目錄

系统级应用开发指南

本系列文章通过对Mesos源码进行分析、借鉴，试图梳理出系统级开发时通用的一些开发技术，并将开源的诸多技术进行总结、整理，使其更加系统化和科学化，并为系统应用开发提供一个最佳的应用开发指南。

这里围绕一个实际的工程项目Mesos来展开学习。Mesos是Apache开源的一个资源调度系统，多用于构建分布式的计算。目中前能够支持上万节点的资源调度，很多大的公司都在使用，包括Apple、Paypal、Airbnb、Uber等。

Mesos具有企业级应用典型的特点：高可靠、分布式、高可用、高性能。对于开发者来说，也是绝佳的学习样本，Mesos项目中使用了libev、libprocess、gmock、glog、protobuf、zookeeper、boost等代码库，涵盖了高性能的事件系统、网络、一致性、日志、单元测试、RPC等系统级开发中经常会用用到的技术。

也正是看到Mesos项目中，本身所具有的诸多优点，本系列文章将围绕Mesos源码，对项目中用到的通用技术进行梳理，一步一应揭开系统级应用开发的面纱，给应用开发这提供一个更加清晰的开发思路。

第一章　项目构建

Mesos代码是使用CMake来管理的，CMake可以根据项目中使用的操作系统环境以及编译环境，生成跨平台的编译指令来构建项目。特别是在大型的项目中，使用CMake来管理整个项目，使得整个项目的组织、构建、测试、发布都能变得更加简便。本章的撰写思路时，先了解CMake的相关概念和基本指南，然后梳理一下Mesos代码结构，结合指南和Mesos源码结构给出CMake实践指南。最后一节对开发中的其他相关工作进行总结，主要包括了版本控制、代码风格、日志和异常管理、以及代码测试等进行简单的介绍。

1.1 CMake使用指南

CMake用于跨平台的编译系统，对于通常的c/c++工程，都是通过make来进行编译的，CMake可以通过指令生成Makefile文件来指导整个项目的编译过程。CMake项目组还有其他的工具来完成项目的打包部署以及测试。

1.1.1 Make使用说明

Make是最常用的构建工具，主要用于C/C++语言的项目。make编译命令通过Makefile文件来构建整个工程。

对于简单的helloworld.cpp这样的代码，直接通过g++编译指令接可以完成项目的构建过程。例如：

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {

    cout << "Hello  World!!" << endl;
    return 0;

}
```

编译生成并可执行可执行文件helloworld

```
g++ -o helloworld helloworld.cpp
./helloworld
```

但是对于稍微大一点的项目，每次都使用个g++来编译，显然不能满足实际开发的需求。那能否通过编写脚本来处理整个项目构建的过程呢？make使用Makefile中的规则来指导编译过程，从而避免每次去编写编译脚本。

Makefile文件由一系列规则构成，每条规则的基本格式如下：

```
<target> : <prerequisites>
[tab]  <commands>
```

构建的规则指明构建的依赖条件以及构建的具体执行过程。规则中首先申明构建的目标和前置条件，目标可以是文件也可以是指令，前置条件通常是文件，用于说明创建目标target之前必须要更新的文件。第二行写明执行的命令，注意的是必须有tab键。"目标"是必需的，不可省略；"前置条件"和"命令"都是可选的，但是两者之中必须至少存在一个。

上例的过程就使用Makefile规则来说明如下：

```
helloworld:
    g++ -o helloworld helloworld.cpp
```

构建执行：

```
make hellowolrd   //生成目标
./helloworld     //执行目标文件
```

Makefile中还可以使用基本的指令，以及shell脚本来控制整个项目构建过程。关于这方面详细的内容不再讲解，读者可以参考参考教程中给出的文章。这里给出一个git上的一个比较全面的Makefile示例文件:

https://gist.github.com/isaacs/62a2d1825d04437c6f08

正如示例开头所讲make非常强大，但是其语法并十分友好，同时对大的项目中每次去编写Makefile文件也是很头大的事儿。而CMake却可以通过简单的指令来帮助程序开发人员生成Makefile文件。

> 参考教程：
>
> http://www.ruanyifeng.com/blog/2015/02/make.html
> https://www.gnu.org/software/make/manual/make.html

1.1.2 CMake使用指南

CMake是一个跨平台的、开源的make系统。CMake使用编译器独立的脚本来生成Makefile文件，完成项目编译过程。CMake使用脚本文件来生成Makefile文件，通常CMakeLists.txt。当然CMakeLists.txt并不是只用于项目构建，也可以用于CPack、CTest等过程。

CMake中包括了内置变量、控制指令、变量以及Message四种语法，涉及的内容很多。这里不再详细说明其语法，通过实际的几个应用场景来说明具体的使用方法。

HelloWorld

仍然以上例中的HelloWorld作为例子,这里先给出项目的代码结构

```
helloworld1
  build
  src
    main.cpp
  CMakeLists.txt
```

CMakeLists.txt

```
# THE HEllOWORLD PROJECT.
#####################
cmake_minimum_required(VERSION 2.8)

project(helloworld)

aux_source_directory(./src SRC_LIST)
add_executable(${PROJECT_NAME} ${SRC_LIST})
```

编译运行

```
cd build
cmake ..
make
./helloworld
```

本例中CMakeLists.txt中，给出了工程名以及需要的cmake版本，第三行给出了依赖的源文件，这里是./src下的所有.hpp .h .cpp .c .hxx格式的文件，第四行给出了编译的目标文件和依赖的源文件。

示例代码下载地址： http://pan.baidu.com/s/1dF2Sf0d

添加配置信息

在实际的工程项目中，版本迭代的速度很快，可以通过脚本来添加版本信息。CMake将对应的版本信息，写到对应的配置头文件中。

```
# THE HEllOWORLD PROJECT.
#####################
cmake_minimum_required(VERSION 2.8)

project(helloworld)

set(HELLOWORLD_VERSION_MAJOR 0)
set(HELLOWORLD_VERSION_MINOR 28)
set(HELLOWORLD_PATCH_VERSION 0)
set(PACKAGE_VERSION
${MESOS_VERSION_MAJOR}.${MESOS_VERSION_MINOR}.${MESOS_PAT
CH_VERSION})
set(HELLOWORLD_PACKAGE_VERSION ${PACKAGE_VERSION})
set(HELLOWORLD_PACKAGE_SOVERSION 0)

#  Configuration
###################
configure_file (
  "${PROJECT_SOURCE_DIR}/demo_config.h.in"
  "${PROJECT_BINARY_DIR}/demo_config.h"
)
include_directories("${PROJECT_BINARY_DIR}")

aux_source_directory(./src SRC_LIST)
add_executable(${PROJECT_NAME} ${SRC_LIST})
```

创建对应的配置文件：demo_config.h.in

```
#define HELLOWORLD_VERSION_MAJOR
@HELLOWORLD_VERSION_MAJOR@
#define HELLOWORLD_VERSION_MINOR
@HELLOWORLD_VERSION_MINOR@
```

在main.cpp中使用配置文件：

```
#include <iostream>
#include "demo_config.h"
using namespace std;

int main(int argc, char *argv[]) {
    cout << argv[0] << " Version is " <<
HELLOWORLD_VERSION_MAJOR << "." <<
            HELLOWORLD_VERSION_MINOR << endl;
    cout << "Hello  World!!" << endl;
    return 0;

}
```

编译运行过程同上，这里也给出工程的代码结构如下：

```
helloworld2
  build
  src
    main.cpp
  CMakeLists.txt
  demo_config.h.in
```

添加代码库

通常会将自己的工程按照不同的工程模块组织起来，不同的模块会编译成不同的目标文件，最后通过链接形成最终的目标文件。在src目录下添加一个demo文件夹，并添加demo_math.h demo_math.cpp文件。并在该目录下创建CMakeLists.txt

```
add_library(mathfunction demo_math.cpp)
```

为了整个工程中能找到对应的目录，需要在整个工程的CMakeLists.txt添加引入指令。

```
include_directories ("${PROJECT_SOURCE_DIR}/src/demo")
add_subdirectory (./src/demo)
```

以上的语句引入demo下的源码，并将其添加到编译目录下，以便编译的时候能找到对应的原型文件。同时这里将对应的源码编译成对应的库文件。最后在目标文件中，链接编译的动态库。

```
add_executable (helloworld SRC_LIST)
target_link_libraries (helloworld mathfunction)
```

编译运行，这里也给出代码的结构，后续的不再赘述：

```
helloworld2
  build
  src
    demo
      CMakeLists.txt
      demo_math.h
      demo_math.cpp
    main.cpp
  CMakeLists.txt
  demo_config.h.in
```

使用系统库也是同样的过程,例如使用math库：

```
target_link_libraries (${PROJECT_NAME} m)
```

Check

用于对相应系统环境中的一些函数或者环境变量的检测，

```
include (CheckFunctionExists)
check_function_exists (log HAVE_LOG)
check_function_exists (exp HAVE_EXP)
```

check的结果可以在demo_config.h.in中来定义

```
#cmakedefine HAVE_LOG
#cmakedefine HAVE_EXP
```

第三方库管理

使用第三方库可以通过直接链接已经编译好的开发库。使用编译好的库时，可以通过find_library来查找系统路径中的库代码。使用发布的源码编译时，要将对应的项目加入到编译目录下。

```
if (EXISTS "${PROJECT_SOURCE_DIR}/gflags/CMakeLists.txt")
  add_subdirectory(gflags)
else ()
  find_library(gflags REQUIRED)
endif ()
```

也可以通过网络来

安装

安装代码

```
install (TARGETS MathFunctions DESTINATION bin)
install (FILES MathFunctions.h DESTINATION include)
```

使用该指令，make install会将对应的目标文件copy到/usr/local/bin目录下，把头文件copy到/usr/local/include目录下。该目录可以通过CMAKE_INSTALL_PREFIX来设置。

测试

CTest一般用于可执行文件的测试，整个项目编译完成后，对整个项目的执行测试。

```
includ(CTest)
add_test (testname 可执行文件 参数列表)  //参数以空格隔开
set_tests_properties (testname
PROPERTIES PASS_REGULAR_EXPRESSION "is 4") //结果验证
```

测试文件也可以通过宏文件来定义

```
#define a macro to simplify adding tests, then use it
macro (do_test arg result)
  add_test (testname${arg} testfun ${arg})
  set_tests_properties (testname${arg}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result})
endmacro (do_test)

# do a bunch of result based tests
do_test (25 "25 is 5")
do_test (-25 "-25 is 0")
```

Package

```
# Package
# build a CPack driven installer package
include (InstallRequiredSystemLibraries)
set (CPACK_RESOURCE_FILE_LICENSE
     "${CMAKE_CURRENT_SOURCE_DIR}/LICENSE")
set (CPACK_PACKAGE_VERSION_MAJOR
"${HELLOWORLD_VERSION_MAJOR}")
set (CPACK_PACKAGE_VERSION_MINOR
"${HELLOWORLD_VERSION_MINOR}")
include (CPack)
```

CMake完成后，会生成PackConfig文件

```
cpack --config CPackConfig.cmake  //二进制
cpack --config CPackSourceConfig.cmake  //源码
```

跨平台

通过开关选项来控制编译参数，并用条件控制指令来选择不同的分支。

```
if (REBUNDLED AND ENABLE_LIBEVENT)
  message(
    WARNING
    "Both `ENABLE_LIBEVENT` and `REBUNDLED` (set to TRUE
by default) flags "
    "have been set. But, libevent does not come rebundled
in Mesos, so it must "
    "be downloaded."
    )
endif (REBUNDLED AND ENABLE_LIBEVENT)

if (WIN32 AND REBUNDLED)
  message(
    WARNING
```

```
    "The current supported version of ZK does not compile
on Windows, and does "
    "not come rebundled in the Mesos repository. It must
be downloaded from "
    "the Internet, even though the `REBUNDLED` flag was
set."
    )
endif (WIN32 AND REBUNDLED)

if (WIN32 AND (NOT ENABLE_LIBEVENT))
  message(
    FATAL_ERROR
    "Windows builds of Mesos currently do not support
libev, the default event "
    "loop used by Mesos. To opt into using libevent, pass
"
    "`-DENABLE_LIBEVENT=1` as an argument when you run
CMake. NOTE: although "
    "the plan is to eventually transition to libevent, it
is still in "
    "experimental support, and the code path is much less
well-exercised."
    )
endif (WIN32 AND (NOT ENABLE_LIBEVENT))
```

完整的示例

```
# THE HEllOWORLD PROJECT.
#####################
cmake_minimum_required(VERSION 2.8)

project(helloworld)

set(HELLOWORLD_VERSION_MAJOR 0)
set(HELLOWORLD_VERSION_MINOR 28)
```

```
set(HELLOWORLD_PATCH_VERSION 0)
set(PACKAGE_VERSION
${MESOS_VERSION_MAJOR}.${MESOS_VERSION_MINOR}.${MESOS_PAT
CH_VERSION})
set(HELLOWORLD_PACKAGE_VERSION ${PACKAGE_VERSION})
set(HELLOWORLD_PACKAGE_SOVERSION 0)

# Check
include (CheckFunctionExists)
check_function_exists (log HAVE_LOG)
check_function_exists (exp HAVE_EXP)

#  Configuration
###################
configure_file (
  "${PROJECT_SOURCE_DIR}/demo_config.h.in"
  "${PROJECT_BINARY_DIR}/demo_config.h"
)
include_directories("${PROJECT_BINARY_DIR}")

# Module or Lib
include_directories ("${PROJECT_SOURCE_DIR}/src/demo")
add_subdirectory (./src/demo)

# Compile
aux_source_directory(./src SRC_LIST)

set_target_properties(
  ${PEOJECT_NAME} PROPERTIES
  VERSION ${HELLOWORLD_PACKAGE_VERSION}
  SOVERSION ${HELLOWORLD_PACKAGE_SOVERSION}
)

add_executable(${PROJECT_NAME} ${SRC_LIST})

target_link_libraries (${PROJECT_NAME} mathfunction)
```

```
target_link_libraries (${PROJECT_NAME} m)

# Install
install (TARGETS ${PROJECT_NAME} DESTINATION bin)

# Test
include(CTest)
add_test (test1 ${PROJECT_NAME})
set_tests_properties (test1
PROPERTIES PASS_REGULAR_EXPRESSION "Version is 0.28")


# Package
# build a CPack driven installer package
include (InstallRequiredSystemLibraries)
set (CPACK_RESOURCE_FILE_LICENSE
     "${CMAKE_CURRENT_SOURCE_DIR}/LICENSE")
set (CPACK_PACKAGE_VERSION_MAJOR
"${HELLOWORLD_VERSION_MAJOR}")
set (CPACK_PACKAGE_VERSION_MINOR
"${HELLOWORLD_VERSION_MINOR}")
include (CPack)
```

CMake包管理

CMake包通过项目目录下的.cmake文件来管理，其查找路径可以通过CMAKE_MODULE_PATH来设置：

```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH}
"${CMAKE_SOURCE_DIR}/cmake")
```

其他

- list list提供列表操作， list子命令APPEND, INSERT, REMOVE_AT, REMOVE_ITEM,REMOVE_DUPLICATES, REVERSE以及SORT在当前的

CMake变量域创建一些新值。

```
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/cmake)
```

- 添加编译器参数
  set(CMAKE_CXX_FLAGS "-Wall -g")
- 开关选项cmakedefine

```
option(USE_MYMATH
  "Use tutorial provided math implementation" ON)
```

使用时在对应的configuretion文件中添加对应的宏定义即可,同时可以作为变量在CMakeLists.txt中使用。

```
#cmakedefine USE_MYMATH
```

- 项目中产生文件

```
# first we add the executable that generates the table
add_executable(MakeTable MakeTable.cxx)

# add the command to generate the source code
add_custom_command (
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/table.h
  COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/table.h
  DEPENDS MakeTable
  )
```

可以通过可执行文件MaekTable来产生table.h头文件

- hpp文件

hpp其实质就是将.cpp的实现代码混入.h头文件当中，定义与实现都包含在同一文件，则该类的调用者只需要include该hpp文件即可，不需再将cpp加入到project中进行编译。而实现代码将直接编译到调用者的obj文件中，不再生成单独的obj,采用hpp将大幅度减少调用 project中的cpp文件数与编译次数，简化了编译过程。很多开源的开发库，例如Boost等，都采用这种方式来编写的。

使用hpp文件缺点是代码对外是完全开放的，所有人都可以看到源码的实现。

使用hpp文件需要注意：

http://baike.baidu.com/subview/3779455/17977377.htm

1、不可包含全局对象和全局函数

由于hpp本质上是作为.h被调用者include，所以当hpp文件中存在全局对象或者全局函数，而该hpp被多个调用者include时，将在链接时导致符号重定义错误。要避免这种情况，需要去除全局对象，将全局函数封装为类的静态方法。

2、类之间不可循环调用。 在.h和.cpp的场景中，当两个类或者多个类之间有循环调用关系时，只要预先在头文件做被调用类的声明即可，如下：

```
class B;
class A{
public:
     void someMethod(B b);
};
class B{
public:
     void someMethod(A a);
};
```

在hpp场景中，由于定义与实现都已经存在于一个文件，调用者必需明确知道被调用者的所有定义，而不能等到cpp中去编译。因此hpp中必须整理类之间调用关系，不可产生循环调用。

3、不可使用静态成员。

静态成员的使用限制在于如果类含有静态成员，则在hpp中必需加入静态成员初始化代码，当该hpp被多个文档include时，将产生符号重定义错误。

> 参考教程
>
> https://cmake.org/Wiki/CMake
>
> https://cmake.org/documentation/
>
> https://cmake.org/cmake-tutorial/
>
> http://hahack.com/codes/cmake/
>
> https://cmake.org/cmake/help/v3.0/

Mesos代码结构

上一节中学习了CMake的相关知识，本节结合CMake和Mesos源码来对mesos的代码结构进行梳理，为后续的学习和使用打下基础。

对于c\/c++代码，项目编译链接过程很好地体现了整个代码的组织逻辑，但是没法体现动态的依赖关系。因此从编译链接过程来进一步理解整个工程依赖关系，比直接查看代码的目录结构更加清晰。本节从mesos的整体结构入手，通过静态目录和动态依赖关系两个角度来分析代码的组织结构，为理顺整个代码做些概念性的认识。

mesos源码地址:https:\/\/github.com\/apache\/mesos, 读者可以自己下载相关的代码。同时在mesos官网下载一份发布版的源代码。这两个版本的代码的区别在于，发布版的代码是打包后的可直接编译执行的代码。

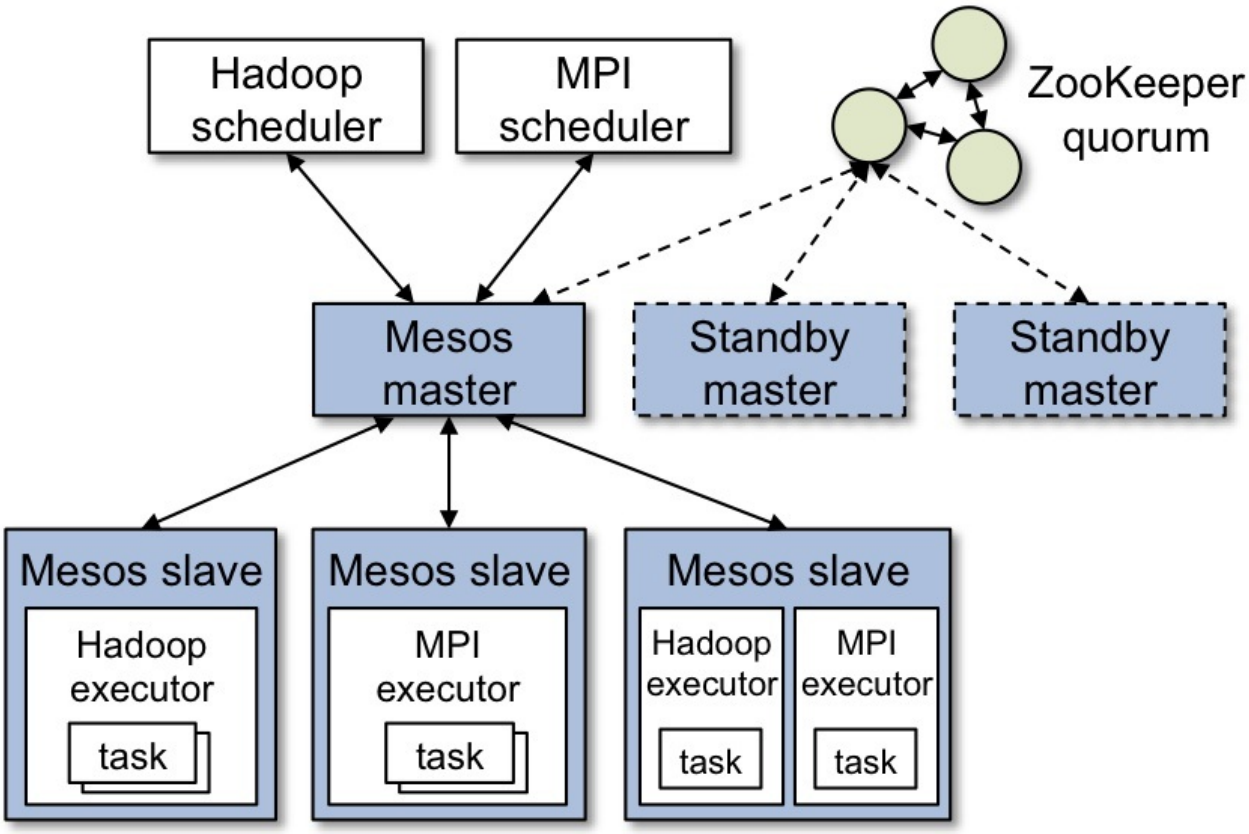
阅读本节前建议先查看下mesos官方的开发者文档，从中也可以窥见mesos项目的特点，便于在后续的理解。

总体架构

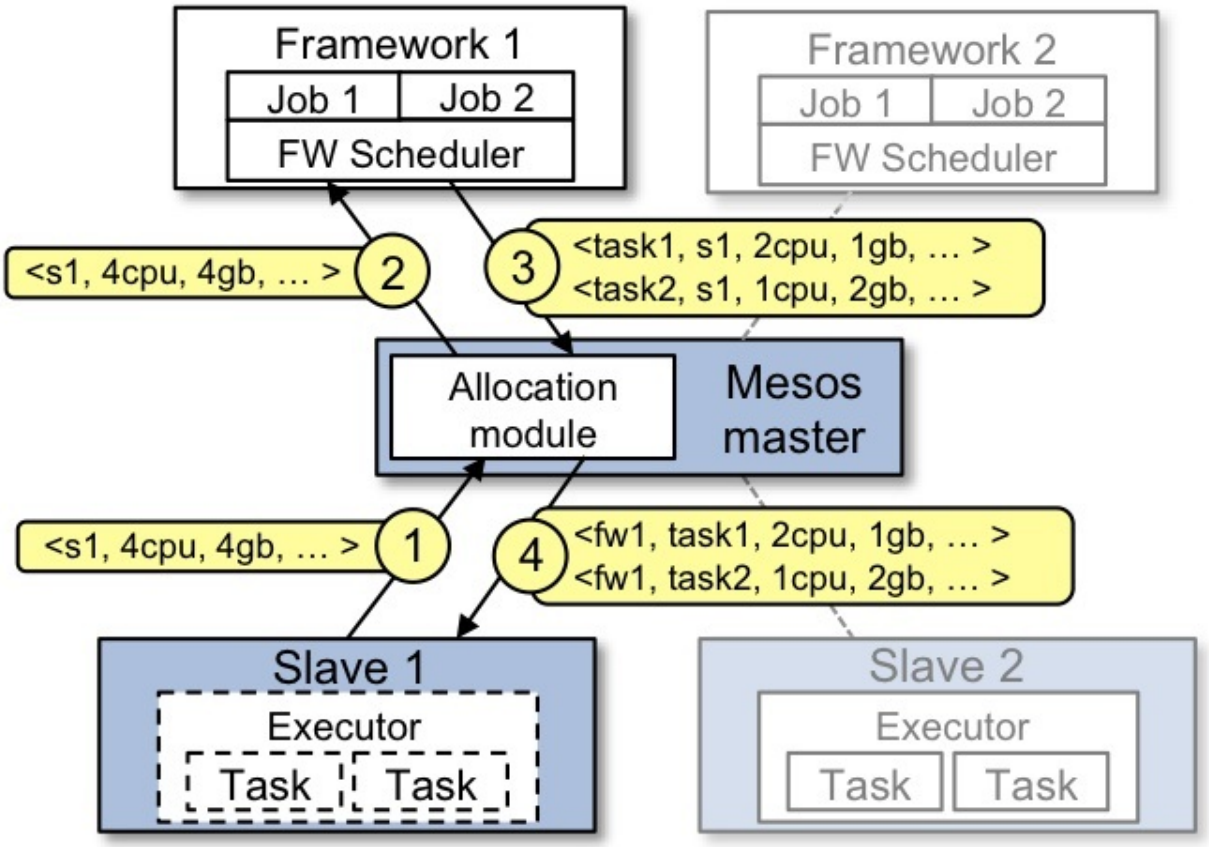
mesos中有四个基本的角色：master, slave, freamework和executor。

- Mesos-master：负责管理framework和slave，收集slave上的资源信息，并将slave上的资源分配给各个framework
- Mesos-slave：管理Node节点上的各个mesos-task，同时为executor分配资源
- Framework：计算框架，如：Hadoop，Spark等，通过MesosSchedulerDiver接入Mesos
- Executor：执行器，用于启动计算框架中的task。

当用户试图添加一种新的计算框架到Mesos中时，需要实现一个Framework scheduler和executor以接入Mesos。



Mesos中按照资源的约束来为不同的框架提供不同的资源，计算框架获取资源后调度并执行对应的任务。



reference:

http:\/\/dongxicheng.org\/apache-mesos\/meso-architecture\/
http:\/\/mesos.apache.org\/documentation\/latest\/architecture\/

目录结构说明

首先看下发布版的mesos代码和git版本的代码区别，基本的代码都是一样的，不同的是git代码中多出了cmake文件夹、以及其他和cmake相关的文件。这就意味着可以通过cmake来构建整个工程项目。(这里发布版为什么不使用cmake？)

整个代码的组织包括了三块 include, src, 3rdparty。mesos中使用的第三方库这里都是源码的形式存放在3rdparty目录下，使用的第三方库包括了libprocess, zookeeper, leveldb。

Zookeeper用于保证Master节点的高可用，保障主节点的可靠性，避免了单点故障。libprocess用于内部节点之间的数据通信和消息处理。

在具体的代码实现中围绕上面的四个角色和不同角色之间的关联来展开讨论。

依赖关系说明

依赖关系说明从CMakeList和Make过程来进行说明。下载git代码后，粗略浏览源码，发现源码中既包含了CMakeLists.txt以及Cmake文件夹，同时又包括了autoconf相关的文件。git版本的代码包含了两种方式的代码构建方式，一种是实用CMake，一种是AutoMake。发布版本都适用了AutoMake来编译，针对1.0版本后新增加了CMake方式，这也为跨平台（特别是Mesos On Windows平台提供了支持）。

CMake编译

首先对，使用cmake对git源码进行编译：

```
mkdir build
cmake ..
make
```

打开文件夹可以看到，build下的代码相对源码少了很多了， 编译完成后，生成了静态的链接库libmesos libprocess libzookeeper liblevedb。

```
3rdparty
    leveldb
    zookeeper
    libprocess
  includ/mesos
  src
    master
      registry.pb.h
    slave
      containerizer
    message
      ...ph.h
```

仔细一看会发现cmake编译的代码基本都是pb.h或者pb.cpp结尾的文件，pb在这里就是protobuf。这就说明这些文件基本都是用于分布式系统之间的消息通信。联想到mesos对外要提供不同计算框架的支持，必然要提供对外的服务接口。因此这里编译的文件只是mesos的一个module，用于mesos对外的应用扩展。

使用**Automake**编译

```
mkdir build
../bootstarp
../configure
make
```

完整的编译整个项目最终生成可执行文件,这个过程和编译发布版代码是一样的。首先编译或安装第三方库：

- libprocess
- zookeeper
- leveldb-1.4
- distribute-0.6.26 pip wheel:python包管理工具

CMake构建Mesos代码说明

mesos源码编译

下载git源码后，在源码目录下：

```
mkdir build
cmake ..
```

cmake执行完成后，可以看到当前目录下的代码结构如下所示：

```
3rdparty
include
src
Makefile
CMakeCache.txt
cmake_install.cmake
CTestTestFile.cmake
```

编译完成后，进入src目录下：

```
# Start mesos master (Ensure work directory exists and
has proper permissions).
$ ./mesos-master --ip=127.0.0.1 --work_dir=/var/lib/mesos

# Start mesos agent (Ensure work directory exists and has
proper permissions).
$ ./mesos-agent --master=127.0.0.1:5050 --
work_dir=/var/lib/mesos

# Test mesos master
$ ./test-framework --master=127.0.0.1:5050
```

mesos源码组织

从mesos源码编译过程来梳理mesos源码的组织结构，以及mesos源码如何通过cmake来组织不同模块的依赖关系。

```
# THE MESOS PROJECT.
######################
cmake_minimum_required(VERSION 2.8.10)

project(Mesos)
set(MESOS_MAJOR_VERSION 1)
set(MESOS_MINOR_VERSION 2)
set(MESOS_PATCH_VERSION 0)
set(PACKAGE_VERSION

${MESOS_MAJOR_VERSION}.${MESOS_MINOR_VERSION}.${MESOS_PAT
CH_VERSION})

set(MESOS_PACKAGE_VERSION ${PACKAGE_VERSION})
set(MESOS_PACKAGE_SOVERSION 0)


# CMAKE MODULE SETUP.
######################
# Paths that are searched when `include(...)` is called.
list(APPEND CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/libprocess/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/stout/cmake)
list(
  APPEND
  CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/libprocess/cmake/macros)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/cmake)
```

```
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/examples/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/master/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/slave/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/tests/cmake)

# Macros.
include(Common)
include(External)
include(PatchCommand)
include(Versions)
include(VsBuildCommand)

# Configuration.
include(MesosConfigure)

# SUBDIRECTORIES.
##################
add_subdirectory(3rdparty)
add_subdirectory(src)

# TESTS.
########
add_custom_target(
  check ${STOUT_TESTS_TARGET}
  COMMAND ${PROCESS_TESTS_TARGET}
  COMMAND ${MESOS_TESTS_TARGET}
  DEPENDS ${STOUT_TESTS_TARGET} ${PROCESS_TESTS_TARGET}
${MESOS_TESTS_TARGET}
  )
```

开发规范

向社区贡献代码

代码风格说明

一般的风格可以参考google c++编码规范，mesos源码遵循google c++编码的相关规范。同时添加了一些不同点，这里对常用的几个规范进行说明。具体的一些细节可以查看参考资料中的相关内容。

异同

- 缩进：类中的Access modifiers不缩进，google使用一个空格
- 模板: template
- 变量名:驼峰

```
Try(State _state, T* _t = NULL, const std::string&
_message = "")
  : state(_state), t(_t), message(_message) {}

常量名，例如
SCREAMING_SNAKE_CASE
```

- 函数调用：

```
// 1: OK.
allocator->resourcesRecovered(frameworkId, slaveId,
resources, filters);

// 2: Don't use.
allocator->resourcesRecovered(frameworkId, slaveId,
                              resources, filters);

// 3: Don't use in this case due to "jaggedness".
allocator->resourcesRecovered(frameworkId,
                              slaveId,
                              resources,
                              filters);

// 3: In this case, 3 is OK.
foobar(someArgument,
       someOtherArgument,
       theLastArgument);

// 4: OK.
allocator->resourcesRecovered(
    frameworkId,
    slaveId,
    resources,
    filters);

// 5: OK.
allocator->resourcesRecovered(
    frameworkId, slaveId, resources, filters);
```

- 变量多行赋值,第二行两格空格

```
Try<Duration> failoverTimeout =
Duration::create(FrameworkInfo().failover_timeout());
```

- 空行：在多个逻辑块之间使用一行空行，类或结构体之间使用两行空行
- 禁止使用模板引用

```
Future<Nothing> f() { return Nothing(); }
Future<bool> g() { return false; }

struct T
{
  T(const char* data) : data(data) {}
  const T& member() const { return *this; }
  const char* data;
};

// 1: Don't use.
const Future<Nothing>& future = f();

// 1: Instead use.
const Future<Nothing> future = f();

// 2: Don't use.
const Future<Nothing>& future = Future<Nothing>
(Nothing());

// 2: Instead use.
const Future<Nothing> future = Future<Nothing>
(Nothing());

// 3: Don't use.
const Future<bool>& future = f().then(lambda::bind(g));

// 3: Instead use.
const Future<bool> future = f().then(lambda::bind(g));

// 4: Don't use (since the T that got constructed is a
temporary!).
```

```
const T& t = T("Hello").member();

// 4: Preferred alias pattern (see below).
const T t("Hello");
const T& t_ = t.member();

// 4: Can also use.
const T t = T("Hello").member();
```

- 头文件的先后次序

头文件的先后此

Example for src/common/foo.cpp:

```
#include "common/foo.hpp"

#include <stdint.h>

#include <string>
#include <vector>

#include <boost/circular_buffer.hpp>

#include <mesos/mesos.hpp>
#include <mesos/type_utils.hpp>

#include <mesos/module/authenticator.hpp>

#include <mesos/scheduler/scheduler.hpp>

#include <process/http.hpp>
#include <process/protobuf.hpp>

#include <stout/foreach.hpp>
#include <stout/hashmap.hpp>

#include "common/build.hpp"
#include "common/protobuf_utils.hpp"

#include "master/flags.hpp"
```

参考文章：

http://pan.baidu.com/s/1i3gc7lF

https://mesos.apache.org/documentation/latest/c++-style-guide/

https://issues.apache.org/jira/browse/MESOS-2629

CMake使用手册

cmake变量使用${}方式取值,但是在IF控制语句中是直接使用变量名。特别注意的是，环境变量使用$ENV{}方式取值,使用SET(ENV{VAR} VALUE)赋值。

CMake指令大小写无关的，指令的基本形式是：

```
指令(参数1 参数2…)
```

参数使用括弧括起,参数之间使用空格或分号分开。下面会从变量和指令来详细介绍CMake语法。

cmake中的变量

内置变量

```
  PROJECT_SOURCE_DIR 工程的根目录

  PROJECT_BINARY_DIR 运行cmake命令的目录,通常是
${PROJECT_SOURCE_DIR}/build

  CMAKE_INCLUDE_PATH 环境变量,非cmake变量

  CMAKE_LIBRARY_PATH 环境变量

  CMAKE_CURRENT_SOURCE_DIR 当前处理的CMakeLists.txt所在的路
径

   CMAKE_CURRENT_BINARY_DIR target编译目录, 使用
ADD_SURDIRECTORY(src bin)可以更改此变量的值

  CMAKE_CURRENT_LIST_FILE 输出调用这个变量的CMakeLists.txt的
完整路径
```

CMAKE_CURRENT_LIST_LINE 输出这个变量所在的行

CMAKE_MODULE_PATH 定义自己的cmake模块所在的路径
SET(CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake),然后可以用INCLUDE命令来调用自己的模块

EXECUTABLE_OUTPUT_PATH 重新定义目标二进制可执行文件的存放位置

LIBRARY_OUTPUT_PATH 重新定义目标链接库文件的存放位置

PROJECT_NAME 返回通过PROJECT指令定义的项目

CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS 用来控制IF ELSE语句的书写方式
系统信息
CMAKE_MAJOR_VERSION cmake主版本号,如2.8.6中的2

CMAKE_MINOR_VERSION cmake次版本号,如2.8.6中的8

CMAKE_PATCH_VERSION cmake补丁等级,如2.8.6中的6

CMAKE_SYSTEM 系统名称,例如Linux-2.6.22

CAMKE_SYSTEM_NAME 不包含版本的系统名,如Linux

CMAKE_SYSTEM_VERSION 系统版本,如2.6.22

MAKE_SYSTEM_PROCESSOR 处理器名称,如i686

UNIX 在所有的类UNIX平台为TRUE,包括OS X和cygwin

WIN32 在所有的win32平台为TRUE,包括cygwin
开关选项

BUILD_SHARED_LIBS 控制默认的库编译方式。如果未进行设置,使用

ADD_LIBRARY时又没有指定库类型,默认编译生成的库都是静态库 （可在t3中稍加修改进行验证）

CMAKE_C_FLAGS 设置C编译选项

CMAKE_CXX_FLAGS 设置C++编译选项

开关选项

CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS,用来控制 IF ELSE 语句的书写方式。

BUILD_SHARED_LIBS这个开关用来控制默认的库编译方式,如果不进行设置,使用 ADD_LIBRARY 并没有指定库类型的情况下,默认编译生成的库都是静态库。如果 SET(BUILD_SHARED_LIBS ON)后,默认生成的为动态库。

CMAKE_C_FLAGS设置 C 编译选项,也可以通过指令 ADD_DEFINITIONS()添加。

CMAKE_CXX_FLAGS设置 C++编译选项,也可以通过指令 ADD_DEFINITIONS()添加。

自定义变量

基本变量 set(name value)

环境变量 set(env{name} value)

cmake常用命令

部分常用命令列表：

- PROJECT
  PROJECT(projectname [CXX] [C] [Java])，指定工程名称,并可

指定工程支持的语言。

- SET

SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])

定义变量(可以定义多个VALUE,如SET(SRC_LIST main.c util.c reactor.c))

- MESSAGE

MESSAGE([SEND_ERROR | STATUS | FATAL_ERROR] “message to display” …)

向终端输出用户定义的信息或变量的值

SEND_ERROR, 产生错误,生成过程被跳过

STATUS, 输出前缀为—的信息

FATAL_ERROR, 立即终止所有cmake过程

- ADD_EXECUTABLE

ADD_EXECUTABLE(bin_file_name ${SRC_LIST})生成可执行文件

- ADD_LIBRARY

ADD_LIBRARY(libname [SHARED | STATIC | MODULE] [EXCLUDE_FROM_ALL] SRC_LIST)

生成动态库或静态库

SHARED 动态库

STATIC 静态库

MODULE 在使用dyld的系统有效,若不支持dyld,等同于SHARED

EXCLUDE_FROM_ALL 表示该库不会被默认构建

- SET_TARGET_PROPERTIES

设置输出的名称,设置动态库的版本和API版本

- ADD_SUBDIRECTORY

ADD_SUBDIRECTORY(src_dir [binary_dir] [EXCLUDE_FROM_ALL])

向当前工程添加存放源文件的子目录,并可以指定中间二进制和目标二进制的存放位置

EXCLUDE_FROM_ALL含义：将这个目录从编译过程中排除

- INCLUDE_DIRECTORIES

INCLUDE_DIRECTORIES([AFTER | BEFORE] [SYSTEM] dir1 dir2 …)

向工程添加多个特定的头文件搜索路径。

- LINK_DIRECTORIES

LINK_DIRECTORIES(dir1 dir2 …)

添加非标准的共享库搜索路径

- TARGET_LINK_LIBRARIES

  TARGET_LINK_LIBRARIES(target lib1 lib2 …)

为target添加需要链接的共享库

- ADD_DEFINITIONS

  向C/C++编译器添加-D定义

  ADD_DEFINITIONS(-DENABLE_DEBUG -DABC),参数之间用空格分隔

- ADD_DEPENDENCIES

  ADD_DEPENDENCIES(target-name depend-target1 depend-target2 …)

定义target依赖的其他target,确保target在构建之前,其依赖的target已经构建完毕

- AUX_SOURCE_DIRECTORY

AUX_SOURCE_DIRECTORY(dir VAR)

发现一个目录下所有的源代码文件并将列表存储在一个变量中

把当前目录下的所有源码文件名赋给变量DIR_HELLO_SRCS

- INCLUDE

INCLUDE(file [OPTIONAL]) 用来载入CMakeLists.txt文件

INCLUDE(module [OPTIONAL])用来载入预定义的cmake模块

OPTIONAL参数的左右是文件不存在也不会产生错误

可以载入一个文件,也可以载入预定义模块（模块会在CMAKE_MODULE_PATH指定的路径进行搜索）

载入的内容将在处理到INCLUDE语句时直接执行

- FIND_
  - FIND_FILE(<VAR> name path1 path2 …)

VAR变量代表找到的文件全路径,包含文件名

  - FIND_LIBRARY(<VAR> name path1 path2 …)

VAR变量代表找到的库全路径,包含库文件名

```
FIND_LIBRARY(libX X11 /usr/lib)
                    IF (NOT libx)
                            MESSAGE(FATAL_ERROR "libX not found")
                    ENDIF(NOT libX)
```

  - FIND_PATH(<VAR> name path1 path2 …)

VAR变量代表包含这个文件的路径

- FIND_PROGRAM(<VAR> name path1 path2 …)

VAR变量代表包含这个程序的全路径

- FIND_PACKAGE(<name> [major.minor] [QUIET] [NO_MODULE] [[REQUIRED | COMPONENTS] [componets …]])

用来调用预定义在CMAKE_MODULE_PATH下的Find<name>.cmake模块,你也可以自己定义Find<name>
模块,通过SET(CMAKE_MODULE_PATH dir)将其放入工程的某个目录供工程使用

基本语法规则：

- IF

语法：

```
IF (expression)
    COMMAND1(ARGS ...)
    COMMAND2(ARGS ...)
    ...
ELSE (expression)
    COMMAND1(ARGS ...)
    COMMAND2(ARGS ...)
    ...
ENDIF (expression) # 一定要有ENDIF与IF对应
```

IF语句中的逻辑判断

```
  IF (not exp), 与上面相反
  IF (var1 AND var2)
  IF (var1 OR var2)
  IF (COMMAND cmd) 如果cmd确实是命令并可调用,为真
  IF (EXISTS dir) IF (EXISTS file) 如果目录或文件存在,为真
  IF (file1 IS_NEWER_THAN file2),当file1比file2新,或file1/file2中有一个不存在时为真,文件名需使用全路径
  IF (IS_DIRECTORY dir) 当dir是目录时,为真
  IF (DEFINED var) 如果变量被定义,为真
  IF (var MATCHES regex) 此处var可以用var名,也可以用${var}
  IF (string MATCHES regex)
```

当给定的变量或者字符串能够匹配正则表达式regex时为真。比如：

```
  IF ("hello" MATCHES "ell")
    MESSAGE("true")
  ENDIF ("hello" MATCHES "ell")
```

数字比较表达式

```
  IF (variable LESS number)
  IF (string LESS number)
  IF (variable GREATER number)
  IF (string GREATER number)
  IF (variable EQUAL number)
  IF (string EQUAL number)
```

按照字母表顺序进行比较

```
  IF (variable STRLESS string)
  IF (string STRLESS string)
  IF (variable STRGREATER string)
  IF (string STRGREATER string)
  IF (variable STREQUAL string)
  IF (string STREQUAL string)
```

一个小例子,用来判断平台差异：

```
  IF (WIN32)
      MESSAGE(STATUS “This is windows.”)
  ELSE (WIN32)
      MESSAGE(STATUS “This is not windows”)
  ENDIF (WIN32)
```

- WHILE

语法：

```
WHILE(condition)
    COMMAND1(ARGS ...)
    COMMAND2(ARGS ...)
    ...
ENDWHILE(condition)
```

其真假判断条件可以参考IF指令

- FOREACH

FOREACH指令的使用方法有三种形式：

a. 列表

语法：

```
FOREACH(loop_var arg1 arg2 ...)
     COMMAND1(ARGS ...)
     COMMAND2(ARGS ...)
 ...
ENDFOREACH(loop_var)
```

示例：

```
AUX_SOURCE_DIRECTORY(. SRC_LIST)
FOREACH(F ${SRC_LIST})
     MESSAGE(${F})
ENDFOREACH(F)
```

b. 范围

语法：

```
FOREACH(loop_var RANGE total)
    COMMAND1(ARGS ...)
    COMMAND2(ARGS ...)
    ...
ENDFOREACH(loop_var)
```

示例：

从0到total以1为步进

```
FOREACH(VAR RANGE 10)
   MESSAGE(${VAR})
ENDFOREACH(VAR)
```

输出：

```
012345678910
```

c. 范围和步进

语法：

```
FOREACH(loop_var RANGE start stop [step])
    COMMAND1(ARGS ...)
    COMMAND2(ARGS ...)
    ...
```

```
ENDFOREACH(loop_var)
FOREACH(A RANGE 5 15 3)
    MESSAGE(${A})
ENDFOREACH(A)
```

CMake实践指南

前面几章已经废话了那么多，说明CMake的使用以及CMake常见的语法，并且针对Mesos代码结构大致讲解了下Mesos项目中如何使用CMake的。这一节中，将围绕Mesos代码中CMake来说明下，自己在创建项目时使用CMake的基本方法，并按照Mesos源码的样本给出一个最佳的实践指南。

从C++代码结构说起

基本的CPP工程代码结构如下，头文件、源码、第三方库文件以及编译文件。

```
include   //头文件
src       //源码
bin       //可执行文件目录
3rdparty  //第三方库
CMakeLists.txt
build
```

对于复杂的项目中，可能还会用到CMake的一些宏文件和配置文件，本身项目中也会增加一些编译时的配置文件，可以在代码中直接使用。

```
include   //头文件
src       //源码
bin       //可执行文件目录
3rdparty  //第三方库
CMakeLists.txt
build
cmake
config.h.in
```

Mesos源码构建分步骤讲解

基础的结构如上所示，分别看一下CMakeLists.txt编写的模版格式：

```
# THE MESOS PROJECT.
######################
cmake_minimum_required(VERSION 2.8.10)

project(Mesos)
set(MESOS_MAJOR_VERSION 1)
set(MESOS_MINOR_VERSION 2)
set(MESOS_PATCH_VERSION 0)
set(PACKAGE_VERSION

${MESOS_MAJOR_VERSION}.${MESOS_MINOR_VERSION}.${MESOS_PAT
CH_VERSION})

set(MESOS_PACKAGE_VERSION ${PACKAGE_VERSION})
set(MESOS_PACKAGE_SOVERSION 0)


# CMAKE MODULE SETUP.
######################
# Paths that are searched when `include(...)` is called.
list(APPEND CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/libprocess/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/stout/cmake)
list(
  APPEND
  CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/3rdparty/libprocess/cmake/macros)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/examples/cmake)
```

```
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/master/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/slave/cmake)
list(APPEND CMAKE_MODULE_PATH
${CMAKE_SOURCE_DIR}/src/tests/cmake)

# Macros.
include(Common)
include(External)
include(PatchCommand)
include(Versions)
include(VsBuildCommand)

# Configuration.
include(MesosConfigure)


# SUBDIRECTORIES.
##################
add_subdirectory(3rdparty)
add_subdirectory(src)


# TESTS.
########
add_custom_target(
  check ${STOUT_TESTS_TARGET}
  COMMAND ${PROCESS_TESTS_TARGET}
  COMMAND ${MESOS_TESTS_TARGET}
  DEPENDS ${STOUT_TESTS_TARGET} ${PROCESS_TESTS_TARGET}
${MESOS_TESTS_TARGET}
  )

# TARGET.
########
```

```
# LINKING LIBRARIES BY DIRECTORY (might generate, e.g., -
L/path/to/thing on Linux
link_directories(${PROCESS_LIB_DIRS})

# THE PROCESS LIBRARY (generates, e.g., libprocess.so,
etc., on Linux).
if (WIN32)
  add_library(${PROCESS_TARGET} STATIC ${PROCESS_SRC})
else (WIN32)
  add_library(${PROCESS_TARGET} SHARED ${PROCESS_SRC})

  set_target_properties(
    ${PROCESS_TARGET} PROPERTIES
    POSITION_INDEPENDENT_CODE TRUE
    )
endif (WIN32)

set_target_properties(
  ${PROCESS_TARGET} PROPERTIES
  VERSION ${PROCESS_PACKAGE_VERSION}
  SOVERSION ${PROCESS_PACKAGE_SOVERSION}
  )

add_dependencies(${PROCESS_TARGET}
${PROCESS_DEPENDENCIES})

# ADD LINKER FLAGS (generates, e.g., -lglog on Linux)
target_link_libraries(${PROJECT_NAME} ${PROCESS_LIBS})
add_executable(${PROJECT_NAME} ${SRC_LIST})
```

Mesos依赖库组织

从Mesos源码中可以看到，Mesos代码开发维护时，使用第三方库时，基本将用到的库源码都在代码中维护下来了。在很多项目中，通常通过在线安装的方式来预先安装好第三方库的头文件和链接库，经常编译一个项目要把第三方库装好，都要费很长时间，特别是依赖多、各个系统版本不一致的情况下。

Mesos源码的第三方库都在源码文件目录下，编译时将所有的源文件重新编译，减少了开发人员维护第三方库的代价。Mesos中如何来组织第三方库的依赖关系呢？我们以glog库为例来说明，只说明编译时的依赖关系如何组织。

阅读时可参考代码：

```
include //头文件
src //源码
bin //可执行文件目录
3rdparty //第三方库
  CMakeList.txt
  cmake
    Versions.cmake
    3rdpartyConfigure.cmake
  glog-0.3.3.tar.gz
  glog-0.3.3.patch
tests
CMakeLists.txt
build
support
m4
docs
cmake
  Common.cmake
  PatchCommand.cmake
  CompliationConfigure.cmake //编译时相关配置信息
  ProjectConfigure.cmake  //项目相关的全局配置信息
  External.cmake //第三方库管理
config.h.in  //源码配置信息
```

Mesos中代码的组织、依赖关系管理按照CMakeList.txt文件来维系的。整个工程项目中使用到的宏文件、以及cmake脚本文件，都保存在cmake子目录下。

基本的依赖组织遵循以下几个步骤：

- 1、定义第三方库相关的变量信息。
- 2、ExternalProject_Add确定第三方库的编译方法
- 3、使用确定库与库之间的依赖关系，确保先使用的库先编译，并且添加第三方库include目录和lib目录。

下面分别看一下每一步骤中用的cmake文件。 1、定义ExternalProject_Add需要的参数信息，以及第三方库基本信息。其中第三方库基本信息，在External函数（External.cmake宏文件中定义）中定义。

```
#############################################################
######
# EXTERNAL defines a few variables that make it easy for
us to track the
# directory structure of a dependency. In particular, if
our library's name is
# boost, we will define the following variables:
#
#   BOOST_VERSION    (e.g., 1.53.0)
#   BOOST_TARGET     (a target folder name to put dep in
e.g., boost-1.53.0)
#   BOOST_CMAKE_ROOT (where to have CMake put the
uncompressed source, e.g.,
#                     build/3rdparty/boost-1.53.0)
#   BOOST_ROOT       (where the code goes in various
stages of build, e.g.,
#                     build/.../boost-1.53.0/src, which
might contain folders
#                     build-1.53.0-build, -lib, and so
on, for each build step
#                     that dependency has)
function(EXTERNAL
  LIB_NAME
```

```
  LIB_VERSION
  BIN_ROOT)

  string(TOUPPER ${LIB_NAME} LIB_NAME_UPPER)

  # Names of variables we will set in this function.
  set(VERSION_VAR    ${LIB_NAME_UPPER}_VERSION)    #
e.g., BOOST_VERSION
  set(TARGET_VAR     ${LIB_NAME_UPPER}_TARGET)     #
e.g., BOOST_TARGET
  set(CMAKE_ROOT_VAR ${LIB_NAME_UPPER}_CMAKE_ROOT) #
e.g., BOOST_CMAKE_ROOT
  set(ROOT_VAR       ${LIB_NAME_UPPER}_ROOT)       #
e.g., BOOST_ROOT

  # Generate data that we will put in the above
variables.
  # NOTE: bundled packages are untar'd into the BIN_ROOT,
which is why we're
  #       pointing the source root into BIN_ROOT rather
than SRC_ROOT.
  # TODO(hausdorff): SRC_DATA doesn't work for HTTP,
LIBEV, GMOCK, or GTEST.
  set(VERSION_DATA    ${LIB_VERSION})
  set(TARGET_DATA     ${LIB_NAME}-${VERSION_DATA})
  set(CMAKE_ROOT_DATA ${BIN_ROOT}/${TARGET_DATA})
  set(ROOT_DATA
${CMAKE_ROOT_DATA}/src/${TARGET_DATA})

  # Finally, EXPORT THE ABOVE VARIABLES. We take the data
variables we just
  # defined, and export them to variables in the parent
scope.
  #
  # NOTE: The "export" step is different from the "define
the data vars" step
```

```
  #       because an expression like ${VERSION_VAR} will
evaluate to
  #       something like "BOOST_VERSION", not something
like "1.53.0". That
  #       is: to get the version in the parent scope we
would do something
  #       like ${BOOST_VERSION}, which might evaluate to
something like
  #       "1.53.0". So in this function, if you wanted to
generate (e.g.) the
  #       target variable, it is not sufficient to write
  #       "${LIB_NAME}-${VERSION_VAR}", because this
would result in
  #       something like "boost-BOOST_VERSION" when what
we really wanted was
  #       "boost-1.53.0". Hence, these two steps are
different.
  set(${VERSION_VAR}    # e.g., 1.53.0
    ${VERSION_DATA}
    PARENT_SCOPE)

  set(${TARGET_VAR}     # e.g., boost-1.53.0
    ${TARGET_DATA}
    PARENT_SCOPE)

  set(${CMAKE_ROOT_VAR} # e.g., build/3rdparty/boost-
1.53.0
    ${CMAKE_ROOT_DATA}
    PARENT_SCOPE)

  set(${ROOT_VAR}       # e.g., build/.../boost-
1.53.0/src
    ${ROOT_DATA}
    PARENT_SCOPE)
endfunction()
```

External函数定义第三方库源码目录、编译目录、编译目标等信息。编译参数信息需要自己根据第三方库编译信息来编写。例如glog库的编译：

```
set(GLOG_CONFIG_CMD  ${GLOG_ROOT}/src/../configure --
prefix=${GLOG_LIB_ROOT})
set(GLOG_BUILD_CMD   make libglog.la)
set(GLOG_INSTALL_CMD make install)
PATCH_CMD(${3RDPARTY_SRC}/glog-0.3.3.patch
GLOG_PATCH_CMD)
```

2、ExternalProject_Add确定第三方库的编译方法

```
ExternalProject_Add(
  ${GLOG_TARGET}
  PREFIX            ${GLOG_CMAKE_ROOT}
  CMAKE_ARGS        -DBUILD_SHARED_LIBS=OFF -
DCMAKE_CXX_FLAGS_DEBUG=${CMAKE_CXX_FLAGS_DEBUG}
  PATCH_COMMAND     ${GLOG_PATCH_CMD}
  CONFIGURE_COMMAND ${GLOG_CONFIG_CMD}
  BUILD_COMMAND     ${GLOG_BUILD_CMD}
  INSTALL_COMMAND   ${GLOG_INSTALL_CMD}
  URL               ${GLOG_URL}
  DOWNLOAD_NAME     glog-${GLOG_VERSION}.tar.gz
  )
```

> 在很多场景下INSTALL_COMMAND都为空，这意味着项目中不需要install对应的lib库。

3、添加第三方库依赖关系，以及编译最终目标时include目录、lib目录、链接库信息。

```
# DEFINE LIBRARY DEPENDENCIES. Tells the process library
build targets download/configure/build all third-party
libraries before attempting to build.
############################################################
#######################
set(TARGET_DEPENDENCIES
  ${TARGET_DEPENDENCIES}
  ${GLOG_TARGET}
)

# Target lib
set(TARGET_LIBS
  ${TARGET_LIBS}
  ${GLOG_LFLAG}
)

# Target include dirs
set(TARGET_INCLUDE_DIRS
  ${TARGET_INCLUDE_DIRS}
  ${GLOG_INCLUDE_DIR}
)

# Target lib dirs
set(TARGET_LIB_DIRS
  ${TARGET_LIB_DIRS}
  ${GLOG_LIB_DIR}
)
```

添加依赖关系

```
add_dependencies(${PROJECT_NAME} ${TARGET_DEPENDENCIES})
```

第二章　工具库介绍

C++常用工具库介绍，包括命令行参数、日志处理、单元测试、性能测试以及常用标准库。

gflags

gflags是google的一个开源的处理命令行参数的库，使用c++开发，可以替代getopt。gflags使用起来比getopt方便，但是不支持参数的简写。

使用入门

gflags中使用宏来定义加载的对应的命令行参数，基本格式如下：

```
DEFINE_xxxxx(变量名，默认值，help-string)
```

gflag中支持的命令行参数类型有一下几种:

```
DEFINE_bool: boolean
DEFINE_int32: 32-bit integer
DEFINE_int64: 64-bit integer
DEFINE_uint64: unsigned 64-bit integer
DEFINE_double: double
DEFINE_string: C++ string
DEFINE_VARIABLE
DEFINE_validator
```

基本的使用示例：

- 添加头文件

  ```
  #include <gflags/gflags.h>
  ```

- 添加参数定义

  ```
    DEFINE_int32(port, 2181, "port")
  ```

- main函数中加入处理代码，并使用命令行参数

```
google::ParseCommandLineFlags(&argc, &argv, true);
printf("%s", FLAGS_mystr);
```

> 注意：使用完成后关闭google:ShutDownCommandLineFlags();避免内存泄露。

这里说明下其中的参数信息，argc和argv是命令参数的数目和具体信息，第三个参数用于gflag处理后参数信息的处理：

> true，则该函数处理完成后，argv中只保留argv[0]，argc会被设置为1。
> false，则argv和argc会被保留，但是注意函数会调整argv中的顺序。 这样，在后续代码中可以使用FLAGS_变量名访问对应的命令行参数了

gflag的命令行参数不支持短类型，参数格式都是 --xxx=vaue或者--xxx value的形式。bool类型除了可以使用--xxx=true/false之外，还可以使用--xxx和--noxxx后面不加等号的方式指定true和false。

高级用法

检验输入参数

gflags库支持定制自己的输入参数检查的函数，如下：

```
static bool ValidatePort(const char* flagname, int32
value) {
   if (value > 0 && value < 32768)   // value is ok
     return true;
   printf("Invalid value for --%s: %d\n", flagname,
(int)value);
   return false;
}
DEFINE_int32(port, 0, "What port to listen on");
static const bool port_dummy =
RegisterFlagValidator(&FLAGS_port, &ValidatePort);
```

判断**flags**变量是否被用户使用：

```
    google::CommandLineFlagInfo info;
    if(GetCommandLineFlagInfo("port" ,&info) &&
info.is_default) {
        FLAGS_port = 27015;
    }
```

定制**help**信息与**version**信息

gflags里面已经定义了-h和--version，可以通过以下方式定制它们的内容

version信息：使用google::SetVersionString设定，使用google::VersionString访问

help信息：使用google::SetUsageMessage设定，使用ProgramUsage访问

注意：google::SetUsageMessage和google::SetVersionString必须在ParseCommandLineFlags之前执行

示例说明

gflag语法很简单，使用起来也很方便，这里给出mesos-master命令行参数的一些简单信息。 首先安装对应的开发库，以ubuntu14.04为例

```
sudo apt-get install libgflags-dev
```

添加option.h文件，用于命令行参数的定义和处理

```
#include <iostream>

#include <gflags/gflags.h>

using namespace std;

DEFINE_int32(port, 0, "What port to listen on");

bool ValidatePort(const char* flagname, int32_t value) {
    if (value > 0 && value<32768)   // value is ok
        return true;
    cout << "Invalid value for --" << flagname << " " <<
(int)value << endl;
    return false;
}

int mainOption(int argc, char *argv[]) {
    std::string usage("This program does nothing.  Sample
usage:\n");
    usage += std::string(argv[0])+" --port 1234 \n or :\n
-flagfile=foo.conf";
    google::SetUsageMessage(usage);
    gflags::SetVersionString("1.0.0");

    google::RegisterFlagValidator (&FLAGS_port,
ValidatePort);
}
```

main

```
#include <iostream>
#include <cmath>
#include "demo_config.h"
#include "./option/option.h"

using namespace std;

int main(int argc, char *argv[]) {

    mainOption(argc, argv);
    google::ParseCommandLineFlags (&argc, &argv, false);

    cout << FLAGS_port << endl;

    google:ShutDownCommandLineFlags();
    return 0;
}
```

编译执行，链接库中添加gflags库

```
target_link_libraries (${PROJECT_NAME} gflags)
```

cmake中使用在线安装方式构建

在cmake中使用已安装的二进制包时，只需要在编译时确认所需要的安装包已经安装完成。CMake使用find_package命令来查找外部库的头文件地址和链接库的地址。基本的使用流程如下：

- 1、find_package
- 2、编写 FindXxxx.cmake文件，写明具体的查找流程，这里使用find_path、find_library来查找对应的头文件和库文件地址。

下面分别看一下find_package和编写FindName.cmake文件的详细过程。

find_package流程

find_package基本语法：

```
FIND_PACKAGE( <name> [version] [EXACT] [QUIET]
[NO_MODULE] [ [ REQUIRED | COMPONENTS ] [ componets... ]
] )
```

这条命令执行后，CMake 会到变量 CMAKE_MODULE_PATH 指示的目录中查找文件 Findname.cmake并执行。当然也可以自己定义Find模块，将其放入工程的某个目录中，通过 SET(CMAKE_MODULE_PATH dir)设置查找路径，供工程FIND_PACKAGE使用。

参数说明：

- name: 查找的lib库名称；
- version:版本号，它是正在查找的包应该兼容的版本号（格式是major[.minor[.patch[.tweak]]]）；
- EXACT选项要求版本号必须精确匹配。如果在find-module内部对该命令的递归调用没有给定[version]参数，那么[version]和EXACT选项会自动地从外部调用前向继承；
- QUIET：会禁掉包没有被发现时的警告信息。对应于Find.cmake模块中的name_FIND_QUIETLY；
  > 注意：name必须完全一致，包括大小写
- REQUIRED： 其含义是指是否是工程必须的，表示如果报没有找到的话，cmake的过程会终止，并输出警告信息。对应于Find.cmake模块中的name_FIND_REQUIRED变量；
  > 注意：name必须完全一致，包括大小写
- COMPONENTS在REQUIRED选项之后，或者如果没有指定REQUIRED选项但是指定了COMPONENTS选项，在它们的后面可以列出一些与包相关（依赖）的部件清单（components list）；

例如：

```
find_package(Gflag REQUIRED)
```

编写查找模块 **Findname.cmake**

查找模块的命名Find.cmake,name必须和find_package中的name保持一致。

编写查找模块的基本步骤：

- 1、定义查找的lib库名，例如gflags
- 2、查找头文件目录和lib库文件目录，并判断查找结果

FindGflag.cmake文件如下：

```
# Licensed to the Apache Software Foundation (ASF) under
one
# or more contributor license agreements.  See the NOTICE
file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this
file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in
compliance
# with the License.  You may obtain a copy of the License
at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in
writing, software
# distributed under the License is distributed on an "AS
IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
express or implied.
# See the License for the specific language governing
permissions and
# limitations under the License.

##########################################################
######
# IMPORTANT NOTE: I (hausdorff) have copied a *lot* of
```

```
stuff in this file from
#                 code I found at in the Akumuli[1]
project. The original version
#                 of this file was also released under
the Apache 2.0 license, and
#                 licensed to the ASF, making it fully
compatible with the Mesos
#                 project.
#  [1]
https://github.com/akumuli/Akumuli/blob/master/cmake/Find
APR.cmake
# This module helps to find the Apache Portable Runtime
(APR) and APR-Util
# packages.
#
# USAGE: to use this module, add the following line to
your CMake project:
#   find_package(Apr)
##############################################################
####################

if (GFLAG_FOUND)
  return()
endif (GFLAG_FOUND)

unset(GFLAG_LIB)
unset(GFLAG_INCLUDE_DIR)
unset(GFLAG_LIBS)

# NOTE: If this fails, stderr is ignored, and the output
variable is empty.
# This has no deleterious effect our path search.
execute_process(
  COMMAND brew --prefix gflags
  OUTPUT_VARIABLE GFLAG_PREFIX
  OUTPUT_STRIP_TRAILING_WHITESPACE)
```

```
set(POSSIBLE_GFLAG_INCLUDE_DIRS
  ${POSSIBLE_GFLAG_INCLUDE_DIRS}
  /usr/local/include/gflags
  /usr/include/gflags
)

set(GFLAG_LIB_NAMES ${GFLAG_LIB_NAMES} gflags)

set(POSSIBLE_GFLAG_LIB_DIRS
  ${POSSIBLE_GFLAG_LIB_DIRS}
  /usr/local/gflags/lib
  /usr/local/lib
  /usr/lib
  )

# SEARCH FOR GFLAG LIBRARIES.
############################
find_path(GFLAG_INCLUDE_DIR gflags.h
${POSSIBLE_GFLAG_INCLUDE_DIRS})

find_library(
  GFLAG_LIB
  NAMES ${GFLAG_LIB_NAMES}
  HINTS ${POSSIBLE_GFLAG_LIB_DIRS}
  )

# Did we find the include directory?
string(
  COMPARE NOTEQUAL
  "GFLAG_INCLUDE_DIR-NOTFOUND"
  ${GFLAG_INCLUDE_DIR} # Value set to GFLAG_INCLUDE_DIR-
NOTFOUND if not found.
  GFLAG_INCLUDE_DIR_FOUND
  )
```

```
# Did we find the library?
string(
  COMPARE NOTEQUAL
  "GFLAG_LIB-NOTFOUND"
  ${GFLAG_LIB} # Value set to `GFLAG_LIB-NOTFOUND` if not
found.
  GFLAG_LIB_FOUND
  )

# GFLAG is considered "found" if we've both an include
directory and an GFLAG binary.
if ("${GFLAG_LIB_FOUND}" AND
"${GFLAG_INCLUDE_DIR_FOUND}")
  set(GFLAG_LIBS ${GFLAG_LIB})
  set(GFLAG_FOUND 1)
else ("${GFLAG_LIB_FOUND}" AND
"${GFLAG_INCLUDE_DIR_FOUND}")
  set(GFLAG_FOUND 0)
endif ("${GFLAG_LIB_FOUND}" AND
"${GFLAG_INCLUDE_DIR_FOUND}")

# Results.
if (GFLAG_FOUND)
  if (NOT GFLAG_FIND_QUIETLY)
    message(STATUS "Found GFLAG headers:
${GFLAG_INCLUDE_DIR}")
    message(STATUS "Found GFLAG library: ${GFLAG_LIBS}")
  endif (NOT GFLAG_FIND_QUIETLY)
else (GFLAG_FOUND)
  if (Gflag_FIND_REQUIRED)
    message(FATAL_ERROR "Could not find GFLAG library")
  endif (Gflag_FIND_REQUIRED)
endif (GFLAG_FOUND)

# (Deprecated declarations.)
set(NATIVE_GFLAG_INCLUDE_PATH ${GFLAG_INCLUDE_DIR} )
```

```
get_filename_component(NATIVE_GFLAG_LIB_PATH ${GFLAG_LIB}
PATH)

# Export libraries variables.
mark_as_advanced(
  GFLAG_LIB
  GFLAG_INCLUDE_DIR
  )
```

查找到的结果，在最终编译时分别追加到${TARGET_INCLUDE_DIRS}、${TARGET_LIB_DIRS}、${TARGET_LIBS}变量下。

cmake中使用源码构建

使用源码构建具体的过程参考，第一章CMake实践指南，这里要注意的是使用源码构建之前，可以单独源码编译一下对应的代码库，确定下编译时需要的一些参数和配置信息。 3rdparty/CMakeLists.txt

```
include(ExternalProject)

# Define build/patch/configure commands for third-party
libs.
##########################################################
####
if (NOT WIN32)
  set(GLOG_CONFIG_CMD  ${GLOG_ROOT}/src/../configure --
prefix=${GLOG_LIB_ROOT})
  set(GLOG_BUILD_CMD   make libglog.la)
  set(GLOG_INSTALL_CMD make install)
  PATCH_CMD(${3RDPARTY_SRC}/glog-0.3.3.patch
GLOG_PATCH_CMD)

  set(GFLAGS_BUILD_CMD   make)
  set(GFLAGS_INSTALL_CMD make install)
```

```
elseif (WIN32)
  set(GLOG_INSTALL_CMD ${CMAKE_NOOP})

endif (NOT WIN32)

# Third-party libraries. Tell the build system how to
pull in and build third-
# party libraries at compile time, using the
ExternalProject_Add macro.
#############################################################
#####################
ExternalProject_Add(
  ${GLOG_TARGET}
  PREFIX            ${GLOG_CMAKE_ROOT}
  CMAKE_ARGS        -DBUILD_SHARED_LIBS=OFF -
DCMAKE_CXX_FLAGS_DEBUG=${CMAKE_CXX_FLAGS_DEBUG}
  PATCH_COMMAND     ${GLOG_PATCH_CMD}
  CONFIGURE_COMMAND ${GLOG_CONFIG_CMD}
  BUILD_COMMAND     ${GLOG_BUILD_CMD}
  INSTALL_COMMAND   ${GLOG_INSTALL_CMD}
  URL               ${GLOG_URL}
  DOWNLOAD_NAME     glog-${GLOG_VERSION}.tar.gz
)

ExternalProject_Add(
  ${GFLAGS_TARGET}
  PREFIX            ${GFLAGS_CMAKE_ROOT}
  BUILD_COMMAND     ${GFLAGS_BUILD_CMD}
  INSTALL_COMMAND   ${GFLAGS_INSTALL_CMD}
  URL               ${GFLAGS_URL}
  DOWNLOAD_NAME     gflags-${GFLAGS_VERSION}.tar.gz
)
```

3rdparty/cmake/3rdpartyConfigure.cmake

> 第三方库基本信息配置，第三方库编译信息信息以及最终编译目标库的头文件地址、链接库地址信息、连接库名称、第三方依赖库信息等。

```
message(STATUS
"*******************************************************
****")
message(STATUS "******************3rd party
Configure**********************")
message(STATUS
"*******************************************************
****")

# DEFINE DIRECTORY STRUCTURE FOR THIRD-PARTY LIBS.
#################################################
set(3RDPARTY_SRC ${CMAKE_SOURCE_DIR}/3rdparty)
set(3RDPARTY_BIN ${CMAKE_BINARY_DIR}/3rdparty)

# DEPENDENCIES
#
# Downloads, configures, and compiles the third-party
libraries
#################################################################
######################
if (NOT WIN32)
  EXTERNAL("glog" ${GLOG_VERSION} "${3RDPARTY_BIN}")
  EXTERNAL("gflags" ${GFLAGS_VERSION} "${3RDPARTY_BIN}")
elseif (WIN32)
  # Glog 0.3.3 does not compile out of the box on
Windows. Therefore, we
  # require 0.3.4.
  EXTERNAL("glog" "0.3.4" "${3RDPARTY_BIN}")

  # NOTE: We expect cURL and zlib exist on Unix (usually
pulled in with a
  # package manager), but Windows has no package manager,
so we have to go
```

```
  # get it.
  EXTERNAL("curl" ${CURL_VERSION} "${3RDPARTY_BIN}")

  EXTERNAL("zlib" ${ZLIB_VERSION} "${3RDPARTY_BIN}")
endif (NOT WIN32)

# Intermediate convenience variables for oddly-structured
directories.
set(GLOG_LIB_ROOT     ${GLOG_ROOT}-lib/lib)
set(GFLAGS_LIB_ROOT   ${GFLAGS_ROOT}-lib/lib)

# Define sources of third-party dependencies.
################################################
set(UPSTREAM_URL ${3RDPARTY_DEPENDENCIES})
set(REBUNDLED_DIR ${CMAKE_CURRENT_SOURCE_DIR})

if (REBUNDLED)
  set(GLOG_URL        ${REBUNDLED_DIR}/3rdparty/glog-
${GLOG_VERSION}.tar.gz)
  set(GFLAGS_URL      ${REBUNDLED_DIR}/3rdparty/gflags-
${GFLAGS_VERSION}.tar.gz)
else (REBUNDLED)
  set(GLOG_URL        ${UPSTREAM_URL}/glog-
${GLOG_VERSION}.tar.gz)
  set(GFLAGS_URL      ${UPSTREAM_URL}/gflags-
${GFLAGS_VERSION}.tar.gz)
else (REBUNDLED)
endif (REBUNDLED)

# Binary lib
# find_package(Gflag REQUIRED)

# Convenience variables for `lib` directories of built
third-party dependencies.
if (WIN32)
  set(GLOG_LIB_DIR        ${GLOG_ROOT}-
```

```
build/${CMAKE_BUILD_TYPE})
  set(GFLAGS_LIB_DIR      ${GFLAGS_ROOT}-build/lib)
else (WIN32)
  set(GLOG_LIB_DIR        ${GLOG_LIB_ROOT}/lib)
  set(GFLAGS_LIB_DIR      ${GFLAGS_ROOT}-build/lib)
endif (WIN32)

# Convenience variables for "lflags", the symbols we pass
to CMake to generate
# things like `-L/path/to/glog` or `-lglog`.
set(GLOG_LFLAG        glog)
set(GFLAGS_LFLAG      gflags)

# Convenience variables for include directories of third-
party dependencies.
if (WIN32)
  set(GLOG_INCLUDE_DIR     ${GLOG_ROOT}/src/windows)
  set(GFLAGS_INCLUDE_DIR   ${GFLAGS_ROOT}-build/include)
else (WIN32)
  set(GLOG_INCLUDE_DIR     ${GLOG_LIB_ROOT}/include)
  set(GFLAGS_INCLUDE_DIR   ${GFLAGS_ROOT}-build/include)
endif (WIN32)

# DEFINE LIBRARY DEPENDENCIES. Tells the process library
build targets
# download/configure/build all third-party libraries
before attempting to build.
############################################################
#######################
set(TARGET_DEPENDENCIES
  ${TARGET_DEPENDENCIES}
  ${GLOG_TARGET}
  ${GFLAGS_TARGET}
)

# Target lib
```

```
set(TARGET_LIBS
  ${TARGET_LIBS}
  ${GLOG_LFLAG}
  ${GFLAGS_LFLAG}
)

# Target include dirs
set(TARGET_INCLUDE_DIRS
  ${TARGET_INCLUDE_DIRS}
  ${GLOG_INCLUDE_DIR}
  ${GFLAGS_INCLUDE_DIR}
)

# Target include lib dirs
set(TARGET_LIB_DIRS
  ${TARGET_LIB_DIRS}
  ${GLOG_LIB_DIR}
  ${GFLAGS_LIB_DIR}
)

message(STATUS "INCLUDE DIR: ${TARGET_INCLUDE_DIRS}")
message(STATUS "TARGET LIB DIR: ${TARGET_LIB_DIRS}")
message(STATUS "TARGET_DEPENDENCIES:
${TARGET_DEPENDENCIES}")
message(STATUS "TARGET_LIBS: ${TARGET_LIBS}")

message(STATUS
"**************************************************
****")
```

参考资料：

> https://gflags.github.io/gflags/
>
> https://github.com/schuhschuh/gflags/releases

日志处理

glog库使用指南

Google Glog,是一个C++语言的应用级日志记录框架，提供了C++风格的流操作和各种助手宏。

使用入门

使用日志文件时，一般会有以下几个最基本的要求，第一要支持不同级别的日志输出，同时可以指定日志输出的目录；第二对日志的输出格式要规范，便于阅读分析和Debug。

glog提供的日志级别有INFO, WARNING, ERROR, FATAL，分别对应数字 0, 1, 2, 3对应级别的日志打印在对应级别的日志文件中。并且高级别的日志同时打印在本级别和低级别中。例如 INFO中会有WARNING级别的输出。

日志文件默认输出在“/tmp/”目录下，可以在程序中设置，FLAGS_log_dir值修改。日志文件名称格式：...log.... 例如：
hello_world.example.com.name.log.INFO.20141225-222411.10474

glog中的常用参数说明：

```
FLAGS_log_dir            日志输出目录
FLAGS_v                  自定义VLOG(m)时，m值小于此处设置值的语句才有输出
FLAGS_max_log_size       每个日志文件最大大小（MB级别）
FLAGS_minloglevel        输出日志的最小级别，即高于等于该级别的日志都将输出。
```

基本函数:

```
LOG_IF(INFO, num_cookies > 10) << "Got lots of cookies";
```

示例说明

首先安装对应的开发库，以ubuntu14.04为例

```
sudo apt-get install libgoogle-glog-dev
```

main

```
#include <iostream>
#include <cmath>

#include <glog/logging.h>

#include "demo_config.h"

#include "./option/option.h"

using namespace std;

int main(int argc, char *argv[]) {

    mainOption(argc, argv);
    google::ParseCommandLineFlags (&argc, &argv, true);
    google::InitGoogleLogging(argv[0]);

    //日志输出目录，可以从命令行传入  --log_dir=./log
    FLAGS_log_dir = "./log";


    Log(INFO) << argv[0] << " Version is " <<
HELLOWORLD_VERSION_MAJOR << "." <<
            HELLOWORLD_VERSION_MINOR << endl;
    LOG(INFO) << FLAGS_port << endl;

    google::ShutdownGoogleLogging();
    google::ShutDownCommandLineFlags();
    return 0;
}
```

编译执行，链接库中添加glog库

```
target_link_libraries (${PROJECT_NAME} glog)
```

单元测试

gtest

gtest是Google公司发布的一个开源C/C++单元测试框架，已被应用于多个开源项目及Google内部项目中，知名的例子包括Chrome Web浏览器、LLVM编译器架构、Protocol Buffers数据交换格式及工具等。

优秀的C/C++单元测试框架并不算少，相比之下gtest仍具有明显优势。与CppUnit比，gtest需要使用的头文件和函数宏更集中，并支持测试用例的自动注册。与CxxUnit比，gtest不要求Python等外部工具的存在。与Boost.Test比，gtest更简洁容易上手，实用性也并不逊色。Wikipedia给出了各种编程语言的单元测试框架列表: http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

gtest源码地址：https://github.com/google/googletest

使用入门

gtest中的测试都是通过测试宏来定义的，使用的时候只需要定义自己的TEST宏，宏的定义格式如下：

```
#include <gtest/gtest.h>

TEST(FooTest, HandleNoneZeroInput) {
    EXPECT_EQ(2, Foo(4, 10));
    EXPECT_EQ(6, Foo(30, 18));
}
```

使用TEST这个宏，它有两个参数，官方的对这两个参数的解释为：[TestCaseName，TestName]，也就是测试用例名和具体的测试名。 对检查点的检查，我们上面使用到了EXPECT*EQ*这个宏，这个宏用来比较两个数字是否相等。*Google*还包装了一系列*EXPECT* 和*ASSERT_*的宏，而EXPECT系列和ASSERT系列的区别是：

1. EXPECT_* 失败时，案例继续往下执行。
2. ASSERT_* 失败时，直接在当前函数中返回，当前函数中ASSERT_*后面的语句将不会执行。

gtest中提供的测试宏有：

- 布尔值检查

```
ASSERT_TRUE(condition);
EXPECT_TRUE(condition);
ASSERT_FALSE(condition);
EXPECT_FALSE(condition);
```

- 数值类型检查

```
ASSERT_EQ(val1, val2);    EXPECT_EQ(val1, val2);
val1 == val2
ASSERT_NE(val1, val2);    EXPECT_NE(val1, val2);
val1 != val2
ASSERT_LT(val1, val2);    EXPECT_LT(val1, val2);
val1 < val2
ASSERT_LE(val1, val2);    EXPECT_LE(val1, val2);
val1 <= val2
ASSERT_GT(val1, val2);    EXPECT_GT(val1, val2);
val1 > val2
ASSERT_GE(val1, val2);    EXPECT_GE(val1, val2);
val1 >= val2
```

- 字符串检查

```
ASSERT_STREQ(str1, str2);   EXPECT_STREQ(str1, str2);
ASSERT_STRNE(str1, str2);    EXPECT_STRNE(str1,
str2);
ASSERT_STRCASEEQ(expected_str, actual_str);
EXPECT_STRCASEEQ(expected_str, actual_str);
ASSERT_STRCASENE(str1, str2);
EXPECT_STRCASENE(str1, str2);
```

> 注意：STREQ和STRNE同时支持char和*wchar_t*类型的，*STRCASEEQ*和*STRCASENE*却只接收char*

- 浮点检测

```
ASSERT_FLOAT_EQ(expected, actual);
EXPECT_FLOAT_EQ(expected, actual);    the two float
values are almost equal
ASSERT_DOUBLE_EQ(expected, actual);
EXPECT_DOUBLE_EQ(expected, actual);
```

- 异常检查

```
ASSERT_THROW(statement, exception_type);
EXPECT_THROW(statement, exception_type);    statement
throws an exception of the given type
ASSERT_ANY_THROW(statement);
EXPECT_ANY_THROW(statement);    statement throws an
exception of any type
ASSERT_NO_THROW(statement);
EXPECT_NO_THROW(statement);    statement doesn't
throw any exception
```

- 类型检查

```
template <typename T> class FooType {
public:
    void Bar() { testing::StaticAssertTypeEq<int, T>(); }
};

TEST(TypeAssertionTest, Demo)
{
    FooType<bool> fooType;
    fooType.Bar();
}
```

事件机制

gtest提供了多种事件机制，非常方便我们在案例之前或之后做一些操作。总结一下gtest的事件一共有3种：

1. 全局的，所有案例执行前后。
2. TestSuite级别的，在某一批案例中第一个案例前，最后一个案例执行后。
3. TestCase级别的，每个TestCase前后执行。

给每一个test用例添加初始启动前和完成后的执行的方法，这个和Java中的Junit就很类似了。例如：

```
class FooCalcTest: public testing::Test {
protected:
    virtual void SetUp() {
       str = "hello world";
    }

    virtual void TearDown() {
        str = "";
    }

    string str;
};

TEST_F(FooCalcTest, testdemotest1) {

    ASSERT_STREQ("hello world", str.c_str());
}
```

使用示例

安装gtest开发库

```
sudo apt-get install libgtest-dev
```

> 安装完成后，这里只安装了程序的头文件，并没有相关的库代码。库代码需要自己编译安装。

```
  cd /usr/src/gtest
  mkdir build
  cd build
  cmake ..
  make
```

example1:

```
# include <gtest/gtest.h>

int Gcd( int a, int b) //计算最大公约数 {

  return 0 == b ? a : Gcd(b, a % b);
}

TEST(GcdTest, IntTest) {
  EXPECT_EQ( 1 , Gcd( 2 , 5 ));
  EXPECT_EQ( 2 , Gcd( 2 , 5 ));
  EXPECT_EQ( 2 , Gcd( 2 , 4 ));
  EXPECT_EQ( 3 , Gcd( 6 , 9 ));
}

int main( int argc, char **argv) {
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

g++ testgcd.cpp -lgtest_main -lgtest -lpthread

example2: 结合CMake 首先给出示例代码的目录结构：

```
gtestdemo
  build
  cmake
    interna_utils.cmake
  include
  src
    option
      option.hpp
    testdemo
      CMakeLists.txt
      testdemo.cpp
      testdemo.h
    main.cpp
  test
    testtestdemo.cc
  CMakeLists.txt
  LICENSE
  demo_config.h.in
```

使用时引入了interna_utils.cmake宏文件，该文件在google gtest工程目录下，读者可以去google gtest的github上下载该文件。整体的代码结构包括了src, include, test三个包， test用于测试存放测试用例，src, include下是源码。cmake中保存用到的cmake宏文件。

编写root下的CMakeLists.txt文件，引入测试相关的宏，以及添加make test target:

```
# Macro
###################################
# Define helper functions and macros used by Google Test.
include(cmake/internal_utils.cmake)
config_compiler_and_linker()  # Defined in
internal_utils.cmake.
...

# Test
###################################
## Unit Test
if (build_tests)
  # This must be set in the root directory for the tests
to be run by
  # 'make test' or ctest.
  enable_testing()


############################################################
###
  # C++ tests built with standard compiler flags.
  set(ld_list gtest gtest_main pthread testdemo)
  #list(APPEND ld_list pthread)
  cxx_test(testtestdemo  "${ld_list}")

endif()

...
```

上例中 cxx_test(testtestdemo "${ld_list}")会自动创建一个test target testtestdemo，这个测试用例对应的就是test/testtestdemo.cc源文件，编译过程中要指定使用的动态库文件，这里包括了gtest相关的库和我们测试代码的动态库。其他相关的内容和 上一节的代码一致。

编译并运行测试代码

```
cd build
cmake ..
make
make test   or ctest
```

高级用法

- 死亡测试

  通常在测试过程中，需要考虑各种各样的输入，有的输入可能直接导致程序崩溃，这时我们就需要检查程序是否按照预期的方式挂掉，这也就是所谓的“死亡测试”。

  ```
  ASSERT_DEATH(statement, regex);
  EXPECT_DEATH(statement, regex);
  ASSERT_EXIT(statement, predicate, regex`);
  EXPECT_EXIT(statement, predicate, regex`);
  ```

  示例

  ```
  void Foo() {
      int *pInt = 0;
      *pInt = 42 ;
  }
  TEST(FooDeathTest, Demo) {
      EXPECT_DEATH(Foo(), "");
  }
  ```

测试用例指南

摘自：http://www.cnblogs.com/coderzh/archive/2010/01/09/beautiful-testcase.html
如何在GTest框架下写出优美的测试案例，有几个比较好的标准：

- 案例的层次结构一定要清晰

```
TEST(TestFoo, JustDemo)
{
  GetTestData(); // 获取测试数据

  CallSUT(); // 调用被测方法

  CheckSomething(); // 检查点验证
}
```

- 案例的检查点一定要明确
- 案例失败时一定要能精确的定位问题
- 案例执行结果一定要稳定
- 案例执行的时间一定不能太长
- 案例一定不能对测试环境造成破坏
- 案例一定独立，不能与其他案例有先后关系的依赖
- 案例的命名一定清晰，容易理解

参考：

> http://developer.51cto.com/art/201108/285290.htm
>
> http://www.cnblogs.com/coderzh/archive/2009/04/06/1426755.html
>
> https://github.com/google/googletest
>
> http://www.cnblogs.com/coderzh/archive/2010/01/09/beautiful-testcase.html

gmock

GoogleMock可以将模块之间的接口mock起来，模拟交互过程。

```
//测试对象, demo/demo_math.h
class DemoMath {
public:
    int addTen(int base) {
        return base + 10;
    }
```

```
    int power(int base, int m) {
        int tmp = 1;
        for(int i = 0; i < m; i++) {
            tmp = tmp * base;
        }
        return tmp;
    }

    int fun(int a) {
        return a;
    }
};

//mock_demo_math.cpp
#include <gmock/gmock.h>
#include <gtest/gtest.h>

#include "demo/demo_math.h"

class MockDemoMath : public DemoMath {
public:
  MockDemoMath();
  virtual ~MockDemoMath();

  MOCK_METHOD1(addTen, int(int base));

  MOCK_METHOD2(power, int(int base, int m));
};


TEST(MockDemoMath, DemoMath) {

  DemoMath demoMath;

  EXPECT_EQ(11, demoMath.addTen(1));
```

```
  EXPECT_EQ(8, demoMath.power(2, 3));

};
```

TDD(Test Driven Development)

如何在开发中也能使用单元测试工具高效地开发呢。很多时候，大家都会又这样的习惯，每写一个新的类或方法时都会进行测试，而且每一次测试时往往都是只针对新的类或方法来测试，而不会测试所有的类或方法。这就引出了一个新的问题，在一次测试中如何只做特定类或者方法的测试。

gtest中提供了测试用例过滤的方法，在运行时加上--gtest_filter参数即可。例如针对以下测试用例：

```
TEST(case2,  functionA_test) {
  .....
}

TEST(case2, functionB_test){
  ....
}
```

```
--gtest_filter=case2.*
```

> case2为测试用例名，*表示所有测试方法

性能测试

性能测试通常是对软件计算用时进行测试和分析，这里主要针对计算时间和基准性能测试给出一个服务的计算能力和服处理务能力。

gprof

gprof 可以为 Linux平台上的程序精确分析性能瓶颈。gprof精确地给出函数被调用的时间和次数，给出函数调用关系。gprof能够让你知道你的代码哪些地方是比较耗时的，哪些函数是被调用次数很多的，并且能够让你一目了然的看到函数与函数之间的调用关系。

gprof是gcc/g++编译器支持的一种性能诊断工具。只要在编译时加上-pg选项，编译器就会在编译程序时在每个函数的开头加一个mcount函数调用，在每一个函数调用之前都会先调用这个mcount函数，在mcount中会保存函数的调用关系图和函数的调用时间和被调次数等信息。最终在程序退出时保存在gmon.out文件中，需要注意的是程序必须是正常退出或者通过exit调用退出，因为只要在exit（）被调用时才会触发程序写gmon.out文件。

gprof 用户手册网站 http://sourceware.org/binutils/docs-2.17/gprof/index.html 。Gprof 是GNU gnu binutils工具之一，默认情况下linux系统当中都带有这个工具。

使用指南

gprof的使用方法主要以下三步：

```
会用-pg参数编译程序
运行程序，并正常退出，回升呈gmon.out文件
使用gprof来分析gmon.out文件
```

gprof工具的基本使用方式：

```
gprof –b a.out gmon.out >report.txt
```

gprof的参数信息说明：

```
-b 不再输出统计图表中每个字段的详细描述。
-p 只输出函数的调用图（Call graph的那部分信息）。
-q 只输出函数的时间消耗列表。
-e Name 不再输出函数Name 及其子函数的调用图（除非它们有未被限制的其它父函数）。可以给定多个 -e 标志。一个 -e 标志只能指定一个函数。
-E Name 不再输出函数Name 及其子函数的调用图，此标志类似于 -e 标志，但它在总时间和百分比时间的计算中排除了由函数Name 及其子函数所用的时间。
-f Name 输出函数Name 及其子函数的调用图。可以指定多个 -f 标志。一个 -f 标志只能指定一个函数。
-F Name 输出函数Name 及其子函数的调用图，它类似于 -f 标志，但它在总时间和百分比时间计算中仅使用所打印的例程的时间。可以指定多个 -F 标志。一个 -F 标志只能指定一个函数。-F 标志覆盖 -E 标志。
-z 显示使用次数为零的例程（按照调用计数和累积时间计算）。
```

gprof生成的report.txt基本信息说明:

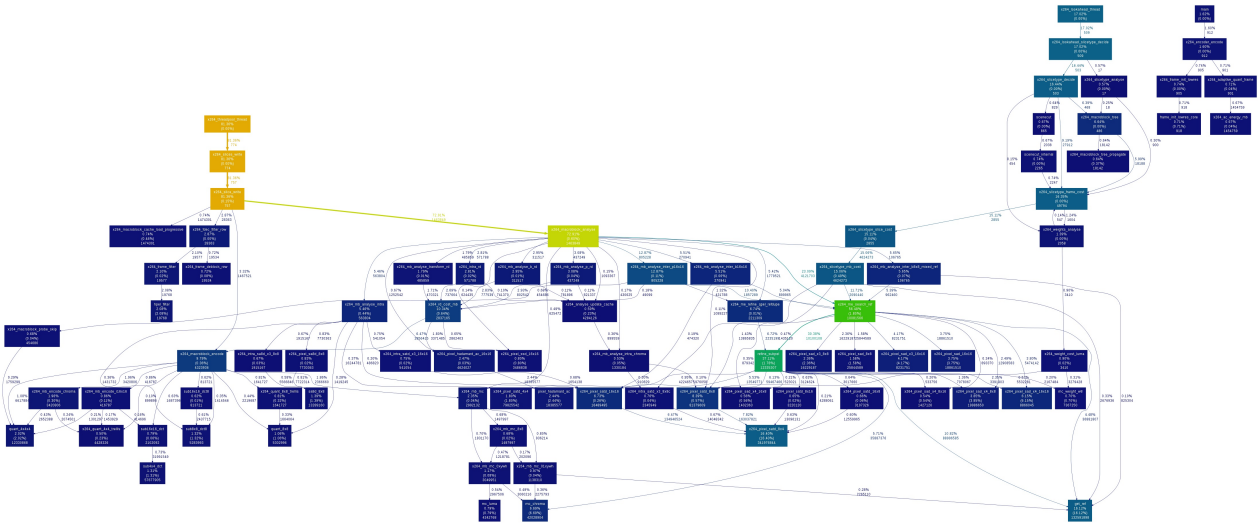
name	%time	Cumulative seconds	Self Seconds	Calls Self	TS/call
函数名	该函数消耗时间占程序所有时间百分比	程序的累积执行时间	该函数本身执行时间	函数被调用次数	函数平均执行时间

> 注意： 程序的累积执行时间只是包括gprof能够监控到的函数。工作在内核态的函数和没有加-pg编译的第三方库函数是无法被gprof能够监控到的，（如sleep（）等） Gprof 的具体参数可以 通过 man gprof 查询。

当然有图形话的工具来生成调用关系图，并给出整个函数的使用时间和调用次数。

```
生成图片
   //安装画图工具
   sudo apt-get install graphviz
   gprof a.out  | ./gprof2dot.py -n0 -e0 | dot -Tpng -o
output.png
   python gprof2dot.py prof.txt | dot -Tpng -o prof.png
```

下图是在对x264源码加速分析时的一个示例：



Q&A

- 共享库的支持

  对于代码剖析的支持是由编译器增加的，因此如果希望从共享库中获得剖析信息，就需要使用 -pg 来编译这些库。提供已经启用代码剖析支持而编译的 C 库版本（libc_p.a）。如果需要分析系统函数（如libc库），可以用 –lc_p替换-lc。这样程序会链接libc_p.so或libc_p.a。这非常重要，因为只有这样才能监控到底层的c库函数的执行时间，（例如memcpy()，memset()，sprintf()等）。

  ```
  gcc example1.c –pg -lc_p -o example1
  注意要用ldd ./example | grep libc来查看程序链接的是libc.so
  还是libc_p.so
  ```

- 关于用户时间与内核时间:

gprof 的最大缺陷：它只能分析应用程序在运行过程中所消耗掉的用户时间，无法得到程序内核空间的运行时间。通常来说，应用程序在运行时既要花费一些时间来运行用户代码，也要花费一些时间来运行“系统代码”，例如内核系统调用sleep()。

有一个方法可以查看应用程序的运行时间组成，在 time 命令下面执行程序。这个命令会显示一个应用程序的实际运行时间、用户空间运行时间、内核空间运行时间。

```
如 time ./program
输出：
real    2m30.295s
user    0m0.000s
sys     0m0.004s
```

> 注意事项
>
> 1. g++在编译和链接两个过程，都要使用-pg选项。
> 2. 只能使用静态连接libc库，否则在初始化*.so之前就调用profile代码会引起“segmentation fault”，解决办法是编译时加上-static-libgcc或-static。
> 3. 如果不用g++而使用ld直接链接程序，要加上链接文件/lib/gcrt0.o，如ld -o myprog /lib/gcrt0.o myprog.o utils.o -lc_p。也可能是gcrt1.o
> 4. 要监控到第三方库函数的执行时间，第三方库也必须是添加 –pg 选项编译的。
> 5. gprof只能分析应用程序所消耗掉的用户时间.
> 6. 程序不能以demon方式运行。否则采集不到时间。（可采集到调用次数）
> 7. 首先使用 time 来运行程序从而判断 gprof 是否能产生有用信息是个好方法。
> 8. 如果 gprof 不适合您的剖析需要，那么还有其他一些工具可以克服 gprof 部分缺陷，包括 OProfile 和 Sysprof。
> 9. gprof对于代码大部分是用户空间的CPU密集型的程序用处明显。对于大部分时间运行在内核空间或者由于外部因素（例如操作系统的 I/O 子系统过载）而运行得非常慢的程序难以进行优化。
> 10. gprof 不支持多线程应用，多线程下只能采集主线程性能数据。原因是gprof采用ITIMER_PROF信号，在多线程内只有主线程才能响应该信号。但是有一个简单的方法可以解决这一问题：http://sam.zoy.org/writings/programming/gprof.html
> 11. gprof只能在程序正常结束退出之后才能生成报告（gmon.out）。 a) 原因： gprof通过在atexit()里注册了一个函数来产生结果信息，任何非正常退出都不会执行atexit()的动作，所以不会产生gmon.out文件。 b) 程序可从main函数中正常退出，或者通过系统调用exit()函数退出。

benchmark

https://github.com/google/benchmark

Benchmark测试在计算机领域中最广泛和最成功的应用是性能测试，主要测试响应时间、传输速率和吞吐量等。

负载测试

其他测试

对于系统级服务，特别是商业中的使用，还要考虑到可靠性可用性测试。

cpack

使用cpack打包项目工程，并生成安装脚步。

```
# Package
#############################################################
# build a CPack driven installer package
include (CPack)
include (InstallRequiredSystemLibraries)
set (CPACK_RESOURCE_FILE_LICENSE
     "${CMAKE_CURRENT_SOURCE_DIR}/LICENSE")
set (CPACK_PACKAGE_VERSION_MAJOR "${VERSION_MAJOR}")
set (CPACK_PACKAGE_VERSION_MINOR "${VERSION_MINOR}")
```

C++项目starter

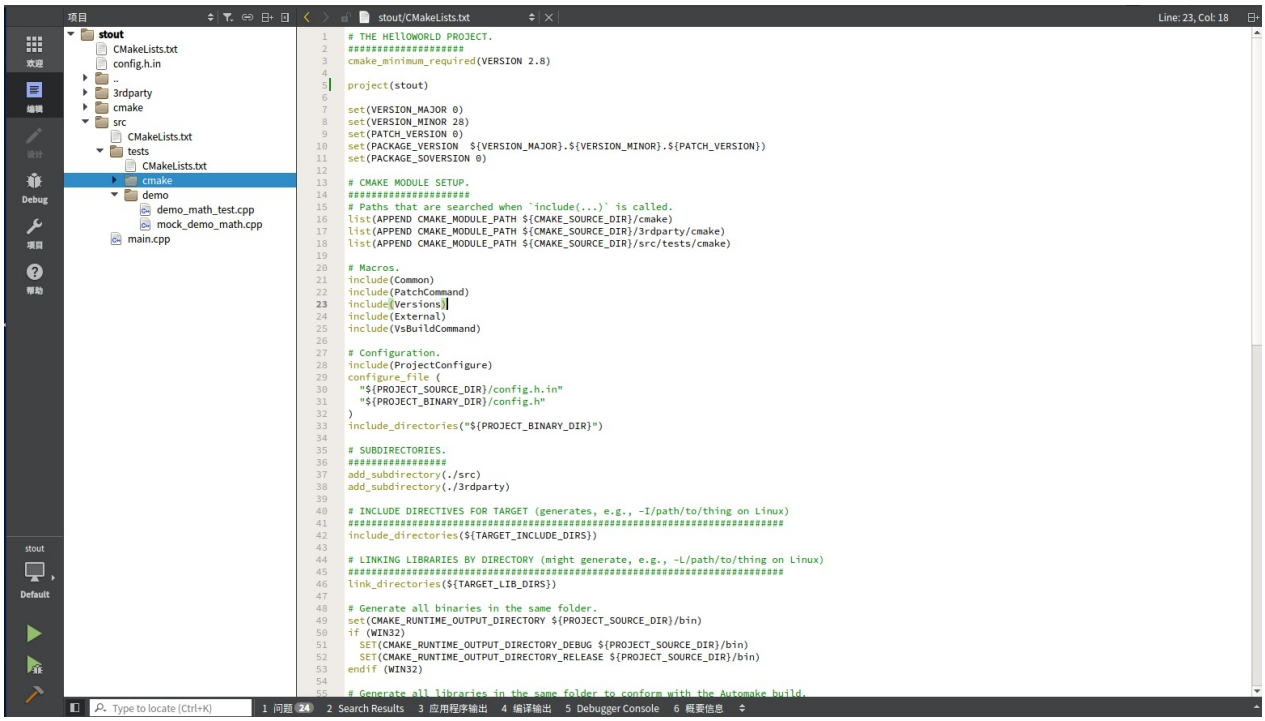
本节综合cmake、ctest、cpack等工具，给出一个完整的工程项目结构，并结合glog、gmock、gtest、stout库给出创建cmake工程项目的完整过程。本章前几节中在介绍各个库的时候没有使用源码的方式，而是直接使用在线安装方式。在本节中使用源码的方式来说明，使用源码优点在于后续维护、代码移植能够更方便一些。建议在创建项目中，对一些关键的类库，使用源码来编译和维护。

https://github.com/3rdparty 中维护了mesos项目中使用的第三方库以及一些常用的代码库，值得大家在C++工程项目中借鉴使用。

starter

IDE

综合多种IDE的使用来看，qt对cmake编译系统的支持是最好的，推荐大家在开发c++项目中使用。



使用qt来构建cmake项目时，在工具栏->选项->构建和运行->构建套件手动配制CMAKE。

项目管理

项目开发中，对项目的开发进度进行追踪管理、以及对外部反馈的快速响应，都能极大地促进项目开发过程。 例如mesos项目实施过程：https://issues.apache.org/jira/browse/MESOS ，使用gitlab私有git仓库时也可以创建对应项目的里程碑、问题、wiki等功能。

第三章 核心库介绍

通信库libprocess

序列化protobuf

Protocol Buffers 是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 RPC 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python 三种语言的 API。

reference:

https://github.com/google/protobuf

并发c++11 concurrency

安全

认证授权

http parser

简介

http请求客户端库，用于http请求的编码和解码处理。

特性：

```
无第三方依赖
可以处理持久消息(keep-alive)
支持解码chunk编码的消息
支持Upgrade协议升级(如无例外就是WebSocket)
可以防御缓冲区溢出攻击
```

解析器可以处理以下类型的HTTP消息：

```
头部的字段和值
Content-Length
请求方法
返回的HTTP代码
Transfer-Encoding
HTTP版本
请求的URL
HTTP消息主体
```

源码地址 ：https://github.com/nodejs/http-parser

学习参考代码 ：https://github.com/nodejs/http-parser/blob/master/test.c

使用说明

http parser处理的基本流程：

```
/***************** 1 ******************/
// 设置回调
http_parser_settings settings;
settings.on_url = my_url_callback;
settings.on_header_field = my_header_field_callback;
/* ... */

/***************** 2 ******************/
http_parser *parser = malloc(sizeof(http_parser));
// 初始化解析器
http_parser_init(parser, HTTP_REQUEST);
// parser调用者的信息保存在 parser->data

/***************** 3 ******************/
// 开始解析
// @parser 解析器对象
// @&settings 解析时的回调函数
// @buf 要解析的数据
// @receved 要解析的数据大小
// @return nparsed已经解析的数据大小
nparsed = http_parser_execute(parser, &settings, buf,
recved);

/***************** 4 ******************/
// 回收内存
 if(parser == NULL) {
   free(parser);
 }
 parser = NULL;
```

下面给出一个完整的示例代码：

```
#include "http_parser.h"
#include <stdlib.h>
#include <assert.h>
```

```
#include <stdio.h>
#include <stdlib.h> /* rand */
#include <string.h>
#include <stdarg.h>

http_parser* parser;

void parser_init (enum http_parser_type type) {
  assert(parser == NULL);
  parser = malloc(sizeof(http_parser));
  http_parser_init(parser, type);
}

void parser_free () {
  assert(parser);
  free(parser);
  parser = NULL;
}

struct message {
  const char *name; // for debugging purposes
  const char *raw;
  enum http_parser_type type;
  enum http_method method;
  int status_code;
  char response_status[MAX_ELEMENT_SIZE];
  char request_path[MAX_ELEMENT_SIZE];
  char request_url[MAX_ELEMENT_SIZE];
  char fragment[MAX_ELEMENT_SIZE];
  char query_string[MAX_ELEMENT_SIZE];
  char body[MAX_ELEMENT_SIZE];
  size_t body_size;
  const char *host;
  const char *userinfo;
  uint16_t port;
  int num_headers;
```

```
  enum { NONE=0, FIELD, VALUE } last_header_element;
  char headers [MAX_HEADERS][2][MAX_ELEMENT_SIZE];
  int should_keep_alive;

  int num_chunks;
  int num_chunks_complete;
  int chunk_lengths[MAX_CHUNKS];

  const char *upgrade; // upgraded body

  unsigned short http_major;
  unsigned short http_minor;

  int message_begin_cb_called;
  int headers_complete_cb_called;
  int message_complete_cb_called;
  int message_complete_on_eof;
  int body_is_final;
};

const message request = {
  .name= "curl get"
  ,.type= HTTP_REQUEST
  ,.raw= "GET /test HTTP/1.1\r\n"
         "User-Agent: curl/7.18.0 (i486-pc-linux-gnu)
libcurl/7.18.0 OpenSSL/0.9.8g zlib/1.2.3
libidn/1.1\r\n"
         "Host: 0.0.0.0=5000\r\n"
         "Accept: */*\r\n"
         "\r\n"
  ,.should_keep_alive= TRUE
  ,.message_complete_on_eof= FALSE
  ,.http_major= 1
  ,.http_minor= 1
  ,.method= HTTP_GET
  ,.query_string= ""
```

```
  ,.fragment= ""
  ,.request_path= "/test"
  ,.request_url= "/test"
  ,.num_headers= 3
  ,.headers=
    { { "User-Agent", "curl/7.18.0 (i486-pc-linux-gnu)
libcurl/7.18.0 OpenSSL/0.9.8g zlib/1.2.3.3 libidn/1.1" }
    , { "Host", "0.0.0.0=5000" }
    , { "Accept", "*/*" }
    }
  ,.body= ""
  };
```

高级用法

boost库

Boost库是一个可移植、提供源代码的C++库，作为标准库的后备，是C++标准化进程的开发引擎之一。

序列化protobuf

protobuf使用自定义的语法格式来定义结构化数据的基本格式，用于消息格式的序列化和反序列化。使用时，先编写对应的proto文件，并将其编译成源码。

基本概念

- .proto文件：数据协议，用于基本的消息通信格式
- modifiers，对应结构话对象属性的更改状态

  required 不可以增加或删除的字段，必须初始化

  optional 可选字段，可删除，可以不初始化

  repeated 可重复字段，生成的是List
- Message，在proto文件里，数据的协议时以Message的形式表现的。
- Build,生成具体的java类时，例如Person.java，同时会存在build方法，用于消息对象的初始化。

使用入门

安装对应开发库

可以通过源码安装，也可以直接安装对应的二进制包。开发者使用时需要安装对应的开发库和proto编译器，用于将对应的proto文件编译成c++/java/python的代码。

```
aptitude search libprotobuf
aptitude search protobufc
sudo apt-get install libprotobuf-dev
sudo apt-get install compiler
```

proto文件

proto文件的编写要遵循特定的语法规范格式，proto文件的基本格式：

```
//License

package mesos;
message message1 {

}

/**
* 这是注释
**/
message message2 {

}
```

message语法

代码风格

3.1 消息与字段名 使用骆驼风格的大小写命名，即单词首字母大写，来做消息名。使用GNU的全部小写，使用下划线分隔的方式定义字段名: message SongServerRequest { required string song_name=1; } 使用这种命名方式得到的名字如下:

```
C++:
    const string& song_name() {...}
    void set_song_name(const string& x) {...}

Java:
    public String getSongName() {...}
    public Builder setSongName(String v) {...}
```

3.2 枚举 使用骆驼风格做枚举名，而用全部大写做值的名字:

```
enum Foo {
    FIRST_VALUE=1;
    SECOND_VALUE=2;
}
```

每个枚举值最后以分号结尾，而不是逗号。

3.3 服务 如果你的 .proto 文件定义了RPC服务，你可以使用骆驼风格:

```
service FooService {
    rpc GetSomething(FooRequest) returns (FooResponse);
}
```

```
/**
 * Describes a command, executed via: '/bin/sh -c value'.
Any URIs specified
 * are fetched before executing the command.  If the
executable field for an
 * uri is set, executable file permission is set on the
downloaded file.
 * Otherwise, if the downloaded file has a recognized
archive extension
 * (currently [compressed] tar and zip) it is extracted
into the executor's
 * working directory. This extraction can be disabled by
setting `extract` to
 * false. In addition, any environment variables are set
before executing
 * the command (so they can be used to "parameterize"
your command).
 */
message CommandInfo {
  message URI {
    required string value = 1;
```

```
    optional bool executable = 2;

    // In case the fetched file is recognized as an
archive, extract
    // its contents into the sandbox. Note that a cached
archive is
    // not copied from the cache to the sandbox in case
extraction
    // originates from an archive in the cache.
    optional bool extract = 3 [default = true];

    // If this field is "true", the fetcher cache will be
used. If not,
    // fetching bypasses the cache and downloads directly
into the
    // sandbox directory, no matter whether a suitable
cache file is
    // available or not. The former directs the fetcher
to download to
    // the file cache, then copy from there to the
sandbox. Subsequent
    // fetch attempts with the same URI will omit
downloading and copy
    // from the cache as long as the file is resident
there. Cache files
    // may get evicted at any time, which then leads to
renewed
    // downloading. See also "docs/fetcher.md" and
    // "docs/fetcher-cache-internals.md".
    optional bool cache = 4;
  }

  repeated URI uris = 1;

  optional Environment environment = 2;
```

```
  // There are two ways to specify the command:
  // 1) If 'shell == true', the command will be launched
via shell
  //        (i.e., /bin/sh -c 'value'). The 'value'
specified will be
  //        treated as the shell command. The 'arguments'
will be ignored.
  // 2) If 'shell == false', the command will be launched
by passing
  //        arguments to an executable. The 'value'
specified will be
  //        treated as the filename of the executable.
The 'arguments'
  //        will be treated as the arguments to the
executable. This is
  //        similar to how POSIX exec families launch
processes (i.e.,
  //        execlp(value, arguments(0), arguments(1),
...)).
  // NOTE: The field 'value' is changed from 'required'
to 'optional'
  // in 0.20.0. It will only cause issues if a new
framework is
  // connecting to an old master.
  optional bool shell = 6 [default = true];
  optional string value = 3;
  repeated string arguments = 7;

  // Enables executor and tasks to run as a specific
user. If the user
  // field is present both in FrameworkInfo and here, the
CommandInfo
  // user value takes precedence.
  optional string user = 5;
}
```

入门示例

编写对应的proto文件

```
package message;
message helloworld
 {
    required int32     id = 1;  // ID
    required string    str = 2;  // str
    optional int32     opt = 3;  //optional field
 }
```

将对应的文件编译成源码

```
基本格式
protoc -I=$SRC_DIR --cpp_out=$DST_DIR
$SRC_DIR/addressbook.proto
具体
g++ test.cpp helloword.pb.cc -lprotobuf
```

高级话题

reference：

http://www.ibm.com/developerworks/cn/linux/l-cn-gpb/

http://blog.csdn.net/menuconfig/article/details/12837173

stout

参考文件：https://github.com/3rdparty/stout

使用时只需要将源码文件放到include文件目录下。部分库依赖第三方库:

- Boost
- Google's glog
- Google's protobuf
- Google's gmock/gtest

stout库包括以下几大类功能：

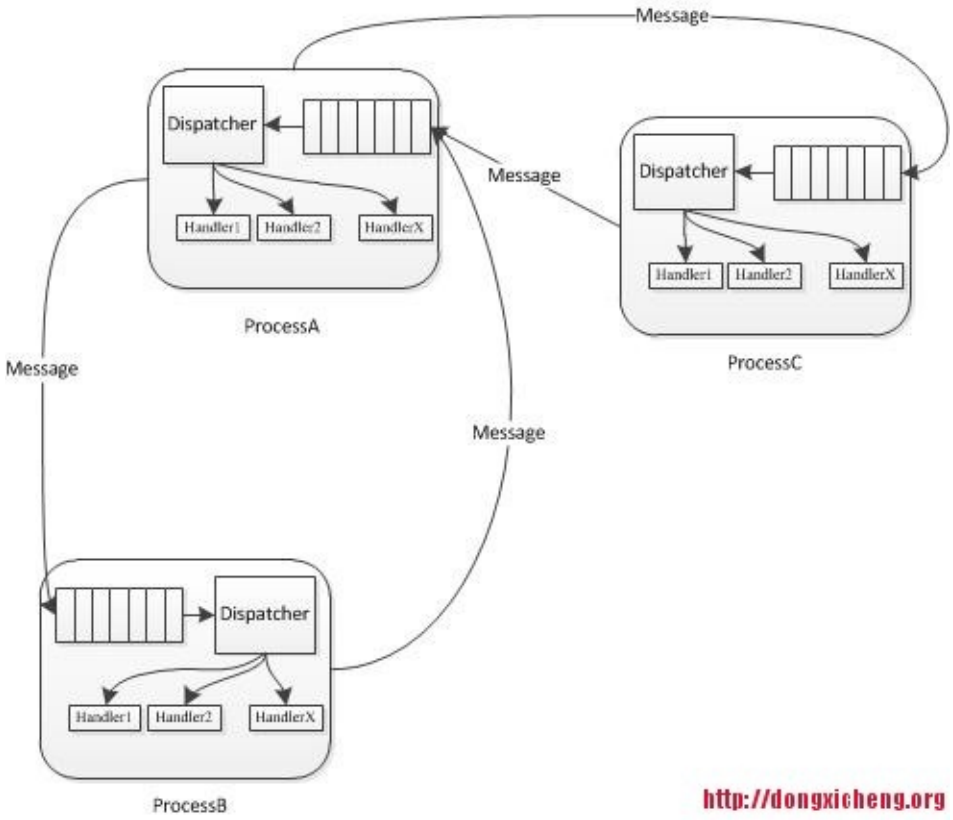
- 基础类库：Duration、Error、None、Nothing、Option、Owned、Result、Try、Stopwatch、UUID
- 集合：cache、hashmap、hashset、multihashmap
- 名空间：fs、gzip、JSON、lambda、net、os、path、protobuf(Requires protobuf)、strings
- 异常：Exceptions
- 其他：copy、EXIT、fatal、gtest、numify、preprocessor、stringify

基础类库

通信库libprocess

Libprocess是一个用c++的库，提供了一个Actor风格消息传递编程模型,利用高效的操作系统事件机制。Libprocess Erlang的流程模型非常相似,包括发送和接收消息的基本构造。

LibProcess采用了基于事件驱动的编程模型，每一个服务（进程）内部实际上运行了一个socket server，而不同服务之间通过消息（事件）进行通信。在一个服务内部，注册了很多消息以及每个消息对应的处理器，一旦它收到某种类型的消息，则会调用相应的处理器进行处理，在处理过程中，可能会产生另外一种消息发送给另一个服务。



基本概念

基本用法

首先给出一个参考实例：摘自官方github example

```
#include <iostream>
#include <sstream>

#include <process/defer.hpp>
#include <process/dispatch.hpp>
#include <process/future.hpp>
#include <process/http.hpp>
#include <process/process.hpp>

using namespace process;

using namespace process::http;

using std::string;

class MyProcess : public Process<MyProcess>
{
public:
  MyProcess() {}
  virtual ~MyProcess() {}

  Future<int> func1()
  {
    promise.future().onAny(
        defer([=] (const Future<int>& future) {
          terminate(self());
        }));
    return promise.future();
  }

  void func2(int i)
  {
    promise.set(i);
  }
```

```
  Future<Response> vars(const Request& request)
  {
    string body = "... vars here ...";
    OK response;
    response.headers["Content-Type"] = "text/plain";
    std::ostringstream out;
    out << body.size();
    response.headers["Content-Length"] = out.str();
    response.body = body;
    return response;
  }

  void stop(const UPID& from, const string& body)
  {
    terminate(self());
  }

protected:
  virtual void initialize()
  {
//     route("/vars", &MyProcess::vars);
    route("/vars", [=] (const Request& request) {
        string body = "... vars here ...";
        OK response;
        response.headers["Content-Type"] = "text/plain";
        std::ostringstream out;
        out << body.size();
        response.headers["Content-Length"] = out.str();
        response.body = body;
        return response;
      });

//     install("stop", &MyProcess::stop);
    install("stop", [=] (const UPID& from, const string&
body) {
        terminate(self());
```

```
      });
  }

private:
  Promise<int> promise;
};


int main(int argc, char** argv)
{
  MyProcess process;
  PID<MyProcess> pid = spawn(&process);

  PID<> pid2 = pid;

// ---------------------------------------

//   Future<int> future = dispatch(pid,
&MyProcess::func1);
//   dispatch(pid, &MyProcess::func2, 42);

//   std::cout << future.get() << std::endl;

//   post(pid, "stop");

// ---------------------------------------

//   Promise<bool> p;

//   dispatch(pid, &MyProcess::func1)
//     .then([=, &p] (int i) {
//         p.set(i == 42);
//         return p.future();
//       })
//     .then([=] (bool b) {
//         if (b) {
```

```
//           post(pid, "stop");
//         }
//         return true; // No Future<void>.
//       });

//   dispatch(pid, &MyProcess::func2, 42);

// ------------------------------------

  dispatch(pid, &MyProcess::func1);
  dispatch(pid, &MyProcess::func2, 42);


  wait(pid);
  return 0;
}
```

libprocess中的几个类说明：

（1）Process类
可通过继承该类，实现一个服务。该类主要包含以下几个方法：
1） Install函数
void install(
const std::string& name,
const MessageHandler& handler)
注册名为name的消息处理器。
2） Send函数
void send(
const UPID& to,
const std::string& name,
const char* data = NULL,
size_t length = 0)
向参数to标识的服务发送名为name的消息，该消息中包含数据data，长度为length。
（2）使用ProtobufProcess类

ProtobufProcess类实现了Process类，与Protocal buffer紧密结合，该类主要有以下几个方法：

1） Install函数

使用ProtobufProcess类，可以很容易创建一个处理Protocal buffer类型消息的消息处理器，可通过install函数注册各种消息处理器，比如如果一个新消息中有2个字段（x1，x2），则可这样注册该消息：

```
install<XXXMessage>(  //使用install函数进行注册
&messageHandler,
&x1，&x2);
}
```

XXXMessage的protocal buffer定义如下：

```
message XXXMessage {
required X x1 = 1;
required Y x2 = 2;
}
```

2） send函数

定义如下：

```
void send(const process::UPID& to,
const google::protobuf::Message& message);
```

它的功能是向参数to标识的服务发送message消息，其中to是UPID类型，它包含了服务ip和端口号。

3） reply函数

定义如下：

```
void reply(const google::protobuf::Message& message);
```

功能：向最近发送消息的服务返回message。

（3）全局函数

1） dispatch

存在多种定义方式，一种方式如下：

```
template <typename R, typename T>
Future<R> dispatch(
const PID<T>& pid,
Future<R> (T::*method)(void))
```

将函数分配给pid进程（服务）执行。注意，函数分配给进程后，不一定会马上执行完后，需要需要等待执行完成，可以结合wait函数使用：

```
Future<bool> added = dispatch(slavesManager, add);
added.await();
```

```
if (!added.isReady() || !added.get()) {
…..
}
```

2） spawn

定义如下：

```
UPID spawn(ProcessBase* process, bool manage=false)；
```

启动进程（服务）process，参数manage表示是否启用垃圾收集机制。

入门实例

假设有两个服务Master和Slave，Slave周期性向Master汇报自己的进度，而Master则不定期地向Slave下达任务，采用LibProcess实现如下：

（1）Master类设计

步骤1 继承ProtobufProcess类。让Master类继承LibProcess中的ProtobufProcess类，该类实际上维护了一个socket server，该server可以处理实现注册好的各种Protocal Buffer定义的Message

```
class Master : public ProtobufProcess<Master> {
//……
}
```

步骤2 注册消息处理器。在初始化函数initialize()中（使用install函数）注册一个ReportProgressMessage类型（需使用Protocal buffer定义）的消息处理器，该消息处理器对应的函数是Master::reportProgress，该函数带有一个参数tasks。这样，Master内部的socket server会监听来自外部的各种消息包，一旦发现ReportProgressMessage类型的消息包，则会调用函数reportProgress进行处理。注意，reportProgress函数中的参数必须正好对应ReportProgressMessage消息定义。

```
void initialize() { //继承ProtobufProcess类后，需实现该函数
install<ReportProgressMessage>(  //使用install函数进行注册
&Master::reportProgress,
&LaunchTasksMessage::tasks);
}
```

步骤3 定义ReportProgressMessage消息。使用Protocal Buffer定义ReportProgressMessage消息如下：

```
//master.proto
message ReportProgressMessage {
repeated Task tasks = 3;
}
message Task {
required string name = 1;
required TaskID task_id = 2;
required float progress =3;
}
message TaskID {
required string value = 1;
}
```

步骤4 编写消息处理器reportProgress。

```
void reportProgress(const vector<Task>& tasks) {
for (int i = 0; i < tasks.size(); i++) {
//update tasks[i] information;
}
```

步骤5 编写main函数启动Master。

```
int main(int argc, char** argv)
{
process::initialize(“master”); //初始化一个名为master的进程
Master* master = new Master();
process::spawn(master); //启动master，实际上是一个socket
server
process::wait(master->self());
delete master;
return 0;
}
```

比较全的代码如下：

```
class Master : public ProtobufProcess<Master> //步骤1
{
Master(): ProcessBase(“master”) {}
void initialize() {
install<ReportProgressMessage>(  //步骤2
&Master::reportProgress,
&LaunchTasksMessage::tasks);
```

```
}
void reportProgress(const vector<Task>& tasks) { //步骤4
for (int i = 0; i < tasks.size(); i++) {
//update tasks[i] information;
}
}
}
```

（2）　　Slave类设计

Slave设计与Master类似，具体如下：

```
class Slave : public ProtobufProcess<Slave>
{
Slave(): ProcessBase(“slave”) {}
void initialize() [
install<LaunchTasksMessage>(
&Master::launchTasks,
&LaunchTasksMessage::id,
&LaunchTasksMessage::tasks);
}
void launchTasks(const int id,
const vector<TaskInfo>& tasks) {
for (int i = 0; i < tasks.size(); i++) {
//launch tasks[i];
}
}
}
```

高级用法

参考：

http://dongxicheng.org/apache-mesos/mesos-libprocess/
http://www.eecs.berkeley.edu/~benh/libprocess/
https://github.com/3rdparty/libprocess

C++项目starter

本节综合cmake、ctest、cpack等工具，给出一个完整的工程项目结构，并结合glog、gmock、gtest、boost、stout、libprocess库给出创建cmake工程项目的完整过程。本章前几节中在介绍各个库的时候没有使用源码的方式，而是直接使用在线安装方式。在本节中使用源码的方式来说明，使用源码优点在于后续维护、代码移植能够更方便一些。建议在创建项目中，对一些关键的类库，使用源码来编译和维护。

https://github.com/3rdparty 中维护了mesos项目中使用的第三方库以及一些常用的代码库，值得大家在C++工程项目中借鉴使用。

第四章 IO和通信范型

- C++编码指南
- 示例代码地址