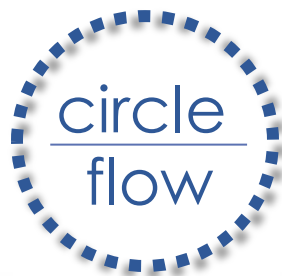


---

# 架构与集成

---



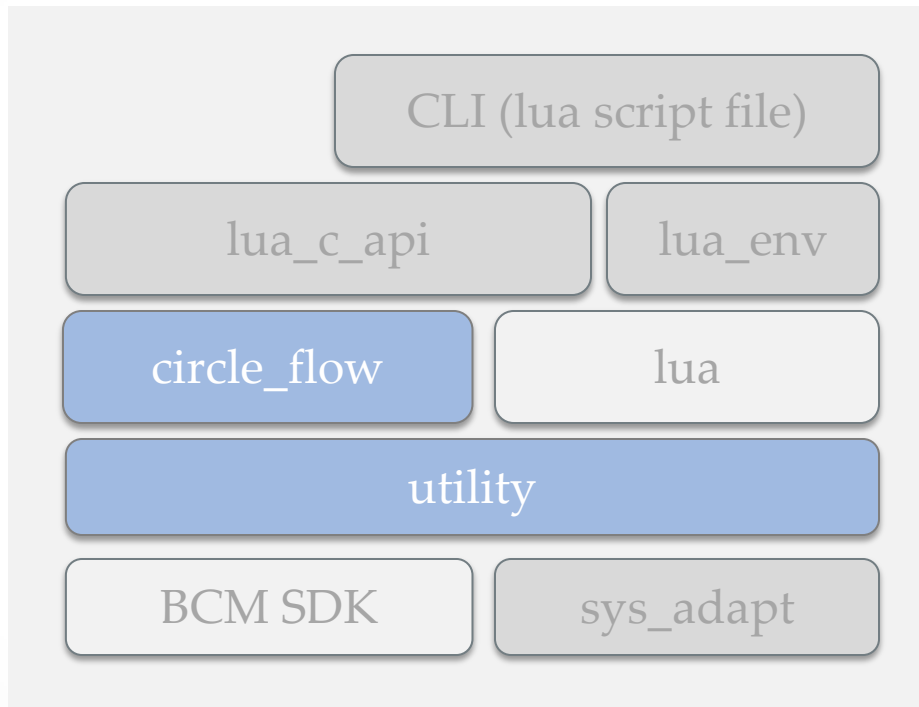
# overview




- 开源项目架构
- 设计原则
- 代码集成指引
- 实现说明

注：

阅读之前，务必对BCM交换芯片和SDK有一个基础的了解。

# whole picture

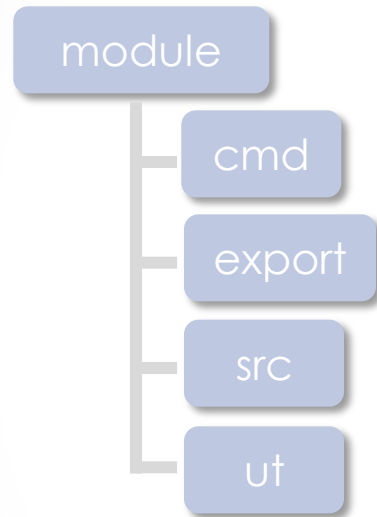


-  : kernel module
-  : system/application module (optional)
-  : third party module

# design rule

- 资源优化为先
- 支持板载多颗同型号芯片
- 开放式平台：API + lua脚本
- 使用exception处理错误
- 多线程安全
- 兼容C++ 98和C++1x

# directory & makefile



#example of makefile

```
SRCS = $(shell /bin/ls -1 src/*.cpp)
SRCS += $(shell /bin/ls -1 src/xxx/*.cpp)
```

...

```
OBJS = $(SRCS:.cpp=.o)
```

#for SDK header file including

```
FLAG += -DINCLUDE_L3 -DBCM_ESW_SUPPORT
        -DSDK_DIR/include
```

#for circle\_flow hal implementation

```
FLAG += -DCF_BCM_56334
```

- export 目录存放对外接口
- src目录存放实现代码， makefile 需自行编写，可参考上面范例
- ut 目录存放unit test对应代码，独立编译运行
- cmd目录下存放模块对应的命令行实现

# unit test

- 测试框架

opmock (0.9x) + gtest

采用opmock的原因是其非侵入式的mock机制避免了ut框架对源码的污染。opmock也有一些bug，比如返回值是常量引用的时候，自动生成的mock代码出现语法错误，已在makefile中通过shell脚本修正。

某些地方会使用ENV\_UT编译宏来选择实现，主要集中在lua\_c\_api模块。

- 运行环境

linux + gcc(g++)

进入ut目录，运行 make即开始编译和测试，

如需单独测试某个子目录，可以 make “子目录”，或者进入该子目录运行make

- 适用性

ut适合纯算法逻辑类的代码，对于硬件配置类的代码则没有太大价值。

因此在源码文件的组织上，尽可能将算法部分独立出来，或者将硬件配置剥离出去。

# dependency

- 芯片规格依赖

有部分实现与硬件规格相关，比如具体资源数量，或者直接操作了寄存器相关代码均通过编译开关CF\_BCM\_XXX进行隔离  
集成时请搜索CF\_BCM\_56334，基于适配芯片具体规格做相应修改

- SDK内部实现依赖

目前有一处实现依赖于SDK的内部数据：  
获取FP entry对应的policer 索引(policer\_hw\_index() ),  
已在v5.6.5 ~ v6.3.4 版本上验证，其他版本需做确认

- 系统依赖

由sys\_adapt模块封装，详见后续介绍

- 库依赖

C++1x模式下，仅依赖标准库  
C++98模式下，需引入boost库

# integration guide

- **circle\_flow**

移植circle\_flow模块, 检视并修改CF\_BCM\_56334编译宏相关代码  
编写makefile

- **utility, sys\_adapt**

详见后文相关模块的说明  
建议使用C++1x编译选项, 可节省大量移植工作

- **lua script**

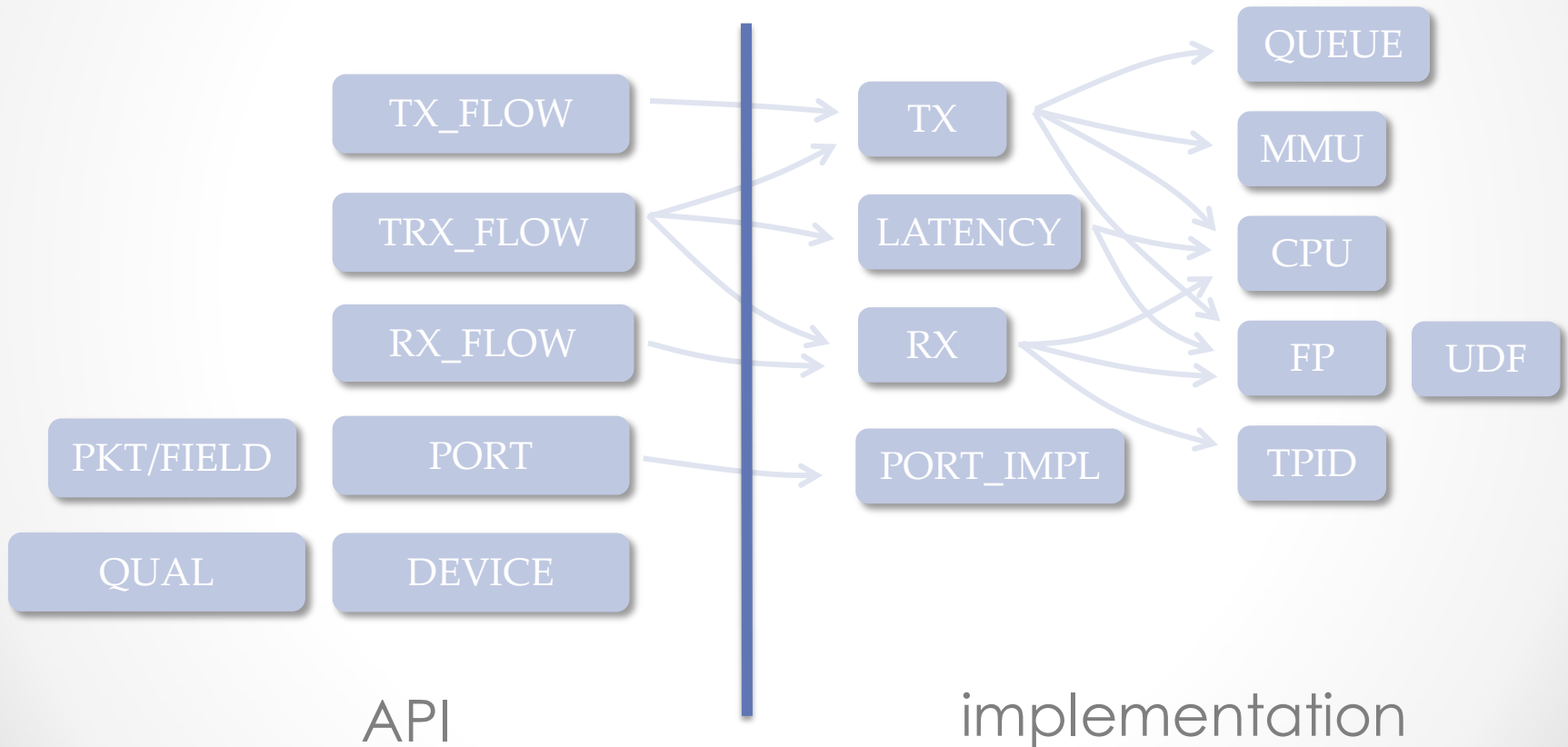
集成lua源码 (从官方网站下载), 注释掉main函数, 编写makefile  
复制lua\_env和lua\_c\_api模块, 编写makefile  
复制lua\_script下的cflow目录到目标运行设备(不是编译设备)的根目录下

- **ut framework**

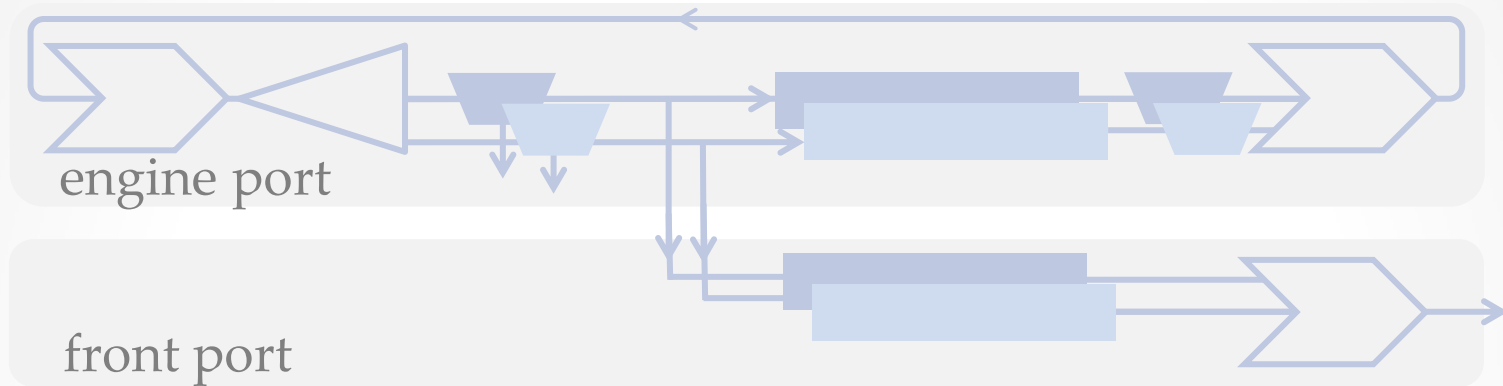
集成 [gtest](#), [opmock](#)和[swig](#) 到ut\_framework路径下, 需从官方网站下载。  
进入opmock/support目录下, make opmock.o



# circle flow insight



# port pair



- **通过一对端口构造一个功能单元**  
产生流量的端口称之为引擎端口，连接外部设备的端口为面板端口，后者向外发送流量以及接收流量
- **可切换的物理接口类型**  
硬件布线可将两个端口都引至面板，分别为RJ45和SFP接口，使用其中一个接口时，另一接口对应的端口即做为引擎端口，这就为面板端口提供了两种不同形式物理接口，带来更灵活的互通性。

# TX flow

- 功能

一个TX flow实例具有独立的速率和burst控制，以及发送计数  
发送的报文流可以是单个报文的重复，也可以是若干个报文的序列  
(可变内容)

- 资源

通过引擎端口的发送队列控制发送速率

分配MMU资源以保证引擎端口队列内pending的报文

通过FP entry实现burst控制，端口复制和队列指定

当流量发送停止时，或者面板端口down时，以上资源自动释放

# RX flow

- 功能

一个RX flow实例可基于指定的报文匹配规则，识别出特定的流量，进行计数，抓包。

- 匹配规则

包括常用的二三四层协议字段，  
或者由用户自定义：偏移量+匹配值/掩码  
需要注意的是，FP（TCAM）按照先后顺序匹配，当一个报文能够匹配两条规则时，排在前面的会被命中。  
比如用户定义了两条RX flow，使用了不同的匹配条件，而某个报文可以同时匹配，实际只会有一条RX flow计数增加。

- 资源

FP entry, UDF window  
CPU 收包分发  
全局TPID

# user define field

- 功能

UDF资源用于RX flow中的自定义匹配规则，以及TRX flow中的flow track的匹配。

可匹配的报文内容的偏移范围，一般为0~127字节。

- 机制

由于BCM SDK的限制，UDF对象的创建必须先于FP group的创建，这无法满足动态配置的需求，因此采用预创建UDF对象以及直接操作硬件UDF offset window 的方式来实现。

- 资源

兼容XGS系列芯片的UDF机制，只需根据芯片具体规格调整block和chunk规格(UDF\_HW\_SPECS)

UDF资源有限，可能会出现资源耗尽导致用户配置失败

# TRX flow

- 功能

将TX flow和RX flow的功能捆绑在一个实例中，并额外提供时延测量的功能。

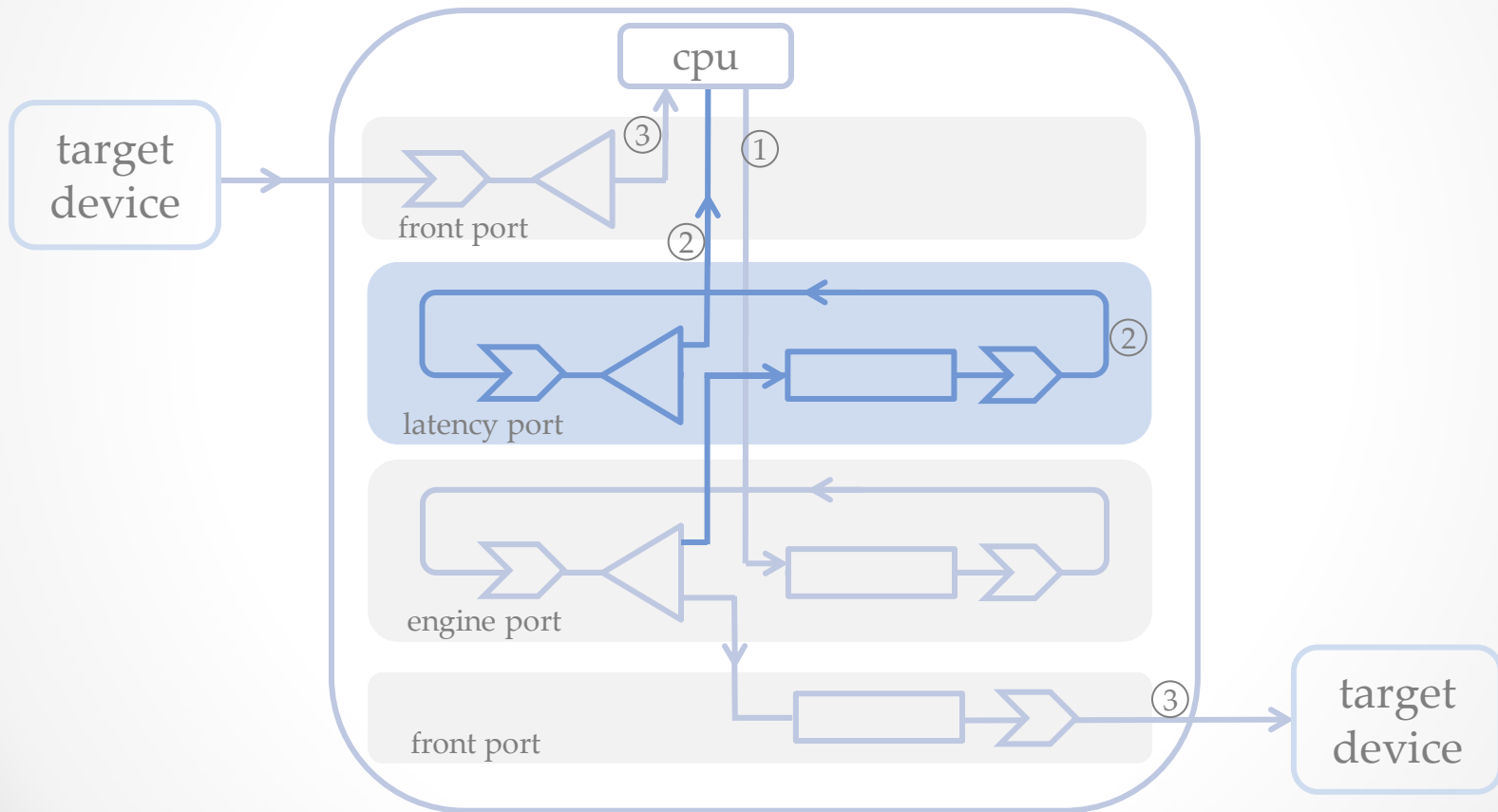
- 机制

发送流量的报文包含了唯一标识符（flow track），接收流量的识别通过匹配该标识符实现，无需其他二三层协议字段，该标识符的匹配通过UDF实现，因此偏移量必须在128字节以内。

- 时延测量

基于流量标识符，和紧随其后的报文序列号，我们可以识别出特定的报文，从而可以测验发送和接收之间的时延。  
相见后文

# latency measurement



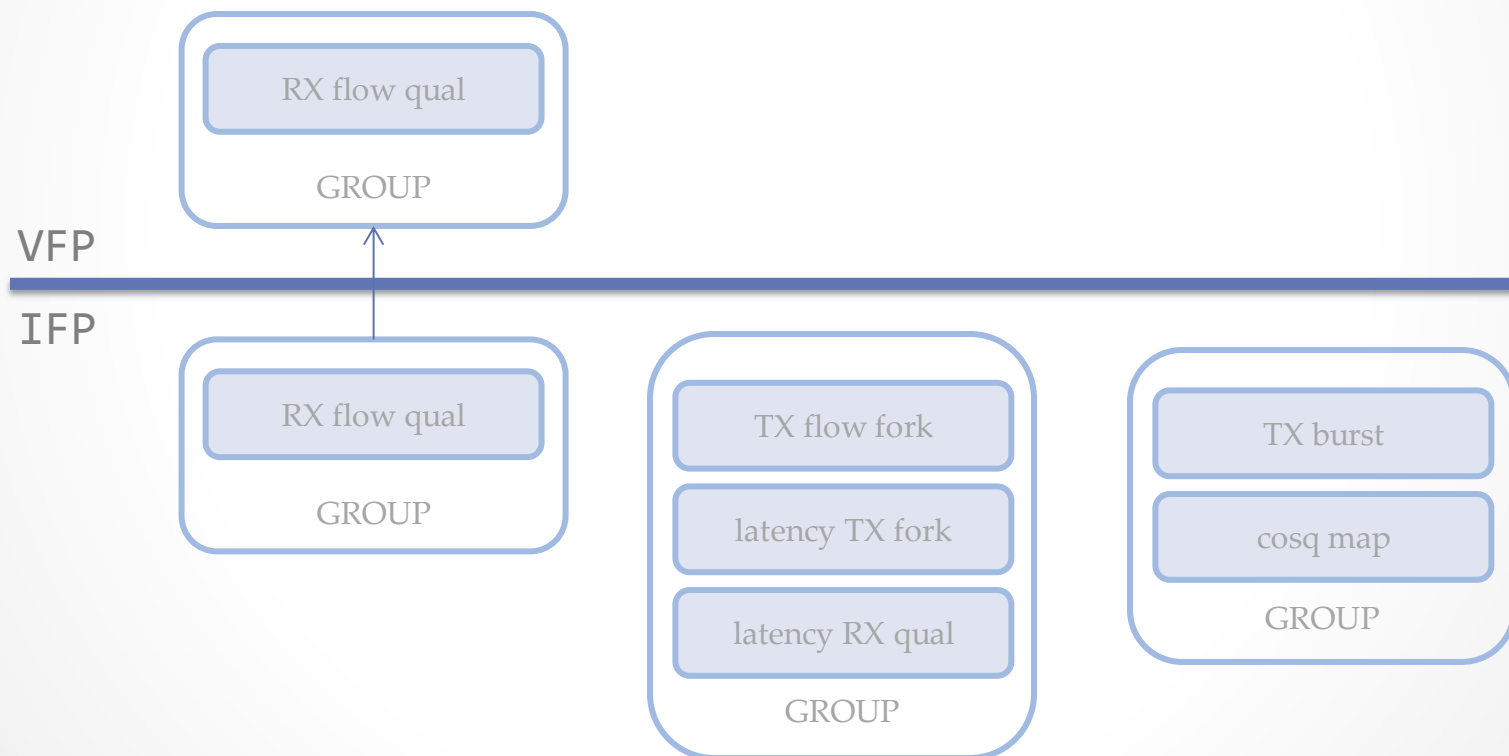
- ① first copy of pkt from CPU
- ② second copy of pkt destined to CPU
- ③ third copy of pkt destined to CPU

# latency measurement

- 由于FP不支持同时复制报文和打时间戳，因此引入了时延端口(latency port)，来提供发送时刻（T1）的时间戳。
- 引擎端口仍旧复制两份报文，分别送往面板端口和时延端口，后者自环后通过FP送往CPU，并打上时间戳T1。
- 经面板端口发送到目标设备的报文，经过处理再次发送回到面板端口，经FP送往CPU，打上接收时刻的时间戳T2。
- 比较上述两个时间戳得到转发时延。



# FP 资源



# TPID

- 背景

芯片允许全局配置若干个外层TPID的值，然后由每个端口选择使用全部或部分，用于报文解析。

外层TPID全局一般为4个，内层TPID全局只允许配置一个。

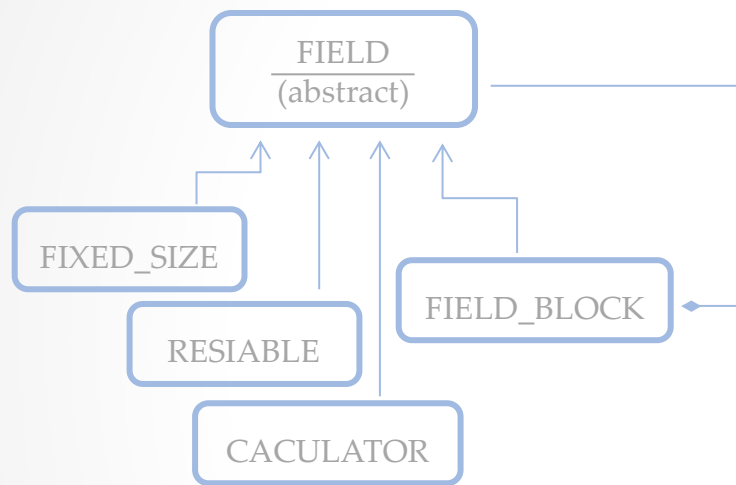
- 功能

作为RX flow的匹配选项，当需要匹配TPID时，意味着该值必须作为全局配置的TPID值，同时对应端口需要选择使能该TPID，这意味着要对TPID资源进行管理。

- 资源

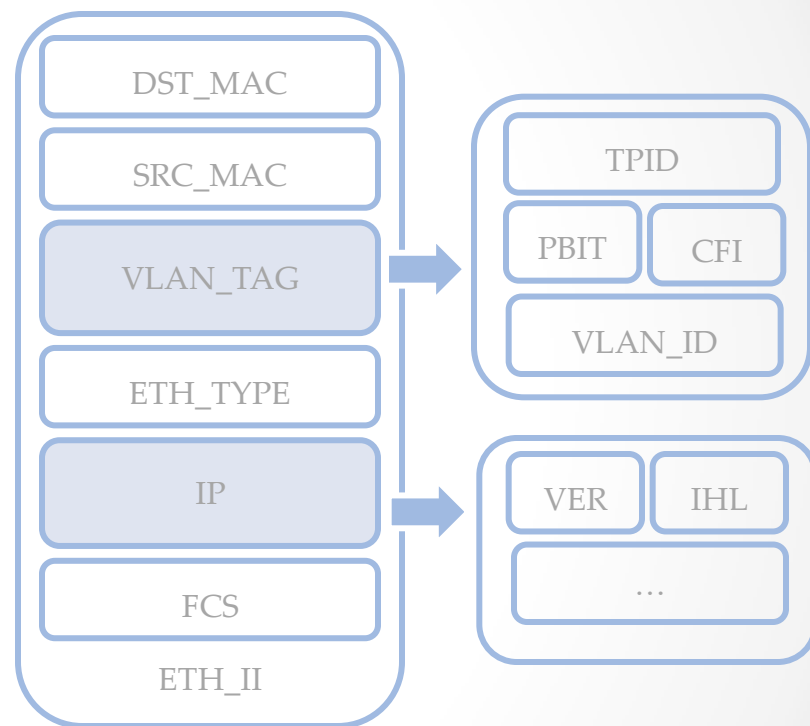
资源有限，可能会出现资源耗尽导致用户配置失败

# field/field block/packet



↑ : inherent  
→ : contains

FIELD\_BLOCK本身既为FIELD实例，  
同时又持有多个FIELD实例



典型报文结构：  
IP over ethernet with vlan tag

# field/field block/packet

- 赋值

field实例可直接赋值字符串或者BYTES (vector<UINT8>), 也可以从field实例直接转换成字符串或者BYTES

- 字符串解析

可自定义parser对象, 并传递给field实例, 用于字符串和BYTES之间的转换。缺省有两个parser实例, 分别针对十六进制和十进制字符串, 默认为前者。

- 内容

field实例可赋值seed对象, 定义变化内容: 递增, 递减, 随机  
field实例可赋值pattern对象, 适用于较长字段的赋值

- field block

一个field block实例中包含若干个field实例, 可对后者的进行增删操作  
packet即为BYTES

一个field block实例对应一个或多个packet, 取决于其中的field是否定义了变化内容

# packet dispatch

- 背景

RX flow的snooping会把报文送到CPU，为了区分报文属于哪一条flow，理想的解决方法是在FP的action中加入match id来指示，但是它无法和timestamp 动作共存，因此通过source port + cpu queue来区分

- 资源

BCM XGS 系列的CPU端口包含48个queue

因此，任意一个面板端口最多能够创建48条使能snooping的RX flow  
对于TRX flow，时延测量使能时需额外再使用一个CPU queue

# utility

常用工具类的抽象和封装  
用于简化编程以及屏蔽系统差异

ERROR/TRACE/LOG	调试手段
CMD	命令行封装
MUTEX	基于RAII机制的信号量
shared_ptr/weak_ptr	基于c++11或boost的智能指针
TIMER/TIME	异步/同步定时器
THREAD	基于RAII的线程
WAIT	条件等待
RANDOM	优化的随机值生成器
UNIQUE	索引号分配/回收管理

# utility integration

- C++98编译模式

须自定义实现若干系统相关接口（详见\_dependency.h）

须集成boost库（下载boost代码库，复制其中的头文件目录boost，无需其他目录，无需编译）

- C++11编译模式（推荐）

绝大部分功能的默认实现基于标准库，无需自定义实现

需在makefile中使能编译宏ENV\_C11

- CMD的实现依赖于目标系统

须自定义实现相关接口（详见\_dependency.h）

# utility : output

error print	当错误发生时，输出调试信息：代码位置，调用栈，时间戳等。 默认使用printf，可定制输出接口
trace print	运行时的上下文详细信息，平常关闭，仅在调试时打开。 默认使用printf，可定制输出接口
cli print	命令行输出



# error clean

- 基于exception的错误传递机制

分离业务处理逻辑和错误处理逻辑，函数内仅需happy path即可  
函数返回值的用途重新回归为函数输出值，有助于简洁代码

- exception safe的编程原则

基于RAII机制的类保证了堆栈回滚时自动释放资源，无需显式地判断  
和处理错误

功能代码不再受错误处理代码的干扰，happy path only.

- 仅在边界处理错误

在API边界处理错误：

重新封装exception类型，并抛出。

或者终结exception，并返回log id，通过该id可获取错误信息。

# smart trace

- 基于object的trace 输出

变参构造trace object, 就像这样: `TRACE("format", a, b, ...);`  
通过object的构造/析构函数, 可实现任何代码段的生命周期的自动化打印。

在关闭输出时, trace object的构造函数仅保存参数, 并不处理格式化字符串, 因此运行成本极低。

- 可控的, 自定义分类的trace输出

通过 trace filter对象来控制trace的输出  
filter对象的构造只需指定一个用以区别其他filter对象的字符串即可

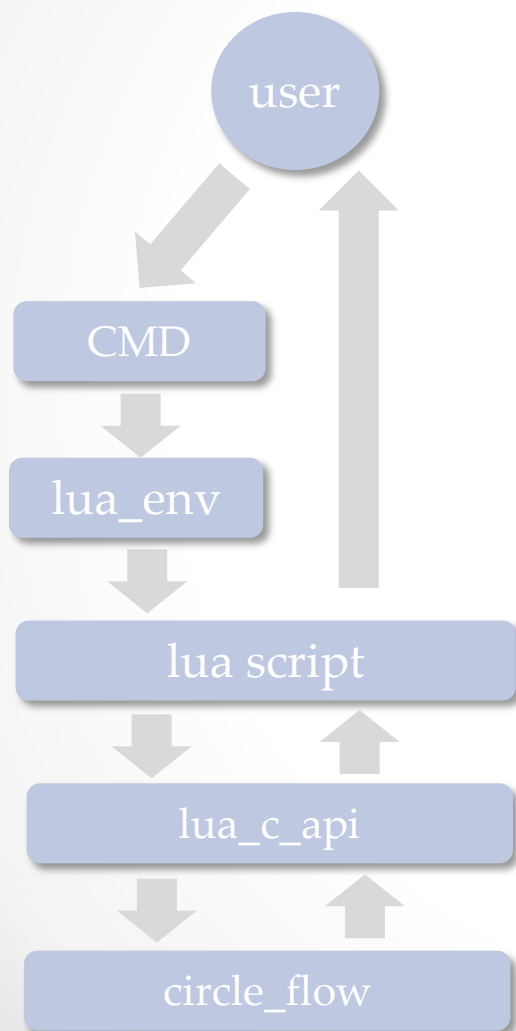
- 错误(exception)触发的trace输出

在发生exception的情况下, trace object的析构函数自动将当前线程的所有trace object按照构造顺序全部dump, 这有助于在不使用core dump的情况下收集更多调用栈的信息。

# sys\_adapt

- 封装因系统而异的接口  
实现那些在utility中需注册的系统依赖性操作接口  
详见utility/export/\_dependency.h
- 目标设备配置参数  
端口配置（port pair，默认端口工作速率等）  
详见device.cpp
- 目标设备初始化入口  
封装上述初始化函数，提供设备初始化入口  
参考sys\_adapt/src/init.cpp

# cli + lua script

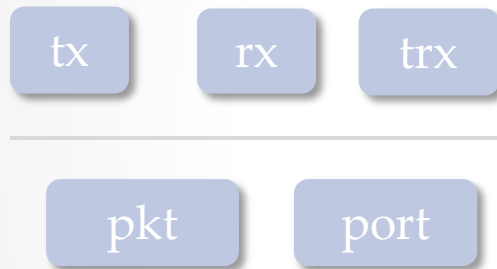


基于脚本的命令行实现  
无需编译加载

lua强大的字符串处理能力  
有助于简化命令行实现

开放的C\_API  
用户可自定义命令

# example of cmds



```
#define packet
> pkt create ipv4 prototype=eth_ipv4 length=64
> pkt set ipv4 dst_mac=00:00:00:00:00:11
> pkt set . src_mac=22
> pkt show .
```

```
#create tx flow
> tx create tx_1
> tx set . port=ge00 rate=100mbps pkt=ipv4
> tx start .
> tx show
> tx cnt
```

```
#create rx flow
> rx create rx_1
> rx set . port=ge01 dst_mac=11 src_mac=22
> rx start .
> rx show
> rx cnt
```

# 注： 句号 (.) 代表上一次有效输入的对象名称

\_\_\_\_\_ find more \_\_\_\_\_  
at  
[www.circleflow.net](http://www.circleflow.net)  
or  
[github.com/circleflow](https://github.com/circleflow)  
\_\_\_\_\_

