Flow Language Extension

Shi Yan

6/26/2016

Index

1 Logic Flow

1.1 Function return means job gets done

When we write a function to implement a task, e.g., sending a block of data to a socket, sending out a request to peer and waiting for its response, or asking user to fill a form, etc., in many times, after the function returns, the task is just done partially. It needs extra input signals to let the task continue. And to prevent from blocking the thread, we don't want to wait indefinitely for those signals. Instead, we register a callback and ask the function to call it back so that the task can resume. Asynchronous programming is very commonly used. But its drawback is that the code to complete one single task has to be split into a few pieces. And since all tasks that make call of asynchronous task have to be split as well, the whole program is heavily fragmented which make it hard to write, read and maintain. It increases chances of producing bugs. The whole developing cycle is greatly prolonged.

The solution is to work out a way to describe task flows in the way how it is designed instead of how it is going to be executed. When a function returns, the task should be completed. Only in this way can we easily put up more functions on it to handle complicate logics.

1.2 Case 1

For a client to connect to a server, usually we need 3 steps:

1. Get connected with the server
2. Send a request data packet
3. Receive a response data packet from server.

It will be quite natural if we can write something like this:

```
do
{
    s = connect(server_ip, server_port);
    send(s, request_data_packet, request_data_len);
    recv(s, response_data_packet, response_data_len);
}
in_case_any_error(...)
{
    print_error();
    return;
}
```

Unfortunately, we don't write such code in most commercial scenarios. None of above three function returns upon job gets done. We have to call connect() and wait for the signal that the socket gets connected successfully, then call send(), if it returns uncompleted then wait for the write_ready signal and call send() again, until all data send out, then keep calling recv() upon read_ready signal until all necessary data received. So usually, we register a callback and use a state machine to handle different signals in all kinds of states. The more steps we need, the uglier the code becomes.

1.3 Case 2

Not only network related programming have such issue, asynchronous call occurs everywhere.

E.g., in social network apps, we often need to show a friend's thumbnail base on his ID. And sometimes it may not exist in the memory. So what we do is:

1. Search that friend's info in the memory.
2. If not found, search database in local disk.
3. If still not found, download one from server.

Naturally, we want to write code like this:

```
CUser* getUserInfo(int user_id)
{
    CUser* pUser = NULL;
    do
    {
        if (!(pUser = search_memory_cache(user_id)))
        {
            if (!(pUser = search_local_disk(user_id)))
            {
                if (!(pUser = download_user_info(user_id)))
                    return NULL;
                add_to_local_disk(pUser);
            }
            add_to_memory_cache(pUser);
        }
    }
    in_case_any_error()
    {
    }
    in_case_cancelled()
    {
    }
    return pUser;
}
```

But in reality, since it takes quite a few milliseconds to search local disk, and it takes much longer to download from server, what we usually do is we maintain two modules, local_database_manager module and server_download_manager module. Each module manage tasks. You can call it to create a task and it returns you a task id. Upon task complete, it calls you back with the task id and your result. And before that, you can also call to cancel a task. Then, to get a user info by its id, we need a state machine, remembering whether we are searching it from memory, or searching from local data or downloading it from server, and store the task id in case the user might want to cancel the whole process.

1.4 Logic flow

By studying the way how a program is executed, we can see a program is usually driven by one or more message/event loops. Each loop per thread. There could be many logic flows running in parallel within a thread. In a thread, only one logic flow runs at a time. One logic flow keeps running until it reaches a waiting point and it cannot go any further, then the thread will either execute other flows (with remaining signals) or the whole thread enters into waiting status.

Logic flows can be nested. One parent flow may create a few child flows.

A flow can be terminated/destroyed when it's in waiting state.

Consider following case: By given a few IP addresses, an application needs to connect to them simultaneously. For each IP address, three connection modes needs to be tried: UDP, TCP/IP and HTTP. Each mode has its own hand shaking protocol. And to give UDP and TCP/IP higher chance to get connected, TCP/IP mode should be started 500ms later after UDP mode is started, and HTTP mode should be started another 500ms later. When the first mode of whichever address gets connected successfully, the application should cancel all other connecting attempts.

From logic flow's point of view, the main flow will at first create N sub flows, according to the number of given IP addresses. Each sub flow creates its own 3 sub flows. The first sub flow connects to the given IP using UDP mode. The second sub flow sleep (logically) 500ms, then connects to server using TCP/IP mode. The third sub flow sleep 1000ms before connects to server using HTTP mode. Whichever sub flow connects to server successfully, it will notify the main flow which will stop the execution of all other sub flows.

1.5 Logic flow vs Threads/Coroutines

Writing code in logic flow looks very much like writing in synchronous mode. A task can be executed using synchronous calls, which block the thread. Although it makes writing code simpler, but we rarely use them in commercial programming. Coroutines can be used to switch among threads, however, that's still not enough. There are three major differences between logic flow and thread/coroutine:

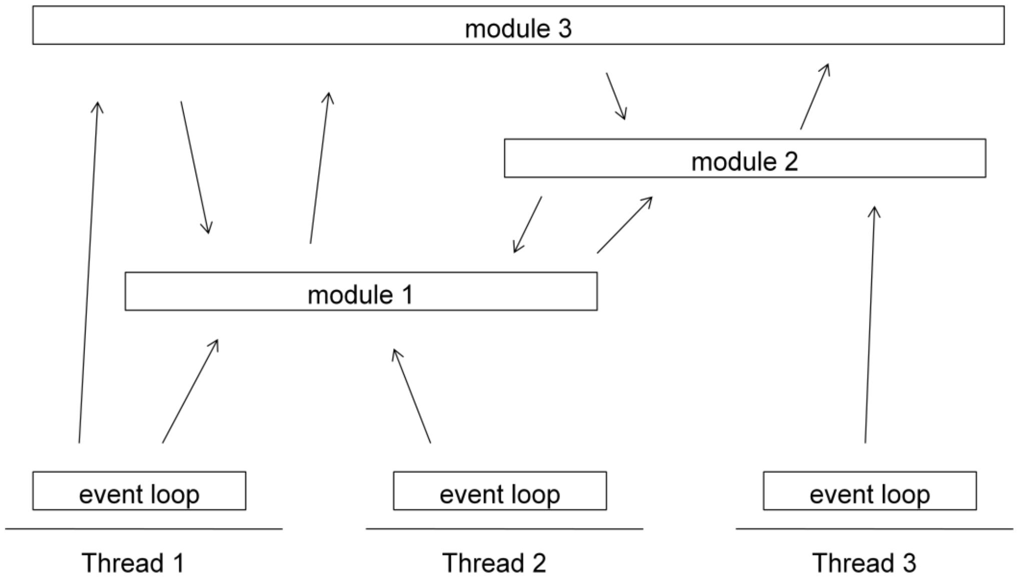
1. Logic flows run in a single thread while different coroutines run in separate threads. So when we write code using logic flow we don't need to worry about thread-safe protection, but in coroutine we must.

2. When a logic flow wants to terminate some other logic flows in the same thread, it can guarantee that those logic flows are in waiting state (because in one thread, only one logic flow is running at a time), so they can be terminated safely. But we cannot guarantee that in thread programming.

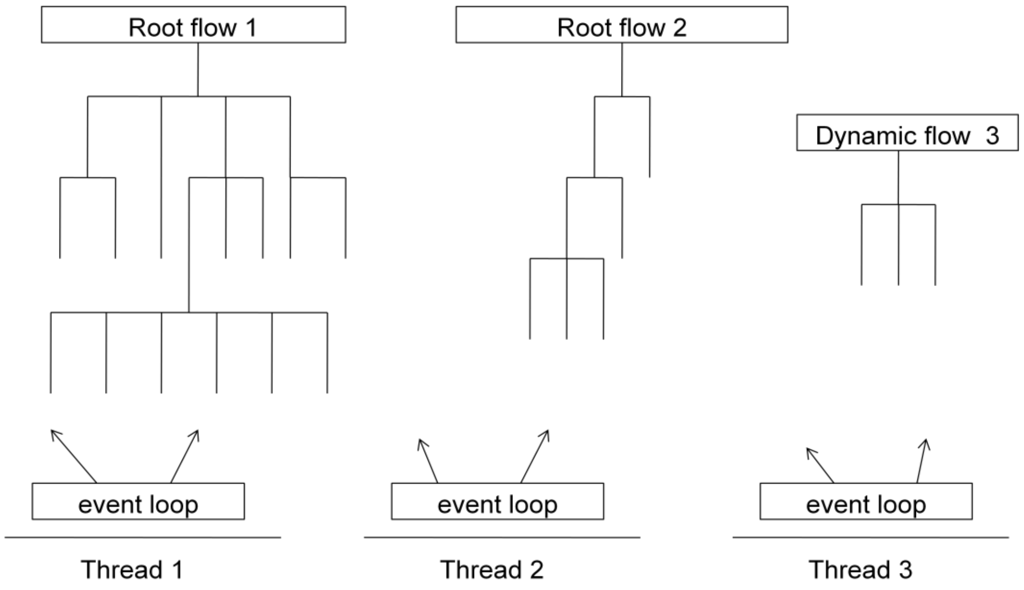
3. We can terminate a logic flow as long as it's in waiting state. But to terminate a thread which is waiting for a semaphore, we'd better to notify the semaphore to get the thread out of the waiting state, then add some code to let it stop the thread proactively. This could make the code rather complicate especially when the task is nested. E.g., in a thread funciton A() make calls of function B(), C() and D(), each of them make calls of some other functions. When we want to terminate this thread, we have no way to know which semaphore it is waiting for. And adding code to terminate the thread right after each wait() call? It's almost as complicate as programming in asynchronous mode.

Logic flows might run across threads. When one logic flow has finished running in one thread and is now in waiting state, another thread could generate an event that triggers the logic flow. Then the logic flow is moved to that thread and starts running there. If the logic flow is triggered by a thread while it is running in another thread, then risk-condition must be handled in the code of the flow.

The following figure shows how different modules work with event loops in asynchronous mode:



Here is how code is organized from flow logic's point of view:



2 Logic Flow Statements

2.1 Flow Language Extension

In this chapter we introduce a few statements and functions to support logic flow programming. Because these statements and functions are not language specific but can be commonly used in any language just like if, for, while, etc., so we call it Flow Language Extension (FLE). To make it brief, we only talk in C++ as examples.

2.2 FLOW_OBJECT, flow_init, flow_wait & flow_signal

FLOW_OBJECT is a data type that we can use to wait and signal in a flow, very much like how we use the semaphore in a thread. flow_init() must be called for initialization. After that, we can call flow_wait() or flow_signal() for waiting or signaling. Unlike wait() is to hold current physical thread until the semaphore is signaled, flow_wait() is to hold current logic flow until the specified FLOW_OBJECT is signaled. If the flow object is already signaled before flow_wait() is called then flow_wait() returns immediately. flow_signal() is to signal a FLOW_OBJECT so that the logic flow which is waiting for that object can continue. If the flow object is not in waiting status yet then the object will just carry a mark. Next time when someone calls flow_wait() to this object, it will return immediately. Once flow_wait() stops waiting and prepare to continue, that mark in the flow object will be cleared so the next flow_wait() will hold there if no signal has been fired.

```
FLOW_OBJECT g_sock_send_object;
flow void send_data_request(int s, char* data_buf, int sz)
{
    int sent_len = ::send(s, data_buf, sz, 0);
    while (sent_len < sz)
    {
        flow_wait(&g_sock_send_object, NULL);
        sent_len += ::send(s, data_buf + sent_len, sz - sent_len, 0);
    }
}
```

2.3 flow_fork

A logic flow can create another logic flow by using flow_fork.

```
flow_fork {
    http_get("google.com");
}
flow_fork {
    http_get("yahoo.com");
}
```

The sample above starts two logic flows which call http_get() logically at the same time. Both logic flows runs in parallel with the main flow. When http_get() returns, that sub logic flow terminates.

We can also write something like this:

```
char* server_ips[4];
for (int i = 0; i < 4; i++)
{
    flow_fork {
        http_get(server_ips[i]);
    }
}
```

Which starts 4 logic flows simultaneously.

Logic flows can be nested. A child logic flow can also create its own child logic flow. But when a parent logic flow terminates, all of its descendent logic flows will be terminated.

2.4 flow_try/flow_catch

flow_try encapsulate a block of code during which is being executed, whenever the flow object defined in flow_catch is signaled, it terminates immediately and the statements in that flow_catch will be executed.

```
flow CUser* getUserInfo(int user_id, FLOW_OBJECT* pFobj)
{
    CUser* pUser = NULL;
    flow_try
    {
        if (!(pUser = search_memory_cache(user_id)))
        {
            if (!(pUser = search_local_disk(user_id)))
            {
                if (!(pUser = download_user_info(user_id)))
                    return NULL;
                add_to_local_disk(pUser);
            }
            add_to_memory_cache(pUser);
        }
    }
    flow_catch(pFobj, NULL)
    {
    }
    return pUser;
}
```

One flow_try can have one or more flow_catches. The signals of those flow objects don't have to be fired from within the flow_try block.

2.5 flow_new/flow_delete

Although flow_fork can generate new flows, however, when parent flow terminates, all its sub flows terminate. Sometimes, we want to create a flow that stays there no matter whether its parent flow lives or dies. That's why flow_new and flow_delete are needed. Below is a example that on server side the main flow is to listen at a port, whenever it detects a socket connects in, it create a sub flow to handle the data in that socket.

```
std::map<int, void*> flow_map;

flow_fork {
   CFlowSocket listen_s;
    listen_s.create();
    listen_s.listen(port);
    while (true)
    {
        int s = listen_s.accept();
        void* flow_ptr = flow_new {
            flow_map[s] = flow_ptr;
            handle_data_socket(s);
        }
    }
}
```

flow_new is to create a totally independent flow. This kind of flow either terminates by itself, or terminates when someone calls flow_delete(...). Below is another example that client has a few server addresses to connect to at the same time, whoever connects first, client will close all other connecting attempts and focus on that connection only.

```
flow int find_first_to_connect(char** ip_address_list, int cnt)
{
    void** flow_table = (void**)malloc(cnt * sizeof(void*));
    for (int i = 0; i < cnt; i++)
    {
        flow_table[i] = flow_new
        {
            connect_to_server(i, ip_address_list[i]);
        }
    }
    void* param;
    flow_wait(&connect_obj, &param);
    int connect_idx = (int)param;
    for (int i = 0; i < cnt; i++)
    {
        if (i != connect_idx)
            flow_delete(flow_table[i]);
    }
    free(flow_table);

    return connect_idx;
}
```

As you can see, with flow_new/flow_delete, we can manipulate with program flows just like we do to program data.

2.6 Flow and flow root functions

You may notice the "flow" modifier in front of the definition of function find_first_to_connect(). This is to tell the compiler that these two functions are flow functions. A flow function is a function that makes calls of flow_wait(), flow_fork, flow_try or other flow functions. flow_wait() is treated as a flow function but flow_signal() is not. Correspondingly, a function without a “flow” modifier is called non-flow function which must not make any flow function call. Generally, event loop should only make calls of flow_signal().

Although not recommended, but it's legal for a function with “flow” modifier but not to make any call to flow_wait() or other flow functions.

A function with “flow_root” modifier indicates that this function is the very start point of flow functions. A flow_root function can call flow functions. But a flow function cannot call flow_root functions. Usually, function main() should declare as “flow_root” if it wants to make any flow call inside. There's also other functions can make such declaration. If a program wants to run multiple threads, and in some thread it wants to make flow calls, then it needs to declare the thread function as “flow_root”.

Usually, a flow_root function initiates the root flow in the beginning, and runs the event loop till the end. It's OK for a thread to run event loop without initiating any flow. But it's not allowed for a thread to initiate the root flow using flow_fork without a event loop in the bottom. Because when there's no event loop, the function cannot hold the logic flow any more, the logic flow will quit unexpectedly.

Since it's the event loop that drives the logic flow moving forward, it needs to be executed without any interruption. So in a flow_root function, only flow_fork and flow_new is allowed before running of event loop, any call of flow_wait/flow_try or flow function must not be used.

2.7 CFlowObject

flow_init/flow_wait/flow_signal can be easily encapsulated in a c++ class.

```
class CFlowObject
{
public:
    CFlowObject()
    {
        flow_init(&m_obj);
    }

    virtual ~CFlowObject()
    {
        cancel_wait();
    }

    flow void wait(void** param = NULL)
    {
        flow_wait(&m_obj, param);
    }

    void signal(void* param = NULL)
    {
        flow_signal(&m_obj, param);
    }

    FLOW_OBJECT* getObj() { return &m_obj; }

protected:
    FLOW_OBJECT  m_obj;
};
```

2.8 time_wait and sample 1

In many scenarios we want to wait for a certain period of time before executing next step, in traditional programming, we need to register a timer callback and implement the next step in that callback. But now with flow extension, we can do that in an easier way. First we add a flow method in class CFlowObject:

```
flow void time_wait(DWORD interval)
{
    m_waiting_time = get_cur_tick() + interval;
    add_timer_to_map(m_waiting_time, this);
    flow_wait(&m_obj, NULL);
 }
```

Then, we can use it like this:

```
flow void heartbeat(int start, int count, int interval)
{
    CFlowObject fo;
    for (int i = 0; i < count; i++)
    {
        printf("%d\n", start + i);
        fo.time_wait(interval);
    }
}

flow_root int main(int argc, char** argv)
{
    flow_fork
    {
        heartbeat(0, 10, 1000);
        stop_event_loop();
    }
    flow_fork
    {
        heartbeat(10000, 5, 2000);
    }

    event_loop();
    return 0;
}
```

It will print result as below:

```
0
10000
1
10001
2
3
10002
4
5
10003
6
7
10004
8
9
10005
10
```

2.9 Sample 2

In this sample, we start two flows which keep printing numbers in different columns and in different intervals. We also randomly choose a time out period in the very beginning after which the two flows terminate at the same time.

```
flow void heartbeat(int col, int max, int interval)
{
    CFlowObject fo;
    for (int i = 0; i < max; i += interval)
    {
         for (int j = 0; j < col; j++)
             printf("\t");
        printf("%d\n", i);
        fo.time_wait(interval * 1000);
    }
}

flow_root int main(int argc, char** argv)
{
    VSOCK::startup();

    flow_fork
    {
        CFlowObject stop_obj;
        srand(get_cur_tick());
        int duration = rand();
        printf("timer is set to %dms\n", duration);
        start_timer(duration, stop_obj.getObj());

        flow_try
        {
            flow_fork
            {
                heartbeat(0, 1000, 2);
            }
            heartbeat(1, 1000, 1);
        }
        flow_catch(stop_obj.getObj(), NULL)
        {
            printf("program terminated\n");
            CFlowSocket::stop_event_loop();
        }
    }

    CFlowSocket::event_loop();
    return 0;
}
```

And here's the running result:

```
timer is set to 26015ms
0
        0
        1
2
        2
        3
4
        4
        5
6
        6
        7
```

```
8
        8
        9
10
        10
        11
12
        12
        13
14
        14
        15
16
        16
        17
18
        18
        19
20
        20
        21
22
        22
        23
24
        24
        25
program terminated
```

2.10 Sample 3

This sample dynamically create four flows which keep printing numbers in different columns and in different intervals. In the mean time, for each flow we start, we randomly choose a time out value as the flow's duration. When time expires, it will signal a designated FLOW_OBJECT then we will kill the correspondent flow.

```
flow void start_timer(int interval, FLOW_OBJECT* pObj, void* param)
{
    flow_fork
    {
        FLOW_OBJECT* pObj2 = pObj;
        void* param2 = param;

        CFlowObject fo;
        fo.time_wait(interval);
        flow_signal(pObj2, param2);
    }
}

flow void heartbeat(int col, int max, int interval)
{
    CFlowObject fo;
    for (int i = 0; i < max; i += interval)
    {
        for (int j = 0; j < col; j++)
            printf("\t");
        printf("%d\n", i);
        fo.time_wait(interval * 1000);
    }
}

flow_root int main(int argc, char** argv)
{
    VSOCK::startup();

    flow_fork
    {
        CFlowObject stop_obj;
        srand(get_cur_tick());

        void* flow_table[4];
        for (int i = 0; i < 4; i++)
        {
            int interval = rand() % 4 + 1;
            int duration = rand();
            printf("start flow %d, interval=%d, duration=%d\n", i, interval,
duration);
            flow_table[i] = flow_new
            {
                heartbeat(i, 1000, interval);
            }
            start_timer(duration, stop_obj.getObj(), (void*)i);
        }

        for (int i = 0; i < 4; i++)
        {
            void* param;
            stop_obj.wait(&param);
            printf("flow %d terminated\n", (int)param);
            flow_delete(flow_table[(int)param]);
        }
        CFlowSocket::stop_event_loop();
    }

    CFlowSocket::event_loop();
    return 0;
}
```

Here's the running result:

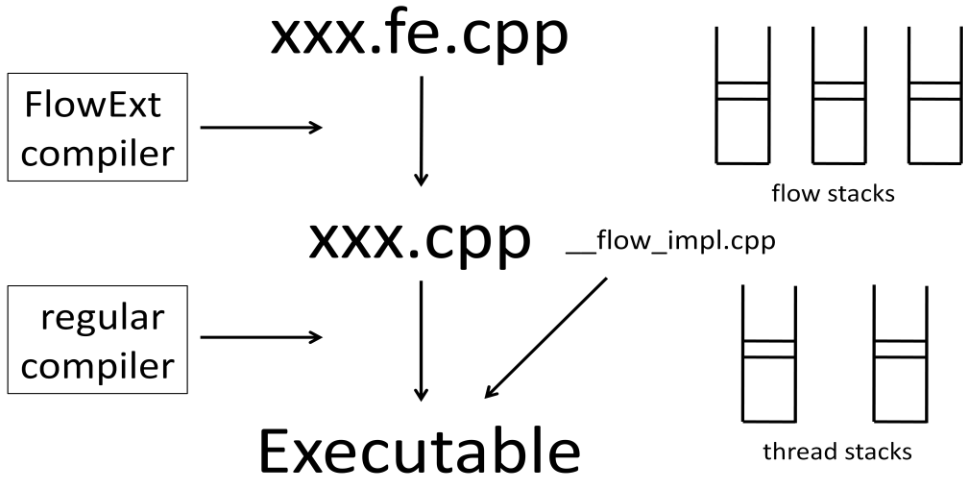
```
start flow 0, interval=1, duration=9643
0
start flow 1, interval=2, duration=8559
      0
start flow 2, interval=4, duration=13352
             0
start flow 3, interval=3, duration=19996
                  0
1
      2
2
                  3
3
             4
      4
4
5
                  6
      6
6
```

```
7
             8
      8
8
flow 1 terminated
                  9
9
flow 0 terminated
             12
                  12
flow 2 terminated
                  15
                  18
flow 3 terminated
```

3 Implementation

3.1 FLE pre-compiler

There might be other ways to implement the FLE statements, but for C/C++, my approach is to implement a compiler which will translate the code using FLE into standard C/C++. Then we can use regular C/C++ compiler to compile it into executables.



The sectors below describe how to compile FLE code into C/C++ code.

3.2 flow function transformation

First, let's study a simple flow function:

```
flow int f1(int param1, char* param2)
{
    FLOW_OBJECT fobj;
    flow_wait(&fobj, NULL);
    return 0;
}
```

In logic flow's definition, the program should hold at flow_wait() if fobj is not signaled. But we don't want the physical execution pointer to hold, we want it to execute other flows during waiting at this flow. So what we can do is we register a callback in flow_wait(), then return the function. When the FLOW_OBJECT is signaled, the callback is called and we need to work out a way to let f1() be re-entered again and go directly below flow_wait() and start to execute from there.

To achieve this, to let a function be re-entered with all the function parameters and local variables keep unchanged, we need to keep them in some place other than the physical function stack, but in a flow stack.

3.3 flow stack

Since each flow runs independently, and each flow works in a single-threaded way, so we'd better give each flow a separate stack. The stack is allocated when the flow is started. Since an application may have much more flows than threads, we may want to let user to decide the stack size so that on one side the stack should be big enough for the flow, and on the other side, not too many stack space are wasted.

When a flow function is called, the caller first reserves a space in the stack to store all function parameters and caller information. If the flow function returns a value, it needs to be added into the struct as well. Then in the callee side, as soon as the function is being entered, it expands the stack space to store all local variables in that function. So for the flow function mentioned above, the data structure for f1() should be defined like this:

```
struct __flow_func_def_f1
{
    FLOW_FUNC caller_func;
    void* caller_data;
    int param1;
    char* param2;
    int ret;
};

struct __flow_func_impl_f1 : public struct __flow_func_d_f1
{
    FLOW_OBJECT fobj;
};
```

So, f1() will be transformed to:

```
flow void f1(void* __flow_param_data)
{
    __flow_func_expand_stack(__flow_param_data, sizeof(__flow_func_impl_f1));
    __flow_func_impl_f1* func_data = (__flow_func_impl_f1*)__flow_param_data;
    flow_wait(&func_data->fobj, NULL);
    func_data->ret = 0;
}
```

In the place where f1 is called, e.g.

```
f1(2, "abc");
```

It will be transformed to:

```
__flow_func_def_f1* f1_data_ptr = (__flow_func_def_f1*)__flow_func_enter(caller_func,
caller_data, sizeof(__flow_func_def_f1));
f1_data_ptr->param1 = 2;
f1_data_ptr->param2 = "abc";
f1(f1_data_ptr);
int ret = f1_data_ptr->ret;
__flow_func_leave(f1_data_ptr);
```

Here, __flow_func_enter is to allocate a data block in the flow stack for function f1(). And __flow_func_leave is to free that data block from the flow stack.

For reference variables, we can change its data type to a pointer of that type to make it able to reset a value while defined in a structure. So,

```
flow void f1(int& n)
{
    int i = n;
    int& m = i;
}
```

will transform to:

```
struct __flow_func_def_f1
{
    int* n;
};

struct __flow_func_impl_f1 : public __flow_func_def_f1
{
    int  i;
    int* m;
};

bool f1(void* __flow_param_data)
{
    __flow_func_expand_stack(__flow_param_data, sizeof(__flow_func_impl_f1));
    __flow_func_impl_f1* func_data = (__flow_func_impl_f1*)__flow_param_data;
    func_data->i = *(func_data->n);
    func_data->m = &(func_data->i);
}
```

3.4 flow_wait

OK, now we want flow_wait to register a callback in the FLOW_OBJECT, and when it's signaled, f1() will be called and it starts to execute from below flow_wait(). So we can continue transforming f1() into this:

```
bool f1(void* __flow_param_data, unsigned int func_signal)
{
    __flow_func_impl_f1* func_data = (__flow_func_impl_f1*)__flow_param_data;
    if (func_signal <= 0)
    {
        __flow_func_expand_stack(__flow_param_data, sizeof(__flow_func_impl_f1));
        if (!__flow_wait(&func_data->fobj, f1, func_data, 1, NULL))
            return false;
    }
    if (func_signal <= 1)
        func_data->ret = 0;

    if (func_signal != 0)
        func_data->caller_func(func_data->caller_data, func_data->caller_signal);
    return func_signal == 0;
}
```

When a flow function is called, the func_signal will always be passed as 0.

__flow_wait is a function that when the FLOW_OBJECT is already signaled , it returns true immediately. Otherwise, it stores the caller function f1() and func_data as well as a number into the FLOW_OBJECT. When this FLOW_OBJECT is signaled, it will call f1() with func_data and the registered number as parameters.

So in the case above, if fobj is already signaled before __flow_wait() is called, __flow_wait() will return true. func_signal remains as 0. Then ret will be set to 0, and the function will return true.

If fobj is not signaled before __flow_wait() is called, __flow_wait() will return false, then f1() will also return false. Later when fobj is signaled, f1() will be called again with func_signal set to 1. Then ret will be set to 0. f1's caller function will be called with registered signal. Then f1 returns false.

Actually all flow functions will be transformed like this. It has two parameters. One points to the data block of the function, the other is a number indicating from where the function is re-entered. func_signal will always be set to 0 when the function is first called. When there's flow_wait or some other flow function to be called, it will pass a higher func_signal to that function so that when the waiting object is signaled, the function will be re-entered with the higher signal so it can start from the correct place.

The transformed function always return a boolean, true means the function hasn't encountered any obstacle, every code needs to execute has been executed. false means the function is not done yet, a callback with the registered signal number is expected.

Now, let's make f1() a little bit complicated:

```
flow int f1(int param1, char* param2)
{
    a1();

    FLOW_OBJECT fobj1;
    flow_wait(&fobj1, NULL);

    b1();
    return f2(param1) + 1;
}
```

In this case, a1() and b1() are non-flow functions, while f2() is a flow function. It will be transformed as below:

```
bool f1(void* __flow_param_data, unsigned int func_signal)
{
    __flow_func_impl_f1* func_data = (__flow_func_impl_f1*)__flow_param_data;
    if (func_signal <= 0)
    {
        __flow_func_expand_stack(__flow_param_data, sizeof(__flow_func_impl_f1));

        a1();

        if (!__flow_wait(&func_data->fobj1, f1, func_data, 1, NULL))
            return false;
    }

    if (func_signal <= 1)
    {
        b1();

        func_data->f2_data_ptr = (__flow_func_def_f2*)__flow_func_enter(f1, func_data,
2, sizeof(__flow_func_def_f2));
        func_data->f2_data_ptr->param1 = func_data->param1;
        if (!f2(func_data->f2_data_ptr, 0))
            return false;
    }

    func_data->ret = func_data->f2_data_ptr->ret + 1;
    __flow_func_leave(func_data->f2_data_ptr);

    if (func_signal != 0)
        func_data->caller_func(func_data->caller_data, func_data->caller_signal);
    return func_signal == 0;
}
```

Based on this algorithm, let's start to transform every basic type of commonly used grammar.

3.5 flow expressions

A flow expression is an expression that includes at least one flow function call. For complex expressions, we need to split them into several steps in a specified order. E.g.,

```
c[f3()] = a1(f1()) + f2();
```

Here, f1(), f2() and f3() are flow functions, a1() is non-flow function. We need to split this expression to a few expressions:

```
temp1 = f1();
temp2 = a1(temp1);
temp3 = f2();
temp4 = f3();
c[temp4] = temp2 + temp3;
```

which will be transformed to:

```
if (func_signal <= 0)
{
    func_data->f1_data_ptr = (__flow_func_def_f1*)__flow_func_enter(f0, func_data, 1, 
sizeof(__flow_func_def_f1));
    if (!f1(func_data->f1_data_ptr, 0))
        return false;
}
if (func_signal <= 1)
{
    func_data->temp1 = func_data->f1_data_ptr->ret;
    __flow_func_leave(func_data->f1_data_ptr);
    func_data->temp2 = a1(func_data->temp1);
    func_data->f2_data_ptr = (__flow_func_def_f2*)__flow_func_enter(f0, func_data, 2, 
sizeof(__flow_func_def_f2));
    if (!f2(func_data->f2_data_ptr, 0))
        return false;
}
if (func_signal <= 2)
{
    func_data->temp3 = func_data->f2_data_ptr->ret;
    __flow_func_leave(func_data->f2_data_ptr);
    func_data->f3_data_ptr = (__flow_func_def_f3*)__flow_func_enter(f0, func_data, 3, 
sizeof(__flow_func_def_f3));
    if (!f3(func_data->f3_data_ptr, 0))
```

```
        return false;
}
if (func_signal <= 3)
{
    func_data->temp4 = func_data->f3_data_ptr->ret;
    __flow_func_leave(func_data->f3_data_ptr);
    c[func_data->temp4] = func_data->temp2 + func_data->temp3;
}
```

The more complicated the transformed code is, the much easier we will write the code using flow extension.

3.6 compound block

If a compound block contains any flow call, then it needs to be transformed. Otherwise, we can leave it untouched. All its local variables will be allocated in the physical stack and freed immediately when the compound block exits. Of course, if there's any expression in the compound block refers to a variable outside the block, we still need to transform it to refer to the variable in the function's data block.

```
int n;

while (a < b)
{
    char buf[100];
    for (int i = 0; i < 99; i++)
        a1(buf, i, i + 1, n);
}
```

will be transformed to:

```
while (a < b)
{
    char buf[100];
    for (int i = 0; i < 99; i++)
        a1(buf, i, i + 1, func_data->n);
}
```

3.7 if/else

There's a typical case, from my own experience, that given a user id we want to show the user's name and icon in a mobile app. We store these information in 3 places: memory cache, local disk and on the server. We first look it up in the memory cache, if not found, we search it in the local disk, which we want to do it asynchronously because it may take some time. If still not found, then we need to issue a http request to the server to download the information. When we get the info from server, we need to add it to local disk as well as the memory cache, then displaying them on the GUI. There are many callbacks will be involved. We need to remember which callback is for which user id so that we can cancel it when user switches the page.

But with Flow programming, things are pretty straight forward. We can just write:

```
CUser* search_memory_cache(int user_id);
flow CUser* search_local_disk(int user_id);
flow CUser* download_user_info(int user_id);
void add_to_memory_cache(CUser* pUser);
void add_to_local_disk(CUser* pUser);

flow CUser* getUserInfo(int user_id, FLOW_OBJECT* pFobj)
{
    CUser* pUser = NULL;
    flow_try
    {
        if (!(pUser = search_memory_cache(user_id)))
        {
            if (!(pUser = search_local_disk(user_id)))
            {
                if (!(pUser = download_user_info(user_id)))
                    return NULL;
                add_to_local_disk(pUser);
            }
            add_to_memory_cache(pUser);
        }
    }
    flow_catch(pFobj, NULL)
    {
    }
    return pUser;
}
```

Here, the code inside the flow_try block will be transformed to:

```
if (__flow_signal <= 0)
{
    func_data->pUser = 0;
}
if (__flow_signal <= 0 && (!(func_data->pUser = search_memory_cache(func_data-
>user_id))) || __flow_signal > 0 && __flow_signal <= 2)
{
    if (__flow_signal <= 0)
    {
        func_data->__flow_temp_0 =
(__flow_func_def_search_local_disk*)__flow_func_enter(getUserInfo, func_data, 1,
sizeof(__flow_func_def_search_local_disk));
        func_data->__flow_temp_0->user_id = func_data->user_id;
        if (!search_local_disk(func_data->__flow_temp_0, 0))
            return false;
    }
    if (__flow_signal <= 1)
    {
        func_data->__flow_temp_1 = func_data->__flow_temp_0->ret;
        __flow_func_leave(func_data->__flow_temp_0);
    }
    if (__flow_signal <= 1 && (!(func_data->pUser = func_data->__flow_temp_1)) ||
__flow_signal == 2)
    {
        if (__flow_signal <= 1)
        {
            func_data->__flow_temp_2 =
(__flow_func_def_download_user_info*)__flow_func_enter(getUserInfo, func_data, 2,
sizeof(__flow_func_def_download_user_info));
            func_data->__flow_temp_2->user_id = func_data->user_id;
            if (!download_user_info(0, func_data->__flow_temp_2))
                return false;
        }
        if (__flow_signal <= 2)
        {
            func_data->__flow_temp_3 = func_data->__flow_temp_2->ret;
            __flow_func_leave(func_data->__flow_temp_2);
        }
        if (__flow_signal <= 2 && (!(func_data->pUser = func_data->__flow_temp_3)) ||
__flow_signal == 2)
            return __flow_signal == 0;
        if (__flow_signal <= 2)
        {
            add_to_local_disk(func_data->pUser);
        }
    }
    if (__flow_signal <= 2)
    {
        add_to_memory_cache(func_data->pUser);
    }
}
```

3.8 while

```
while (a < b)
{
    ...
}
```

will transform to:

```
int bMode = signal <= 0 ? 0 : 1;
while (signal <= 3)
{
    if (bMode > 1 && signal > 1)
       signal = 1;
    if (signal <= 1)
    {
        if (!(func_data->a < func_data->b))
            break;
    }
    bMode = 2;

    ...
}
```

3.9 do

```
do
{
    ...
} while (a < b);
```

will transform to:

```
int bMode = signal <= 0 ? 0 : 1;
while (signal <= 3)
{
    if (bMode > 1 && signal > 1)
       signal = 1;
    if (signal <= 1)
    {
        if (bMode != 0 && !( func_data->a < func_data->b))
            break;
    }
    bMode = 2;

    ...
}
```

3.10 for

```
for (i = 0; i < n; i++)
{
    ...
}
```

will transform to:

```
if (signal <= 0)
{
    func_data->i = 0;
}
int bMode = signal <= 0 ? 0 : 1;
while (signal <= 3)
{
    if (bMode > 1 && signal > 1)
        signal = 1;
    if (signal <= 1)
    {
        if (!(func_data->a < func_data->b))
            break;
        if (bMode != 0)
            func_data->i++;
    }
    bMode = 2;

    ...
}
```

3.11 switch

```
switch (a[n++])
{
    ...
}
```

will transform to:

```
if (signal <= 0)
{
    func_data->temp1 = func_data->a[func_data->n++];
}
if (signal <= 10)
{
    switch (func_data->temp1)
    {
        ...
    }
}
```

3.12 flow_fork

In flow_fork, we want to create a new flow.

```
flow void f0()
```

```
{
    int n = 5;

    flow_fork
    {
        f1();
        if (n > 3)
            return;
        f2();
    }

    a2();
}
```

will transform to:

```
bool f0_sub1(void* __flow_param_data, unsigned int signal)
{
    __flow_func_impl_f0_sub1* func_data =
(__flow_func_impl_f0_sub1*)__flow_param_data;
    if (signal <= 0)
    {
        __flow_func_expand_stack(__flow_param_data,  sizeof(__flow_func_impl_f0_sub1));
        func_data->f1_data_ptr = (__flow_func_def_f1*)__flow_func_enter(func_data-
>this_flow, f0_sub1, func_data, 1, sizeof(__flow_func_def_f1));
        if (!f1(func_data->f1_data_ptr, 0))
            return false;
    }
    if (signal <= 1)
    {
        __flow_func_leave(func_data->f1_data_ptr);
        if (func_data->parent->n < 3)
        {
            __flow_end(func_data->this_flow);
            return false;
        }
        func_data->f2_data_ptr = (__flow_func_def_f2*)__flow_func_enter(func_data-
>this_flow, f0_sub1, func_data, 2, sizeof(__flow_func_def_f2));
        if (!f2(func_data->f2_data_ptr, 0))
            return false;
    }
    __flow_func_leave(func_data->f2_data_ptr);
    __flow_end(func_data->this_flow);
    return false;
}

bool f0(void* __flow_param_data, unsigned int signal)
{
    __flow_func_impl_f0* func_data = (__flow_func_impl_f0*)__flow_param_data;
    if (signal <= 0)
    {
        __flow_func_expand_stack(__flow_param_data, sizeof(__flow_func_impl_f0));
        func_data->n = 5;
        func_data->temp_flow1 = __flow_start(func_data->this_flow);
        func_data->f0_sub1_data_ptr =  (__flow_func_def_f0_sub1
*)__flow_func_enter(func_data->temp_flow1, NULL, NULL, 0,
sizeof(__flow_func_def_f0_sub1));
        func_data->f0_sub1_data_ptr->parent = func_data;
        f0_sub1(func_data->f0_sub1_data_ptr, 0);

        a2();
    }
    return signal == 0;
}
```

__flow_start() is to allocate a new flow object under the parent flow.
In the data block of each function, we need to add a member "this_flow" to store the pointer to the flow.
When calling __flow_func_enter(), we need to pass this_flow member so that every function knows in which flow it is running.
The whole flow_fork block is moved to a new function f0_sub1.
When f0 calls f0_sub1, it passes NULL as caller_func and caller_data so that f0_sub1 won't call back when finished.
We need to add a member called "parent" in f0_sub1's data block and set it to f0's func_data so that expressions in flow_fork can make use of variables defined in f0().
At last, before each "return" statement in f0_sub1, we need to add __flow_end() so that it will terminate this new flow nice and clean.

3.13 what need to be done when a flow terminates

1. cancel all the waiting of FLOW_OBJECTS that initiate from this flow.
2. terminates all flows that previously forked from this flow.
3. clear the stack and free the buffer allocated for this flow.

Actually, we also need to call the destructor of all C++ objects created in this flow, but we'll cover that later.

3.14 flow_new

The transformation of flow_new is almost the same as that of flow_fork. The only difference is when calling __flow_start(), the parent flow passed as parameter is set as NULL so that the child flow will not be affected when the parent flow terminates.

```
flow void f0()
{
    void* flow_ptr;
    flow_ptr = flow_new
    {
        f1();
    }
}
```

will transform to:

```
bool f0_sub1(void* __flow_param_data, unsigned int signal)
{
    __flow_func_impl_f0_sub1* func_data =
(__flow_func_impl_f0_sub1*)__flow_param_data;
    if (signal <= 0)
    {
        __flow_func_expand_stack(__flow_param_data,  sizeof(__flow_func_impl_f0_sub1));
        func_data->f1_data_ptr = (__flow_func_def_f1*)__flow_func_enter(func_data-
>this_flow, f0_sub1, func_data, 1, sizeof(__flow_func_def_f1));
        if (!f1(func_data->f1_data_ptr, 0))
            return false;
    }
    if (signal <= 1)
    {
        __flow_func_leave(func_data->f1_data_ptr);
    }
    __flow_end(func_data->this_flow);
    return false;
}

bool f0(void* __flow_param_data, unsigned int signal)
{
    __flow_func_impl_f0* func_data = (__flow_func_impl_f0*)__flow_param_data;
    if (signal <= 0)
    {
        __flow_func_expand_stack(__flow_param_data, sizeof(__flow_func_impl_f0));
        func_data->temp_flow1 = __flow_start(NULL);
        func_data->flow_ptr = func_data->temp_flow1;
        func_data->f0_sub1_data_ptr =
(__flow_func_def_f0_sub1*)__flow_func_enter(func_data->temp_flow1, NULL, NULL, 0,
sizeof(__flow_func_def_f0_sub1));
        func_data->f0_sub1_data_ptr->parent = func_data;
        f0_sub1(func_data->f0_sub1_data_ptr, 0);
    }

    return signal == 0;
}
```

3.15 flow_try/flow_catch

flow_try doesn't fork a new flow, but we still use __flow_start() create a flow to run the flow_try block. Because by doing this way, when a signal is caught by flow_catch, we can terminates what's running in the flow_try block with ease.

```
flow_try
{
    ...
}
flow_catch(&fobj1, NULL)
{
    ...
}
flow_catch(&fobj2, NULL)
{
    ...
}
```

will transform to:

```
if (signal <= 0)
{
    func_data->sub_flow = __flow_start(func_data->this_flow);
    func_data->pTempWait1 = &func_data->fobj1;
    func_data->pTempWait2 = &func_data->fobj2;
    if (__flow_wait(func_data->pTempWait1, f0, func_data, 1, NULL))
        signal = 1;
    else if (__flow_wait(func_data->pTempWait2, f0, func_data, 2, NULL))
        signal = 2;
    else
    {
        __flow_func_def_f0_sub1 * temp1 = __flow_func_enter(sub_flow, caller_func,
caller_data, 3, sizeof(__flow_func_def_f0_sub1));
        temp1->parent = func_data;
        if (!f0_sub1(temp1, 0))
            return false;
    }
}
if (signal == 1)
{
    __flow_end(func_data->sub_flow);
    flow_cancel_wait(func_data->pTempWait2);
    ...
}
else if (signal == 2)
{
    __flow_end(func_data->sub_flow);
    flow_cancel_wait(func_data->pTempWait1);
    ...
}
else if (signal == 0 || signal == 3)
{
    __flow_end(func_data->sub_flow);
    flow_cancel_wait(func_data->pTempWait1);
    flow_cancel_wait(func_data ->pTempWait2);
}
```

3.16 flow_root functions

flow_root functions are kind of special in that its function header cannot be transformed. But since it can only includes flow_fork and flow_new, so the function never need to re-enter. All we need to do is to create a flow for the main flow and terminate it before the function exits. We also need to call __flow_func_enter for the flow_root function itself so that we can store all local variables as well as function parameters into the data block.

```
flow_root int main(int argc, char** argv)
{
    flow_fork {
        http_get(argv[1]);
    }
    event_loop();
    return 0;
}
```

will transform to:

```
int main(int argc, char** argv)
{
    void* temp_flow1 = __flow_start(NULL);
    __flow_func_impl_main * main_data_ptr =  (__flow_func_impl_main
*)__flow_func_enter(temp_flow1, NULL, NULL, 0, sizeof(__flow_func_impl_main));
    main_data_ptr->argc = argc;
    main_data_ptr->argv = argv;

    void* temp_flow2 = __flow_start(temp_flow1);
    __flow_func_def_main_sub1* main_sub1_data_ptr =  (__flow_func_def_main_sub1
*)__flow_func_enter(temp_flow2, NULL, NULL, 0, sizeof(__flow_func_def_main_sub1));
    main_sub1_data_ptr->parent = main_data_ptr;
    main_sub1(main_sub1_data_ptr, 0);

    event_loop();
    __flow_end(main_data_ptr);
    return 0;
}
```

3.17 flow_impl.h

```
#ifndef _FLOW_IMPL_H_
#define _FLOW_IMPL_H_

extern "C" {

typedef bool (*FLOW_FUNC)(unsigned, void*);

struct FLOW_OBJECT {
 FLOW_FUNC   callback_func;
 void*       callback_data;
    unsigned    callback_signal;
 void*       param;
 int         value;
 void*       waiting_flow;
 FLOW_OBJECT*    prev_waiting_obj;
 FLOW_OBJECT*    next_waiting_obj;
};

void flow_init(FLOW_OBJECT* pObj);
void flow_signal(FLOW_OBJECT* pObj, void* param);
void flow_delete(void* pFlow);
void flow_cancel_wait(FLOW_OBJECT* pObj);

struct __FLOW_FUNC_BLOCK {
    void*         this_flow;
 FLOW_FUNC    this_func;
    FLOW_FUNC    caller_func;
    void*        caller_data;
    unsigned     caller_signal;
 unsigned     delete_counter;
};

bool __flow_wait(FLOW_OBJECT* pObj, FLOW_FUNC callback_func, unsigned signal, void*
callback_data, void** param);
void* __flow_start(void* pFlow, unsigned long stack_size); // 0 means using default
value
void __flow_end(void* pFlow);
void* __flow_func_enter(void* pFlow, FLOW_FUNC caller_func, void* caller_data,
unsigned caller_signal, long block_size);
void __flow_func_expand_stack(void* pFlow, void* caller_data, long block_size);
void __flow_func_leave(void* pFlow, void* pFuncBlock);

}

#endif // _FLOW_IMPL_H_
```

3.18 flow_impl.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "flow_impl.h"

#define __FLOW_CALL_SIGNAL_DELETE_OBJECT  0x80000000
#define FLOW_STACK_DEFAULT_SIZE      512000

struct __FlowBlock
{
 __FlowBlock*           parent_flow;
 __FlowBlock*           next_flow;
 __FlowBlock*           prev_flow;
 __FlowBlock*           sub_flows;
 __FLOW_FUNC_BLOCK*     cur_stack;
 FLOW_OBJECT*           first_waiting_object;
 unsigned long          stack_allocated;
 unsigned long          stack_used;
 char*                  stack;
};

void __flow_assert(bool b)
{
 assert(b);
}

void flow_init(FLOW_OBJECT* pObj)
{
     memset(pObj, 0, sizeof(FLOW_OBJECT));
}

void flow_signal(FLOW_OBJECT* pObj, void* param)
{
 FLOW_FUNC callback_func = pObj->callback_func;
 if (callback_func)
 {
     flow_cancel_wait(pObj);
     if (pObj->param)
         *((void**)pObj->param) = param;
     callback_func(pObj->callback_signal, pObj->callback_data);
 }
 else
 {
     pObj->value = 1;
     pObj->param = param;
 }
}

bool __flow_wait(FLOW_OBJECT* pObj, FLOW_FUNC callback_func, unsigned signal, void*
callback_data, void** param)
{
 __FLOW_FUNC_BLOCK* pCallingBlock = (__FLOW_FUNC_BLOCK*)callback_data;

     if (pObj->value)
     {
     pObj->value = 0;
     if (param)
         *param = pObj->param;
     return true;
     }

 __FlowBlock* pCallingFlow = (__FlowBlock*)pCallingBlock->this_flow;
 pObj->next_waiting_obj = pCallingFlow->first_waiting_object;
 if (pCallingFlow->first_waiting_object)
     pCallingFlow->first_waiting_object->prev_waiting_obj = pObj;
     pCallingFlow->first_waiting_object = pObj;

 pObj->waiting_flow = pCallingFlow;
 __flow_assert(pObj->waiting_flow != NULL);
 pObj->callback_func = callback_func;
 pObj->callback_data = callback_data;
 pObj->callback_signal = signal;
 pObj->param = (void*)param;
```

```
 return false;
}

void flow_cancel_wait(FLOW_OBJECT* pObj)
{
 (unsigned long)pObj, (unsigned long)pObj->callback_func, (unsigned long)pObj-
>waiting_flow);
 if (!pObj->callback_func)
     return;

 __FlowBlock* pWaitingFlow = (__FlowBlock*)pObj->waiting_flow;
 __flow_assert(pWaitingFlow != NULL);
 if (pWaitingFlow->first_waiting_object == pObj)
 {
     __flow_assert(pObj->prev_waiting_obj == NULL);
     pWaitingFlow->first_waiting_object = pObj->next_waiting_obj;
 }
 else if (pObj->prev_waiting_obj)
     pObj->prev_waiting_obj->next_waiting_obj = pObj->next_waiting_obj;
 if (pObj->next_waiting_obj)
     pObj->next_waiting_obj->prev_waiting_obj = pObj->prev_waiting_obj;

 pObj->prev_waiting_obj = pObj->next_waiting_obj = NULL;
 pObj->value = 0;
 pObj->waiting_flow = NULL;
 pObj->callback_func = NULL;
}

void* __flow_start(void* parent_flow, unsigned long stack_size)
{
 __FlowBlock* pParentFlow = (__FlowBlock*)parent_flow;
 if (stack_size == 0)
     stack_size = FLOW_STACK_DEFAULT_SIZE;
 __FlowBlock* pFlow = (__FlowBlock*)malloc(sizeof(__FlowBlock) + stack_size);
 memset(pFlow, 0, sizeof(__FlowBlock));
 pFlow->parent_flow = pParentFlow;
 if (pParentFlow)
 {
     pFlow->next_flow = pParentFlow->sub_flows;
     if (pParentFlow->sub_flows)
         pParentFlow->sub_flows->prev_flow = pFlow;
     pParentFlow->sub_flows = pFlow;
 }
 pFlow->stack = (char*)pFlow + sizeof(__FlowBlock);
 pFlow->stack_allocated = stack_size;
 return pFlow;
}

void* __flow_func_enter(void* flow, FLOW_FUNC caller_func, void* caller_data,
unsigned caller_signal, long block_size)
{
 __FlowBlock* pFlow = (__FlowBlock*)flow;

 __flow_assert(pFlow->cur_stack == NULL || caller_data == pFlow->cur_stack);
 __flow_assert(pFlow->stack_used + block_size < pFlow->stack_allocated);
 __FLOW_FUNC_BLOCK* pFuncBlock = (__FLOW_FUNC_BLOCK*)(pFlow->stack + pFlow-
>stack_used);
 (long)caller_data, (long)pFuncBlock);
 pFuncBlock->this_flow = pFlow;
 pFuncBlock->this_func = NULL;
 pFuncBlock->caller_func = caller_func;
 pFuncBlock->caller_data = caller_data;
 pFuncBlock->caller_signal = caller_signal;
 pFuncBlock->delete_counter = 0;
 pFlow->cur_stack = pFuncBlock;
 pFlow->stack_used += block_size;

 return pFuncBlock;
}

void __flow_func_expand_stack(void* flow, void* caller_data, long block_size)
{
 __FlowBlock* pFlow = (__FlowBlock*)flow;

 __flow_assert(caller_data == pFlow->cur_stack);
 __flow_assert(block_size >= (char*)pFlow->cur_stack - (char*)caller_data);
 block_size -= (char*)pFlow->cur_stack - (char*)caller_data;
 __flow_assert(pFlow->stack_used + block_size < pFlow->stack_allocated);
 pFlow->stack_used += block_size;
}

void __flow_func_leave(void* flow, void* func_data)
{
 __FlowBlock* pFlow = (__FlowBlock*)flow;
 __FLOW_FUNC_BLOCK* pFuncBlock = (__FLOW_FUNC_BLOCK*)func_data;

 __flow_assert(pFuncBlock == pFlow->cur_stack);

 pFlow->stack_used = (char*)pFuncBlock - pFlow->stack;
 pFlow->cur_stack = (__FLOW_FUNC_BLOCK*)pFuncBlock->caller_data;
}

// delete the whole stack, all sub_flows, remove itself from parent flow
void __flow_end(void* flow)
{
 __FlowBlock* pFlow = (__FlowBlock*)flow;

 // cancel all waiting objects
 while (pFlow->first_waiting_object)
 {
     flow_cancel_wait(pFlow->first_waiting_object);
 }

 while (pFlow->cur_stack)
 {
     __FLOW_FUNC_BLOCK* pPrevStack = (__FLOW_FUNC_BLOCK*)pFlow->cur_stack-
>caller_data;
     if (pFlow->cur_stack->this_func)
         pFlow->cur_stack->this_func(__FLOW_CALL_SIGNAL_DELETE_OBJECT, pFlow-
>cur_stack);
     pFlow->cur_stack = pPrevStack;
 }

 while (pFlow->sub_flows)
 {
     __flow_end(pFlow->sub_flows);
 }

 __FlowBlock* pParent = pFlow->parent_flow;
 if (pParent)
 {
     if (pParent->sub_flows == pFlow)
     {
         pParent->sub_flows = pFlow->next_flow;
         if (pParent->sub_flows)
             pParent->sub_flows->prev_flow = NULL;
```

```
        }
        else
        {
            pFlow->prev_flow->next_flow = pFlow->next_flow;
            if (pFlow->next_flow)
                pFlow->next_flow->prev_flow = pFlow->prev_flow;
        }
    }

    free(pFlow);
}

void flow_delete(void* flow)
{
    __FlowBlock* pFlow = (__FlowBlock*)flow;

    __flow_assert(pFlow->parent_flow == NULL);
    __flow_end(pFlow);
}
```

4 Flow Extension Transformation for C++

4.1 constructor

Since we are moving all the variables in a flow function from the physical stack to the flow stack, for object variables, we cannot utilize the standard compiler to call their constructor but to do it by our own. So

```
...
A a(10);
...
```

will transform to:

```
func_data->a.A::A(10);
```

4.2 destructor

destructor is a bit of headache. It needs to be called whenever the program quit the compound where it is used. Say in a while compound there defined 3 objects. Then before every break and continue statement, we need to insert commands to call the destructor of these 3 objects.

While that's still doable, but considering when we terminate a flow, we need to destruct every object in every function in the flow stack. How do we know where in a function had the program ran into and which object needs to be destructed?

To make things worse, the objects defined in a function need to be destructed in the exact reverse order of how they were constructed.

To solve the problem, one solution is during transformation, we give each object defined in the function an index number, starting from 0. And in the data block of the function, we add a table to store the number of those objects that having been constructed. Since the number of objects defined in a function is finite, the table size is foreseeable for each function.

Every time when an object is constructed, we append the object's index number to the table. And every time when we exit a compound block, we call the destructor of those objects defined in that block and remove them from the table from back to front.

Since we cannot convert any class's destructor to a C function, we need to call it explicitly. We can put the code in the end of the function.

```
void f0()
{
    A a(10);
    while (a < 5)
    {
        B b;
        if (a < b)
            continue;
        f1();
        C c;
        if (b > c)
            break;
    }
}
```

will transform to:

```
bool f0(void* __flow_param_data, unsigned int signal)
{
    __flow_func_impl_f0* func_data = (__flow_func_impl_f0*)__flow_param_data;
    if (signal <= 0)
    {
        __flow_func_expand_stack(__flow_param_data, sizeof(__flow_func_impl_f1));
        func_data->a.A::A(10);
        func_data->__flow_obj_table[func_data->obj_counter++] = 0;
    }
    int bMode = signal <= 0 ? 0 : 1;
    while (signal <= 2)
    {
        if (bMode > 1 && signal > 1)
            signal = 1;
        if (signal <= 1)
        {
            if (!(a < 5))
                break;
        }
        bMode = 2;

        if (signal <= 1)
        {
            func_data->b.B::B();
            func_data->__flow_obj_table[func_data->obj_counter++] = 1;
            if (a < b)
            {
                f0(func_data, 0x80000001);
                continue;
            }
            func_data->f1_data_ptr =
(__flow_func_def_f1*)__flow_func_enter(func_data->this_flow, f0, func_data, 2,
sizeof(__flow_func_def_f1));
            if (!f1(func_data->f1_data_ptr, 0))
                return false;
        }
        if (signal <= 2)
        {
            __flow_func_leave(func_data->f1_data_ptr);
            func_data->c.C::C();
            func_data->__flow_obj_table[func_data->obj_counter++] = 2;
            if (b > c)
            {
                f0(func_data, 0x80000001);
                break;
            }
        }
    }
    if (signal <= 2)
    {
        f0(func_data, 0x80000000);
        return signal == 0;
    }
    while (0x80000000 + func_data->obj_counter > signal)
    {
        switch (func_data->__flow_obj_table[--func_table->obj_counter])
        {
        case 0:
            func_data->a.A::~A();
            break;
        case 1:
            func_data->b.B::~B();
            break;
        case 2:
            func_data->c.C::~C();
            break;
        }
    }
    return false;
}
```

We add a while block in the bottom of f0() so that when f0() is called with signal >= 0x80000000, it will call destructor and shrink the obj_table until it contains (signal - 0x80000000) items.

So before the break and continue statements in the first while, f0(func_data, 0x80000001) is inserted so that only 1 item (the object a in this case) will remain in the obj_table.

Before f0() returns, f0(func_data, 0x80000000) will be called to destruct all objects. When a flow terminates, every function in the flow stack will be called with signal set to 0x80000000 to destruct all objects as well.

4.3 flow class method

Just like flow functions, class method can also be defined as flow method. To transform flow methods, some extra work need to be done.

First, since only C function pointer can be stored in the flow stack, so we need to convert the class method to a C function. To achieve this, for each flow class method, we need to define a static method in the same class whose only purpose is to call the C++ method.

Then, when we transform the C++ method, wherever we call __flow_func_enter to pass down the caller function pointer, we pass the static method instead of the C++ method.

At last, in the place where this class method is called, besides all parameters are set into the data block, the pointer to the object also need to be pass into. So,

```
class CFlowObject
{
    ...
    flow void wait(void** param = NULL)
    {
        flow_wait(&m_obj, param);
    }
    ...
};
```

will transform to:

```
class CFlowObject
```

```
{
    ...
    static bool __flow_staic_wait(void* __flow_param_data, unsigned int signal)
    {
        __flow_func_def_wait* func_data = (__flow_func_def_wait*)__flow_param_data;
        func_data->__flow_caller_this->wait(func_data, signal);
    }

    bool wait(void* __flow_param_data, unsigned int signal)
    {
        __flow_func_impl_wait* func_data = (__flow_func_impl_wait*)__flow_param_data;
        if (signal <= 0)
        {
            __flow_func_expand_stack(__flow_param_data,
sizeof(__flow_func_impl_wait));
            if (!__flow_wait(&m_obj, CFlowObject::__flow_static_wait, 1, func_data,
func_data->param))
                return false;
        }
        if (signal <= 1)
        {
            if (signal > 0 && func_data->caller_func)
                func_data->caller_func(func_data->caller_data, func_data-
>caller_signal);
            return signal == 0;
        }
        return false;
    }
    ...
};
```

And on the caller side,

```
CFlowObject fo;
...
fo.wait();
```

will transform to:

```
func_data->temp0 = (CFlowObject::__flow_func_def_wait *)__flow_func_enter(func_data-
>this_flow, my_func, func_data, 1, sizeof(CFlowObject::__flow_func_def_wait));
func_data->temp0->__flow_caller_this = &fo;
func_data->temp0->param = 0;
if (!func_data->temp0->__flow_caller_this->wait(func_data->temp0, 0))
    return false;
```

5 Open Source Project

I created a project called FlowExt at github.com. fec/ sub directory stores source code that pre-compiles FLE code into standard C++ code. Run ./make.sh to compile it. Output file is "flowext".

Then copy "flowext" into samples/ sub folder. There are flow_helper.fe.cpp and three other .fe.cpp files which are exactly the same as the 3 samples showing in chapter 2.7, 2.8 and 2.9. For each .fe.cpp file. Run fec.sh to convert fe.cpp into standard cpp file:

```
./fec.sh xxx
```

which convert xxx.fe.cpp into xxx.cpp.

Then, for each sample file, you can compile and link and run using following command:

```
g++ -o test timer_test.cpp flow_helper.cpp flow_impl.cpp socklib.cpp
./test
```

Although the conversion logic is simple and direct, C++ language analyzing is a huge effort. So the pre-compiler flowext is kind of shaky because it is written from scratch. But it can successfully convert the provided sample files on ubuntu 14.04 with gcc 4.6.3 and Microsoft Visual Studio 2010 on Windows 8.