD, /*新测量. */
OFPMC_MODIFY, /*修改指定的测量. */
OFPMC_DELETE, /*删除指定的测量. */
};
```

对于添加请求(OFPMC_ADD) ，如果指定测量标识符的测量项已经存在，交换机必须拒绝添加测量项，并且必须发送一个含有 OFPET_METER_MOD_FAILED 类和 OFPMMFC_METER_EXISTS 码的 ofp_error_msg 消息。

对于修改请求(OFPMC_MODIFY)，如果指定测量标识符的一个测量项已经存在,这个测量项,包括其带宽,必须删除,添加新测量项。如果指定表标识符的测量项不存在,那么交换机必须拒绝 meter mod 并且发送一个 含 OFPET_METER_MOD_FAILED 类和 OFPMMFC_UNKNOWN_METER 码的 ofp_error_msg 消息。

如果交换机由于缺少空间而不能添加进入的测量项，交换机必须发送一个带有 OFPET_METER_MOD_FAILED 类和 OFPMMFC_OUT_OF_METERS 代码的 ofp_error_msg 消息。

由于带宽数量限制的约束（硬件和其他方面），交换机不能添加进入的测量项，必须拒绝添加测量项，并且发送一个带有 OFPET_METER_MOD_FAILED 类和 OFPMMFC_OUT_OF_BANDS 码的 ofp_error_msg 消息。

对于删除请求(OFPMC_DELETE)，如果指定测量标识符的测量项没有存在,就没有错误记录,也没有表发生修改。否则,将删除测量,在指令集中包含此测量的所有流也会被删除。

对于 delete 请求只需要指定测量标识符,其他字段如带宽可以省略。

用 OFPM_ALL 作为测量值的单个消息可删除所有的测量。虚拟的测量永远不会删除,当删除所有的测量时，也不会消除。

7、openflow 协议

Openflow 交换机规范的核心就是建立 Openflow 协议消息的结构体。

下面描述的结构体，定义和枚举都是源于文件 include/openflow/openflow.h，此文件是 openflow 标准规范的一部分。所有结构体都是用填充和 8 字节来对齐，并且由声明核查。所有消息都是以 big-endian 格式发送。

7.1 openflow 头部

每个 openflow 消息都是以 head 开头

```
/*  所有 OpenFlow 的头部 */

struct ofp_header {

uint8_t version; /* OFP_VERSION.协议版本 */

uint8_t type; /* 一个 OFPT_ 常数 */

uint16_t length; /* 包含头部的长度 */

uint32_t xid; /* 与包有关的事件 ID，回复配对请求时使用相同的 ID */

};

OFP_ASSERT(sizeof(struct ofp_header) == 8);
```

openflow 版本指正在使用的协议版本，早期阶段的 openflow 草稿中，用最高位表示实验版本，低位表明协议版本号。现在介绍的协议 1.3.2，版本是 0x04。

长度字段表明了消息的总长度，因此并没有用额外的帧标识去区分每个帧。

类型可以有以下值：

```
enum  ofp_type  {
/*  不可改变的消息  */
```

```
OFPT_HELLO   =  0,    /*  对称消息 */
OFPT_ERROR   =  1,    /* 对称消息 */
OFPT_ECHO_REQUEST    =  2,    /* 对称消息 */
OFPT_ECHO_REPLY  =  3,    /* 对称消息 */
OFPT_EXPERIMENTER    =  4,    /* 对称消息 */

/* 交换机配置消息  */
OFPT_FEATURES_REQUEST =  5, /*  控制器/交换机消息 */
OFPT_FEATURES_REPLY   =  6, /*  控制器/交换机消息 */
OFPT_GET_CONFIG_REQUEST       =  7, /*  控制器/交换机消息 */
OFPT_GET_CONFIG_REPLY     =  8, /*  控制器/交换机消息 */
OFPT_SET_CONFIG   =  9, /*  控制器/交换机消息 */

/* 异步消息 */
OFPT_PACKET_IN    =  10,  /*  异步消息 */
OFPT_FLOW_REMOVED=  11,  /*  异步消息  */
OFPT_PORT_STATUS =  12,  /* 异步消息  */

/* 控制器命令消息  */
    OFPT_PACKET_OUT  =  13,  控制器/交换机消息 */
    OFPT_FLOW_MOD    =  14,  控制器/交换机消息 */
    OFPT_GROUP_MOD   =  15,  控制器/交换机消息 */
    OFPT_PORT_MOD    =  16,  控制器/交换机消息 */
    OFPT_TABLE_MOD   =  17,  控制器/交换机消息 */

    /* 复合消息 */
    OFPT_MULTIPART_REQUEST=  18,  控制器/交换机消息  */
    OFPT_MULTIPART_REPLY =  19,  控制器/交换机消息  */

    /* 屏障消息（见 6.1.1） */
    OFPT_BARRIER_REQUEST  =  20,  控制器/交换机消息   */
    OFPT_BARRIER_REPLY    =  21,  控制器/交换机消息  */

    /* 队列配置消息.  */
    OFPT_QUEUE_GET_CONFIG_REQUEST = 22,/* 控制器/交换机消息*/
    OFPT_QUEUE_GET_CONFIG_REPLY    =  23, /*控制器/交换机消息*/

    /*  控制器角色改变请求消息  */
    OFPT_ROLE_REQUEST     =  24,  控制器/交换机消息     */
    OFPT_ROLE_REPLY  =  25,  控制器/交换机消息  */

    /*  异步消息配置  */
    OFPT_GET_ASYNC_REQUEST=  26,  控制器/交换机消息  */
```

```
    OFPT_GET_ASYNC_REPLY  =  27,  控制器/交换机消息    */
    OFPT_SET_ASYNC    =  28,  控制器/交换机消息      */

/* 测量器和速率限制器配置消息  */
OFPT_METER_MOD   =  29,  控制器/交换机消息    */
};
```

7.1.1 填充

大多数 openflow 消息都包含填充字段，各种类型消息和各种共有的结构体中都有。这些填充字段实际上它们的名字以 pad 开始，填充字段的目的就是使多字节实体和实际处理器边界对齐。

所有消息中共有结构体都是 64 比特对齐。其它类型则根据需要对齐，例如 32 位整数在 32 位边界中对齐。一个填充规则例外就是 OXM 匹配字段，它从不进行填充(7.3.2)。通常 openflow 消息除非明确说明，否则并不填充；另一方面，大多数共有的结构体都在结尾填充。

填充字段应设置为零。一个 openflow 实现必须能接受填充字段中设置的任何值，而且必须忽略填充域中的内容。

7.2 共有结构体

本节介绍各种消息类型使用的结构体。

7.2.1 端口结构体

Openflow 流水线通过端口收发数据包。交换机可以定义物理端口和逻辑端口，openflow 规范定义一些保留端口（4.1）。

物理端口，交换机定义的逻辑端口，OFPP_LOCAL 保留端口结构体描述如下：

```
/*端口说明*/
struct ofp_port {
uint32_t port_no;
uint8_t pad[4];
uint8_t hw_addr[OFP_ETH_ALEN];
uint8_t pad2[2]; /* Align to 64 bits. */
char name[OFP_MAX_PORT_NAME_LEN]; /* Null-terminated */
uint32_t config; /* Bitmap of OFPPC_* flags. */
uint32_t state; /* Bitmap of OFPPS_* flags. */
/* Bitmaps of OFPPF_* that describe features. All bits zeroed if
* unsupported or unavailable. */
uint32_t curr; /* Current features. */
uint32_t advertised; /* Features being advertised by the port. */
uint32_t supported; /* Features supported by the port. */
uint32_t peer; /* Features advertised by peer. */
uint32_t curr_speed; /* Current port bitrate in kbps. */
uint32_t max_speed; /* Max port bitrate in kbps */
};
OFP_ASSERT(sizeof(struct ofp_port) == 64);
```

Port-no 唯一标识交换机内的端口。hw_addr 通常是端口 MAC 地址；OFP_ETH_ALEN 为 6. 名字字段是一个空终止字符串，包含一个可读接口名称。OFP_MAC_PORT_NAME_LEN 值为 16.

配置字段描述端口的管理设置，有以下结构体：

```
/* 物理端口的行为标志。在 ofp_port 用这个标志描述当前端口配置 。用来设定端口行为 。
*/

enum ofp_port_config {

OFPPC_PORT_DOWN = 1 << 0, /* 端口在管理下关闭 */

OFPPC_NO_RECV = 1 << 2, /* 丢弃端口接收的所有包 */

OFPPC_NO_FWD = 1 << 5, /* 丢弃所有转发给该端口的包 */

OFPPC_NO_PACKET_IN = 1 << 6 /* 不向端口发送 packet-in 消息*/

};
```

这个 OFPPC_PORT_DOWN 位表明端口在管理下断开，OpenFlow 不应该使用。这个 OFPPC_NO_RECV 位表明从哪个端口收到的包应该忽略。 这个 OFPPC_NO_FWD 位表明不应该发送包给这个端口。这个 OFPPFL_NO_PACKET_IN 位表明这个端口的包发生漏表行为，不会触发 packet-in 消息给控制器。

一般来说,端口配置位由控制器设置并且不被交换机改变。这些位在交换机实现协议(如：STP 或者 BFD)时非常有用。如果端口配置位被交换机通过其他管理接口改变了，交换机发送一个 OFPT_PORT_STATUS 消息通知控制器发生了改变。 （STP：生成树协议 BFD:快速故障检测标准协议）

状态字段描述了端口的内部状态，有以下结构：

```
/* 物理端口的当前状态，并不是由控制器配置的 */
enum ofp_port_state {
OFPPS_LINK_DOWN = 1 << 0, /* 目前没有物理连接 */
OFPPS_BLOCKED = 1 << 1, /* 端口堵塞 */
OFPPS_LIVE = 1 << 2, /* 有效的快速故障备份组 */
};
```

端口状态位代表物理连接或 openflower 以外的交换机协议。OFPPS_LINK_DOWN 位表明协议连接断开。OFPPS_BLOCKED 位表明 openflowe 以外的交换，例如 802.1D 生成树协议，防止 OFPP_FLOOD 端口的使用。

所有端口状态位都是只读的，不能被控制器改变。当端口标志改变时，交换机发送 OFPT_PORT_STATUS 消息通知控制器。

端口号使用以下规则：

```
/* 端口编号。端口从 1 开始编号。 */

enum ofp_port_no {

OFPP_MAX = 0xffffff00, /* 物理端口和逻辑交换端口的最大号码 */

/* 保留 OpenFlow 端口 (假装输出 "ports"). */

OFPP_IN_PORT = 0xfffffff8, /* 将包从输入端口发出。为了将包发回给输入端口，这个保留端口必须要明确使用。*/
OFPP_TABLE = 0xfffffff9, /* 将包提交给第一个表。NB: 这个目的端口只能在 packet-out 消息中使用 */

OFPP_NORMAL = 0xfffffffa, /*处理常规的 L2/L3 交换 */

OFPP_FLOOD = 0xfffffffb, /*VLAN 中所有物理端口,除了输入端口和阻塞的或者链路断开的端口。 */

OFPP_ALL = 0xfffffffc, /* 除了输入端口外的所有物理端口 */

OFPP_CONTROLLER = 0xfffffffd, /*发给控制器 */

OFPP_LOCAL = 0xfffffffe, /*本地 openflow "port". */

OFPP_ANY = 0xffffffff   /* 通配端口只能在流修改（删除）、流统计请求中使用。筛选所有流，无论是什么输出端口（包括没有输出端口的流）  */

};
```

Curr,advertised,supported,peer 字段表示链路模式（speed and duplexity；10M 到 10G，全双工、半双工），链路类型（铜/光纤），链路特征（自动协商和暂停）。Curr 是当前连接方式、类型、特征；advertised 是广播给对方的连接方式、类型、特征；supported 是支持的链路方式、类型、特征。Peer 是对方的连接方式、类型、特征。

端口特征由如下结构体表示：

```
/* 数据通道中有效端口的特征. */

enum ofp_port_features {

OFPPF_10MB_HD = 1 << 0, /* 支持 10M 半双工速率 */
OFPPF_10MB_FD = 1 << 1, /* 支持 10M 全双工速率 */
OFPPF_100MB_HD = 1 << 2, /* 支持 100M 半双工速率 */
OFPPF_100MB_FD = 1 << 3, /* 支持 100M 全双工速率 */
OFPPF_1GB_HD = 1 << 4, /* 支持 1 Gb 半双工速率 */
```

```
OFPPF_1GB_FD = 1 << 5, /* 支持 1 Gb 全双工速率  */
OFPPF_10GB_FD = 1 << 6, /* 支持 10Gb 全双工速率 */
OFPPF_40GB_FD = 1 << 7, /* 支持 40 Gb 全双工速率 */
OFPPF_100GB_FD = 1 << 8, /* 支持 100 Gb 全双工速率 */
OFPPF_1TB_FD = 1 << 9, /* 支持 1Tb 全双工速率*/
OFPPF_OTHER = 1 << 10, /*其他速率，不在列表中 */
OFPPF_COPPER = 1 << 11, /*铜线媒质. */
OFPPF_FIBER = 1 << 12, /* 光纤媒质 */
OFPPF_AUTONEG = 1 << 13, /* 自动协商 */
OFPPF_PAUSE = 1 << 14, /* 暂停 */
OFPPF_PAUSE_ASYM = 1 << 15 /* Asymmetric pause. */
};
```

多个标志可同时进行设置。如果没有设置端口速度标志，就会使用 max_speed 或者 curr_speed。

curr_speed 表明链路当前传输速率，max_speed 是最大传输速率，以 kbps 为单位。将数字四舍五入适应通常的用法。例如，一个光学的 10Gb 以太网端口应该设置字段为 10000000 而不是 10312500，一个 OC_192 端口应该将字段设置为 100000000 而不是 9953280.
Max_speed 表明了链路的最大速率，而 curr_speed 表明了当前速率。一个有三条链路的 LAG 端口最大速率为 1Gb/s,其中一个端口关闭,一个端口自动协商速率为 1Gb/s,一个端口自动协商速率 100Mb/s,最大速率为 3Gb/s,当前速率为 1.1Gb/s。

7.2.2 队列结构体

一个 openflow 交换机通过简单的排队机制提供有限的 QoS 服务。一个（或多个）队列可以连接到端口，用来与流表项映射。流表项映射的某个队列，就根据这个队列的配置处理。（注：将队列绑定在某个端口，实现有限的流量控制操作）

队列由一个 ofp_packet_queue 结构体来描述：

```
/* 队列的完整描述  */

    struct  ofp_packet_queue  {
    uint32_t  queue_id; /* 指定队列的 ID  */
    uint32_t  port; /*  队列所属的端口 */
    uint16_t  len;  /*  队列的字节长度*/
    uint8_t  pad[6];/*  64-bit  校正*/（与上面的凑成 64 倍数）
    struct  ofp_queue_prop_header  properties[0];  /*特性列表 */
    };
    OFP_ASSERT(sizeof(struct  ofp_packet_queue)  ==  16);
     一组属性，类型和配置来进一步描述每个队列。
    enum  ofp_queue_properties  {
    OFPQT_MIN_RATE  =  1,    /*  最低速率保证  */
    OFPQT_MAX_RATE  =  2,    /* 最大速率 */
```

```
    OFPQT_EXPERIMENTER = 0xffff   /* 实验定义属性 */
    };
```

每个队列属性描述以共同的头部开始：

```
    /* 队列的一般描述。*/
    struct ofp_queue_prop_header {
    uint16_t property; /*  OFPQT_.中的某一个 */
    uint16_t len; /* 属性的长度，包括头部的长度 */
    uint8_t pad[4];    /* 64-bit alignemnt. */
    };
OFP_ASSERT(sizeof(struct ofp_queue_prop_header) == 8);
```

一个最小速率的队列特性使用下列结构和字段：

```
     /* Min-Rate 队列特征描述 */
    struct ofp_queue_prop_min_rate {
    struct ofp_queue_prop_header prop_header; /* prop: OFPQT_MIN, len: 16.
    */
    uint16_t rate; /* 以 0.1%为单位; 禁止大于 1000 */
    uint8_t pad[6];    /* 64-bit 对齐 */
    };
OFP_ASSERT(sizeof(struct ofp_queue_prop_min_rate) == 16);
```

如果没有设定速率，则速率设置为 OFPQ_MIN_RATE_UNCFG，即 0xfff。

一个最大速率队列用的结构体和字段如下：

```
     /* Max-Rate 队列特征描述 */
    struct ofp_queue_prop_max_rate {
    struct ofp_queue_prop_header prop_header; /* prop: OFPQT_MAX, len: 16. */
    uint16_t rate; /*以 0.1%为单位; 禁止大于 1000 */
    uint8_t pad[6];    /* 64-bit 对齐 */
    };
OFP_ASSERT(sizeof(struct ofp_queue_prop_max_rate) == 16);
```

如果没有设定速率，则速率为 OFPQ_MAX_RATE_UNCFG，即 0fff。

一个实验队列属性用的结构体和字段如下：

```
/* Experimenter 队列特征描述 */

struct ofp_queue_prop_experimenter {

struct ofp_queue_prop_header prop_header; /* prop: OFPQT_EXPERIMENTER, len:
16. */
```

```
uint32_t  experimenter; /* Experimenter ID  与 ofp_experimenter_header 结构体形式相同*/

uint8_t  pad[4];    /*  64-bit  对齐  */

uint8_t  data[0];   /*  实验者定义的数据  */

};

OFP_ASSERT(sizeof(struct  ofp_queue_prop_experimenter)  ==  16);
```

标准 Openflow 处理并没有解释其余的实验队列属性，而是由实验者自己定义。

7.2.3 流匹配结构体

一个 openflow 匹配表是由流匹配头部和一序列 0 或者一些流匹配字段组成。

7.2.3.1 流匹配头部

用 ofp_match 结构体描述匹配头部:

```
/* 流的匹配字段 */
struct ofp_match {
uint16_t type; /* One of OFPMT_* */
uint16_t length; /* ofp_match 长度(包括填充字段) */
/* Followed by:
* - Exactly (length - 4) (possibly 0) bytes containing OXM TLVs, then
* - Exactly ((length + 7)/8*8 - length) (between 0 and 7) bytes of
* all-zero bytes
*总之, 需要时就在 ofp_match 中进行填充, 使其为 8 的倍数，结构长度对齐。
*/
uint8_t oxm_fields[0]; /* 0 or more OXM match fields */
uint8_t pad[4]; /* Zero bytes - see above for sizing */
};
OFP_ASSERT(sizeof(struct ofp_match) == 8);
```

type 字段设置为 OFPMT_OXM ，长度字段为包括所有匹配字段在内的 ofp_match 结构体的实际长度。OpenFlow 有效载荷是一组 OXM 流匹配字段。

```
/* 匹配类型表明正在使用的匹配结构体（构成匹配的字段组）。所有匹配结构体中匹配类型在类型字段的开始。"OpenFlow Extensible Match" 类型 与下面介绍的 OXM TLV 格式相符，并且必须所有交换机都支持。 Extensions that define other match types may be published on the ONF wiki. 扩展项支持是可选的。
*/
enum ofp_match_type {
OFPMT_STANDARD = 0, /* 不赞成 */
OFPMT_OXM = 1, /* OpenFlow Extensible Match */
```

};

此规范中唯一有效的匹配类型是 OFPMT_OXM, 不支持 OpenFlow 1.1 匹配类型 OFPMT_STANDARD。如果使用另一种匹配类型，匹配字段和有效载荷可能变得会不一样，但是这超出了本规范的范围。

7.2.3.2 Flow Match Field Structures

流匹配字段用 OpenFlow 可扩展的匹配(OXM)格式来描述，即一种简化的形式 type-length-value (TLV)。每个 OXM TLV 长度从 5 到 259 (包括全部) 字节。 OXM TLVs 没有相同的长度,也没有采取填充来对齐。一个 OXM TLV 的前四个字节是头部,下面接的是 body 部分。

一个 OXM TLV 的头部可看成一个网络字节顺序的 32-bit 字 (见图 4)。

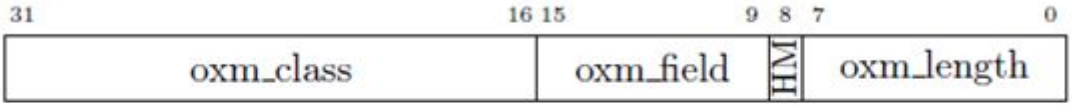

Figure 4: OXM TLV header layout.

OXM TLV 的头部定义如 Table 9

Name		Width	Usage
oxm_type	oxm_class	16	Match class: member class or reserved class
	oxm_field	7	Match field within the class
	oxm_hasmask	1	Set if OXM include a bitmask in payload
	oxm_length	8	Length of OXM payload

Table 9: OXM TLV header fields.

oxm_class 是一个 OXM 匹配类，包含相关的匹配类型，在 7.2.3.3.节有相关介绍。oxm_field 是一个 class-specfic 值, 在匹配类里区分某一个匹配类型。 oxm_class 和 oxm_field 两者(the most-signicant 23 bits of the header)都是 oxm_type。oxm_type 通常表示一个协议的头部字段，例如以太网类型，也可以适用于元数据，例如数据包到达的那个交换机端口。

oxm_hasmask 定义如果 OXM TLV 包含位掩码，在 7.2.3.5 部分有详细介绍。

oxm_length 是以字节为单位的正整数，表示 OXM TLV 负荷长度。OXM TLV 的长度，包括头部，是精确地 4+oxm_length 个字节。

对于一个指定的 oxm_class, oxm_field, and oxm_hasmask 值, oxm_length 是一个常量。包括只允许软件最低限度的解析 OXM TLV 的未知类型。（类似的，若指定 oxm_class, oxm_field, and oxm_length,则 oxm_hasmask 是一个常量。）

7.2.3.3 OXM classes

匹配类型使用 OXM 匹配类进行了结构化，openflow 规范可以区分 OXM 的两种匹配类，ONF 成员类和 ONF 保留类，使用高位比特区分。高位比特为 1 是 ONF 保留类，供 openflow 规范本身使用。高位比特为 0 的是 ONF 成员类，必要时由 ONF 分配，标识一个 ONF 成员，并且可任意使用。对 ONF 成员类的支持是可选的。

OXM 类定义如下：

```
/* OXM Class IDs.
* 高位比特区分保留类和成员类
* 0x0000 到 0x7FFF 的是成员类，由 ONF 分配
*  0x8000 到 0xFFFE 的是保留类，用于标准化。
*/
    enum ofp_oxm_class {
OFPXMC_NXM_0 = 0x0000, /* 与 NXM 后向兼容 */
OFPXMC_NXM_1 = 0x0001, /* 与 NXM 后向兼容 */
OFPXMC_OPENFLOW_BASIC = 0x8000, /* OpenFlow 基本类 */
OFPXMC_EXPERIMENTER = 0xFFFF, /* 实验者类 */
};
```

OPPXMC_OPENFLOW_BASIC 类包含 openflow 匹配域的基本设置（见 7.2.3.7）。可选类 OFPXMC_EXPERIMENTER 用于实验者匹配（见 7.2.3.8）。其他 ONF 保留类用于将来使用，如用于规范的模块化。ONF 成员类的前两个 OFPXMC_NXM_0 和 OFPXMC_NXM_1 用于与 Nicira Extensible Match(NXM)规范的后向兼容。

7.2.3.4 Flow Match

一个 zero—length 的 openflow 匹配(即没有 OXM TLVs)可以匹配每个包。应被通配的匹配字段都从 openflow 匹配中忽略。

OXM TLV 对由 openflow 匹配进行匹配的包设置限制：

如果 oxm_hasmask 为 0，OXM TLV 的 body 包含该字段的值，称为 oxm_value。OXM TLV 匹配仅与相应字段等于 oxm_value 的包相匹配。

如果 oxm_hasmask 为 1，oxm_entry 的 body 包含该字段的值（oxm_value），紧接着是一个相同长度的位掩码，叫为 oxm_mask。oxm_mask 里的每 1_bit 限制 OXM TLV 只能匹配字段与 oxm_value 相应位相等的包。oxm_mask 的 0_bit 没有这样的限制。

当使用掩码时，oxm_mask 中的 0_bit 与 oxm_value 中的 1_bit 一致时产生错误。交换机必须用 OFPET_BAD_MATCH 类型的消息和 OFPBMC_BAD_WILDCARDS 事件报告错误。

下表总结了相对应的 oxm_mask 与 oxm_value 位之间在使用掩码时的约束关系。忽略 oxm_mask 等同于提供一个全是 1-bit 的 oxm_mask。

	oxm_value	
oxm_mask	0	1
0	no constraint	error
1	must be 0	must be 1

Table 10: OXM mask and value.

当有多个 OXM TLV 时，要满足所有的约束关系：包字段必须匹配 openflow 匹配中所有 OXM TLV 的部分。若没有 OXM TLV 的字段，则通配成 ANY，忽略的 OXM TLV 全部掩码为零。

7.2.3.5 流匹配字段掩码

当 oxm_hasmask 为 1，OXM TLV 包含一位掩码并且长度有效地增加一倍，所以 oxm_length 总是奇数,位掩码在字段值之后，使用相同的方式编码。掩码定义某个指定的位为 0，表明与其相应字段相同位的匹配是一种“don’t care”匹配，而 1 意味着要精确匹配。

一个全0比特oxm_mask等同于彻底忽略OXM TLV,一个全1比特oxm_mask等同于指定oxm_hasmask为0，并且忽略oxm_mask。

一些oxm_type不支持掩码的通配符，也就是说，当这些字段被指定时oxm_hasmask必须为0。例如，标识数据包接收的输入端口（接受包的端口）字段可能未被掩码。

一些oxm_type 不支持掩码通配符，可能只支持特定oxm_mask模式。例如，一些有IPv4地址的字段可能被限制为CIDR掩码（子网掩码）。

个别字段的限定会在规范中详细说明。交换机可能接受规范不允许的oxm_hasmask oxm_mask值，只要交换机能正确执行对oxm_hasmask 或oxm_mask值的支持，交换机必须拒绝尝试创建包含它不支持的oxm_hasmask或oxm_mask值的流表项（见6.4）。

7.2.3.6 流匹配字段基础

给定oxm_type的OXM TLV的存在是受到其他OXM TLV的存在或值的限制，一般来说，只有当openflow匹配明确地匹配了相应的协议，才能匹配协议的头部字段。

例如：

只有在另一个表项中oxm_type=OXM_OF_ETH_TYPE,oxm_hasmask=0, 并且oxm_value=0x0800的前提下，才允许OXM TLV 中xom_type=OXM_OF_IPV4_SRC。也就是说，只有以太网的类型明确设置为IPV4时，IPV4源地址的匹配才被允许。

只有在一个表项中oxm_type=OXM_OF_ETH_TYPE,oxm_hasmask=0, oxm_value=0x0800或0x6dd,并且另一个表项中oxm_type=OXM_OF_IP_PROTO,oxm_mask=0,oxm_value=6的前提下，0才允许OXM TLV中 xom_type=OXM_OF_TCP_SRC。也就是说，只有当以太网的类型为IP,IP协议为TCP时，在TCP源端口的匹配才允许。

只有在一个表项中 oxm_type=OXM_OF_ETH_TYPE,oxm_hasmask=0,oxm_value=0x8847或0x8848时，才允许OXM TLV中oxm_type=OXM_OF_MPLS_LABEL。

只有在一个表项中oxm_type=OXM_OF_VLAN_VID,oxm_value!=OFPVID=NONE时,才允许OXM TLV中oxm_type=OXM_OF_VLAN_PCP。

这些对个别字段的限制在规范中进行了说明（见7.2.3.7）。交换机实现时可放宽这些限制。例如，交换机可能在没有任何先决条件的情况下接受。交换机必须拒绝建立试图违反限

制的流表项（见6.4），并且必须处理那些由于缺乏先决条件而不一致的匹配（如，既匹配TCP源端口又匹配UDP目的端口）。

由成员（成员类或作为实验者字段）定义的新的匹配字段可为已有匹配字段提供了备用条件。例如，通过替换ETH_TYPE，在可选链路技术（如PPP）上重复使用用已有的IP匹配字段（对于PPP,那可能是一个假设的PPP_PROTOCOL字段）。

有条件约束的OXM TLV必须出现在那些作为条件的OXM TLV后面，那些在openflow匹配里的OXM TLV的顺序不受约束。

任一指定的oxm_type最多在openflow匹配中出现一次，否则，交换机必须产生一个错误（见6.4）。交换机实现时可放松这个规则，允许在某些情况下多次出现oxm_type,但是，这种匹配行为由具体实现来决定。

如果一个流表实现指定的OXM TLV,这个流表必须接受包含其先决条件的有效匹配，即使这个流表并不支持先决条件指定的匹配域的所有可能值。例如，一个流表匹配IPv4源地址，这个流表必须接受以太网类型精确匹配IPV4，但是不需要支持以太网类型匹配其他任何值。

7.2.3.7 流匹配域

本规范为oxm_class=OFPAMC_OPENFLOW_BASIC的匹配字段定义了一个默认的集合,其值如下：

```
/* OXM Flow match field types for OpenFlow basic class. */
enum oxm_ofb_match_fields {
OFPXMT_OFB_IN_PORT = 0, /* Switch input port. */
OFPXMT_OFB_IN_PHY_PORT = 1, /* Switch physical input port. */
OFPXMT_OFB_METADATA = 2, /* Metadata passed between tables. */
OFPXMT_OFB_ETH_DST = 3, /* Ethernet destination address. */
OFPXMT_OFB_ETH_SRC = 4, /* Ethernet source address. */
OFPXMT_OFB_ETH_TYPE = 5, /* Ethernet frame type. */
OFPXMT_OFB_VLAN_VID = 6, /* VLAN id. */
OFPXMT_OFB_VLAN_PCP = 7, /* VLAN priority. */
OFPXMT_OFB_IP_DSCP = 8, /* IP DSCP (6 bits in ToS field). */
OFPXMT_OFB_IP_ECN = 9, /* IP ECN (2 bits in ToS field). */
OFPXMT_OFB_IP_PROTO = 10, /* IP protocol. */
OFPXMT_OFB_IPV4_SRC = 11, /* IPv4 source address. */
OFPXMT_OFB_IPV4_DST = 12, /* IPv4 destination address. */
OFPXMT_OFB_TCP_SRC = 13, /* TCP source port. */
OFPXMT_OFB_TCP_DST = 14, /* TCP destination port. */
OFPXMT_OFB_UDP_SRC = 15, /* UDP source port. */
OFPXMT_OFB_UDP_DST = 16, /* UDP destination port. */
OFPXMT_OFB_SCTP_SRC = 17, /* SCTP source port. */
OFPXMT_OFB_SCTP_DST = 18, /* SCTP destination port. */
OFPXMT_OFB_ICMPV4_TYPE = 19, /* ICMP type. */
OFPXMT_OFB_ICMPV4_CODE = 20, /* ICMP code. */
OFPXMT_OFB_ARP_OP = 21, /* ARP opcode. */
OFPXMT_OFB_ARP_SPA = 22, /* ARP source IPv4 address. */
OFPXMT_OFB_ARP_TPA = 23, /* ARP target IPv4 address. */
```

```
OFPXMT_OFB_ARP_SHA = 24, /* ARP source hardware address. */
OFPXMT_OFB_ARP_THA = 25, /* ARP target hardware address. */
OFPXMT_OFB_IPV6_SRC = 26, /* IPv6 source address. */
OFPXMT_OFB_IPV6_DST = 27, /* IPv6 destination address. */
OFPXMT_OFB_IPV6_FLABEL = 28, /* IPv6 Flow Label */
OFPXMT_OFB_ICMPV6_TYPE = 29, /* ICMPv6 type. */
OFPXMT_OFB_ICMPV6_CODE = 30, /* ICMPv6 code. */
OFPXMT_OFB_IPV6_ND_TARGET = 31, /* Target address for ND. */
OFPXMT_OFB_IPV6_ND_SLL = 32, /* Source link-layer for ND. */
OFPXMT_OFB_IPV6_ND_TLL = 33, /* Target link-layer for ND. */
OFPXMT_OFB_MPLS_LABEL = 34, /* MPLS label. */
OFPXMT_OFB_MPLS_TC = 35, /* MPLS TC. */
OFPXMT_OFP_MPLS_BOS = 36, /* MPLS BoS bit. */
OFPXMT_OFB_PBB_ISID = 37, /* PBB I-SID. */
OFPXMT_OFB_TUNNEL_ID = 38, /* Logical Port Metadata. */
OFPXMT_OFB_IPV6_EXTHDR = 39, /* IPv6 Extension Header pseudo-field */
};
```

交换机在它的流水线中必须支持表11中列出的required匹配域。每个required匹配域在交换机中必须至少有一个流表支持：流表必须保证匹配那个字段，匹配域的先决条件在表中必须已列出（见7.2.3.6）。required域并不需要在所有流表中实现，也不需要在相同流表中实现。流表可以支持no-requried和实验者匹配域。控制器可以查询交换机中每个流表支持哪些匹配域。

Field		Description
OXM_OF_IN_PORT	*Required*	Ingress port. This may be a physical or switch-defined logical port.
OXM_OF_ETH_DST	*Required*	Ethernet destination address. Can use arbitrary bitmask
OXM_OF_ETH_SRC	*Required*	Ethernet source address. Can use arbitrary bitmask
OXM_OF_ETH_TYPE	*Required*	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_IP_PROTO	*Required*	IPv4 or IPv6 protocol number
OXM_OF_IPV4_SRC	*Required*	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	*Required*	IPv4 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_SRC	*Required*	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	*Required*	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_TCP_SRC	*Required*	TCP source port
OXM_OF_TCP_DST	*Required*	TCP destination port
OXM_OF_UDP_SRC	*Required*	UDP source port
OXM_OF_UDP_DST	*Required*	UDP destination port

Table˜11: Required match fields.

每个匹配域有不同的大小，先决条件，和掩码能力，如表12所示。如果没有明确说明，每个字段类型指的就是数据包头域最外层事件。

Field	Bits	Mask	Pre-requisite	Description
OXM_OF_IN_PORT	32	No	None	Ingress port. Numerical representation of incoming port, starting at 1. This may be a physical or switch-defined logical port.
OXM_OF_IN_PHY_PORT	32	No	IN_PORT present	Physical port. In ofp_packet_in messages, underlying physical port when packet received on a logical port.
OXM_OF_METADATA	64	Yes	None	Table metadata. Used to pass information between tables.
OXM_OF_ETH_DST	48	Yes	None	Ethernet destination MAC address.
OXM_OF_ETH_SRC	48	Yes	None	Ethernet source MAC address.
OXM_OF_ETH_TYPE	16	No	None	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_VLAN_VID	12+1	Yes	None	VLAN-ID from 802.1Q header. The CFI bit indicate the presence of a valid VLAN-ID, see below.
OXM_OF_VLAN_PCP	3	No	VLAN_VID!=NONE	VLAN-PCP from 802.1Q header.
OXM_OF_IP_DSCP	6	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	Diff Serv Code Point (DSCP). Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_ECN	2	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	ECN bits of the IP header. Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_PROTO	8	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	IPv4 or IPv6 protocol number.
OXM_OF_IPV4_SRC	32	Yes	ETH_TYPE=0x0800	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	32	Yes	ETH_TYPE=0x0800	IPv4 destination address. Can use subnet mask or arbitrary bitmask

Table 12 – Continued on next page

Table 12 – concluded from previous page

Field	Bits	Mask	Pre-requisite	Description
OXM_OF_TCP_SRC	16	No	IP_PROTO=6	TCP source port
OXM_OF_TCP_DST	16	No	IP_PROTO=6	TCP destination port
OXM_OF_UDP_SRC	16	No	IP_PROTO=17	UDP source port
OXM_OF_UDP_DST	16	No	IP_PROTO=17	UDP destination port
OXM_OF_SCTP_SRC	16	No	IP_PROTO=132	SCTP source port
OXM_OF_SCTP_DST	16	No	IP_PROTO=132	SCTP destination port
OXM_OF_ICMPV4_TYPE	8	No	IP_PROTO=1	ICMP type
OXM_OF_ICMPV4_CODE	8	No	IP_PROTO=1	ICMP code
OXM_OF_ARP_OP	16	No	ETH_TYPE=0x0806	ARP opcode
OXM_OF_ARP_SPA	32	Yes	ETH_TYPE=0x0806	Source IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_TPA	32	Yes	ETH_TYPE=0x0806	Target IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_SHA	48	Yes	ETH_TYPE=0x0806	Source Ethernet address in the ARP payload.
OXM_OF_ARP_THA	48	Yes	ETH_TYPE=0x0806	Target Ethernet address in the ARP payload.
OXM_OF_IPV6_SRC	128	Yes	ETH_TYPE=0x86dd	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	128	Yes	ETH_TYPE=0x86dd	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_FLABEL	20	Yes	ETH_TYPE=0x86dd	IPv6 flow label.
OXM_OF_ICMPV6_TYPE	8	No	IP_PROTO=58	ICMPv6 type
OXM_OF_ICMPV6_CODE	8	No	IP_PROTO=58	ICMPv6 code
OXM_OF_IPV6_ND_TARGET	128	No	ICMPV6_TYPE=135 or ICMPV6_TYPE=136	The target address in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_SLL	48	No	ICMPV6_TYPE=135	The source link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_TLL	48	No	ICMPV6_TYPE=136	The target link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_MPLS_LABEL	20	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The LABEL in the first MPLS shim header.
OXM_OF_MPLS_TC	3	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The TC in the first MPLS shim header.
OXM_OF_MPLS_BOS	1	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The BoS bit (Bottom of Stack bit) in the first MPLS shim header.
OXM_OF_PBB_ISID	24	Yes	ETH_TYPE=0x88E7	The I-SID in the first PBB service instance tag.
OXM_OF_TUNNEL_ID	64	Yes	None	Metadata associated with a logical port.
OXM_OF_IPV6_EXTHDR	9	Yes	ETH_TYPE=0x86dd	IPv6 Extension Header pseudo-field.

Table˜12: Match fields details.

输入端口 OXM_OF_IN_PORT 是一个有效的标准 openflow 端口，可以是物理端口，逻辑端口，OFPP_LOCAL 保留端口或是 OFPP_CONTROLLER 保留端口。物理端口 OXM_OF_IN_PHY_PORT 在包输入消息中用来标识逻辑端口下层的物理端口（见 7.4.1）。

在多个表间查找时，元数据域 OXM_OF_METADATA 用来传递信息。这个值可被随意覆盖。

隧道 ID 字段 OXM_OF_TUNNEL_ID 携带与逻辑端口有关的可选元数据。元数据的映射由逻辑端口实现定义。如果逻辑端口不支持这样的数据或者数据包从物理端口接收，则值为 0。例如，一个通过 GRE 隧道接收的数据包包括一个（32 位）密钥，密钥存储在低 32 位，高 32 位为 0。对 MPLS 逻辑端口，低 20 位表示 MPLS 标记。对 VxLAN 逻辑端口，低 24 位表示 VNI。

忽略 OFPXMT_OFB_LAN_VID 字段表明流表项应该匹配数据包,无论数据包是否包含相应的标签。对于 VLAN 标签定义了下列特殊值，允许其匹配任何标签的数据包，与标签的值无关，并且支持匹配没有 VLAN 标签的数据包。为 OFPXMT_OFB_VLAN_VID 定义的特殊值如下：

```
/*  VLAN id 是 12-bits, 所以可以用总共 16 位来表示特殊情况
*
*/
```

```
enum ofp_vlan_id {
OFPVID_PRESENT = 0x1000, /* 表示 VLAN id 的 bit 位*/
OFPVID_NONE = 0x0000, /* 没有设置 VLAN id  */
};
```

当 OFPXMT_OFB_VLAN_VID 字段为通配时（不是当前）或者 OFPXMT_OFB_VLAN_VID 设置为 OFPVID_NONE 时必须拒绝 OFPXMT_OFB_VLAN_PCP 字段。表 13 总结了通配位的组合，以及特殊的 VLAN 标签匹配的域值。

OXM field	oxm_value	oxm_mask	Matching packets
absent	-	-	Packets *with* and *without* a VLAN tag
present	OFPVID_NONE	absent	Only packets *without* a VLAN tag
present	OFPVID_PRESENT	OFPVID_PRESENT	Only packets *with* a VLAN tag regardless of its value
present	*value* | OFPVID_PRESENT	absent	Only packets with VLAN tag and VID equal *value*

Table 13: Match combinations for VLAN tags.

OXM_OF_IPV6_EXTHDR 是一个伪字段，表明数据包头部中不同的 IPV6 扩展头部。IPV6 扩展头部比特组合在 OXM_OF_EXTHDR 字段中，这些比特位有下列值：

```
/* IPv6扩展报头定义位为pseudo-field */
enum ofp_ipv6exthdr_flags {
OFPIEH_NONEXT = 1 << 0, /* "No next header" encountered. */
OFPIEH_ESP = 1 << 1, /* Encrypted Sec Payload header present. */
OFPIEH_AUTH = 1 << 2, /* Authentication header present. */
OFPIEH_DEST = 1 << 3, /* 1 or 2 dest headers present. */
OFPIEH_FRAG = 1 << 4, /* Fragment header present. */
OFPIEH_ROUTER = 1 << 5, /* Router header present. */
OFPIEH_HOP = 1 << 6, /* Hop-by-hop header present. */
OFPIEH_UNREP = 1 << 7, /* Unexpected repeats encountered. */
OFPIEH_UNSEQ = 1 << 8, /* Unexpected sequencing encountered. */
};
```

如果IPv6逐跳选项扩展报头作为数据包中第一个扩展报头则OFPIEH_HOP设为1。

如果目前是一个IPv6路由扩展报头则设置OFPIEH_ROUTER为1。

如果目前是IPv6分片扩展报头则设OFIEH_FRAG为1。

如果目前是一个或多个IPv6目的地选项扩展报头则设OFPIEH_DEST为1。IPv6数据包中有一个或两个很正常（见RFC2460）。

如果目前是IPv6身份验证扩展报头则设置OFPIEH_AUTH为1。

如果目前是IPv6封装安全净荷扩展报头则设置OFPIEH_ESP为1。

如果目前IPv6没有下一个头扩展报头怎设OFIEH_NONNEXT设为1。

如果IPv6扩展报头不是RFC的优选项（不要求的），则设OFPIEH_UNSEQ为1。

如果多个IPv6扩展报头不期而遇，则设OFPIEH_UNREP为1。(若是两个目的地的可选报头，将不需要设置该位)。

7.2.3.8 实验者流匹配字段

对 experimenter-specific 流匹配字段的支持是可选的。Experimenter-specific 可使用 oxm_class=OFPXMC_EXPERIMENTER 来定义流匹配字段。OXM TLV body 前 4 个字节包含实验者身份，结构体同 ofp_experimenter（见 7.5.4）。Oxm_field 和 OXM TLV 其余部分都是 experimenter-defined 并且不需要填充或对齐。

```
/* Header for OXM experimenter match fields. */
struct ofp_oxm_experimenter_header {
uint32_t oxm_header; /* oxm_class = OFPXMC_EXPERIMENTER */
uint32_t experimenter; /* Experimenter ID which takes the same
form as in struct ofp_experimenter_header. */
};
OFP_ASSERT(sizeof(struct ofp_oxm_experimenter_header) == 8);
```

7.2.4 流指令结构体

当流匹配了流表项，就执行与这个流表项相关的流指令。定义的指令列表如下：

```
enum ofp_instruction_type {
OFPIT_GOTO_TABLE = 1, /* Setup the next table in the lookuppipeline */
OFPIT_WRITE_METADATA = 2, /* Setup the metadata field for use later inpipeline */
OFPIT_WRITE_ACTIONS = 3, /* Write the action(s) onto the datapath actionset */
OFPIT_APPLY_ACTIONS = 4, /* Applies the action(s) immediately */
OFPIT_CLEAR_ACTIONS = 5, /* Clears all actions from the datapathaction set */
OFPIT_METER = 6, /* Apply meter (rate limiter) */
OFPIT_EXPERIMENTER = 0xFFFF /* Experimenter instruction */
};
```

5.9 节中介绍的指令集，流表可能支持指令类型的子集。指令定义包含指令类型，长度及相关数据。

```
/* 所有指令通用头部。长度包括头部和为了实现 64-bit 对齐的填充部分。
* NB: 指令长度必须为 8 的倍数 */
struct ofp_instruction {
uint16_t type; /* Instruction type */
uint16_t len; /* Length of this struct in bytes. */
};
OFP_ASSERT(sizeof(struct ofp_instruction) == 4);
```

OFPIT_GOTO_TABLE 指令用以下结构体和字段：

```
/*  OFPIT_GOTO_TABLE 的指令结构*/
struct ofp_instruction_goto_table {
uint16_t type; /* OFPIT_GOTO_TABLE */
uint16_t len; /* Length of this struct in bytes. */
uint8_t table_id; /* Set next table in the lookup pipeline */
uint8_t pad[3]; /* Pad to 64 bits. */
};
```

```
OFP_ASSERT(sizeof(struct ofp_instruction_goto_table) == 8);
```

table_id 表示数据包处理流水线中的下一个表。

OFPIT_WRITE_METADATA 指令用以下结构和字段：

```
/* OFPIT_WRITE_METADATA 的指令结构*/
struct ofp_instruction_write_metadata {
uint16_t type; /* OFPIT_WRITE_METADATA */
uint16_t len; /* Length of this struct in bytes. */
uint8_t pad[4]; /* Align to 64-bits */
uint64_t metadata; /* Metadata value to write */
uint64_t metadata_mask; /* Metadata write bitmask */
};
OFP_ASSERT(sizeof(struct ofp_instruction_write_metadata) == 24);
```

下一个表要查找元数据,可以用 metadata 和 metadata_mask 写入匹配字段的指定比特。如果这个指令没有规定，则元数据发送时不需改变。

OFPIT_WRITE_ACTIONS，OFPIT_APPLY_ACTIONS, 和 OFPIT_CLEAR_ACTIONS 指令用以下结构和字段：

```
/*OFPIT_WRITE/APPLY/CLEAR_ACTIONS 的结构体指令*/
struct ofp_instruction_actions {
uint16_t type;                          /* OFPIT_*_ACTIONS 的一种 */
uint16_t len;                           /*此结构体的比特长度 */
uint8_t pad[4];                         /* 64 位对齐*/
struct ofp_action_header actions[0];
/* 0 个或多个与 OFPIT_WRITE_ACTIONS 和 OFPIT_APPLY_ACTIONS 相关的行动 */
};
OFP_ASSERT(sizeof(struct ofp_instruction_actions) == 8);
```

对于 Apply-Actions 指令，行动字段被当做一个列表，此行动应用于有序包。对于 Write-Actions 指令，行动字段作为一个集合，此行动并入当前行动集。

对于 Clear-Actions 指令，此结构体不包含任何行动。

指令 OFPIT_METER 使用以下的结构体和字段。

```
/*OFPIT_METER 的结构体指令*/
struct ofp_instruction_meter {
uint16_t type;          /* OFPIT_METER 的类型*/
uint16_t len;           /* Length is 8. */
uint32_t meter_id;      /* Meter 实例的 ID. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_meter) == 8);
```

meter_id 显示被应用在包上的测量。

指令 OFPIT_EXPERIMENTER 使用以下的结构体和字段：

```
/*OFPIT_EXPERIMENTER 的结构体指令*/
struct ofp_instruction_experimenter {
uint16_t type;                      /* OFPIT_EXPERIMENTER */
```

```
uint16_t len;                          /*结构体的比特长度 */
uint32_t experimenter;       /*Experimenter ID，其格式同 ofp_experimenter_header */
/* 实验者自定义的其它数据. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_experimenter) == 8);
```

7.2.5 行动的结构

许多行动是和流表项、组或数据相关的，目前定义的行动类型如下：

```
   Enum ofp_action_type {
OFPAT_OUTPUT    =0  /*输出到交换机端口*/
OFPAT_COPY_TTL_OUT  =11 /*申请复制 TTL outwards 去打包，从相邻外层到最外层*/
OFPAT_COPY_TTL_IN  =12/*申请复制 TTL inwards 打包，从最外层到相邻外层*/
OFPAT_SET_MPLS_TTL = 15, /* MPLS TTL*/
OFPAT_DEC_MPLS_TTL = 16，/*减少 MPLS TTL*/
OFPAT_PUSH_VLAN = 17, /* 压入一个新的 VLAN 标签
OFPAT_POP_VLAN = 18, /* 弹出最外面的 VLAN 标签*/
OFPAT_PUSH_MPLS = 19, /*压入一个新的 MPLS 标签*/
OFPAT_POP_MPLS = 20, /* 弹出最外面的 MPLS 标签 */
OFPAT_SET_QUEUE = 21, /* 设置输出端口一个队列的 ID*/
OFPAT_GROUP = 22, /*申请组 */
OFPAT_SET_NW_TTL = 23, /* IP TTL. */
OFPAT_DEC_NW_TTL = 24, /*减少 IP TTL. */
OFPAT_SET_FIELD = 25, /* 用 OXM TLV 格式设置一个头部字段*/
OFPAT_PUSH_PBB = 26, /* 压入一个新的 PBB 服务标签 (I-TAG) */
OFPAT_POP_PBB = 27, /* 弹出外面的 PBB 服务标签 (I-TAG) */
OFPAT_EXPERIMENTER = 0xffff
};
```

输出、组、建立队列的行动在 5.12 节进行了描述，压入/弹出的行动在表格 6 进行了介绍，表 12 用 OXM 类型对 Set-Field 的行动进行了描述，一个行动的定义包含了操作的类型、长度和任何关联的数据。

```
   /*行动的头部是所有行动共有的，长度包括它的头部和填充，使行动达到 64-bit。
    *NB：操作的长度必须是 8 的倍数，*/
struct ofp_action_header {
uint16_t type; /* OFPAT _* 之中的一个*/
uint16_t len; /*行动的长度, 包括它的头部和填充，使行动达到 64-bit */
uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_action_header) == 8);
```

一个输出行动是使用下面的结构和字段：

```
    /*OFPAT_OUTPUT 是发送数据包到端口的行动结构。
```

当‘port’是 OFPP_CONTROLLER 时，’max_len’表示发送的最大字节数。一个 0 的’max_len’表示没有字节需要被发送。一个 OFPCML_NO_BUFFER 的 ’max_len’表示数据包没有缓冲并且整个数据包将会被发送到控制器。

```
struct ofp_action_output {
uint16_t type; /* OFPAT_OUTPUT. */
uint16_t len; /*长度是 16. */
uint32_t port; /* 输出端口. */
uint16_t max_len; /*发送给控制器的最大长度. */
uint8_t pad[6]; /* 填充到 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_action_output) == 16);
```

端口指的是数据包将发送的端口。max_len 表示从 OFPP_CONTROLLER 端口发送出去的数据包中数据的最大值。如果 max_len 是 0，这个交换机必须发送 0 字节。OFPCML_NO_BUFFER 中的 max_len 表示整个数据包将发送，并且它不能缓冲。

```
   enum ofp_controller_max_len {
   OFPCML_MAX = 0xffe5,  /* 用来请求一个指定长度的 max_len 最大值*/
   OFPCML_NO_BUFFER = 0xffff /* 表明没有缓冲，整个数据包将会发送到控制器 */
};
```

Group 行动使用下面的结构和字段：

```
/* Action structure for OFPAT_GROUP. */
struct ofp_action_group {
uint16_t type; /* OFPAT_GROUP. */
uint16_t len; /* 长度是 8. */
uint32_t group_id; /* 数组标识符*/
};
OFP_ASSERT(sizeof(struct ofp_action_group) == 8);
```

group_id 标识处理数据包的组，存储段集的使用依赖于组的类型。

Set-Queue 的行动设置队列 id，用来映射一个流表项到已配置的端口队列，不考虑 ToS 和 VLAN PCP，这个数据包不会由于 Set-Queue 行动而发生改变。如果交换机内部处理需要设置 ToS/PCP 比特，那么在发送数据包之前应该恢复其原来的值。

一个交换机可能只支持一个特定的 PCP/ToS 队列。在那种情况下，我们不能随意映射一个流表到那个特定的队列，因此 Set-Queue 行动不被支持。用户仍然可以使用这些队列，并通过设置有关字段(ToS, VLAN PCP)等来映射流表项到队列。

Set-Queue 行动使用下面的结构和字段：

```
/* OFPAT_SET_QUEUE action struct: 发送数据包到端口的给定队列。 */
struct ofp_action_set_queue {
uint16_t type; /* OFPAT_SET_QUEUE. */
uint16_t len; /* Len is 8. */
uint32_t queue_id; /* 数据包的队列 id. */
};
OFP_ASSERT(sizeof(struct ofp_action_set_queue) == 8);
```

Set MPLS TTL 行动使用下面的结构和字段:

```
/*  OFPAT_SET_MPLS_TTL 的行动结构。 */
struct ofp_action_mpls_ttl {
uint16_t type; /* OFPAT_SET_MPLS_TTL. */
uint16_t len; /* 长度是 8. */
```

```
uint8_t mpls_ttl; /* MPLS TTL */
uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_mpls_ttl) == 8);
```

mpls_ttl 字段由 MPLS TTL 来设置。

Decrement MPLS TTL 行动无需协商而且只包含一个ofp_action_header,行动使MPLS TTL值减少。

Set_IPv4 TTL 行动使用下列结构体和字段:

```
/* Action structure for OFPAT_SET_NW_TTL. */
struct ofp_action_nw_ttl {
uint16_t type;            /* OFPAT_SET_NW_TTL. */
uint16_t len;              /* Length is 8. */
uint8_t nw_ttl;           /* IP TTL */
uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_nw_ttl) == 8);
```

nw_ttl 字段是 IP 头部的 TTL 地址。

Decrement IPv4 TTL 行动无需协商而且只包含一个 ofp_action_header，行动使 IP 头部的 TTL 值减少。

Copy TTL outwards 行动无需协商而且只包含一个 ofp_action_header,行动从紧邻最外层的头部复制 TTL 值到最外层头部的 TTL。

Copy TTL inwards 行动无需协商而且只包含一个 ofp_action_header，行动从最外层的头部复制 TTL 值到紧邻最外层头部的 TTL。

Push VLAN header 行动，Push MPLS header 和 Push PBB header 行动使用下列结构体和字段：

```
/* Action structure for OFPAT_PUSH_VLAN/MPLS/PBB. */
struct ofp_action_push {
uint16_t type; /* OFPAT_PUSH_VLAN/MPLS/PBB. */
uint16_t len; /* Length is 8. */
uint16_t ethertype; /* Ethertype */
uint8_t pad[2];
};
```

OFP_ASSERT(sizeof(struct ofp_action_push) == 8);

ethertype 表示 Ethertype 的新标签。当压入一个新的 VLAN 标签、新的 MPLS 头部或者 PBB 业务头部时会使用.

Push PBB header 行动逻辑上会压入一个新的 PBB 业务实例头部到数据包 (I-TAG TCI)，并复制原始的数据包 ethertype 地址到标签的客户地址 (C-DA and C-SA)。 I-TAG 的客户地址就封装在原始的 Ethertype 地址的位置上，因此这个操作可看作把骨干 MAC-in-MAC 头部和 I-SID 字段加到数据包前面。这个 Push PBB header 行动不会把骨干 VANL 头部（B-TAG）加给数据包, 它可在压入 PBB 头部操作之后再通过 Push VlAN header 的操作添加。这个操作之后，常规的 set-field 行动可用来修改外部 Ethertype 地址（B-DA 和 B-SA)。

Pop VLAN header 行动不需要参数，仅由通用 ofp_action_header 组成，这个行动会弹出数据包最外面的 VLAN 标签。

Pop PBB header 行动不需要参数，仅由通用 ofp_action_header 组成。这个行动逻辑上会弹出数据包最外面的 PBB 业务实例头部，并且会复制数据包 Ethernet 地址里的客户地址。这个操作表示从数据包前面移走骨干 MAC-in-MAC 头部和 I-SID 字段。 Pop PBB header 行动不会移走数据包的骨干VLAN头部(B-TAG),它应在此操作之前通过Pop VLAN header行动移走。

Pop MPLS header 的行动使用以下结构体和字段：

```
/*行动结构 for OFPAT_POP_MPLS. */
struct ofp_action_pop_mpls {
uint16_t type;        /* OFPAT_POP_MPLS. */
uint16_t len;        /*长度是 8. */
uint16_t ethertype; /* ethertype*/
uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_pop_mpls) == 8);
```

ethertype 表示 MPLS 负荷的 Ethertype。不管“bottom of stack (BoS)”比特是否在 MPLS 垫片上进行了设置，ethertype 用来作为即将形成的数据包的 Ethertype。 推荐流表项使用这个行动来匹配 MPLS 标签和 MPLS BoS 字段来避免给 MPLS 负荷错误的 ethertype。

当 BoS 不等于 1 时，MPLS 规范不允许对 MPLS 负荷设置任意的 ethertype。控制器负责遵守这个要求，并且只能设置 0x8847 或 0x8848 作为那些 MPLS 负荷的 ethertype。交换机可以随意的执行 MPLS 需求：此情况下，交换机应该拒绝任何与通配 BoS 相匹配的流表项,和在 Pop MPLS header 行动中，利用错误的 ethertype 匹配 BoS 为 0 的流表项。这两种情况都应该返回一个 ofp_error_msg，带有 OFPET_BAD_ACTION 类型和 OFPBAC_MATCH_INCONSISTENT 代码.

Set Field 行动使用以下结构体和行动:

```
/* Action structure for OFPAT_SET_FIELD. */
struct ofp_action_set_field {
uint16_t type;        /* OFPAT_SET_FIELD. */
uint16_t len;        /*长度会被填补到 64 字节. */
/* 接着:
* - 精确的 oxm_len bytes 包括一个单一 OXM TLV, then
* - Exactly ((oxm_len + 4) + 7)/8*8 - (oxm_len + 4) (between 0 and 7)
* bytes of all-zero bytes
*/
uint8_t field[4];            /* OXM TLV - Make compiler happy */
};
OFP_ASSERT(sizeof(struct ofp_action_set_field) == 8);
```

Field 包含使用单一 OXM TLV 结构体的头部字段。 Set-Field 行动由 oxm_type、 OXM TLV 类型、修改相应的数据包头部字段的值 oxm_value 以及 OXM 的负荷定义的。oxm_hasmask 值必须是零而且不包括 oxm_mask。流表项匹配必须包含设置的 OXM 有关字段 （见 7.2.3.6），否则将会生成一个错误（见 6.4）。

set-field 行动的类型可以是任何有效的 OXM 头类型,7.2.3.7 节和表 12 描述了可能的 OXM 类型。因为不是头部字段，不支持 Set_Field 行动的 OXM 类型 OFPXMT_OFB_IN_PORT,OXM_OF_IN_PHY_PORT 和 OFPXMT_OFB_METADATA。Set-Field 行动覆盖指定的 OXM 类型头部字段,并执行所需头部 CRC 计算。OXM 字段是指头部最外层,除非字段类型明确指定其它字段,因此一般 set-field 行动适用于 outermost-possible 头(例如“Set VLAN ID“set-field 行动总是设置最外层 VLAN 标签的 ID)。

一个实验者行动使用以下结构和字段:

```
/* Action header for OFPAT_EXPERIMENTER.
* The rest of the body is experimenter-defined. */
struct ofp_action_experimenter_header {
uint16_t type;          /* OFPAT_EXPERIMENTER. */
uint16_t len;           /*长度是 8 的倍数. */
uint32_t experimenter; /* 实验者 ID 与 ofp_experimenter_header 的结构相同 */
};
OFP_ASSERT(sizeof(struct ofp_action_experimenter_header) == 8);
```

Experimneter 字段表示实验者 ID，需要使用与 ofp_experimenter 相同的形式。

7.3 Controller-to-Switch Messages

7.3.1 握手

控制器使用 OFPT_FEATURES_REQUEST 消息识别交换机和读取它的基本功能。在会话建立(见 6.3.1)基础上，控制器应该发送一个 OFPT_FEATURES_REQUEST 消息。这个消息只包含 OpenFlow 头部。交换机必须发送一个 OFPT_FEATURES_REPLY 响应消息:

```
/* 交换机功能 */
struct ofp_switch_features {
struct ofp_header header;
uint64_t datapath_id; /* 数据通路唯一的 ID。低 48-bits 是 MAC 地址,高 16 位是开发者
                         定义。 */
uint32_t n_buffers;   /*一次缓冲最大的数据包数。 */
uint8_t n_tables;     /* 数据通路支持的表数量。 */
uint8_t auxiliary_id; /*标识辅助连接/
uint8_t pad[2];        /* 64 位对齐. */
/* 功能 */
uint32_t capabilities; /* 位图的支持"ofp_capabilities". */
uint32_t reserved;
};
OFP_ASSERT(sizeof(struct ofp_switch_features) == 32);
```

datapath_id 唯一标识数据通路。低 48 位用于交换机 MAC 地址,高 16 位表示由开发者使用。一个例子就是使用高 16 位作为 VLAN ID 在一个物理交换机里区分多个虚拟交换机实例。这些字段应该被控制器视为一个不透明的位字符串。

n_buffers 字段表示交换机使用 packet-in 消息向控制器发送数据包时指定的最大缓冲

数据包数量。

n_tables 字段描述交换机的表支持的数量,每一种表都可以对支持的匹配字段,行动和表项数量有不同的设置。当控制器和交换机第一次通信,控制器从特性回复中会发现有多少表交换机支持。如果它希望知道查询的表大小、类型和顺序,控制器发送一个 OFPMP_TABLE_FEATURES 复合请求(见 7.3.5.5)。一个交换机必须按照数据包经过表的次序返回表。

auxiliary_id 字段表示交换机到控制器连接的类型,主连接这个字段的设置为零,辅助连接这个字段设置为非零值（见 6.3.5)。

Capabilities 字段使用以下的组合标记：

```
/* Capabilities supported by the datapath. */
enum ofp_capabilities {
OFPC_FLOW_STATS = 1 << 0, /*流量统计. */
OFPC_TABLE_STATS = 1 << 1, /* 表统计. */
OFPC_PORT_STATS = 1 << 2, /* 端口统计. */
OFPC_GROUP_STATS = 1 << 3, /*组统计. */
OFPC_IP_REASM = 1 << 5, /* 可以重新组装 IP 碎片. */
OFPC_QUEUE_STATS = 1 << 6, /*队列统计. */
OFPC_PORT_BLOCKED = 1 << 8 /* 交换机将阻塞循环端口. */
};
```

OFPC_PORT_BLOCKED 位表示交换机协议，这在 OpenFlow 以外,如 802.1 D 生成树,将检测拓扑环路并阻塞端口防止数据包循环。如果没有设置,在大多数情况下,控制器应该实现一个机制来防止数据包循环。

7.3.2 交换机配置

控制器分别使用 OFPT_SET_CONFIG 和 OFPT_GET_CONFIG_REQUEST 消息可以设置和查询交换机配置参数。交换机对配置的要求做出回应 OFPT_GET_CONFIG_REPLY 消息;它没有对设置配置请求进行回复。

OFPT_GET_CONFIG_REQUEST 只有 OpenFlow 头部。OFPT_SET_CONFIG 和 OFPT_GET_CONFIG_REPLY 使用以下结构:

```
/* Switch configuration. */
struct ofp_switch_config {
struct ofp_header header;
uint16_t flags;          /*  OFPC_* flags 位图. */
uint16_t miss_send_len; /*数据通路应该发送给控制器的数据包最大字节数。见
                         ofp_controller_max_len 有效值*/
};
OFP_ASSERT(sizeof(struct ofp_switch_config) == 12);
```

配置标志包括以下：

```
enum ofp_config_flags {
/* Handling of IP fragments. */
OFPC_FRAG_NORMAL = 0,         /* 分组没有特殊处理. */
```

```
OFPC_FRAG_DROP = 1 << 0,        /* 丢弃分组. */
OFPC_FRAG_REASM = 1 << 1,       /* 重新组装(只有 OFPC_IP_REASM 设置). */
OFPC_FRAG_MASK = 3,
};
```

OFPC_FRAG_ *标志表示 IP 分组是否应该正常处理、丢弃,或重组。“normal”处理分组意味着让分组穿过 OpenFlow 表。如果任何字段不存在(如 TCP / UDP 端口不合适),这个数据包应不匹配任何表项里已设置的字段。

当不使用输出行动对 OFPP_CONTROLLER 逻辑端口时,miss_send_len 字段定义了 OpenFlow 流水线发送到控制器的数据包字节数,例如,如果此消息被使能,则发送携带无效 TTL 的数据包。如果这个字段等于 0,交换机必须发送 ofp_packet_in 消息中零字节长的数据包。如果该值设置为 OFPCML_NO_BUFFER,消息中必须包含完整的包,而不会被缓冲。

7.3.3 Flow Table Configuration

流表编号从 0 和任意可取值直到 OFPTT_MAX。OFPTT_ALL 是一个保留值。

```
/* 表编号。表可以使用任何数值直至 OFPT_MAX. */
enum ofp_table {
/* 最后一个可用的表数. */
OFPTT_MAX = 0xfe,
/* 假表. */
OFPTT_ALL = 0xff /* 通配符表用于表配置、流统计和删除. */
};
```

控制器可以分别使用 OFP_TABLE_MOD 和 OFPMP_TABLE 请求对交换机配置和查询表状态。交换机使用 OFPT_MULTIPART_REPLY 消息响应表的复合请求。OFP_TABLE_MOD 使用以下结构和字段:

```
/* 流表的配置/修改行为 */
struct ofp_table_mod {
struct ofp_header header;
uint8_t table_id;     /* 表的 ID,OFPTT_ALL 表示所有表*/
uint8_t pad[3];      /* 填补到 32 字节 */
uint32_t config;     /* 位图的 OFPTC_* flags */
};
OFP_ASSERT(sizeof(struct ofp_table_mod) == 16);
```

table_id 表示配置更改时应选择的表。如果 table_id 是 OFPTT_ALL,则配置应用于交换机中的所有表。

Config 字段是一个位图,早期版本的规范提供向后兼容性,保留以供将来使用。目前该定义的字段没有标记。该字段的值定义如下:

```
/*配置表的标记,保留供将来使用. */
enum ofp_table_config {
OFPTC_DEPRECATED_MASK = 3, /* 废弃的比特 */
};
```

7.3.4 Modify State Messages

7.3.4.1 Modify Flow Entry Message

利用 OFPT_FLOW_MOD 消息控制器完成一个流表修改:

```
/* 流建立和拆除 (controller -> datapath). */
struct ofp_flow_mod {
struct ofp_header header;
uint64_t cookie;        /* 不透明 controller-issued 标识符. */
uint64_t cookie_mask; /* 限制 cookie 位的掩码，当使用 OFPFC_MODIFY *或 OFPFC_DELETE
                         *命令时必须匹配 cookie。值为 0 表示没有限制*/
/* 流行动 */
uint8_t table_id;     /* 放入流的表 ID。对 OFPFC_DELETE_*命令,OFPTT_ALL 也可以用来
                                    删除所有表里匹配的流。 */
uint8_t command;         /*  OFPFC_*之一. */
uint16_t idle_timeout; /*丢弃之前的空闲时间(seconds). */
uint16_t hard_timeout; /* 丢弃之前的最大时间 (seconds). */
uint16_t priority;      /*流表项优先级. */
uint32_t buffer_id; /*缓冲的包使用。或 OFP_NO_BUFFER。对 OFPFC_DELETE *则无意义*/
uint32_t out_port;    /* 对于 OFPFC_DELETE *命令,要求匹配的表项包含这个作为输出端
                         口。OFPP_ANY 表示没有限制*/
uint32_t out_group; /* 对于 OFPFC_DELETE *命令,要求匹配的表项包括这个作为输出组。
                         值 OFPG_ANY 表示没有限制。*/
uint16_t flags;      /* OFPFF_*的标记位图. */
uint8_t pad[2];
struct ofp_match match;    /* 字段相匹配，变量的大小. */
/* 变量的大小和填充匹配总是被指令跟着 */
//struct ofp_instruction instructions[0]; /* . */
};
OFP_ASSERT(sizeof(struct ofp_flow_mod) == 56);
```

cookie 字段是控制器选择的一个不透明的数据值。这个值出现在流删除消息和流统计,也可以用于过滤流统计,流修改和删除(见 6.4)。数据包处理流水线不使用它,因此不需要驻留在硬件里。值-1(0xffffffffffffffff)保留,不得使用。当一个表项通过 OFPFC_ADD 消息插入流表中,其 cookie 字段设置为提供的值。当一个流表项修改(OFPFC_MODIFY 或 OFPFC_MODIFY_STRICT 消息),其 cookie 字段不变。

如果 cookie_mask 字段不为零，当修改或删除流表项时，和 cookie 字段一起限制流匹配。OFPFC_ADD 消息忽略这个字段。cookie_mask 字段的行动在 6.4 节说明。

table_id 字段表示哪个表中的流表项应插入、修改或删除。Table 0 表示流水线上第一个表。OFPTT_ALL 只对删除请求有效。

Command 字段必须是下列之一：

```
enum ofp_flow_mod_command {
```

```
OFPFC_ADD = 0,              /* 新流表. */
OFPFC_MODIFY = 1,           /* 修改所有匹配的流表. */
OFPFC_MODIFY_STRICT = 2, /*修改严格匹配通配符和优先级的表项. */
OFPFC_DELETE = 3,              /* 删除所有匹配的流动表 */
OFPFC_DELETE_STRICT = 4, /*删除严格匹配通配符和优先级的表项. */
}
```

OFPFC_MODIFY 和 OFPFC_MODIFY_STRICT 之间的区别在 6.4 节和解释，OFPFC_DELETE 和 OFPFC_DELETE_STRICT 之间的区别在 6.4 节进行阐述。

Idle_timeout 和 hard_timeout 字段控制流表项过期的速度(见 5.5)。当一个流表项插入表中,其 idle_timeout 和 hard_timeout 字段设置为消息中的值。当一个流表项修改(OFPFC_MODIFY 或 OFPFC_MODIFY_STRICT 消息),idle_timeout 和 hard_timeout 字段被忽略。

如果设置了 idle_timeout 而 hard_timeout 为零,表项在 idle_timeout 秒后没有收到信息后必须过期。如果 idle_timeout 为零而设置了 hard_timeout，无论是否有数据包正碰撞表项，表项到 hard_timeout 秒必须过期。如果 idle_timeout 和 hard_timeout 都设置了,流表项将在没有信息后 idle_timeout 秒，或 hard_timeout 秒后超时,以先到期者为准。如果 idle_timeout 和 hard_timeout 都是零,表项认为是永久的,不会超时。它可以被 OFPFC_DELETE 类型的 flow_mod 消息移除。

Priority 表示指定的流表中表的优先级。较高的数据表示较高的优先级。这一字段仅用于 OFPFC_ADD 消息匹配和添加流表项时,和当 OFPFC_MODIFY_STRICT 或 OFPFC_DELETE_STRICT 消息匹配流条目时。

buffer_id 指向交换机缓冲的数据包并用 packet-in 消息发送给控制器。如果没有缓冲数据包与 flow mod 关联,则必须设置为 OFP_NO_BUFFER。在流插入后,包括一个有效 buffer_id 的 flow mod 移走缓冲区里对应数据包，并从第一个流表开始，穿过整个 OpenFlow 流水线进行处理。这实际上相当于发送了 flow mod 的一个双消息和 packet-out 转发到 OFPP_TABLE 逻辑端口(见 7.3.7),同时要求交换机必须在数据包输出前完全处理 flow mod。无论 flow mod 指向哪个表,或是 flow mod 中包含的指令都适用这些语法。OFPFC_DELETE 和 OFPFC_DELETE_STRICT 消息忽略这一字段。

输出端口和组利用 out_port 和 out_group 字段选择性的过滤 OFPFC_DELETE 和 OFPFC_DELETE_STRICT 消息。如果 out_port 或 out_group 包含一个分别大于 OFPP_ANY 或 OFPG_ANY 的值，就在匹配时引入了一个约束。这个约束就是对端口或组来说，流表项必须包含一个输出行动。其它约束如 ofp_match 结构和优先仍会使用；这纯粹是一个额外的约束。注意若禁用输出过滤,out_port 和 out_group 必须分别设置为 OFPP_ANY 和 OFPG_ANY。这些字段由 OFPFC_ADD、OFPFC_MODIFY 或 OFPFC_MODIFY_STRICT 消息忽略。

Flags 字段可能包含的下列标记:

```
enum ofp_flow_mod_flags {
OFPFF_SEND_FLOW_REM = 1 << 0, /* 发送删除消息当流过期或被删除. */
OFPFF_CHECK_OVERLAP = 1 << 1, /* 首先要看的是重叠的表项. */
```

```
OFPFF_RESET_COUNTS = 1 << 2, /* 重置流数据包和字节计数. */
OFPFF_NO_PKT_COUNTS = 1 << 3, /* 不要监测的数据包计数. */
OFPFF_NO_BYT_COUNTS = 1 << 4, /* 不要监测的字节计数. */
};
```

OFPFF_SEND_FLOW_REM 标志设置时,当流表项过期或删除时交换机必须发送一个流删除消息。

OFPFF_CHECK_OVERLAP 标志设置时,交换机在向流表中插入前,必须检查没有相同优先级的表项冲突。如果有的话,flow mod 失败并返回一个错误信息（见 6.4)。

OFPFF_NO_PKT_COUNTS 标志被设置时,交换机不需要监测流的数据包计数。OFPFF_NO_BYT_COUNTS 标记被设置时,交换机不需要监测流的字节计数。设置这些标记可能减少一些 OpenFlow 交换机的处理负荷,尽管这些计数器在流表项删除消息和流统计时可能不可用。交换机不需要注意这些标记，可跟踪流计数并返回它不管相关标记的设置。如果一个交换机不跟踪流计数,相应的计数器不可用,它必须设置为字段最大值。

当流表项插入到表中,它的 flags 字段设置为消息中的值。当一个流表项匹配和修改(OFPFC_MODIFY 或 OFPFC_MODIFY_STRICT 消息),则 flags 字段被忽略。

Instructions 字段包含了流表项添加或修改表项的指令集。如果指令集是无效或不支持,交换机必须生成一个错误信息（见 6.4)。

7.3.4.2 修改组表项消息

控制器利用 OFPT_GROUP_MOD 消息对组表进行修改:

```
/* 组建立和拆除 (controller -> datapath). */
struct ofp_group_mod {
struct ofp_header header;
uint16_t command;          /* OFPGC_*之一. */
uint8_t type;              /*OFPGT_*之一. */
uint8_t pad;               /*填补到 64 位. */
uint32_t group_id;         /* 组标识 */
struct ofp_bucket buckets[0]; /* 存储段数组的长度，从头部的长度字段计算 */
};
OFP_ASSERT(sizeof(struct ofp_group_mod) == 16);
```

语义类型和组字段解释 6.5 节。

Command 字段必须是下列之一:

```
/* Group commands */
enum ofp_group_mod_command {
OFPGC_ADD = 0,        /* 新建群组. */
OFPGC_MODIFY = 1,     /*修改所有匹配的组. */
OFPGC_DELETE = 2,     /* 删除所有匹配的组。 */
};
```

Type 字段必须是下列之一:

```
/* 组类型。值[128，255] 保留给实验者使用. */
enum ofp_group_type {
OFPGT_ALL = 0,          /* 所有的(多播/广播)组. */
OFPGT_SELECT = 1,       /* 所选定组. */
OFPGT_INDIRECT = 2,     /*间接组 */
OFPGT_FF = 3,            /* 快速故障转移组. */
};
```

group_id 字段唯一地标识在交换机中的组。指定的组标识符定义如下:

```
/* 组编号。组可以使用任何不超过 OFPG_MAX 的编号。 */
enum ofp_group {
/* 最后一个可用的组号. */
OFPG_MAX = 0xffffff00,
/* 假的组. */
OFPG_ALL = 0xfffffffc, /* 表示所有组的组删除命令. */
OFPG_ANY = 0xffffffff    /* 通配符组仅用于流统计请求。选择所有流(包括没有组的流)
*/
};
```

Buckets 字段是一个存储段的数组。对间接组，数组必须包含一个存储段(见 5.6.1),其它组类型在数组里可能有多个存储段。对快速故障转移组,存储段次序就定义了其优先级(见 5.6.1),通过修改组可以改变存储段的排序 (例如使用带 OFPGC_MODIFY 命令的 OFPT_GROUP_MOD 消息)。

数组中的存储段使用以下结构:

```
/*用于组的存储段 */
struct ofp_bucket {
uint16_t len;            /* 存储段字节长度,包括头部和任何对齐 64 位的填充. */
uint16_t weight;         /* 存储段的有关重量，为选择的组定义. */
uint32_t watch_port;/*状态会影响存储段是否活跃的端口。只需要快速转移故障的组。*/
uint32_t watch_group; /*状态会影响存储段是否活跃的组。只需要快速转移故障的组。*/
uint8_t pad[4];
struct ofp_action_header actions[0]; /*与存储段关联的 0 或多个行为，行动列表长度
                                      依据存储段长度计算 */
};

OFP_ASSERT(sizeof(struct ofp_bucket) == 16);
```

Weight 字段只是为所选组定义的，它是否支持是可选的。被组处理的存储段共享信息由组中存储段重量之和平均后的单个存储段重量来定义。当一个端口拆除,信息量分布的改变没

有定义，交换机的数据包分布精确度应与存储段重量匹配，但没有定义。

watch_port 和 watch_group 字段只有快速故障转移组需要,也可能选来实现用在其它组类型上。这些字段表示端口或组，其活跃控制这个存储段是否为转发的候选者。对快速转移故障组,定义的第一个存储段是最高优先级,而且只有最高优先级活跃存储段被使用(见 5.6.1)。

actions 字段是与存储段相关的行动集。当存储段被数据包选中时,其行动集就应用到数据包(见 5.10)。

7.3.4.3 Port Modification Message

控制器使用 OFPT_PORT_MOD 消息修改端口的行为:

```
/* 物理端口的修改行为 */
struct ofp_port_mod {
struct ofp_header header;
uint32_t port_no;
uint8_t pad[4];
uint8_t hw_addr[OFP_ETH_ALEN]; /* 这是不可配置的硬件地址。这是用来对请求进行正常检查,因此它必须返回相同 ofp_port 结构. */
uint8_t pad2[2];      /* 填充到 64 字节. */
uint32_t config;      /*  OFPPC_* 位图标记. */
uint32_t mask;        /* 位图 OFPPC_* 标记的改变. */
uint32_t advertise;   /* 位图的 OFPPF_ *。所有位为零以防止任何行动发生. */
uint8_t pad3[4];       /* 填充到 64 字节. */
};
OFP_ASSERT(sizeof(struct ofp_port_mod) == 40);
```

Mask 字段用来在 config 字段中选择比特去改变。Advertise 字段没有掩码;所有端口特性一起改变。

7.3.4.4 计量器修改消息

利用 OFPT_METER_MOD 消息完成控制器的计量器修改：

```
/* 计量器配置. OFPT_METER_MOD. */
struct ofp_meter_mod {
struct ofp_header header;
uint16_t command;      /* OFPMC_*之一. */
uint16_t flags;        /* OFPMF_*位图的标记. */
uint32_t meter_id;     /* 计量器实例. */
struct ofp_meter_band_header bands[0]; /* 带宽列表长度，从头部长度字段计算. */
};
OFP_ASSERT(sizeof(struct ofp_meter_mod) == 16);
```

meter_id 字段在交换机里唯一地标识计量器。计量器从 meter_id = 1 开始定义，直至交换机可以支持的最大数。OpenFlow 协议还定义了一些额外的不与流表相关的虚拟计量器:

```
/*计量器编号。OFPM_MAX 流表计可以使用任何数直至 OFPM_MAX */
enum ofp_meter {
/* 最后一个可用的计量器. */
OFPM_MAX = 0xffff0000,
/* 虚拟仪表. */
OFPM_SLOWPATH = 0xfffffffd,     /* 用于慢速数据通道的计量器. */
OFPM_CONTROLLER = 0xfffffffe,  /*用于控制器连接的计量器。 */
OFPM_ALL = 0xffffffff, /*表示用于统计请求命令的所有计量器 */
};
```

OpenFlow 现有实现提供支持的虚拟计量器，在新实现中鼓励使用常规流计量器或优先级队列。

OFPM_CONTROLLER:虚拟计量器使用 CONTROLLER 保留端口或其他处理,控制所有数据包通过 Packet-in mes-sages 发送到控制器, (见 6.1.2)。可用来限制发送到控制器的流量。

OFPM_SLOWPATH:虚拟计量器控制所有数据包被缓慢数据通路的交换机处理。许多交换机实现有一条快和慢数据通路,例如硬件交换机可能有一条慢速软数据通路,或软交换机可能有一条慢速的用户空间数据通道。

Command 字段的命令必须是下列之一:

```
/* Meter commands */
enum ofp_meter_mod_command {
OFPMC_ADD,            /* 新的计量器. */
OFPMC_MODIFY,         /* 修改指定的计量器. */
OFPMC_DELETE,         /* 删除指定的计量器. */
};
```

Flags 字段可能包含以下标记的组合:

```
/* 计量器配置标记 */
enum ofp_meter_flags {
OFPMF_KBPS = 1 << 0,       /* 速率值在 kb / s(kilo-bit 每秒. */
OFPMF_PKTPS = 1 << 1,      /* 速率值用包/秒记. */
OFPMF_BURST = 1 << 2,      /* 突发的尺寸. */
OFPMF_STATS = 1 << 3,      /* 收集统计. */
};
```

Bands 字段是一个速率带宽列表。它可以包含任意数量的频带,当可以理解的话，每个频带类型是可以重复的。一次只能使用一个频带,如果当前数据包速度超过多个频带速率,频带则用最高的速率配置。

所有的频带都使用相同的通用头部来定义:

```
/* 所有计量器共同的头部 */
```

```
struct ofp_meter_band_header {
uint16_t type;        /* OFPMBT_*.之一 */
uint16_t len;         /*此频带的字节长度. */
uint32_t rate;        /* 此频带的速率. */
uint32_t burst_size;  /* 突发的的大小 */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_header) == 12);
```

Rate 字段表示速率值，在此频带基础上可以传送数据包(见 5.7.1)。速率值单位是千比特/秒,除非 flags 字段包括 OFPMF_PKTPS,此时速率是包/秒。

burst_size 字段只有在 flags 字段包含 OFPMF_BURST 时使用。对所有长度超过突发值的数据包和突发字节,它规定了计量器的粒度,计量器的速率将严格限制。突发值单位是千比特,除非 flags 字段包括 OFPMF_PKTPS,在这种情况下,突发值单位是包。

Type 字段必须是下列之一：

```
/* 计量器频带类型*/
enum ofp_meter_band_type {
OFPMBT_DROP = 1, /* 丢弃包. */
OFPMBT_DSCP_REMARK = 2, /* IP 头部的 DSCP. */
OFPMBT_EXPERIMENTER = 0xFFFF /* 实验者计量器频带. */
};
```

OpenFlow 交换机可能不支持所有频带类型,并可能不允许在所有计量器上使用它所支持的频带,即一些计量器可能是专用的。

频带 OFPMBT_DROP 定义了简单的速率限制器，超过频带速率值的话数据包将丢弃，并使用以下结构:

```
/* OFPMBT_DROP band - drop packets */
struct ofp_meter_band_drop {
uint16_t type;        /* OFPMBT_DROP. */
uint16_t len;         /* 这个频带的字节长度. */
uint32_t rate;        /* 开始丢弃包的速率 */
uint32_t burst_size; /* 突发大小 */
uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_meter_band_drop) == 16);
```

频带 OFPMBT_DSCP_REMARK 定义了一个简单的 DiffServ 策略，对超过频带速率的数据包的 IP 头部 DSCP 字段，检测其丢弃的优先级，并使用以下结构:

```
/* OFPMBT_DSCP_REMARK 频带- IP 头部 DSCP*/
struct ofp_meter_band_dscp_remark {
uint16_t type;        /* OFPMBT_DSCP_REMARK. */
```

```
uint16_t len;          /*频带的字节长度. */
uint32_t rate;         /* 开始检测数据包的速率. */
uint32_t burst_size; /* 突发大小 */
uint8_t prec_level;  /* 添加的丢弃优先级数。*/
uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_meter_band_dscp_remark) == 16);
```

prec_level 字段表示如果超过频带速率超过某个数值，数据包的丢弃优先级应该增加。

频带 OFPMBT_EXPERIMENTER 是实验者定义并使用以下结构:

```
/* OFPMBT_EXPERIMENTER 频带-行动集中的写行动*/
struct ofp_meter_band_experimenter {
uint16_t type;           /* OFPMBT_*之一. */
uint16_t len;            /*这个频带的字节长度. */
uint32_t rate;           /* 这个频带的速率*/
uint32_t burst_size;     /* 突发大小. */
uint32_t experimenter;   /* 实验者 ID 与 ofp_experimenter_header 结构相同 */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_experimenter) == 16);
```

7.3.5 复合消息

复合消息用于编码请求或应答那些可能携带大量数据而且不能装进单个 OpenFlow 消息（仅限于 64 kb）的情况。请求或应答是编码序列与特定的多部分类型的多部分消息,并佐的接收器。多部分消息主要用于请求数据或从交换机状态信息。

请求被放在一个或多个 OFPT_MULTIPART_REQUEST 消息:

```
struct ofp_multipart_request {
struct ofp_header header;
uint16_t type;          /*  OFPMP_* 之一. */
uint16_t flags;         /* OFPMPF_REQ_* 标记. */
uint8_t pad[4];
uint8_t body[0];        /*请求的 body。0 或多个字节. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_request) == 16);

enum ofp_multipart_request_flags {
     OFPMPF_REQ_MORE = 1 << 0       /* 更多的请求跟随. */
};
```

交换机用一个或多个 OFPT_MULTIPART_REPLY 消息进行响应:

```
struct ofp_multipart_reply {
```

```
struct ofp_header header;
uint16_t type;          /* 一个 OFPMP_*常数. */
uint16_t flags;         /* OFPMPF_REPLY_* 标记. */
uint8_t pad[4];
uint8_t body[0];        /* 回答的 body，0 或更多字节. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_reply) == 16);

enum ofp_multipart_reply_flags {
OFPMPF_REPLY_MORE = 1 << 0          /* 更多回复跟随 */
};
```

在请求和回复中为 flags 定义的值是唯一的,是否更多的请求/回复会跟随这个——这个值就是 0x0001。为了方便实现,控制器允许发送请求和交换机允许发送回复时不携带额外的表项(即一个空的 body)。然而,另一个消息必须使用更多的标志设置去跟随一个消息。跨越多个消息(有一个或多个消息设置了更多的标志)的请求或应答, 消息队列中的所有消息必须使用相同的复合类型和事务 id(xid)。如果多个未回答的复合请求或回复同时在传输中，复合请求或应答的消息可能与其他 OpenFlow 消息类型交替,包括其他复合请求或回复,但必须有不同的事务 id。回复的事务 id 必须匹配激起他们的请求。

在请求和响应中，type 字段指所传递信息的种类，并决定如何解析 body 字段:

```
enum ofp_multipart_type {
/* 描述这个 OpenFlow 交换机。*请求 body 是空的。*回复主体是 struct ofp_desc. */
OFPMP_DESC = 0,

/* 单独流统计。* 请求 body 是 ofp_flow_stats_request 结构。*回复 body 是一个 struct ofp_flow_stats 数组。. */
OFPMP_FLOW = 1,

/*总的流统计。* 请求 body 是 ofp_aggregate_stats_request 结构。*回复 body 是 ofp_aggregate_stats_reply 结构. */
OFPMP_AGGREGATE = 2,

/* 流表统计。*请求 body 是空的。*回复 body 是 ofp_table_stats 结构数组。 */
OFPMP_TABLE = 3,

/* 端口统计。* 请求 body 是 ofp_port_stats_request 结构。*回复 body 是 ofp_port_stats 结构数组。 */
OFPMP_PORT_STATS = 4,



/*一个端口的队列统计
```

```
 *请求的 body 是 ofp_queue_stats_request 结构体.
 *回复 body 是 ofp_queue_stats 结构体数组*/
OFPMP_QUEUE=5,

/*组计数器统计。
*请求的 body 是 ofp_group_stats_request 结构体.
*回复是 ofp_group_stats 结构体数组*/
OFPMP_GROUP=6,

/*组描述。
*请求 body 是空的。
/*回复 body 是 ofp_group_desc 结构体数组*/
OFPMP_GROUP_DESC=7,

/*组特征。
*请求 body 为空。
*回复 body 是 ofp_group_features 结构体。*/
OFPMP_GROUP_FEATURES=8,

/*计量器统计
*请求 body 是 ofp_meter_multipart_requests 结构体。
*回复 body 是 ofp_meter_stats 结构体数组*/
OFPMP_METER=9,

/*计量器配置
*请求 body 是 ofp_meter_multipart_requests 结构体.
*回复 body 是 ofp_meter_config 结构体数组。*/
OFPMP_METER_CONFIG==10,

/*计量器特征
*请求 body 为空。
*回复 body 是 ofp_meter_features 结构体。*/
OFPMP_METER_FEATURES=11,

/*表特征
*请求 body 是空的或包含 ofp_table_features 结构体数组,包含控制器所需的交换机视图。如果交换机不能设置特定的视图，就会返回一个 error.
*回复 body 是一组 ofp_table_features 结构体.*/
OFPMP_TABLE_FEATURES=12,

/*端口说明
*请求 body 是空的。
*回复 body 是 ofp_port 结构体数组*/
OFPMP_PORT_DESC=13,
```

```
/*实验者扩展项
*请求 body 和回复 body 都以 ofp_experimenter_multipart_header 结构体开始。
*而请求 body 和回复 body 是由实验者定义的。*/
OFPMP_EXPERIMENTER=0xffff
};
```

如果一个复合请求跨越多个消息并且扩展到交换机不能缓冲的大小，那么交换机必须回复一个 OFPET_BAD_REQUEST 类型的错误消息并且编码为 OFPBRC_MULTIPART_BUFFER_OVERFLOW。如果一个复合请求包含一个不支持的类型，交换机必须回复一个 OFPET_BAD_REQUEST 类型的错误消息并且编码为 OFPBRC_BAD_MUTIPART.

在所有包含统计数字复合应答中，如果在交换机中没有指定的数字计数器，其值必须被设置为字段最大值（无符号数=-1）。计数器是无符号的并且循环时无溢出指示。

7.3.5.1 说明

关于交换机厂商、硬件修订、软件修订、序列号和描述字段的信息，可以从 OFPMP_DESC multipart 请求类型中获得：

```
/*OFPMP_DESC 请求的回复部分。每个表项都是 NULL 结束的 ASCII 码字符串*/
struct ofp_desc {
char mfr_desc[DESC_STR_LEN]; /* 厂商说明. */
char hw_desc[DESC_STR_LEN]; /* 硬件说明. */
char sw_desc[DESC_STR_LEN]; /* 软件说明. */
char serial_num[SERIAL_NUM_LEN]; /* 序列号. */
char dp_desc[DESC_STR_LEN]; /* 可读的数据通道描述. */
};
OFP_ASSERT(sizeof(struct ofp_desc) == 1056);
```

每个表项都是 ASCII 码格式并且用空字节（\0）从右边填充。DESC_STR_LEN 是 256，SERIAL_NUM_LEN 是 32。dp_desc 是一个自由形式的字符串，调试时用来描述数据通道，例如：“switch3 in room 3120”。因此，它不能被保证是唯一的并且不应用作数据通道的主要标识——使用交换机的 datapath_id 字段来代替。

7.3.5.2 单个的流统计

关于单个流表项的信息使用 OFPMP_FLOW 复合请求类型进行请求：

```
/*OFPMP_FLOW 的 ofp_multipart_request 部分*/
struct ofp_flow_stats_request {
uint8_t table_id;            /* 要读的table_ID (from ofp_table_stats),
                                OFPTT_ALL 表示所有表. */
```

```
uint8_t pad[3];                /* 对齐到32 bits. */
uint32_t out_port;          /* 作为一个输出端口要求包含这个匹配项. 一个OFPP_ANY值表
                               示没有限制。*/
uint32_t out_group;         /* 作为一个输出组要求包含这个匹配项，一个OFPG_ANY值表示
                               没有限制*/
uint8_t pad2[4];            /* 对齐到  64 bits. */
uint64_t cookie;            /* 要求匹配项要包含这个cookie值。*/
uint64_t cookie_mask;       /* 掩码用来限制那些必须匹配的cookie bits . 0 表示没有
                               限制*/
Struct ofp_match match;     /*用来匹配的字段，可变大小*/
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request)==40);
```

匹配字段包含需匹配的流表项说明，还可含有通配符和掩码字段。这些字段的匹配行为在6.4中介绍。

Table_id 表示一个将读取的一个流表的索引值，OFPTT_ALL表示所有流表。

Out_port和out_group字段可又输出端口和组选择过滤。如果out_port或out_group包含一个分别不同于OFPP_ANY和OFPG_ANY的值，它会在匹配的时候引进一个约束。这个约束就是流表项必须包含一个指定端口或组的输出行动。其它约束例如ofp_match结构仍然在使用;这是纯粹一个额外的约束。注意为了禁止输出过滤，out_port和out_group都必须分别设置为OFPP_ANY和OFPG_ANY。

Cookie和cookie_mask字段的用法见6.4部分。

回复一个OFPMP_FLOW 复合请求的部分由下列队列组成：

```
/*回复一个OFPMP_FLOW请求的部分。*/
struct ofp_flow_stats {
uint16_t length;          /* 这个项的长度 */
uint8_t table_id;         /* 流的来源表ID */
uint8_t pad;
uint32_t duration_sec;    /* 流已生成的时间，以秒记 */
uint32_t duration_nsec;   /* 流存在的时间超过duration_sec的时间，以ns记 */
uint16_t priority;        /* 流表项的优先权 */
uint16_t idle_timeout;    /* 在到期之前闲置的秒数 */
uint16_t hard_timeout;    /* 到期之前的秒数*/
uint16_t flags; /*OFPFF_* flags位图. */
uint8_t pad2[4]; /* 对齐到64-bits. */
uint64_t cookie; /* 控制器发出的不透明标识符 */
uint64_t packet_count; /* 流里的包的数目。 */
uint64_t byte_count; /* 流的字节数. */
struct ofp_match match; /* 字段说明. 可变大小 */
/* 可变大小和填充的匹配总是跟随着指令. */
//struct ofp_instruction instructions[0]; /* 指令集-- 0 或者更多. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 56);
```

由创建流表项的flow mod提供的字段（见7.3.4.1），加上插入到流表项中的table_id，packet_count和byte_count对所有被流表项处理的数据包进行计数。

duration_nsec和duration_sec字段显示流表项安装在交换机中已过去的时间。总共持续的时间可用duration_sec*10^9+duration_nsec计算出。实现的话要求提供第二精度；在能够获得的情况下鼓励用更高的精度。

7.3.5.3 总计的流统计

关于多个的流表项的总计信息用 OFPMP_AGGREGATE 复合请求类型来请求：

```
/*OFPMP_AGGREGATE 类型的 ofp_aggregate_atats_request 部分*/
struct ofp_aggregate_stats_request {
uint8_t table_id;  /* 要读取的表ID(来自ofp_table_stats)，OFPTT_ALL为所有的表*/
uint8_t pad[3]; /*排列到32 bits. */
uint32_t out_port; /*包含此项匹配的流表项要求作为一个输出端口. OFPP_ANY值表示没有
                     限制*/
uint32_t out_group; /*包含此项匹配的流表项要求作为一个输出组，OFPG_ANY值表示没有
                    限制*/
uint8_t pad2[4]; /* 排列到 64 bits. */
uint64_t cookie; /*要求正在匹配的流表项去包含这些cookie值*/
uint64_t cookie_mask;/*掩码(mask)用来限制那些必须匹配的cookie比特，0表示没有限制.
                      */
struct ofp_match match; /* 将匹配的字段. 可变大小. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_request) == 40);
```

在这个消息的中的字段与在单独的流统计请求（OFPMP_FLOW）里有相同的含义。

回复部分的组成如下：

```
/*OFPMP_AGGREGATE请求的回复部分. */
struct ofp_aggregate_stats_reply {
uint64_t packet_count; /*流中的数据包数. */
uint64_t byte_count; /* 流中的字节数. */
uint32_t flow_count; /*流的数目. */
uint8_t pad[4]; /* 排列到64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_reply) == 24);
```

7.3.5.4 表统计

请求关于表的信息使用 OFPMP_TABLE 复合请求类型。在请求内不包含任何数据。

回复部分由如下一个数组组成：

```
/*回复给OFPMP_TABLE请求的部分. */
struct ofp_table_stats {
uint8_t table_id; /*表标识符.低数值的表第一个访问 */
uint8_t pad[3]; /*排列到32-bits. */
uint32_t active_count; /* 活跃的流表项数量. */
uint64_t lookup_count; /*表中处理过的数据包数量. */
uint64_t matched_count; /*表中匹配的数据包数量。*/
};
OFP_ASSERT(sizeof(struct ofp_table_stats) == 24);
```

在交换机支持的每一个表中，数组都有一个结构。流表项是以数据包通过表的顺序返回的。

7.3.5.5 表特征

OFPMP_TABLE_FEATURES 复合类型允许一个控制器去查询现存的表的能力和选择要求交换机去重新配置表去匹配一个已有的配置。一般来讲，表的能力体现了一个表的所有可能的特征，然而一些能力互不兼容，目前的能力结构不允许我们去解决兼容问题。

7.3.5..5.1 表特征请求和回复

如果 OFPMP_TABLE_FEATURES 请求是空的，交换机将返回一组包含当前配置流表能力的 ofp_table_features 结构体。

如果请求部分包含一个或更多ofp_table_features结构体的队列，交换机将尝试去改变它的流表去匹配被请求的流表配置。这个操作配置整个流水线，流水线里的流表设置必须和请求的设置匹配，否则必须返回一个error。尤其是，如果这个配置成功进行了设置，则给一个或更多交换机支持的流表的请求结构若不包含一个ofp_table_features结构体，这些流表将从流水线中被移除。配置改变成功会修改请求里面的所有流表的特性，也就是说，要么在请求里被指定的所有流表都修改，要么一个都不改，并且新的流表的功能必须是超集或者和被请求的功能相同。如果流表配置成功了，已删除的流表或在新旧配置里能力改变的流表，其流表项都将从流表中删除。如果交换机不能设置所请求的配置，将返回一个OFPET_TABLE_FEATURES_FAILED类型错误并带有适当错误代码。

含有ofp_table_features的请求和回复至少要满足以下要求:

●每个ofp_table_features结构的table_id字段的值在消息所有的ofp_table_features结构里应该是唯一的。

●每个ofp_table_features结构的属性字段必须包含一个ofp_table_feature_prop_type属性，以及另外两个。第一，the_MISS后缀可能被忽略，如果它和常规流表项相应的属性相同；第二，OFPTFPT_EXPERIMENTER和OFPTFPT_EXPRIMENTER_MISS类型的属性可能会被多次忽略或使用。排序未规定，但鼓励使用在规范上列出的次序。（见7.3.5.5.2）

一个交换机接收到一个请求，若不满足所需请求，应该返回一个OFPET_TABLE_FEATURES_FALLED类型的错误并带有适当的代码。

下面的结构说明表特性请求和回复的部分：

```
/* OFPMP_TABLE_FEATURES类的ofp_multipart_request 部分./
/*回复OFPMP_TABLE_FEATURES 请求的部分. */
struct ofp_table_features {
uint16_t length;       /*长度被填至 64 bits. */
uint8_t table_id;       /*表的标识符. 低编号先询问. */
uint8_t pad[5];        /*排到64-bits. */
char name[OFP_MAX_TABLE_NAME_LEN];
uint64_t metadata_match; /*表能匹配元数据的位数. */
uint64_t metadata_write; /*表能写的元数据位数. */
uint32_t config;         /*OFPTC_*值的位图 */
uint32_t max_entries;    /*流表项被支持的最大数. */
/* 表特征属性列举 */
struct ofp_table_feature_prop_header properties[0]; /*属性举例 */
};
OFP_ASSERT(sizeof(struct ofp_table_features) == 64);
```

数组有一个结构给每个被交换机支持的流表。流表项总是按照数据表通过流表的顺序返回。OFP_MAX_TABLLE_NAME_LEN 是32。

该metadata_match字段表示流表利用ofp_match结构体元数据字段时可匹配的元数据字段的位数，。值0xFFFFFFFFFFFFFFFF表明表可以匹配整个元数据字段。

该metadata_write字段显示表可以使用ofpit_write_metadata指令写元数据字段的位数。值0xFFFFFFFFFFFFFFFF表明表可以写整个元数据字段。

配置字段是表的配置，通过表的配置信息对表进行设置（见 7.3.3）

Max_entries 字段表示能插入表的流表项最大数量。由于现代的硬件的限制，该 max_entries 值应考虑建议值和接近表最大努力的性能。不考虑表的高层抽象，在实践中通过一个单一的流表项所消耗的资源是不固定的。例如，一个流表项可能消耗多个流表项的资源，取决于它的匹配参数（e.g.IPv4 vs. IPv6）。而且，在同个 OPENFLOW 层不同的表可能实际上共享相同的底层物理资源。进一步说，基于 OpenFlow 混合交换机，这些表可能与非 OpenFlow 功能部分共享。结果是交换机实现者应该报告所支持的总的流表项值，并且控制器的作者不应该把这个值设为固定的，物理常数。

properties字段是表特征属性的一个列表，描叙表的不同性能。

7.3.5.5.2 表特征属性

目前定义的表特征属性类型的清单是：

```
/* 表特征属性类型.
* 最低位被清除表示一个常规流表项的一个属性.
* 最低位被设置表示Table-Miss Flow Entry的一个属性.*/
enum ofp_table_feature_prop_type {
OFPTFPT_INSTRUCTIONS = 0,        /* 指令属性. */
OFPTFPT_INSTRUCTIONS_MISS = 1,  /* table-miss的指令. */
OFPTFPT_NEXT_TABLES = 2,          /* Next Table 属性. */
OFPTFPT_NEXT_TABLES_MISS = 3,     /* Next Table for table-miss. */
OFPTFPT_WRITE_ACTIONS = 4,        /* Write Actions 属性. */
```

```
OFPTFPT_WRITE_ACTIONS_MISS = 5,    /* Write Actions for table-miss. */
OFPTFPT_APPLY_ACTIONS = 6,         /* Apply Actions 属性. */
OFPTFPT_APPLY_ACTIONS_MISS = 7,    /* Apply Actions for table-miss. */
OFPTFPT_MATCH = 8,                 /* 匹配属性. */
OFPTFPT_WILDCARDS = 10,            /* Wildcards 属性. */
OFPTFPT_WRITE_SETFIELD = 12,       /* Write Set-Field属性. */
OFPTFPT_WRITE_SETFIELD_MISS = 13, /* Write Set-Field for table-miss. */
OFPTFPT_APPLY_SETFIELD = 14,       /* Apply Set-Field 属性. */
OFPTFPT_APPLY_SETFIELD_MISS = 15, /* Apply Set-Field for table-miss. */
OFPTFPT_EXPERIMENTER = 0xFFFE,     /* Experimenter 属性. */
OFPTFPT_EXPERIMENTER_MISS = 0xFFFF, /* Experimenter for table-miss. */
};
```

_MISS 后缀的属性描述漏表流表项的功能(见 5.4),而其他属性描述常规流表项的功能。如果一个指定属性没有任何功能（例如不支持 Set_Field），一个空列表属性也必须被包含在属性列表里。当漏表流表项的一个属性和常规流表项相应的属性相同,(i.e.两个属性都有相同的功能清单)，这个漏表属性可以从属性列表里清除。

一个属性定义包含属性类型、长度和关联的数据：

```
/* 所有表特征属性常用的标头 */
struct ofp_table_feature_prop_header {
uint16_t type; /* OFPTFPT_*中之一. */
uint16_t length; /*这个属性占字节长度. */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_header) == 4);
```

OFPTFPT_INSTRUCTIONS和OFPTFPT_INSTRUCTIONS_MISS属性使用如下结构和字段:

```
/*指令属性*/
struct ofp_table_feature_prop_instructions {
uint16_t type;       /*OFPTFPT_INSTRUCTIONS和OFPTFPT_INSTRUCTIONS_MISS之一. */
uint16_t length;      /*这个属性的字节长度. */
/* 接下来是:
* - 准确地讲 (length - 4)个字节包含指令的 id, 其次
* - 精确地(length + 7)/8*8 - (length) (between 0 and 7)
* 全零字节的字节数*/
struct ofp_instruction instruction_ids[0]; /* 指令列表*/
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_instructions) == 4);
```

Instruction_ids 是被这个表支持的指令列表（见 5.9)。列表里的元素是可变大小的，用于实验者表述指令的，非实验者指令是 4 字节。

OFPTFPT_NEXT_TABLES 和 OFPTFPT_NEXT_TABLES_MISS 属性使用如下结构和字段：

```
/* Next Tables 属性*/
struct ofp_table_feature_prop_next_tables {
uint16_t type;    /* OFPTFPT_NEXT_TABLES和OFPTFPT_NEXT_TABLES_MISS之一. */
uint16_t length;  /* 这个属性占的字节长度. */
```

```
/* Followed by:
* - Exactly (length - 4) bytes containing the table_ids, then
* - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
* bytes of all-zero bytes */
uint8_t next_table_ids[0]; /*表地址列表. */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_next_tables) == 4);
```

next_table_ids 是表的数组，使用该 OFPIT_GOTO_TABLE 指令可以直接到达本表。（见 5.1）。

OFPTFPT_WRITE_ACTIONS,OFPTFPT_WRITE_ACTIONS_MISS,OFPTFPT_APPLY_ACTIONS 和 OFPTFPT_APPLY_ACTIONS_MISS 属性使用如下结构和字段：

```
/* 行动属性 */
struct ofp_table_feature_prop_actions {
uint16_t type; /*OFPTFPT_WRITE_ACTIONS,
                 OFPTFPT_WRITE_ACTIONS_MISS,
                 OFPTFPT_APPLY_ACTIONS,
                  FPTFPT_APPLY_ACTIONS_MISS.中之一 */
uint16_t length; /* 该属性字节长度. */
/* Followed by:
* - Exactly (length - 4) bytes containing the action_ids, then
* - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
* bytes of all-zero bytes */
struct ofp_action_header action_ids[0]; /* List of actions */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_actions) == 4);
```

Action_ids是特征的行动列表（见5.12）。列表里的元素是可变大小的，用于实验者表述指令，非实验者指令是4字节。OFPTFPT_WRITE_ACTIONS和OFPTFPT_WRITE_ACTIONS_MISS属性描述使用OFPIT_WRITE_ACTIONS指令的表所支持的行动，而OFPTFPT_APPLY_ACTIONS和OFPTFPT_APPLY_ACTIONS_MISS属性描述被正在使用OFPIT_APPLY_ACTIONS指令的表支持的行动。

OFPTFPT_MATCH,OFPTFPT_WILDCARDS,OFPTFPT_WRITE_SETFIELD,OFPTFPT_WRITE_SETFIELD_MISS, OFPTFPT_APPLY_SETFIELD 和 OFPTFPT_APPLY_SETFIELD_MISS属性使用如下结构和字段：

```
/* Match, Wildcard 或 Set-Field 属性 */
struct ofp_table_feature_prop_oxm {
uint16_t type;    /*OFPTFPT_MATCH,OFPTFPT_WILDCARDS,
                  OFPTFPT_WRITE_SETFIELD,
                  OFPTFPT_WRITE_SETFIELD_MISS,
                   OFPTFPT_APPLY_SETFIELD,
                    OFPTFPT_APPLY_SETFIELD_MISS.中之一 */
uint16_t length; /* Length in bytes of this property. */
/* Followed by:
* - Exactly (length - 4) bytes containing the oxm_ids, then
```

```
* - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
* bytes of all-zero bytes */
uint32_t oxm_ids[0]; /*OXM头数组*/
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_oxm) == 4);
```

oxm_ids 是针对特征的 OXM 类型列表(见 7.2.3.2)。列表里是 32 位 OXM 头或者实验者使用的 64 位 OXM 头。

OFPTFPT_MATCH属性表示特别的表所支持匹配的字段（见7.2.3.7)。例如：如果表能够匹配入端口，一个OXM_OF_IN_PORT类型的OXM头应该被包括在列表里。如果在OXM头中HASMASK位被设置，那么交换机必须支持为所给的掩码类型。OFPTFPT_WILDCARDS属性表示特别的表所支持通配（省略）的字段。例如，一个直接查找的哈希表将把列表变空，而一个TCAM表或者有序查询表将设置成和OFPTFPT_MATCH属性相同的值。

OFPTFPT_WRITE_SETFIELD 和 OFPTFPT_WRITE_SETFIELD_MISS 属性利用 OFPIT_WRITE_ACTIONS 指令，来描述表所支持的 Set_Field 行动类型。而 OFPTFPT_APPLY_SETFIELD 和 OFPTFPT_APPLY_SETFIELD_MISS 属性利用 OFPIT_APPLY_ACTIONS 指令，来描述表所支持的Set-Field 行动类型。

控制器max_entries字段异常，在ofp_table_features中的所有字段可能被要求改变。此字段是只读的，并且由交换机返回。

OFPTFPT_APPLY_ACTIONS， OFPTFPT_APPLY_ACTIONS_MISS, OFPTFPT_APPLY_SETFIELD, 和OFPTFPT_APPLY_SETFIELD_MISS属性包含表能运用的行动和字段。对于每一个列表，如果一个元素is present，意味着表至少能使该元素隔离一次。目前没有办法表示哪些元素能一起运用，按哪种顺序，一个元素能在一个单一的流表项里被运用多少次。

OFPTFPT_EXPERIMENTER 和OFPTFPT_EXPERIMENTER_MISS属性使用如下结构和字段：

```
/*实验者的表的特征属性*/
struct ofp_table_feature_prop_experimenter {
uint16_t type;      /*OFPTFPT_EXPERIMENTER和OFPTFPT_EXPERIMENTER_MISS中之一. */
uint16_t length;    /* 这个属性字节长度. */
uint32_t  experimenter;  /*  Experimenter  ID  在  ofp_experimenter_header  结
                          构中有相同形式. */
uint32_t exp_type; /* 实验者定义. */
/* Followed by:
* - Exactly (length - 12) bytes containing the experimenter data, then
* - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
* bytes of all-zero bytes */
uint32_t experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_experimenter) == 12);
```

实验者字段是Experimenter ID,在ofp_experimenter结构体中有相同形式（见7.5.4）.

7.3.5.6 端口统计

关于端口统计的信息使用 OFPMP_PORT_STATS 复合请求类型进行请求：

```
/* Body for ofp_multipart_request of type OFPMP_PORT. */
```

```
struct ofp_port_stats_request {
uint32_t port_no;      /* OFPMP_PORT消息必须请求统计，要么给一个单一的端口(在
                        port_no中指定的) 或者所有端口 (port_no == OFPP_ANY). */
uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_port_stats_request) == 8);
```

port_no字段有选择的过滤统计请求到给定的端口。若访问所有端口的统计,port_no必须设置成 OFPP_ANY.

回复部分由以下数组组成:

```
/* OFPMP_PORT请求的回复部分.如果一个计数器不被支持,设置该字段给所有的（ones）. */
struct ofp_port_stats {
uint32_t port_no;
uint8_t pad[4]; /*排到 64-bits. */
uint64_t rx_packets; /* 收到包的数量. */
uint64_t tx_packets; /*已传送包的数量 */
uint64_t rx_bytes; /*收到的字节数. */
uint64_t tx_bytes; /*传送的字节数. */
uint64_t rx_dropped; /*被 RX丢弃的包数. */
uint64_t tx_dropped; /*被TX丢弃的包数. */
uint64_t rx_errors; /* 接受到错误的数量. 这是一个超集（super-set）更具体的接收错误，应该大于或等于所有rx_ * _err值的总和。*/
uint64_t tx_errors; /* 传送错误的数量.这是一个超集更具体的传输错误，应该大于或等于所有tx_ * _err值的总和（没有当前定义的。）*/
uint64_t rx_frame_err; /* 帧调整的错误数量. */
uint64_t rx_over_err; /* 在RX溢出的数据包数. */
uint64_t rx_crc_err; /* CRC错误数. */
uint64_t collisions; /* 冲突数量. */
uint32_t duration_sec; /* 端口已经生存了的秒数. */
uint32_t duration_nsec; /*端口已生存的时间超过duration_sec的纳秒数. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats) == 112);
```

duration_sec 和 duration_nsec 字段表示端口已配置成 OpenFlow 通道已过去的时间。持续的总的纳秒数可以被算成 duration_sec*10^9+duration_nsec. 实现时要求提供第二精度，在能获得的条件下鼓励提供更高的精度。

7.3.5.7 端口描述

端口描述请求 OFPMP_PORT_DESCRIPTION能使控制器获得支持OpenFlow系统里所有的端口描述。请求部分是空的。回复部分由如下数组构成:

```
/* 一个端口的描述 */
struct ofp_port {
uint32_t port_no;
uint8_t pad[4];
uint8_t hw_addr[OFP_ETH_ALEN];
```

```
uint8_t pad2[2];                    /*排到64 bits. */
char name[OFP_MAX_PORT_NAME_LEN];  /* Null-terminated */
uint32_t config;                    /* OFPPC_* flags的位图. */
uint32_t state;                     /* OFPPS_* flags的位图. */
/*  描述特性的OFPPF_*的位图. 如果不支持或不可用的所有位清零*/
uint32_t curr;                    /* 当前特征 */
uint32_t advertised;              /* 端口公布的特性. */
uint32_t supported;               /* 端口支持的特性. */
uint32_t peer;                    /*对端公布的特性 . */
uint32_t curr_speed;               /* 当前端口的比特率，kbps. */
uint32_t max_speed;               /* 最大端口比特率kbps */
};
OFP_ASSERT(sizeof(struct ofp_port) == 64);
```

这个结构被描述在 7.2.1.

7.3.5.8 队列统计

OFPMP_QUEUE 复合请求消息提供队列统计给一个或多个端口和一个或多个队列。请求部分包含一个 port_no 字段为请求的统计标识出 OpenFlow 端口，或者 OFPP_ANY 去查阅所有的端口。queue_id 字段识别优先队列之一。或者 OFPQ_ALL 指向在指定端口配置的所有队列。OFPQ_ALL 是 0xffffffff。

```
struct ofp_queue_stats_request {
uint32_t port_no;       /* 如果是 OFPP_ANY表示所有端口. */
uint32_t queue_id;      /* 如果是OFPQ_ALL表示所有队列. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats_request) == 8);
```

回复部分由下列结构组成:

```
struct ofp_queue_stats {
uint32_t port_no;
uint32_t queue_id; /* 队列i.d */
uint64_t tx_bytes; /* 传送的字节数. */
uint64_t tx_packets; /* 传送的包数. */
uint64_t tx_errors; /* 由于溢出丢弃的数据包数量. */
uint32_t duration_sec; /* 队列已经生成的秒数. */
uint32_t duration_nsec; /* 队列已经生成超出duration_sec的纳秒数. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats) == 40);
```

Duration_sec 和 duration_nsec 字段表示队列已经安装在交换机里过去了的时间。持续的总时间可以用 duration_sec*10^9+duration_nsec.实现时要求提供第二精度，在能获得的条件下鼓励提供更高的精度。

7.3.5.9 组统计

OFPMP_GROUP 复合请求消息为一个或多个组提供统计。请求部分由 group_id 字段组成。它也能被设置为 OFPG_ALL 表示交换机里所有的组。

```
/*OFPMP_GROUP请求部分. */
struct ofp_group_stats_request {
uint32_t group_id; /* 如果是 OFPG_ALL表示所有的表. */
uint8_t pad[4]; /* 拍到 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_group_stats_request) == 8);
```

回复部分由如下结构组成：

```
/* 回复给 OFPMP_GROUP请求的部分. */
struct ofp_group_stats {
uint16_t length; /*这个流表项的长度. */
uint8_t pad[2]; /* 排到64 bits. */
uint32_t group_id; /* 组标识符. */
uint32_t ref_count; /* 直接指向该组的流或组的数量 */
uint8_t pad2[4]; /*排到64 bits. */
uint64_t packet_count; /*组处理过的包的数量. */
uint64_t byte_count; /* 组处理过的字节数量. */
uint32_t duration_sec; /* 组已经生存的秒数. */
uint32_t duration_nsec; /* 组已经生存的超过duration_sec的纳秒数. */
struct ofp_bucket_counter bucket_stats[0]; /*每个存储段设置一个计数器. */
};
OFP_ASSERT(sizeof(struct ofp_group_stats) == 40);
```

字段包括创建组的 group_mod 提供的，加上 ref_count 表示计算与组相关的流的数量，packet_count，和 byte_count 计算所有组处理的数据包。

Duration_sec 和 duration_nsec 字段表示组安装在交换机里已经过去的时间。总的持续的纳秒数可以用 duration_sec*10^9+duration_nsec.实现时要求提供第二精度，在能获得的条件下鼓励提供更高的精度。

Bucket_stats 字段由一组 ofp_bucket_counter 结构体组成：

```
/* 用来做组统计回复. */
struct ofp_bucket_counter {
uint64_t packet_count; /* 存储段处理的包数. */
uint64_t byte_count; /* 存储段处理的字节数. */
};
OFP_ASSERT(sizeof(struct ofp_bucket_counter) == 16);
```

7.3.5.10 组描述

OFPMP_GROUP_DESC 复合请求消息提供一个方式将交换机上的组设置列出来,以及它们相应的存储段行动。请求部分是空的，而回复部分是如下结构的数组：

```
/*回复给OFPMP_GROUP_DESC 请求的部分. */
struct ofp_group_desc {
uint16_t length;        /*流表项的长度. */
uint8_t type;           /*OFPGT_*中之一. */
uint8_t pad;            /* 填充至64 bits. */
uint32_t group_id;      /* 组标识符. */
struct ofp_bucket buckets[0]; /*存储段列表 -- 0 或更多. */
};
OFP_ASSERT(sizeof(struct ofp_group_desc) == 8);
```

描述组的字段和那些 ofp_group_mod 结构体使用的一样(见 7.3.4.2)。

7.3.5.11 组特征

OFPMP_GROUP_FEATURES 复合请求消息提供一个方式列出交换机上组的功能。请求部分是空的，而回复部分是如下结构：

```
/*回复给OFPMP_GROUP_FEATURES 请求的部分. 组特征. */
struct ofp_group_features {
uint32_t types;          /*  OFPGT_* 值支持的位图. */
uint32_t capabilities;  /* OFPGFC_* 功能支持的位图. */
uint32_t max_groups[4]; /* 每个类型的组的最大数量. */
uint32_t actions[4];    /* 支持的OFPAT_* 的位图. */
};
OFP_ASSERT(sizeof(struct ofp_group_features) == 40);
```

max_groups字段是每个类型的组的最大数量。行动是每个组类型支持的行动。功能使用如下标志的一个组合：

```
/* 组配置标志*/
enum ofp_group_capabilities {
OFPGFC_SELECT_WEIGHT = 1 << 0,    /* 选择组支持的重量 */
OFPGFC_SELECT_LIVENESS = 1 << 1, /*选择组支持的活跃度*/
OFPGFC_CHAINING = 1 << 2,         /* 支持链接组 */
OFPGFC_CHAINING_CHECKS = 1 << 3, /* 为循环检查和删除链接*/
```

7.3.5.12 计量器统计

OFPMT_METER 统计请求消息给一个或多个计量器提供统计。请求部分由一个 meter_id 字段组成，它能设置成 OFPM_ALL 表示交换机上所有计量器。

```
/* OFPMP_METER 和 OFPMP_METER_CONFIG 请求部分 */
struct ofp_meter_multipart_request {
uint32_t meter_id;      /* 计量器实例, 或者是 OFPM_ALL. */
uint8_t pad[4];          /* 排列到 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_meter_multipart_request) == 8);
```

回复部分是如下结构：

```
/*OFPMP_METER 请求的回复部分. 计量器统计. */
struct ofp_meter_stats {
uint32_t meter_id;        /*计量器例子 */
uint16_t len;             /* 统计的字节长度. */
uint8_t pad[6];
uint32_t flow_count;      /* 跳至计量器的流数量. */
uint64_t packet_in_count; /* 输入数据包的数量. */
uint64_t byte_in_count;   /* 输入的字节数. */
uint32_t duration_sec;    /* 计量器已经生成了的秒数. */
uint32_t duration_nsec;   /* 计量器已经生存超出 duration_sec 的纳秒数. */
struct ofp_meter_band_stats band_stats[0];
                          /*band_stats长度是字段长度里推出来的 */
};
OFP_ASSERT(sizeof(struct ofp_meter_stats) == 40);
```

packet_in_count 和 byte_in_count 对计量器处理过的所有数据包进行计数。duration_sec和duration_nsec 字段表示计量器安装在交换机上过去了的时间。总的持续时间可以用 duration_sec*10^9+duration_nsec。实现时要求提供第二精度，在能获得的条件下鼓励提供更高的精度。

band_stats字段由ofp_meter_band_stats 结构组成:

```
/* 给每个计量带宽的统计 */
struct ofp_meter_band_stats {
uint64_t packet_band_count; /*带内的数据包数量. */
uint64_t byte_band_count; /* 带内的字节数量. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_stats) == 16);
```

packet_band_count和 byte_band_count 对带宽处理的数据包进行计数。带宽统计的顺序必须和OFPMT_METER_CONFIG统计回复里的一样。

7.3.5.13 计量器配置统计

OFPMT_METER_CONFIG 统计请求消息给一个或多个计量器提供配置。请求部分由一个meter_id字段组成,它可以被设置成 OFPM_ALL表示交换机上所有的计量器。

```
/*OFPMP_METER和OFPMP_METER_CONFIG 请求部分. */
struct ofp_meter_multipart_request {
uint32_t meter_id;      /* 计量器实例,或者OFPM_ALL. */
uint8_t pad[4];         /* 对齐到64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_meter_multipart_request) == 8);
```

回复部分由一个如下结构组成：

```
/*OFPMP_METER_CONFIG请求的回复部分.计量器配置 */
struct ofp_meter_config {
uint16_t length; /*流表项的长度. */
uint16_t flags; /* All OFPMC_* 表示申请. */
uint32_t meter_id; /*计量器实例. */
struct ofp_meter_band_header bands[0]; /* 带宽的长度从字段长度里推导出 */
};
OFP_ASSERT(sizeof(struct ofp_meter_config) == 8);
```

该字段和用来配置计量器的字段相同(见 7.3.4.4)

7.3.5.14 仪表功能统计

OFPMT_METER_FEATRUES 统计请求消息提供了计量子系统功能集合。

请求部分是空的，和回复部分由以下结构组成：

```
/*回复 ofpmp_meter_features 请求。计量功能*/
struct ofp_meter_features {
uint32_t max_meter;       /*计量器的最大数值. */
uint32_t band_types;     /*支持位图ofpmbt_ *的值*/
uint32_t capabilities;  /*ofp_meter_flags 的位图*/
uint8_t max_bands;       /*每个计量的的最大频带*/
uint8_t max_color;       /*最大的彩色值*/
uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_meter_features) == 16);
```

7.3.5.15 实验者复合

实验者特定复合消息通过 OFPMP_EXPERIMENTER 复合类型进行请求,请求的第一个字节和回复部分结构如下：

```
/*ofp_multipart_request 部分/回复OFPMP_EXPERIMENTER. */
struct ofp_experimenter_multipart_header {
```

```
uint32_t experimenter;
                    /*实验者标识符，与 ofp_experimenter_header 中格式相同 */
uint32_t exp_type; /*实验者定义 */
/*实验者自定义附加数据 */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_multipart_header) == 8);
```

剩余的请求和回复部分是由实验者定义

experimenter 字段是实验者的 ID，与 OFPMP_EXPERIMENTER 中的格式相同。

7.3.6 队列配置消息

队列配置超出了 OpenFlow 协议范围，可通过一个命令行工具，或通过一个外部的专用配置协议。

控制器使用下列结构来查询交换机端口上已配置队列：

```
/*查询端口队列配置*/
struct ofp_queue_get_config_request {
struct ofp_header header;
uint32_t port;  /*查询的端口；应该指向一个有效的物理端口（或 OFPP_ANY 要求所有配
                    置队列）*/
uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_queue_get_config_request) == 16);
```

交换机使用 OFP_QUEUE_GET_CONFIG_REPLY 命令回复，其中包含配置队列的列表。

```
/*给定端口的队列配置*/
struct ofp_queue_get_config_reply {
struct ofp_header header;
uint32_t port;
uint8_t pad[4];
struct ofp_packet_queue queues[0]; /*已配置的队列列表*/
};
OFP_ASSERT(sizeof(struct ofp_queue_get_config_reply) == 16);
```

7.3.7 分组报文

当控制器通过数据通路发送一个数据包，就使用 OFPT_PACKET_OUT 消息：

```
/*发送数据包（控制器->通路）*/
struct ofp_packet_out {
struct ofp_header header;
uint32_t buffer_id;   /*指定的标识符通过数据通路（OFP_NO_BUFFER 如果没有）*/
uint32_t in_port;     /*分组的输入端口或 OFPP_CONTROLLER*/
uint16_t actions_len; /*以字节数组的大小的作用 */
uint8_t pad[6];
struct ofp_action_header actions[0]; /*行动列表-0 或更多*/
```

```
/*可变大小的行动清单可在后面跟着分组数据。
*如果 buffer_id = -1 目前的数据是有意义的。
/ *uint8_t 数据[ 0 ]; */   /*分组数据   长度是从长度字段的头部计算的 */
};
OFP_ASSERT(sizeof(struct ofp_packet_out) == 24
```

buffer_id 与 ofp_packet_in 消息中给出的是相同的。如果 buffer_id 是 OFP_NO_BUFFER，则数据包位于数据阵列中，和封装在消息里的数据包由行动的消息进行处理。OFP_NO_BUFFER 等于 0xffffffff。如果 buffer_id 是有效的，相关的端口被行动消息从缓冲区删除。

in_port 字段表示入端口，端口必须与 OpenFlow 处理数据包相关。它必须设置为一个有效的标准交换机端口（见 4.2）或 OFPP_CONTROLLER.例如，当使用 OFPP_TABLE, OFPP_IN_PORT, OFPP_ALL 和组处理数据包时就使用这个字段。

Action 是一个行动列表定义交换机如何处理数据包的字段。可包括包修改，组处理和一个输出端口。一个 OFPT_PACKET_OUT 行动清单消息也可以指定 OFPP_TABLE 保留端口作为输出行动来处理 OpenFlow 流水线中的数据包，从第一个流表开始（见 4.5）。如果 OFPP_TABLE 被指定，in_port 作为查找时流表的入端口。

在某些情况下，发送到 OFPP_TABLE 的数据包，经过流表项行动或漏表后，可能转发给控制器。对这样控制器到交换机回路的检测和执行行动不在本规范范围内。一般来说，OpenFlow 的消息不保证是顺序处理。

因此，如果一个使用 OFPP_TABLE 的 OFPP_TABLE 消息依赖于最近发送到交换机（与 OFPT_FLOW_MOD 消息）的流，一个 OFPT_BARRIER_REQUEST 消息需要在 OFPT_PACKET_OUT 消息之前，确保在执行 OFPP_TABLE 之前把流表项交给流表。

7.3.8 屏障消息

当控制器要想确保消息已送到或想接收到通知来完成操作，它可以使用一个 OFPT_BARRIER_REQUEST 消息。此消息没有内容。收到后，在执行基于 Barrier Request 任何消息之前，交换机必须处理完所有先前收到的信息，包括发送相应的回复或错误信息。当处理完成，交换机必须发送一个消息 OFPT_BARRIER_REPLY 携带 XID 最初的请求。

7.3.9 任务请求消息

当控制器要转变角色，它使用 OFPT_ROLE_REQUEST 消息，结构如下：

```
/*请求和回复消息的角色*/
struct ofp_role_request {
struct ofp_header header; /* ofpt_role_request / ofpt_role_reply 型 */
uint32_t role;            /* ofpcr_role_ *之一 */
uint8_t pad[4];           /*对齐到 64 位*/
uint64_t generation_id;   /*主设备选择产生的标识符
 */
};
OFP_ASSERT(sizeof(struct ofp_role_request) == 24);
```

role 字段就是控制器要担任的新角色，并且可以有以下值：

```
/*控制器的角色*/
enum ofp_controller_role {
OFPCR_ROLE_NOCHANGE = 0, /*不改变目前的角色*/
OFPCR_ROLE_EQUAL = 1,     /*默认角色，完全访问。 */
OFPCR_ROLE_MASTER = 2,     /*完全访问，最多一个主设备。*/
OFPCR_ROLE_SLAVE = 3,      /*只读访问*/
};
```

如果消息中的角色值是 OFPCR_ROLE_MASTER 或 OFPCR_ROLE_SLAVE，交换机必须验证 generation_id 检查过期消息（见 6.3.4）。如果验证失败，交换机必须抛弃角色要求和返回 OFPET_ROLE_REQUEST_FAILED 类型和 OFPRRFC_STALE 代码的错误消息。

如果角色值是 OFPCR_ROLE_MASTER,所有其他角色是 OFPCR_ROLE_MASTER 的控制器都要改变为 OFPCR_ROLE_SLAVE（见 6.3.4）。如果角色值是 OFPCR_ROLE_NOCHANGE,控制器当前的角色就不改变；这使控制器查询其目前的角色而无需改变它。

收到一个 OFPT_ROLE_REQUEST 的消息后，如果没有错误，交换机必须返回一个 OFPT_ROLE_REPLY 消息。这则消息的结构与 OFPT_ROLE_REQUEST 消息是完全一样的，role 字段就是当前控制的角色。generation_id 设置为当前 generation_id（上一次成功角色请求的 generation_id，角色是 OFPCR_ROLE_MASTER 或 OFPCR_ROLE_SLAVE），如果目前的 generation_id 控制器没有设置，回复中的 generation_id 必须设置为最大字段(无符号值相当于- 1)。

7.3.10 设置异步配置信息

在一个给定的 OpenFlow 通道上，分别使用 OFPT_SET_ASYNC 和 OFPT_GET_ASYNC_REQUEST 消息，控制器能设置和查询它想要接收的异步消息（除错误消息）。

交换机使用 OFPT_GET_ASYNC_REPLY 消息响应一个 OFPT_GET_ASYNC_REQUEST 消息；它不回复一个设置配置的请求。

基于 OpenFlow 头部的 OFPT_GET_ASYNC_REQUEST 消息没有 body 。OFPT_SET_ASYNC 和 OFPT_GET_ASYNC_REPLY 消息的格式如下：

```
/*异步消息的配置。 */
struct ofp_async_config {
struct ofp_header header;        /* OFPT_GET_ASYNC_REPLY or OFPT_SET_ASYNC. */
uint32_t packet_in_mask[2];      /* ofpr_ *值的位掩码*/
uint32_t port_status_mask[2];    /* ofppr_ *值的位掩码。 */
uint32_t flow_removed_mask[2];    /* ofprr_ *值的位掩码。. */
};
OFP_ASSERT(sizeof(struct ofp_async_config) == 32);
```

结构 struct ofp_async_config 包含三个二维数组。每个数组控制是否接收控制器一个特定的枚举 ofp_type 异步消息。当控制器是 OFPCR_ROLE_EQUAL 或 OFPCR_ROLE_MASTER 角色时，每个数组中的元素 0 表示感兴趣的消息。当控制器是 OFPCR_ROLE_SLAVE 角色时，元素为 1。每个数组元素的位掩码中的 0 禁止接收与比特位

置有关的 reason 代码消息。而 1 则可以接收。例如，在 port_status_mask [1]中比特值为 2^2=4，当控制器的角色是 OFPCR_ROLE_SLAVE 时，决定控制器是否接收理由是 OFPPR_MODIFY（值为 2）的 OFPT_PORT_STATUS 消息。

OFPT_SET_ASYNC 设置控制器是否应该接受一个交换机产生的异步消息。其他的 OpenFlow 功能控制是否产生一个给定的消息；例如，在 OFPFF_SEND_FLOW_REM 标志控制交换机在流表项删除时是否产生 OFPT_FLOW_REMOVED 消息。。

一个交换机的配置，例如使用 OpenFlow 配置协议，一旦 OpenFlow 建立连接，就可以设置异步消息的初始配置。交换机没有配置的话，初始配置应：

在“master”或“equal”的角色，使能所有的 OFPT_PACKET_IN 消息，除了那些带有 OFPR_INVALID_TTL 原因的，使能所有的 OFPT_PORT_STATUS 和 OFPT_FLOW_REMOVED 消息。

在“slave”角色，使能所有 OFPT_PORT_STATUS 消息，并禁用所有 OFPT_PACKET_IN 和 OFPT_FLOW_REMOVED 的消息。

利用 OFPT_SET_ASYNC 的配置设置是针对一个特定的 OpenFlow 的通道，它不会影响任何其他 OpenFlow 通道，无论是目前建立的或是以后建立的。
配置设置 OFPT_SET_ASYNC 不过滤器或影响的错误消息。

7.4 异步消息

7.4.1 包输入消息

当数据包通过数据通路进行接收和发送给控制器，都使用 OFPT_PACKET_IN 消息：

```
/*端口接收到的数据包（数据通路->控制器） */
struct ofp_packet_in {
struct ofp_header header;
uint32_t buffer_id; /*分配的数据通路 ID。 */
uint16_t total_len; /*帧全长 */
uint8_t reason; /*包被发送的原因（一个ofpr_ *）*/
uint8_t table_id; /*被查询的流表标识符*/
uint64_t cookie; /*查询的流表项 cookie */
struct ofp_match match; /*分组元数据。可变尺寸。 */
/*可变大小和填充匹配总是跟着：
* 2 个全零填充字节，然后是以太网帧，其长度是根据 header.length 推断。
*以太网帧前填充的字节，确保以太网报头后的 IP 头部（如果有的话）是 32 位
*/
//uint8_t pad[2]; /*  对齐 64 位+ 16 位 */
//uint8_t data[0]; /*以太网帧 */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 32);
```

buffer_id 是个不透明的值，数据通路使用它识别一个缓冲数据包。当一个数据包进行缓存，消息的某些字节将包含在消息的数据部分。如果数据包是因为“发送给控制器”的行动而发送，那么流建立请求的 ofp_action_output 会发送 max_len 字节。如果数据包其他原因发送，如无效的 TTL,然后 OFPT_SET_CONFIG 消息至少发送 miss_send_len 字节。默认 miss_send_len 是 128 字节。如果数据包没有缓冲-要么是因为没有可用的缓冲区，或因明确

表示通过 OFPCML_NO_BUFFER 请求-整个数据包都包含在数据部分，而且 buffer_id 就是 OFP_NO_BUFFER。

实现缓冲的交换机希望通过文档公开，可用的缓冲数量和缓冲区可重复使用前的时间长度。交换机必须妥善处理这样的事件，当一个 packet_in 消息缓冲，而控制器没有反应的情况。交换机应防止缓冲区被重复使用，直到它已由控制器处理，或经过一段时间（文档中说明）。

data 字段包含的数据包本身，或缓冲包的一小部分。包头部反映了前面处理对数据包的任何变化。

reason 字段可以是这些值的某个：

```
/*为什么这个包被发送到控制器？ */
enum ofp_packet_in_reason {
OFPR_NO_MATCH = 0, /*  没有匹配的流（漏表流表项） */
OFPR_ACTION = 1,    /*输出给控制器的行动。*/
OFPR_INVALID_TTL = 2, /*包有无效的 TTL */
};
```

OFPR_INVALID_TTL 表明携带无效的 IP TTL 或 MPLS TTL 被 OpenFlow 流水线拒绝并传给控制器。不需要对每一个数据包都进行无效的 TTL 检查，但它至少在每次 OFPAT_DEC_MPLS_TTL 或 OFPAT_DEC_NW_TTL 行动应用到一个包时进行检查。

cookie 字段包含导致包被发送到控制器的流表项 cookie。如果一个 cookie 不能与一个特定的流有关的，这字段必须设置为-1（0xffffffffffffffff）。例如，如果包输入是在组存储段里或从行动集产生的。

当有事件触发包输入消息发生，同时包含一组OXM TLVs，match字段就反映分组的头部和上下午。这个上下文包括与数据包以前处理发生的任何改变，包括已经执行的行动，行动集的任何变化，除了无任何变化的。这OXM TLVs必须包括上下文字段，也就是，其值不能从数据包确定的字段。标准的上下文字段是OFPXMT_OFB_IN_PORT, OFPXMT_OFB_IN_PHY_PORT,OFPXMT_OFB_METADATA和 OFPXMT_OFB_TUNNEL_ID.所有位都是零的字段应忽略。OXM TLVs可包括先前从分组中提取的数据包头部字段，也包括那些在处理过程中的任何修改。

当在物理端口直接收到一个包时，而且未被逻辑端口处理，OFPXMT_OFB_IN_PORT 和 OFPXMT_OFB_IN_PHY_PORT 有相同的值，OpenFlow 物理端口的 port no。OFPXMT_OFB_IN_PHY_PORT 如果与 OFPXMT_OFB_IN_PORT 有相同的值则应忽略。

当一个包是通过一个物理端口的逻辑端口上接收的，OFPXMT_OFB_IN_PORT 是逻辑端口的端口的 port no，OFPXMT_OFB_IN_PHY_PORT 是物理端口的 port no。例如，考虑一个包在隧道接口上接收，接口是由两个物理端口的链路汇聚组（LAG）进行定义的。如果隧道接口是绑定到 OpenFlow 的逻辑端口，那么 OFPXMT_OFB_IN_PORT 是隧道 port no，OFPXMT_OFB_IN_PHY_PORT 是已配置隧道的 LAG 物理端口 port no 成员。

OFPXMT_OFB_IN_PORT TLV 引用的端口必须是用来匹配流表项的端口（见 5.3），也必须对 OpenFlow 处理有用（即 OpenFlow 可以转发数据包到该端口，依靠端口标志）。OFPXMT_OFB_IN_PHY_PORT 不需要有效匹配或进行 OpenFlow 处理。

7.4.2 流删除消息

如果控制器请求通知流表项超时或从表中删除（见 5.5），数据通路使用 OFPT_FLOW_REMOVED 消息：

```
/*流删除（数据通路->控制器） */
struct ofp_flow_removed {
struct ofp_header header;
uint64_t cookie;     /*不透明的控制器发出标识符。 */
uint16_t priority;    /*流表项优先级。*/
uint8_t reason;       /*一个 ofprr_ *
uint8_t table_id;     /*流表的标识符*/
uint32_t duration_sec; /* Time flow was alive in seconds. */
uint32_t duration_nsec;
                /* Time flow was alive in nanoseconds beyond duration_sec. */
uint16_t idle_timeout; /*从原来的flow mod  空闲超时*/
uint16_t hard_timeout; /*从原来的 flow mod 硬超时。*/
uint64_t packet_count;
uint64_t byte_count;
struct ofp_match match; /*字段描述。可变尺寸。*/
};
OFP_ASSERT(sizeof(struct ofp_flow_removed) == 56);
```

Match，cookie，和 priority 字段与那些在 flow mod 请求里使用的相同。

Reason 字段是下列之一：

```
/*为什么流中删除？*/
enum ofp_flow_removed_reason {
OFPRR_IDLE_TIMEOUT = 0, /*流idle_timeout空闲时间超过*/
OFPRR_HARD_TIMEOUT = 1, /*时间超过 hard_timeout。 */
OFPRR_DELETE = 2,       /*由 DELETE flow mod 消除。 */
OFPRR_GROUP_DELETE = 3, /*删除的组。*/
};
```

duration_sec 和 duration_nsec 字段在第 7.3.5.2 描述。

idle_timeout 和 hard_timeout 字段是直接从创建流表项的 flow mod 复制。

利用上述三个字段，你可以找到流表项活跃的时间，以及流表项接收信息的时间。

packet_count 和 byte_count 分别表示与流表项相关联的包和字节的数值。这些计数器应像其他统计计数器（见 7.3.5）；如果没有设置，它们都是无符号的，并应设置为字段最大值。

7.4.3 端口的状态信息

当端口被添加，修改，从数据通路中删除，利用 OFPT_PORT_STATUS 消息告知控制器：

```
/*数据通路的一个物理端口改变 */
struct ofp_port_status {
struct ofp_header header;
uint8_t reason;          /*一个 ofppr_*/
uint8_t pad[7];          /*对齐到64位。 */
struct ofp_port desc;
```

```
};
OFP_ASSERT(sizeof(struct ofp_port_status) == 80);
```

reason可以是下列值之一：

```
/*是什么改变了关于物理端口*/
enum ofp_port_reason {
OFPPR_ADD = 0, /*端口添加。 */
OFPPR_DELETE = 1, /*端口被删除*/
OFPPR_MODIFY = 2, /*某些端口的属性已经改变了。*/
};
```

7.4.4 错误消息

交换机需要把问题通知给控制器。这是通过 OFPT_ERROR_MSG 消息完成：

```
/* ofpt_error：错误消息（数据通路->控制器）。*/
struct ofp_error_msg {
struct ofp_header header;
uint16_t type;
uint16_t code;
uint8_t data[0]; /*可变长度的数据。用类型和代码区分。无填充*/
};
OFP_ASSERT(sizeof(struct ofp_error_msg) == 12);
```

Type 的值表示错误的高层次的类型。代码值是在类型基础上解析。数据是可变长度的并且基于类型和代码解析。除非另有规定 data 字段包含至少 64 字节的失败请求，导致错误消息产生，如果失败请求小于 64 字节，它应该是完整的请求而无填充。

如果错误消息是来自控制器的响应，例如，OFPET_BAD_REQUEST,

OFPET_BAD_ACTION, OFPET_BAD_INSTRUCTION, OFPET_BAD_MATCH, 或 OFPET_FLOW_MOD_FAILED,那么头部的 xid 字段必须匹配那个引起错误的消息。

以_EPERM 结束的错误代码表示一个允许产生的错误，例如，控制器和交换机之间插入一个 OpenFlow 管理程序。

目前定义的错误类型是：

```
/*ofp_error_message 里的“type”值。这些值是不变的：他们不会在未来版本的协议改变（虽然新值可以添加）。*/
enum ofp_error_type {
OFPET_HELLO_FAILED = 0, /* hello 协议失败*/
OFPET_BAD_REQUEST = 1, /*请求不被理解 */
OFPET_BAD_ACTION = 2, /*在行动中描述的错误。*/
OFPET_BAD_INSTRUCTION = 3, /*指令表中错误。*/
OFPET_BAD_MATCH = 4, /*错误匹配。 */
OFPET_FLOW_MOD_FAILED = 5, /*修改流表项问题。*/
OFPET_GROUP_MOD_FAILED = 6, /*修改组表项问题。*/
OFPET_PORT_MOD_FAILED = 7, /*端口属性请求失败。 */
OFPET_TABLE_MOD_FAILED = 8, /*  流表请求失败*/
OFPET_QUEUE_OP_FAILED = 9, /*队列操作失败*/
```

```
OFPET_SWITCH_CONFIG_FAILED = 10, /*交换机配置请求失败*/
OFPET_ROLE_REQUEST_FAILED = 11, /*控制器任务请求失败*/
OFPET_METER_MOD_FAILED = 12, /*计量器错误*/
OFPET_TABLE_FEATURES_FAILED = 13, /*设置流表功能失败*/
OFPET_EXPERIMENTER = 0xffff /*实验者的错误消息*/
};
```

为 OFPET_HELLO_FAILED 错误 type，下面是目前定义的 codes：

```
/*为OFPET_HELLO_FAILED ofp_error_msg “code”的值。Data包含ASCII文本字符串，可以提供详细的故障。*/
enum ofp_hello_failed_code {
OFPHFC_INCOMPATIBLE = 0, /*不兼容的版本  */
OFPHFC_EPERM = 1,        /*  允许失败    */
};
```

Data 字段包含 ASCII 文本字符串，加上为什么发生错误的细节。

为 OFPET_BAD_REQUEST 错误 type，下面是目前定义的 code：

```
/*为 OFPET_BAD_REQUESTofp_error_msg “code”的值。Data 至少包含请求失败的第一个 64 字节。*/
enum ofp_bad_request_code {
OFPBRC_BAD_VERSION = 0, /* ofp_header.版本不支持。 */
OFPBRC_BAD_TYPE = 1,        /* ofp_header.type 不支持的类型*/
OFPBRC_BAD_MULTIPART = 2,   /* ofp_multipart_request.type 类型不支持 */
OFPBRC_BAD_EXPERIMENTER = 3, /*不支持的实验者身份 ID（在 ofp_experimenter_header
                              或 ofp_multipart_request 或 ofp_multipart_reply）。*/
OFPBRC_BAD_EXP_TYPE = 4, /*实验者类型不支持。 */
OFPBRC_EPERM = 5,        /*允许误差*/
OFPBRC_BAD_LEN = 6,      /*类型错误的请求长度。*/
OFPBRC_BUFFER_EMPTY = 7, /*已被使用的指定缓冲区。*/
OFPBRC_BUFFER_UNKNOWN = 8, /*指定的缓冲区不存在。*/
OFPBRC_BAD_TABLE_ID = 9, /*指定的表标识符无效或不存在。*/
OFPBRC_IS_SLAVE = 10, /*拒绝因为控制器是从设备*/
OFPBRC_BAD_PORT = 11, /*无效的端口 */
OFPBRC_BAD_PACKET = 12, /*包输出里无效的数据包*/
OFPBRC_MULTIPART_BUFFER_OVERFLOW = 13, /* ofp_multipart_request 分配的缓冲区溢出
*/
```

下面是目前定义OFPET_BAD_ACTION错误type的code：

```
/*OFPET_BAD_INSTRUCTION ofp_error_msg 'code'的值。数据至少包含请求失败的第一个64字节*/
enum ofp_bad_action_code {
OFPBAC_BAD_TYPE = 0,        /*未知的行动类型。*/
OFPBAC_BAD_LEN = 1,         /*行动的长度问题。 */
OFPBAC_BAD_EXPERIMENTER = 2, /*未知的实验者指定标识符 */
OFPBAC_BAD_EXP_TYPE = 3, /*实验者的标识符。未知的行动 */
OFPBAC_BAD_OUT_PORT = 4, /*问题有效输出端口。*/
OFPBAC_BAD_ARGUMENT = 5, /*不良行为的论证    */
```

```
OFPBAC_EPERM = 6,          /*允许的错误*/
OFPBAC_TOO_MANY = 7,       /*不能处理这么多的行动。*/
OFPBAC_BAD_QUEUE = 8,      /*问题验证输出队列。*/
OFPBAC_BAD_OUT_GROUP = 9, /*在转发的动作无效组标识符*/
OFPBAC_MATCH_INCONSISTENT = 10, /*行动不能申请匹配，或设置字段丢失的前提。*/
OFPBAC_UNSUPPORTED_ORDER = 11, /*应用行动指令的行动列表不支持行动排序*/
OFPBAC_BAD_TAG = 12,           /*行动使用不支持的标签/封装。*/
OFPBAC_BAD_SET_TYPE = 13,      /*在set_field行动不支持的类型。*/
OFPBAC_BAD_SET_LEN = 14,       /*在set_field行动长度问题。*/
OFPBAC_BAD_SET_ARGUMENT = 15,  /*在set_field行动不好的论据  */
};
```

下面是目前定义为OFPET_BAD_INSTRUCTION错误type的code：

```
/*为OFPET_BAD_INSTRUCTION ofp_error_msg“code”的值。数据至少包含请求失败的第一个64字节。*/
enum ofp_bad_instruction_code {
OFPBIC_UNKNOWN_INST = 0,   /*未知的指令*/
OFPBIC_UNSUP_INST = 1,     /*交换机或表不支持的指令。*/
OFPBIC_BAD_TABLE_ID = 2,   /*指定流表标识符无效。 */
OFPBIC_UNSUP_METADATA = 3, /*元数据值不支持的数据通路。 */
OFPBIC_UNSUP_METADATA_MASK = 4, /*元数据的掩码值不支持的数据通路。*/
OFPBIC_BAD_EXPERIMENTER = 5,  /*未知的实验者指定标识符。 */
OFPBIC_BAD_EXP_TYPE = 6,       /*实验者未知指令标识符。*/
OFPBIC_BAD_LEN = 7,            /*在指令长度的问题。 */
OFPBIC_EPERM = 8,              /*允许的错误。 */
};
```

为OFPET_BAD_MATCH错误type，下面是目前定义的code：

```
/*为OFPET_BAD_MATCHofp_error_msg “code”的值。数据至少包含请求失败的第一个64字节。*/

enum ofp_bad_match_code {
OFPBMC_BAD_TYPE = 0, /*不支持匹配指定的匹配类型*/
OFPBMC_BAD_LEN = 1,        /*匹配长度问题*/
OFPBMC_BAD_TAG = 2,        /*匹配采用不支持的标签/封装。*/
OFPBMC_BAD_DL_ADDR_MASK = 3,
                    /*不支持的数据地址掩码 - 交换机不支持任意链路地址掩码。 */
OFPBMC_BAD_NW_ADDR_MASK = 4,
                     /*不支持的网络地址掩码 - 交换机不支持任意网络地址掩码。 */
OFPBMC_BAD_WILDCARDS = 5,    /*不支持的域掩蔽或在匹配中省略的组合*/
OFPBMC_BAD_FIELD = 6,        /*在匹配中不支持的字段类型。*/
OFPBMC_BAD_VALUE = 7,         /*在匹配域不受支持的值。*/
OFPBMC_BAD_MASK = 8,
                   /*不支持匹配中指定的掩码，字段不是dl_address或nw_address。 */
```

```
OFPBMC_BAD_PREREQ = 9, /*一个先决条件不满足。*/
OFPBMC_DUP_FIELD = 10, /*一种字段类型是重复的。*/
OFPBMC_EPERM = 11,     /*允许的错误。 */
};
```

For the OFPET_FLOW_MOD_FAILED error type, the following codes are currently de_ned:

对于OFPET_TABLE_MOD_FAILED错误type，下面是目前定义的代码：

```
/*对于OFPET_TABLE_MOD_FAILED  的ofp_error_msg“code”的值。”data至少包含请求失败第一个64字节的。 */
enum ofp_flow_mod_failed_code {
OFPFMFC_UNKNOWN = 0,      /*未指定的错误*/
OFPFMFC_TABLE_FULL = 1,   /*因为表已满，流不能添加。 */
OFPFMFC_BAD_TABLE_ID = 2, /*流表不存在    */
OFPFMFC_OVERLAP = 3,       /*使用check_overlap标志尝试添加重叠的流*/
OFPFMFC_EPERM = 4,         /*允许的错误。   */
OFPFMFC_BAD_TIMEOUT = 5, /*因为不支持空闲/硬超时，流表不被添加。 */
OFPFMFC_BAD_COMMAND = 6, /*不支持的或未知的命令。*/
OFPFMFC_BAD_FLAGS = 7,    /*不支持的或未知的标志。*/
};
```

对于OFPET_SWITCH_CONFIG_FAILED错误type，下面是目前定义的code：

```
/*为OFPET_SWITCH_CONFIG_FAILED的ofp_error_msg“code”的值。”data至少包含请求失败第一个64字节的*/
enum ofp_group_mod_failed_code {
OFPGMFC_GROUP_EXISTS = 0, /*组不添加因为组添加试图取代一个已经存在的组*/
OFPGMFC_INVALID_GROUP = 1,       /*组不能添加因为组指定的组是无效的*/
OFPGMFC_WEIGHT_UNSUPPORTED = 2, /*交换机在所选组中不支持不均等的负荷分担*/
OFPGMFC_OUT_OF_GROUPS = 3,       /*组表已满。*/
OFPGMFC_OUT_OF_BUCKETS = 4,      /*组的行动存储段已超过最大值。*/
OFPGMFC_CHAINING_UNSUPPORTED = 5, /*  交换机不支持需要转发去的组 */
OFPGMFC_WATCH_UNSUPPORTED = 6, /*这个组不能监视w指定的atch_port或watch_group。*/
OFPGMFC_LOOP = 7,              /*组表项会引起循环*/
OFPGMFC_UNKNOWN_GROUP = 8,   /*组不能修改，因为修改组试图修改一个不存在的组*/
OFPGMFC_CHAINED_GROUP = 9,   /*组不能删除因为另一组是向其转发。     */
OFPGMFC_BAD_TYPE = 10,       /*不支持的或未知的组类型*/
OFPGMFC_BAD_COMMAND = 11,      /*不支持的或未知的命令。 */
OFPGMFC_BAD_BUCKET = 12,       /*存储段里的错误*/
OFPGMFC_BAD_WATCH = 13,     /*在观察端口/组的错误。*/
OFPGMFC_EPERM = 14,          /*允许的错误。*/
};
```

下面是目前定义为OFPET_PORT_MOD_FAILED错误type的代码：

```
/*OFPET_METER_MOD_FAILED的 ofp_error_msg“code”的值。Data至少包含请求失败
```

的第一个64字节*/
```
enum ofp_port_mod_failed_code {
OFPPMFC_BAD_PORT = 0,     /*指定的端口号不存在*/
OFPPMFC_BAD_HW_ADDR = 1, /*指定的硬件地址不匹配端口号*/
OFPPMFC_BAD_CONFIG = 2,  /*指定的配置无效*/
OFPPMFC_BAD_ADVERTISE = 3, /*指定的标识符是无效的。*/
OFPPMFC_EPERM = 4,         /*允许的错误. */
};
```

下面是目前定义为OFPET_METER_MOD_FAILED错误type的code：

/*OFPET_METER_MOD_FAILED的 ofp_error_msg“code”的值。Data至少包含请求失败的第一个64字节*/
```
enum ofp_table_mod_failed_code {
OFPTMFC_BAD_TABLE = 0, /*指定的流表不存在*/
OFPTMFC_BAD_CONFIG = 1, /*指定的配置无效*/
OFPTMFC_EPERM = 2, /*允许的错误*/
};
```

下面是目前定义为OFPET_TABLE_MOD_FAILED错误type的code：

/*OFPET_TABLE_MOD_FAILED的 ofp_error_msg“code”的值。Data至少包含请求失败的第一个64字节*/
```
enum ofp_queue_op_failed_code {
OFPQOFC_BAD_PORT = 0, /*无效的端口（或端口不存在）*/
OFPQOFC_BAD_QUEUE = 1, /*  队列不存在 */
OFPQOFC_EPERM = 2, /*允许的错误*/
};
```

下面是目前定义为 OFPET_SWITCH_CONFIG_FAILED 错误 type 的 code：

/*OFPET_SWITCH_CONFIG_FAILED的 ofp_error_msg“code”的值。Data至少包含请求失败的第一个64字节*/
```
enum ofp_switch_config_failed_code {
OFPSCFC_BAD_FLAGS = 0, /*指定的标志无效*/
OFPSCFC_BAD_LEN = 1,    /*指定的长度无效 */
OFPSCFC_EPERM = 2,       /*允许的错误*/
};
```

下面是目前定义为OFPET_ROLE_REQUEST_FAILED错误type的code：

/*对于OFPET_ROLE_REQUEST_FAILED的 ofp_error_msg“code”的值。Data至少包含请求失败的第一个64字节*/
```
enum ofp_role_request_failed_code {
OFPRRFC_STALE = 0,       /*过时消息: 旧的 generation_id. */
OFPRRFC_UNSUP = 1,      /*控制器不支持角色的变化*/
OFPRRFC_BAD_ROLE = 2, /*无效的角色*/
```

```
};
```

下面是目前定义为 OFPET_METER_MOD_FAILED 错误 type 的 code：

```
/*对于OFPET_METER_MOD_FAILED 的ofp_error_msg“code”的值。Data至少包含请求失败的第一个64字节*/
enum ofp_meter_mod_failed_code {
OFPMMFC_UNKNOWN = 0,          /*未指定的错误. */
OFPMMFC_METER_EXISTS = 1,     /*计量器不加因为试图取代现有的计量器 */
OFPMMFC_INVALID_METER = 2,    /*计量器不加因为指定的计量器是无效的*/
OFPMMFC_UNKNOWN_METER = 3, /*计量器不修改因为试图修改一个不存在的计量器 */
OFPMMFC_BAD_COMMAND = 4,      /*不支持的或未知的命令*/
OFPMMFC_BAD_FLAGS = 5,        /*配置的标志不支持*/
OFPMMFC_BAD_RATE = 6,         /*速率不支持*/
OFPMMFC_BAD_BURST = 7,       /* 不支持的突发大小. */
OFPMMFC_BAD_BAND = 8,        /*频带不支持*/
OFPMMFC_BAD_BAND_VALUE = 9, /*频带值不被支持 */
OFPMMFC_OUT_OF_METERS = 10, /*没有更多的计量器有效  */
OFPMMFC_OUT_OF_BANDS = 11, /*对于计量器的属性数量已超过最大值. */
};
```

下面是目前定义为 OFPET_TABLE_FEATURES_FAILED 错误 type 的 codes：

```
/*对于OFPET_TABLE_FEATURES_FAILED的 ofp_error_msg“code”的值。Data至少包含请求失败的第一个64字节*/
enum ofp_table_features_failed_code {
OFPTFFC_BAD_TABLE = 0,       /*指定的流表不存在*/
OFPTFFC_BAD_METADATA = 1, /*无效的元数据的掩码*/
OFPTFFC_BAD_TYPE = 2,       /*未知的属性类型. */
OFPTFFC_BAD_LEN = 3,         /*属性长度问题*/
OFPTFFC_BAD_ARGUMENT = 4, /*不支持的属性值*/
OFPTFFC_EPERM = 5,          /*  允许的错误 */
};
```

对于为OFPET_EXPERIMENTER错误type,错误信息由以下结构体和字段定义，后面跟着实验者定义的数据：

```
/* OFPET_EXPERIMENTER:错误消息（数据通路->控制器） */
struct ofp_error_experimenter_msg {
struct ofp_header header;
uint16_t type;              /* ofpet_experimenter*/
uint16_t exp_type;          /*实验者定义*/
uint32_t experimenter;    /实验者ID与ofp_experimenter_header中相同*/
uint8_t data[0];            /*可变长度的数据。基于类型和代码解析。无填充 */
};
OFP_ASSERT(sizeof(struct ofp_error_experimenter_msg) == 16);
```

实验者字段是实验者ID，与ofp_experimenter结构体中的相同（见7.5.4）。

7.5对称信息

7.5.1 Hello

OFPT_HELLO消息由一OpenFlow头加一组可变尺寸hello元素。

/* OFPT_HELLO.此消息包含零个或多个可变尺寸的元素。未知的元素类型必须忽略/跳过，允许未来的扩展 */

```
struct ofp_hello {
struct ofp_header header;
/* hello元素列表*/
struct ofp_hello_elem_header elements[0]; /*  元素列表，0或更多*/
};
OFP_ASSERT(sizeof(struct ofp_hello) == 8);
```

Header字段中的version字段（见7.1）必须设置为发送者（见6.3.1）支持的最高OpenFlow协议版本。

Elemnets字段是一组hello元素，包含连接进行初次握手时传递的可选数据。实现时必须忽略（跳过）Hello消息中不支持的所有元素。列表中的元素类型定义如下：

```
/* Hello元素类型*/
enum ofp_hello_elem_type {
OFPHET_VERSIONBITMAP = 1, /*支持的版本位图*/
};
```

元素定义包含元素的类型，长度，和任何相关的数据：

```
/*所有hello元素共同的头*/
struct ofp_hello_elem_header {
uint16_t type;        /* OFPHET_*之一*/
uint16_t length;      /*在本单元的字节长度*/
};
OFP_ASSERT(sizeof(struct ofp_hello_elem_header) == 4);
```

OFPHET_VERSIONBITMAP 元素使用下列结构和字段：

```
/*hello元素的版本位图  */
struct ofp_hello_elem_versionbitmap {
uint16_t type;        /* OFPHET_VERSIONBITMAP. */
uint16_t length;      /*在本元素的字节长度 */
/*其次是：
  包含位图的（长度-4）字节，然后（长度+ 7）/ 8×8-(长度)(0～7)的全零字节*/
uint32_t bitmaps[0]; /*位图列表-支持的版本*/
};
```

```
OFP_ASSERT(sizeof(struct ofp_hello_elem_versionbitmap) == 4);
```

Bitmaps字段表示设备支持的OpenFlow交换协议版本，可在版本协商后使用（见6.3.1）。对位图的比特通过协议的ofp_version数索引；如果左位移数值表示的比特等于设置的ofp_version，就支持这个OpenFlow版本。字段中包含的位图数值取决于支持的最高版本号：ofp_version 0-31 在第一个位图中编码，ofp_version 32-63 在第二个位图中编码，以此类推。例如，如果一个交换机只支持1.0版（ofp_version= 0x01）和1.3版（ofp_version= 0x04），第一个位图将被设置为0x00000012。

7.5.2回送请求

回送请求消息包括一个OpenFlow头加一个任意长度的数据字段。数据字段可能是一个消息的时间戳来检查延迟，不同长度来测量带宽，或零大小来验证交换机和控制器之间是否活跃。

7.5.3回送答复

回送应答消息包括一个OpenFlow头加上回声请求消息中未修改的数据字段。

一个OpenFlow协议在实现时划分成多个层，回声请求/应答的逻辑应该在“最深”的开发层实现。例如，在OpenFlow的实现里，用户空间进程靠近内核模块，回声请求/应答是则在内核模块实现。接收一个格式正确的回应则表示端到端功能比在用户空间的进程中实现的回声请求/应答更可靠，同时也提供了更精确的端到端延迟时间。

7.5.4实验者

实验者消息的定义如下：

```
/*实验者的扩展 */
struct ofp_experimenter_header {
struct ofp_header header;     /*  ofpt_experimenter*/
uint32_t experimenter;        /*  实验者标识符
                              * MSB 0：低位字节由IEEE OUI定义。
                              * MSB! = 0：ONF定义的 */
uint32_t exp_type;            /*实验者定义  */
/*实验者定义任意附加数据 */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_header) == 16);
```

Experimenter的字段是一个32位的值，唯一标识实验者。如果最高位字节是零，接下来的三个字节是实验者的IEEE OUI 。如果最高位字节不为零，这是一个ONF分配的值。如果实验者没有（或希望使用）他们的OUI，他们应该与ONF联系获得一个唯一的实验者标识符。

标准的OpenFlow处理对其余部分不进行解析，可由实验者任意定义。

如果一个交换机不理解实验者的扩充，它必须发送一个带有OFPBRC_BAD_EXPERIMENTER错误代码和OFPET_BAD_REQUEST错误类型的OFPT_ERROR消息。