# **dp单元测试框架介绍与实践指导**

## **〇、quick start**

1. 以agent仓为例，在 agent/cpe-agent/src/t 目录下，新建一个.c文件，如 lightwanNewT.c
2. 在agent/cpe-agent/src/t/Makefile.src中添加如下一行。如果是平台相关的，或者特性宏相关的，加上对应判断。文件中都有先例

TEST\_SRC\_FILES += $(UNIT\_TEST\_DIR)/lightwanNewT.c

1. .c文件的内容

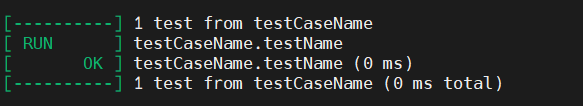
#include "lightwanUnitTestFramework.h"  
  
// testFunc 原型自拟，无限制  
LW\_ERR\_T  
testFunc(  
 void  
 )  
{  
 return LW\_SUCCESS;  
}  
  
// 这两个name自行拟定，第一个表示测试用例，第二个表示这个测试用例里具体的测试点（称为测试特例）  
LW\_TEST(testCaseName, testName)   
{  
 EXPECT\_EQ(LW\_SUCCESS, testFunc()); // 只有两者相等，才测试通过  
}

1. 编译agent仓（以centos + cpe为例）

agent/cpe-agent/src/check\_build\_error.sh TAG\_PLATFORM=centos TAG\_MODE=cpe

编译结果件是agent/cpe-agent/src/autoTest

1. 目前如果在centos机器上编译centos + cpe，会自动运行单元测试，上面的例子运行结果如下



也可以用如下命令（两条命令）手动运行autoTest

cd agent/cpe-agent/src

LD\_LIBRARY\_PATH=$(pwd); for line in $(ls -d $(pwd)/modules/\*/); do LD\_LIBRARY\_PATH="$LD\_LIBRARY\_PATH:$line"; done; LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH ./autoTest

1. 需要更进阶的功能，
   1. 先查看“gtest框架/ktf框架”小节，了解gtest/ktf的使用流程，以及除了EXPECT\_EQ以外的判定宏；
   2. 再查看“框架基本用法总结”小节，了解框架还有哪些添加用例的方式；
   3. 最后查看“编写用例”、“运行用例”、“部署用例”章节，了解如何在业务代码里，设计并完成完整的测试用例。

## **单元测试的目的**

倡导“**码未动，测试现行**”。我们在编写单元测试的过程中，其实就是在设计我们的代码将要处理哪些问题。先思考函数如何设计，将要处理/完成哪些功能，如何处理异常/边界值。单元测试越全面，就代表你的代码更加健壮。

切勿理想化，编译通过且仅仅达到了某些预期值，就“完成了”。切勿有“浪费时间”，“有Smoke test”的观念。Bug在整个研发流程中被发现的越早，修改的代价就越低；否则大量的时间将被花费在跟踪那些包含在独立单元里的简单的Bug上面。

## **框架介绍**

### **简介**

简单来说，dp单元测试框架分是 google的gtest框架（内核态是ktf框架） + lightwan封装。

整个测试框架的编写与执行过程：

* 1. 各个.c里调用框架API注册测试用例（testCase），每个测试用例里可以有多个测试特例（test），每个测试特例对应一个具体的测试功能点
  2. main函数调用RUN\_ALL\_TEST()
  3. 运行global.setUp()
  4. 运行testCase1.setUp()
  5. 运行testCase1.test1.setUp()
  6. 运行testCase1.test1，输出测试结果
  7. 运行testCase1.test1.tearDown()
  8. 按注册顺序，对testCase1里的后续test重复e~g
  9. 运行testCase1.tearDown()
  10. 按注册顺序，对后续testCase重复d~i
  11. 运行global.tearDown()
  12. 输出整体测试结果

### **gtest框架**

[三方卡片内容: https://project.appexnetworks.com.cn/projects/cloudwan-full/wiki/%E3%80%90Framework%E3%80%91GoogleTest%E6%A1%86%E6%9E%B6](https://project.appexnetworks.com.cn/projects/cloudwan-full/wiki/%E3%80%90Framework%E3%80%91GoogleTest%E6%A1%86%E6%9E%B6)

除了上面的wiki外，网上也有很多gtest的资料，这里不再赘述。

### ktf框架

[三方卡片内容: https://project.appexnetworks.com.cn/projects/cloudwan-full/wiki/%E3%80%90Framework%E3%80%91Kernel\_Test\_Framework\_%E5%86%85%E6%A0%B8%E5%8D%95%E5%85%83%E6%B5%8B%E8%AF%95%E6%B5%8B%E8%AF%95%E6%A1%86%E6%9E%B6](https://project.appexnetworks.com.cn/projects/cloudwan-full/wiki/%E3%80%90Framework%E3%80%91Kernel_Test_Framework_%E5%86%85%E6%A0%B8%E5%8D%95%E5%85%83%E6%B5%8B%E8%AF%95%E6%B5%8B%E8%AF%95%E6%A1%86%E6%9E%B6)

### **lightwan框架**

[lightwan单元测试框架介绍](https://project.appexnetworks.com.cn/projects/cloudwan-full/wiki/%E3%80%90Framework%E3%80%91Lightwan_Test_Framework)

这里补充说明几点：

* 1. wiki里的 LW\_CASEOPERATIONS 封装，主要是提供了一个标准化的test执行流程，并且封装了灵活的init和release，可以实现每个test前后的统一处理，避免每个test都要写一遍。
  2. LW\_CASEOPERATIONS 的封装较灵活但也相对较重量级一点，且不能实现testCase级别的init和release（同一testCase里，第一个test执行前init，最后一个test执行后release）。可以使用封装的 LW\_TEST\_F 宏解决这些问题，这里setUp和tearDown分别对应init和release，具体用法见宏定义。
  3. 如果要实现整个单元测试级别的init和release，也就是wiki里提到的全局事件，可以如下修改psbc仓里的UnitTest/lightwanUnitTestFramework/lightwanUSpaceTest.c：

class MyEnvironment : public testing::Environment  
{  
 public:  
 virtual void SetUp()  
 {  
 }  
   
 virtual void TearDown()  
 {  
 }  
};  
  
int main(  
 int argc, char \*\*argv  
 )  
{  
 // 添加全局事件，可添加多个  
 testing::AddGlobalTestEnvironment(new MyEnvironment);  
  
 testing::InitGoogleTest(&argc, argv);  
 return RUN\_ALL\_TESTS();  
}

### 框架基本用法总结

* 1. 包含 #include "lightwanUnitTestFramework.h" 即可。
  2. 添加测试用例
     + 普通用例

LW\_TEST(testCaseName, testName)  
{  
 // check func  
}

* + - 需要测试特例级别init和release（setUp和tearDown）的用例

// define myCaseType  
// define myTestInfo  
  
// implement init  
// implement release  
// implement implement  
// implement check  
  
LW\_ERR\_T  
caseCheck(  
 \_\_in int32\_t CaseType  
 )  
{  
 myTestInfo testInfo =  
 {  
 .Op =   
 {  
 .CaseType = CaseType,  
 .CaseInit = init,  
 .ImplementAPI = implement,  
 .Check = check,  
 .CaseRelease = release  
 }  
 };  
  
 return LW\_CaseRun(&testInfo.Op);  
}  
  
LW\_TEST(testCaseName, testName1)  
{  
 EXPECT\_EQ(LW\_SUCCESS, caseCheck(myCaseType1))  
}  
LW\_TEST(testCaseName, testName2)  
{  
 EXPECT\_EQ(LW\_SUCCESS, caseCheck(myCaseType2))  
}

或

// implement setUp  
// implement tearDown  
// implement check1  
// implement check2  
  
LW\_TEST\_F(testCaseName, testName, emptyFunc, emptyFunc, setUp, tearDown)  
{  
 EXPECT\_EQ(LW\_SUCCESS, check1())  
}  
LW\_TEST\_F(testCaseName, testName)  
{  
 EXPECT\_EQ(LW\_SUCCESS, check2())  
}

* + - 需要测试用例级别init和release（setUp和tearDown）的用例

// implement setUpCase  
// implement tearDownCase  
// implement setUp  
// implement tearDown  
// implement check1  
// implement check2  
  
LW\_TEST\_F(testCaseName, testName, setUpCase, tearDownCase, setUp, tearDown)  
{  
 EXPECT\_EQ(LW\_SUCCESS, check1())  
}  
LW\_TEST\_F(testCaseName, testName)  
{  
 EXPECT\_EQ(LW\_SUCCESS, check2())  
}

* 1. 调用agent代码
     + 单元测试可执行文件里的符号是和cpeagent几乎相同的（在编译框架一节会详细介绍），所以测试用例.c里一般像写agent代码一样调函数，包头文件就行
     + gtest是用c++编译的，所以目前单元测试的.c也都是用g++编译的，在包头文件时，有的头文件没加extern "C"，会导致找不到符号，建议是在要包的头文件里加上（注意要同步加 #ifdef \_\_cplusplus）
     + 静态函数也可以调用，在编译单元测试时，会把static符号忽略掉，所有函数都是全局符号（因为流水线那边要编autoTest，这个改动会影响正常出包，所以目前用TAG进行控制，默认不忽略static，详细的TAG可以看对应代码仓的Makefile，搜索LW\_UNIT\_TEST。目前可以先本地加TAG调试自己的case，合上去前Makefile.src里注释掉，不要跑）

## **编写用例**

### **编写原则**

单元测试作为整个开发流程中开发人员自测的其中一环，并不能替代后续的系统测试（对应我们的模块级测试case）、以及自动化测试，其应该只关注代码本身的逻辑，不应该依赖任何外部环境。换句话说，只要有代码和对应架构的机器，就应该能运行，故和外部环境相关的内容都需要打桩处理，具体怎么打桩见第4小节。

另外，单元测试是一种白盒测试，其本身也有一定的开发与维护成本，所以“单元”的定义比较重要，在人力有限的情况下，所有代码都加单元测试不太现实，应该优先覆盖那些功能明确、或者历史容易出错（逻辑复杂）的代码单元。（这不代表单元测试的代码覆盖率一定不高，设计优良的模块通过不算太多的用例也能达到较高的代码覆盖率。通常70%以上都是可以达到的）

一般来说，需要单元测试的代码单元包括 模块的对外API 和 模块内部功能。

### 模块对外API**用例**

模块所有对外API都需要加用例。我们的protobuf消息都是封装了消息中间层的，这些中间层给业务模块提供的消息转换接口也是对外API。

编写用例时，可以借鉴ChatGPT的意见，从输入、分支、结果等几个方面来设计，完整覆盖整个API。

举例：编写用例测试LW\_GetHAInterfaceIp这个API

LW\_NONNULL((1, 2, 3))  
BOOL  
LW\_GetHAInterfaceIp(  
 \_\_in const char \*IfName,  
 \_\_out uint32\_t \*Ip,  
 \_\_out uint8\_t \*Mask  
 )  
{  
 LW\_ERR\_T ret = LW\_SUCCESS;  
 size\_t loop;  
 LW\_HA\_CONF \*haConf = NULL;  
  
 if ((NULL == IfName) || (NULL == Ip))  
 {  
 ret = -LW\_EINVAL;  
 LW\_LOGE("Parameters error.\n");  
 goto CommonReturn;  
 }  
  
 haConf = LW\_AgentMemZeroAlloc(sizeof(\*haConf));  
 if (NULL == haConf)  
 {  
 LW\_LOGE("Apply memory for ha conf failed\n");  
 goto CommonReturn;  
 }  
  
 ret = LW\_AgentGetHAConf(haConf);  
 if (ret < LW\_SUCCESS)  
 {  
 LW\_LOGW("Get agent ha conf failed\n");  
 goto CommonReturn;  
 }  
  
 for (loop = 0; loop < haConf->HAIntfInstance.HAInterfaceNum; loop++)  
 {  
 if (strcmp(haConf->HAIntfInstance.HAInterface[loop], IfName) == 0)  
 {  
 if (LW\_AgentHaIsMaster())  
 {  
 \*Ip = haConf->HAIntfInstance.HAMasterIp[loop];  
 \*Mask = haConf->HAIntfInstance.HAMasterMask[loop];  
 }  
 else if (LW\_AgentHaIsSlave())  
 {  
 \*Ip = haConf->HAIntfInstance.HASlaveIp[loop];  
 \*Mask = haConf->HAIntfInstance.HASlaveMask[loop];  
 }  
 break;  
 }  
 }  
  
CommonReturn:  
  
 if (haConf)  
 {  
 LW\_AgentMemFree(haConf);  
 }  
 return ret;  
}

TEST(LW\_GetHAInterfaceIpTest, NullPointerInput)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(NULL, &Ip, &Mask);  
 EXPECT\_EQ(-LW\_EINVAL, ret);  
  
 ret = LW\_GetHAInterfaceIp(IfName, NULL, &Mask);  
 EXPECT\_EQ(-LW\_EINVAL, ret);  
}  
  
TEST(LW\_GetHAInterfaceIpTest, MemoryAllocationFailure)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
 // Simulate memory allocation failure  
 EXPECT\_CALL(mock, LW\_AgentMemZeroAlloc(sizeof(LW\_HA\_CONF)))  
 .WillOnce(Return(nullptr));  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(IfName, &Ip, &Mask);  
 EXPECT\_EQ(LW\_FAILURE, ret);  
}  
  
TEST(LW\_GetHAInterfaceIpTest, GetAgentHAConfFailure)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
 // Simulate successful memory allocation  
 EXPECT\_CALL(mock, LW\_AgentMemZeroAlloc(sizeof(LW\_HA\_CONF)))  
 .WillOnce(Return(new LW\_HA\_CONF));  
  
 // Simulate GetAgentHAConf failure  
 EXPECT\_CALL(mock, LW\_AgentGetHAConf(\_))  
 .WillOnce(Return(-LW\_FAILURE));  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(IfName, &Ip, &Mask);  
 EXPECT\_EQ(LW\_FAILURE, ret);  
}  
  
TEST(LW\_GetHAInterfaceIpTest, MatchingInterfaceFoundAsMaster)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
 // Simulate successful memory allocation  
 EXPECT\_CALL(mock, LW\_AgentMemZeroAlloc(sizeof(LW\_HA\_CONF)))  
 .WillOnce(Return(new LW\_HA\_CONF));  
  
 // Simulate successful GetAgentHAConf  
 EXPECT\_CALL(mock, LW\_AgentGetHAConf(\_))  
 .WillOnce(Return(LW\_SUCCESS));  
  
 // Set haConf values to match the interface as master  
 haConf->HAIntfInstance.HAInterfaceNum = 1;  
 haConf->HAIntfInstance.HAInterface[0] = "eth0";  
 haConf->HAIntfInstance.HAMasterIp[0] = 192.168.0.1;  
 haConf->HAIntfInstance.HAMasterMask[0] = 255.255.255.0;  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(IfName, &Ip, &Mask);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
 EXPECT\_EQ(192.168.0.1, Ip);  
 EXPECT\_EQ(255.255.255.0, Mask);  
}  
  
TEST(LW\_GetHAInterfaceIpTest, MatchingInterfaceFoundAsSlave)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
 // Simulate successful memory allocation  
 EXPECT\_CALL(mock, LW\_AgentMemZeroAlloc(sizeof(LW\_HA\_CONF)))  
 .WillOnce(Return(new LW\_HA\_CONF));  
  
 // Simulate successful GetAgentHAConf  
 EXPECT\_CALL(mock, LW\_AgentGetHAConf(\_))  
 .WillOnce(Return(LW\_SUCCESS));  
  
 // Set haConf values to match the interface as slave  
 haConf->HAIntfInstance.HAInterfaceNum = 1;  
 haConf->HAIntfInstance.HAInterface[0] = "eth0";  
 haConf->HAIntfInstance.HASlaveIp[0] = 192.168.0.2;  
 haConf->HAIntfInstance.HASlaveMask[0] = 255.255.255.0;  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(IfName, &Ip, &Mask);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
 EXPECT\_EQ(192.168.0.2, Ip);  
 EXPECT\_EQ(255.255.255.0, Mask);  
}  
  
TEST(LW\_GetHAInterfaceIpTest, NoMatchingInterfaceFound)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
 // Simulate successful memory allocation  
 EXPECT\_CALL(mock, LW\_AgentMemZeroAlloc(sizeof(LW\_HA\_CONF)))  
 .WillOnce(Return(new LW\_HA\_CONF));  
  
 // Simulate successful GetAgentHAConf  
 EXPECT\_CALL(mock, LW\_AgentGetHAConf(\_))  
 .WillOnce(Return(LW\_SUCCESS));  
  
 // Set haConf values to not match the interface  
 haConf->HAIntfInstance.HAInterfaceNum = 1;  
 haConf->HAIntfInstance.HAInterface[0] = "eth1";  
 haConf->HAIntfInstance.HAMasterIp[0] = 192.168.0.1;  
 haConf->HAIntfInstance.HAMasterMask[0] = 255.255.255.0;  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(IfName, &Ip, &Mask);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
 // Assert that Ip and Mask are not modified  
 EXPECT\_EQ(0, Ip);  
 EXPECT\_EQ(0, Mask);  
}

### 模块内部功能**用例**

除了对外API以外，模块的其余函数都算内部函数。我们需要按具体模块的功能划分，从中挑选出重点的功能函数、以及不同功能函数的组合，设计用例。

编写用例时，对于单个功能函数的用例，内容和对外API差别不大，可以按具体函数来决定重点关注输入、分支、结果的哪几部分。部分用例还要关注执行过程前后系统资源的变化情况（内存，cpu使用率等等）。对于多个函数组合的用例，可以自己构造调用顺序，来测试可能会出现的业务流程。

举例：接口地址有netio上报 + dynamic上报，编写用例测试同时上报的情况

TEST(LW\_IfmIntfIpNotify, NetioNotify)  
{  
 // netio  
 char ifName[] = "eth0";  
 LW\_NETIO\_STATE\_NOTIFY NetioNotify = {  
 XXX = XX,  
 ...  
 }  
 LW\_ERR\_T ret = LWCtrl\_IfmInterfaceNotifyHandle(&NetioNotify);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
  
 // check if conf  
 LWCTRL\_INTERFACE\_CONF ifConf;  
 ret = LWCtrl\_IfmInterfaceGetConfFromListByIfname(ifName, &ifConf);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
 EXPECT\_EQ(LW\_SUCCESS, LW\_CheckIfConf(&ifConf));  
}  
  
TEST(LW\_IfmIntfIpNotify, DynamicNotify)  
{  
 // dynamic  
 char notify[] = "XXX";  
 int len = strlen(notofy);  
 LW\_ERR\_T ret = LWCtrl\_IfmInterfaceDynamicCallBack(notify, len);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
  
 // check if conf  
 LWCTRL\_INTERFACE\_CONF ifConf;  
 ret = LWCtrl\_IfmInterfaceGetConfFromListByIfname(ifName, &ifConf);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
 EXPECT\_EQ(LW\_SUCCESS, LW\_CheckIfConf(&ifConf));  
}  
  
TEST(LW\_IfmIntfIpNotify, NetioDynamicNotify)  
{  
 // netio  
 char ifName[] = "eth0";  
 LW\_NETIO\_STATE\_NOTIFY NetioNotify = {  
 XXX = XX,  
 ...  
 }  
 LW\_ERR\_T ret = LWCtrl\_IfmInterfaceNotifyHandle(&NetioNotify);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
   
 // dynamic  
 char notify[] = "XXX";  
 int len = strlen(notofy);  
 LW\_ERR\_T ret = LWCtrl\_IfmInterfaceDynamicCallBack(notify, len);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
  
 // check if conf  
 LWCTRL\_INTERFACE\_CONF ifConf;  
 ret = LWCtrl\_IfmInterfaceGetConfFromListByIfname(ifName, &ifConf);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
 EXPECT\_EQ(LW\_SUCCESS, LW\_CheckIfConf(&ifConf));  
}  
  
TEST(LW\_IfmIntfIpNotify, DynamicNetioNotify)  
{  
 // dynamic  
 char notify[] = "XXX";  
 int len = strlen(notofy);  
 LW\_ERR\_T ret = LWCtrl\_IfmInterfaceDynamicCallBack(notify, len);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
  
 // netio  
 char ifName[] = "eth0";  
 LW\_NETIO\_STATE\_NOTIFY NetioNotify = {  
 XXX = XX,  
 ...  
 }  
 LW\_ERR\_T ret = LWCtrl\_IfmInterfaceNotifyHandle(&NetioNotify);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
  
 // check if conf  
 LWCTRL\_INTERFACE\_CONF ifConf;  
 ret = LWCtrl\_IfmInterfaceGetConfFromListByIfname(ifName, &ifConf);  
 EXPECT\_EQ(LW\_SUCCESS, ret);  
 EXPECT\_EQ(LW\_SUCCESS, LW\_CheckIfConf(&ifConf));  
}

### 常用封装处理（持续补充）

假设要打桩的函数叫XXX，

#### **Stub库打桩**

做法：单元测试代码里，桩函数任意命名（比如XXX\_new），用例执行前，定义Stub stub对象，使用stub.set(XXX, XXX\_new)临时修改XXX的函数地址为XXX\_new，使用stub.reset(XXX)恢复XXX的函数地址。stub对象被销毁时，也会恢复XXX的函数地址，所以写在某个TEST内部的话，只会在这个TEST内部生效。可以先打桩模拟函数返回失败，模拟完后再reset恢复正常的函数，达到覆盖函数执行成功与失败的情况。

优点：不影响正式代码，不修改单元测试的Makefile

缺点：目前内核态符号、用户态.so里的符号暂时不支持打桩，后续会对Stub库进行扩展支持

例：

#include "lightwanUnitTestFramework.h"  
  
#include "lightwanAgentMem.h"  
#include "lightwanAgentHA.h"  
  
static int gofail()  
{  
return -LW\_EINVAL;  
}  
  
static int gosuccess()  
{  
return LW\_SUCCESS;  
}  
  
static void \*retNull()  
{  
//printf("\*\*\*\*\*\*\*\*\*\*\* ret null \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\r\n");  
return NULL;  
}  
  
static void \*retNotNull(size\_t len)  
{  
return malloc(len);  
}  
  
TEST(LW\_GetHAInterfaceIpTest, NullPointerInput)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(NULL, &Ip, &Mask);  
 EXPECT\_EQ(-LW\_EINVAL, ret);  
  
 ret = LW\_GetHAInterfaceIp(IfName, NULL, &Mask);  
 EXPECT\_EQ(-LW\_EINVAL, ret);  
}  
  
TEST(LW\_GetHAInterfaceIpTest, MemoryAllocationFailure)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
Stub stub;  
stub.set(LW\_AgentMemZeroAlloc, retNull);  
 // Simulate memory allocation failure  
 // EXPECT\_CALL(mock, LW\_AgentMemZeroAlloc(sizeof(LW\_HA\_CONF)))  
 // .WillOnce(Return(nullptr));  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(IfName, &Ip, &Mask);  
 EXPECT\_NE(LW\_SUCCESS, ret);  
}  
  
TEST(LW\_GetHAInterfaceIpTest, GetAgentHAConfFailure)  
{  
 const char \*IfName = "eth0";  
 uint32\_t Ip;  
 uint8\_t Mask;  
  
Stub stub;  
stub.set(LW\_AgentMemZeroAlloc, retNotNull);  
// Simulate successful memory allocation  
 // EXPECT\_CALL(mock, LW\_AgentMemZeroAlloc(sizeof(LW\_HA\_CONF)))  
 // .WillOnce(Return(new LW\_HA\_CONF));  
  
stub.set(LW\_AgentGetHAConf, gofail);  
 // Simulate GetAgentHAConf failure  
 // EXPECT\_CALL(mock, LW\_AgentGetHAConf(\_))  
 // .WillOnce(Return(-LW\_FAILURE));  
  
 LW\_ERR\_T ret = LW\_GetHAInterfaceIp(IfName, &Ip, &Mask);  
 EXPECT\_NE(LW\_SUCCESS, ret);  
}

#### **wrap函数**

做法：单元测试代码里，桩函数命名为\_\_wrap\_XXX，单元测试编译时链接参数增加 -Wl,--wrap=XXX

优点：不修改正式代码

缺点：需频繁修改单元测试的Makefile

例：

执行shell命令：lightwanUnitTestSystemWrap.c

#### **weak选项**

做法：单元测试代码里，桩函数命名为XXX；正式代码里，XXX函数声明时增加 #ifdef LW\_UNIT\_TEST \_\_attribute\_\_((weak)) #endif

优点：不修改单元测试的Makefile

缺点：需要修改正式代码，打桩很多的话，正式代码会变得比较难看

例：

暂无

### **公共用例提取**（持续补充）

类似的用例可以提取出一套用例模型，方便后续其他人编写。提取方法可参考下面这个wiki

[三方卡片内容: https://project.appexnetworks.com.cn/projects/cloudwan-full/wiki/%E3%80%90Framework%E3%80%91Common\_case\_methodmodule](https://project.appexnetworks.com.cn/projects/cloudwan-full/wiki/%E3%80%90Framework%E3%80%91Common_case_methodmodule)

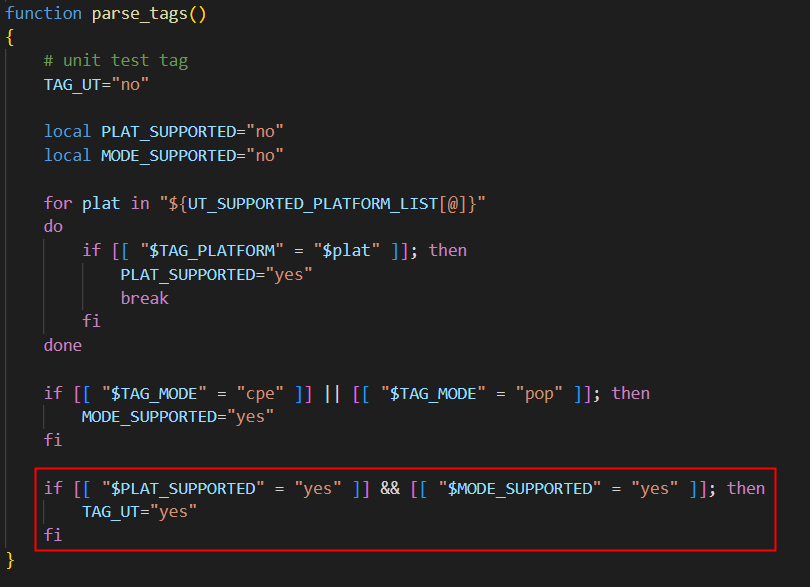
## **执行用例**

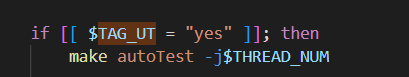
### **编译与**执行

各个代码仓单独编译autoTest，单独执行。

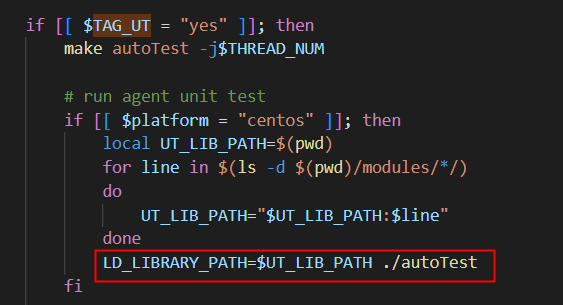
用户态以agent仓为例，其他仓大同小异。在 agent/cpe-agent/src/Makefile 里：

在编cpe或者pop时，会默认 make autoTest

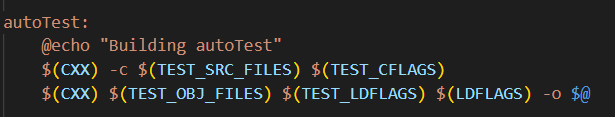




如果编的平台是centos，还会顺手运行（当前默认编译机平台和要编的平台相同）



autoTest使用g++编译，.c文件通过TEST\_SRC\_FILES指定，TEST\_SRC\_FILES在agent/cpe-agent/src/t/Makefile.src里赋值。



cflags继承agent编译时的所有cflags（特性宏什么的都有）



ldflags指定gtest库和libcpeagent.a，并且上面编译命令有指定agent的ldflags，所以可以调用agent进程里的所有符号。但要注意libcpeagent.a是用gcc编译的，autoTest是用g++编译的，c++代码调用c代码时，需要让编译器知道被调用的函数是以c的方式编译的，也就是说被调用函数声明的地方（所在的头文件）里，需要加上extern "C"。



cflags和ldflags还有一些其他的特殊适配，这里没有展示，可到Makefile里自行查看。

内核态以psbc仓为例，首先编译psbc的内核态代码

cd psbc

./build.sh MOD=k KDIR=/usr/src/kernels/3.10.0-514.26.2.el7.x86\_64

接着编译autoTest的ko

cd linux-kern/UnitTest

make

最后加载ko，并运行ktfrun

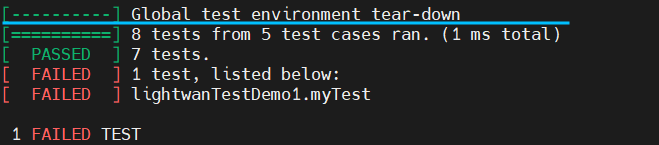
make install

### **结果分析**

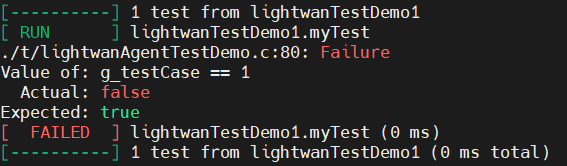
用户态：

在 Global test environment tear-down 之后，就是当前 autoTest 的整体执行结果。

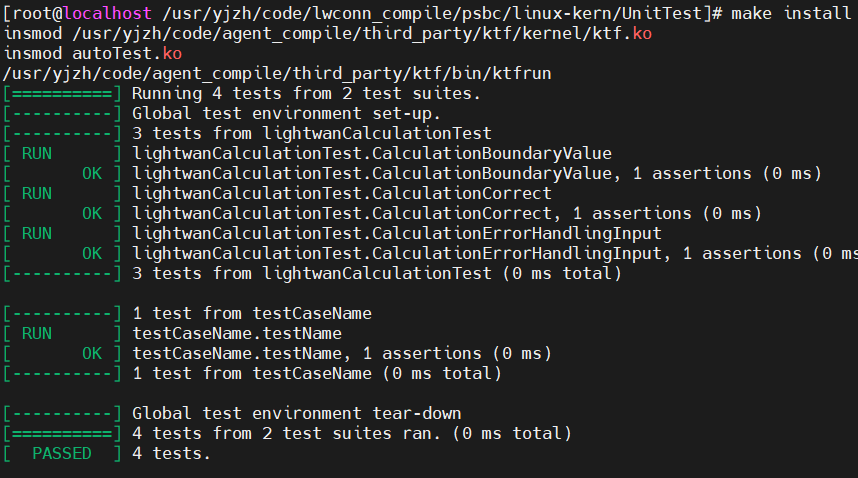
8 tests指的就是8个测试特例，5 test cases就是5个测试用例。



有测试特例出错时，根据测试特例/用例名，找到对应的打印，可以比较清晰的看到出错的原因。



内核态：

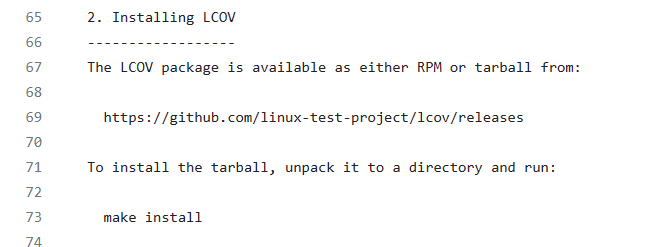


### **覆盖率统计**

当前agent仓默认不统计覆盖率，需要时在check\_build\_error.sh脚本命令后面加上TAG\_UT\_COV=yes即可，默认生成的覆盖率结果在agent/cpe-agent/src/utCovReport目录，打开目录下的index.html即可。下面简单介绍一下覆盖率统计原理。

一般使用gcov + lcov统计代码覆盖率。

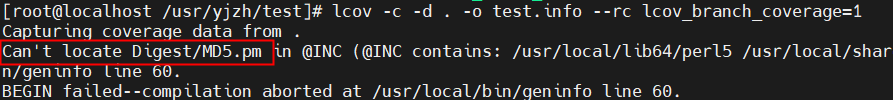
gcov是gcc自带的，lcov一般需要手动安装下，镜像源没有安装包的话可以到源码仓下载安装，参考[lcov/README · GitHub](https://github.com/linux-test-project/lcov/blob/master/README)里的这一段：



使用方法如下：

* 1. 在所有需要统计的代码的CFLAGS和LDFLAGS里加上 **-fprofile-arcs -ftest-coverage**，告诉gcc/g++生成.gcno文件，并在代码里插入gcov桩
  2. 正常编译
  3. 运行编译出来的可执行文件，这里是autoTest。运行时插入的gcov桩会生成.gcda文件
  4. 用lcov收集整理代码覆盖率 **lcov -c -d . -o utCov.info --rc lcov\_branch\_coverage=1**，-c表示收集代码覆盖率，-d .表示收集当前目录，收集的就是.gcda文件；-o表示输出文件，即lcov根据.gcno + .gcda分析生成的最终覆盖率结果，不过这个.info文件可读性比较差，下面还会转换一下；--rc表示修改lcov的配置项，lcov\_branch\_coverage=1表示打开分支覆盖率统计。
* 注意：

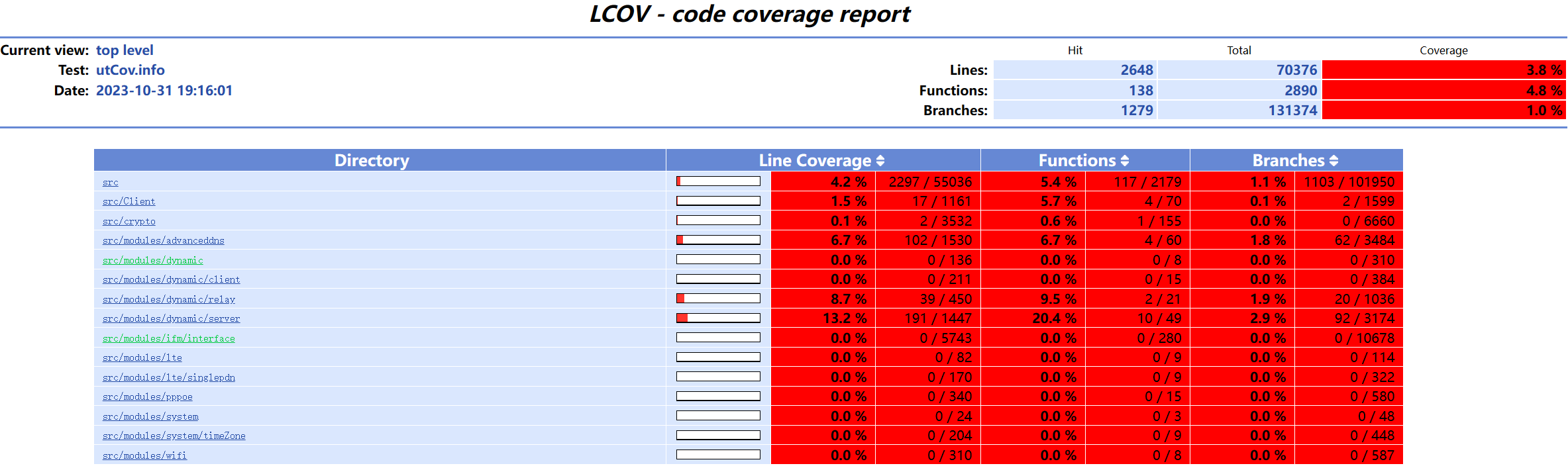
（1）lcov运行依赖一些perl的包，如果报错和perl相关，根据报错信息，安装对应名称的包即可

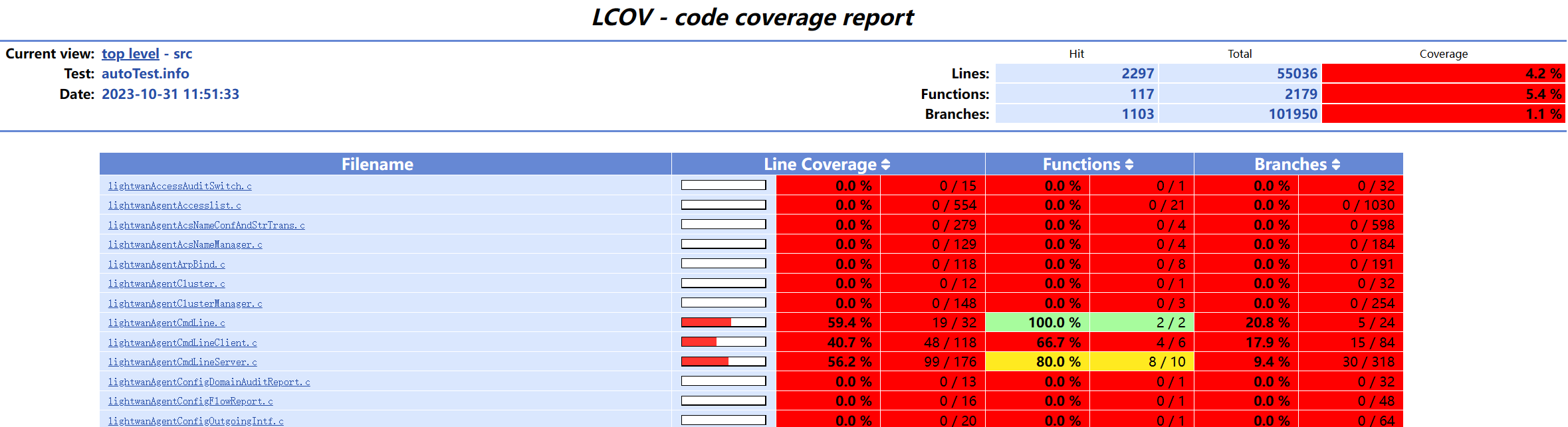




（2）lcov默认是关闭分支覆盖率统计的，如果需要分支统计的话，每个lcov命令都要加上--rc lcov\_branch\_coverage=1，不然那条命令就会按不需要分支统计的方式去执行，可能会丢失数据。

* 1. 生成网页版统计 **genhtml --branch-coverage -o utCovReport utCov.info**，genhtml是lcov自带的转换工具，--branch-coverage表示转换分支覆盖率统计；-o表示转换完网页的存放目录，可以自动创建；最后再指定要转换的源文件为lcov生成的.info文件即可。网页效果如下：



文件夹点开可以看到每个文件的统计数据，每个文件也可以点开，看到具体行、函数、分支的覆盖情况。

## **部署用例**（待流水线优化完成后补充）

### **部署原则**

### **本地部署**

### **流水线部署**

#### **编译流水线部署**

#### **smoke流水线部署**

## **常见问题QA**

Q：内核态报 error: ‘self’ undeclared (first use in this function) 错误？

A：ktf框架的EXPECT、ASSERT等判定宏只能写在LW\_TEST/LW\_TEST\_F里面，不能写在自己的check函数里。