BLADE 统一构建系统

腾讯公司
俞欢(搜索广告平台部)
陈峰, 彭崇(基础架构部)
2011.4

目前的问题

- Makefile维护起来比较麻烦
  - 我只想用libA，却不得不去指定libA依赖的libB,libC,还有libB所依赖的libD...
  - libA不是我的Makefile生成的，它的源代码变了我不知道需要重新编译。
- 编译速度慢
  - 即使我们引入了ccache等工具，编译速度依然比较慢。

BLADE的目标

- 统一的代码构建环境，提高代码复用程度
- 更快的构建速度
- 保持简洁的接口，便于使用，便于构建模式的扩展，例如远程/并行编译等
- 接口可扩展，目前以支持C++项目为主
- 我们的理念：解放程序员，提高生产力。用工具来解决非创造性的技术问题。

支持的特性

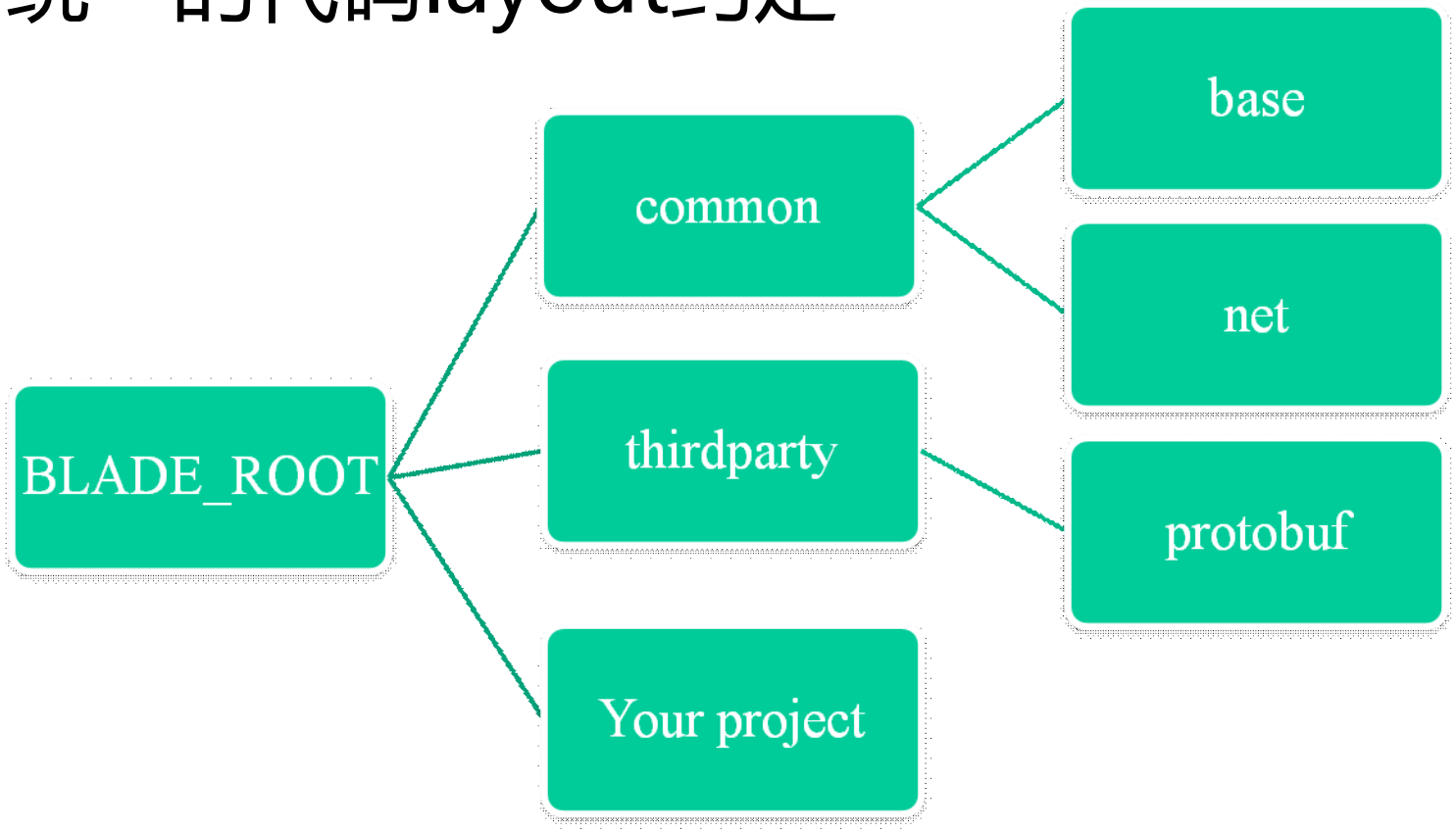
- 模块依赖性分析，构建特定的目标只触发需要的动作。
- 依赖自动传递，只需要写出直接依赖，间接的依赖blade为你搞定。
- 构建过程所有步骤的cache，并可跨用户共享
- 输出至终端时，以彩色高亮显示构建过程。
- 出错时，对出错信息关键行进行彩色高亮。
- 方便的批量运行测试。

构建速度提升

- 以common库为例
  - 700多个源文件，27万行代码。
- Makefile单任务构建：
  - 12分钟
- Makefile并行4任务构建：
  - 4分钟
- Makefile带ccache重新构建：
  - 1分钟
- blade 单任务构建:
  - 8分钟
- blade 并行4路构建:
  - 3分钟
- blade 开启cache重新构建:
  - 8秒！

BLADE使用环境

- 统一的代码layout约定



BLADE配置文件的组织

- BLADE_ROOT文件所在的目录为代码树的根，一个代码树中只能有一BLADE_ROOT 文件。
- 核心配置文件：BUILD
- 每个BUILD含有对当前目录目标(targets)的描述
- 每个target可以指定对本目录或其它目录中targets的依赖，依赖关系自动传递
- 每个BUILD内容是自解释的，没有import等嵌套概念

BUILD文件的一个例子

```
cc_library(
    name = 'base',
    deps = [
        ':string',
        '//common/base/compatible:compatible',
        '#ptread'
    ]
)

cc_test(
    name = 'any_ptr_test',
    srcs = ['any_ptr_test.cpp'],
    deps = [':base']
)

cc_binary(
    name = 'test_app',
    srcs = 'test_main.cpp',
    deps = ':base'
)
```

blade运行的例子

```
phongchen@TengDa_10_6_222_16:~/work/tiny_infra/src/common/readline> blade ... -j8
scons: building associated VariantDir targets: build64_release
Compiling shared ==> build64_release/common/readline/bind.c.o
Compiling shared ==> build64_release/common/readline/callback.c.o
Compiling shared ==> build64_release/common/readline/compat.c.o
Compiling shared ==> build64_release/common/readline/misc.c.o
Compiling shared ==> build64_release/common/readline/xmalloc.c.o
Compiling shared ==> build64_release/common/readline/history.c.o
Compiling shared ==> build64_release/common/readline/histexpand.c.o
Compiling shared ==> build64_release/common/readline/histfile.c.o
Compiling shared ==> build64_release/common/readline/histsearch.c.o
Compiling shared ==> build64_release/common/readline/shell.c.o
Compiling shared ==> build64_release/common/readline/mbutil.c.o
Compiling shared ==> build64_release/common/readline/tilde.c.o
Linking Static Library ==> build64_release/common/readline/libreadline.a
Ranlib Library ==> build64_release/common/readline/libreadline.a
Linking Static Library ==> build64_release/common/readline/libhistory.a
Ranlib Library ==> build64_release/common/readline/libhistory.a
Linking Program ==> build64_release/common/readline/example
```

BUILD目前支持的targets

- cc_library
- cc_binary
- cc_test
- proto_library
- lex_yacc_library
- java_jar
- resource_library
- swig_library
- gen_rule

target的通用属性

- name: 字符串，和路径一起成为target的唯一标识，也决定了构建的输出命名
- srcs: 列表或字符串，构建该对象需要的源文件，一般在当前目录，或相对于当前目录的子目录中
- deps: 列表或字符串，该对象所依赖的其它targets

deps的命名约定

- “//path/to/dir/:name” 的形式
  - 其他目录下的target，path为从BUILD_ROOT出发的路径，name为被依赖的目标名。看见就知道在哪里。
- “:name”
  - 当前目录下的target， path可以省略。
- “#pthread”
  - 系统库。直接写#跟名字即可。

cc_binary

- dynamic_link=True
- 目前我们的binary默认为全静态编译以适应云计算平台使用。
- 如果有应用需要动态编译方式，可以使用此参数指定，此时被此target依赖的所有库都会自动生成对应的动态库供链接。

cc_library

- cc_library同时用于构建静态和动态库，默认只构建静态库，只有被dynamic_link=1的cc_binary依赖时才生成动态链接库。
- link_all_symbols=True
  - 库在被静态链接时，确保库里所有的符号都被链接，以保证依赖全局对象构造函数，比如自动注册器的代码能够正常工作。相当于原来的force_link，不过是在用在生成而不是使用库时。
  - 需要全部链接的部分最好单独拆分出来做成全部链接的库，而不是整个库全都全部链接，否则会无端增大可执行文件的大小。
- pre_build=True
  - 主要应用在thirdparty中从rpm包解来的库，使用这个参数表示不从源码构建。对应的二进制文件必须存在 lib{32,64}_{release,debug} 这样的子目录中。不区分debug/release时可以只有两个实际的目录。

cc_test

- 相当于cc_binary，再加上自动链接gtest和gtest_main
- 还支持testdata参数， 列表或字符串，文件会被链接到输出所在目录*name*.runfiles子目录下，比如：testdata/a.txt =>*name*.runfiles/testdata/a.txt
- 用blade test命令，会在成功构建后到*name*.runfiles目录下自动运行，并输出总结信息。

proto_library

- deps 为import所涉及的其他proto_library
- 自动依赖protobuf，使用者不需要再显式指定。
- 构建时自动调用protoc生成cc和h，并且编译成对应的cc_library

lex_yacc_library

- srcs 必须为二元列表，后缀分别为ll和yy
- 构建时自动调用flex和bison, 并且编译成对应的cc_library

java_jar

- srcs 为源代码所在目录, 如 src/com/soso/poppy
- deps 指定依赖的外部jar包，包括pre_build = 1的已编译好的jar 包和需要编译打包的其它项目jar包。还可以指定依赖protoc 或者 swig 生成的java代码，打包时会自动把依赖到的这部分class 编译打包到目标jar包中

swig_library

- swig_library 对指定的接口文件使用swig生成对应的python和java代码，并且生成对应的python模块和java native库。
- 支持swig选项(extra_swigflags)有：

- cpperraswarn : Bool	python and java
- java_package : package name	only java
- java_lib_packed : Bool	only java

gen_rule

- 用于定制自己的目标
- outs = []，表示输出的文件列表
- cmd, 字符串，表示被调用的命令行
- cmd中可含有如下变量，运行时会被替换成srcs和outs中的对应值
  - $SRCS
  - $OUTS
  - $FIRST_SRC
  - $FIRST_OUT

CACHE

- blade支持cache机制，可以大幅度加快构建速度。
- cache目录依次按一下顺序检测：
  - 命令行参数--cache-dir
  - 环境变量BLADE_CACHE_DIR
- 如果均未配置，则不启用cache。
- 空的BLADE_CACHE_DIR变量或者不带参数值的--cache-dir=禁止cache。

blade命令行参数(1)

- blade action [options] [targets]
- actions:
  - build
  - test
  - run
  - clean
  - query
- 就像svn的用法，svn up/svn ci等等

blade命令行参数(2)

- targets的表示方法
  - path:name 表示path中的某个target
  - path表示path中所有targets
  - path/... 表示path中所有targets，并递归包括所有子目录
  - :name表示当前目录下的某个target
  - 默认表示当前目录

blade命令行参数(3)

- 选项
  - -m32 或 -m64
    - 指定构建模式，默认为自动检测
  - –p PROFILE
    - Profile可以为release或者debug，默认release
  - -j N,--jobs=N
    - N路并行编译，多CPU机器上适用
  - --verbose
    - 完整输出所运行的每条命令行

blade命令行参数(4)

- --cache-dir=DIR
  - 指定一个cache目录，如果是多人公用开发机还可以协商指定至公共可读写目录，共享cache，若不指定则不启用cache功能
- --help
  - 寻求帮助

blade命令行参数(5)

- --color=yes/no/auto
  - 是否开启彩色，默认是auto，也就是，输出到终端时有彩色，方便观看；重定向到文件时没有彩色，方便程序解析。
  - 错误信息太多需要输出到less但是又想保留彩色时可以开启这个选项：blade --color=yes 2>&1 | less -R

blade命令行参数(6)

- --generate-dynamic
  - 我们主要采用全静态链接构建和部署。默认情况下，库目标是只生成静态库，不生成动态库的，这个选项强制生成动态库。
  - 这个选项还有个副作用可以利用：生成动态库时，所有的依赖都必须解决，可以检查到BUILD文件写的不规范导致的依赖不正确。

blade命令行参数示例

- blade build
  - 构建当前目录下的所有目标
- blade build ...
  - 构建当前目录及其子目录下的所有目标
- blade build base/string/... system/...
  - 构建base/string和system子目录下的所有target
- blade build :regex
  - 构建当前目录下的regex目标
- blade build –m32 –pdebug
  - 构建32位debug版
- blade test :mempool_test
  - 自动构建并运行当前目录下的mempool_test目标。

命令行补全

- blade 的子命令以及目录名，选项都是可以按tab自动补全的
- 比如 blade b<TAB>，自动补全为 blade build
- blade test --f<TAB>，自动补全为 blade test --full-test
- 目录也会出现在补全列表里

测试支持

- 增量测试
  - blade默认采用增量测试，即一个测试如果成功，并且本身及其以来的环境没有变化，一定时期内不再重新运行
  - 除非指定--full-test参数
- 并发测试
  - --test-jobs=4，blade就会并发运行4个进程
- 增量测试和并发测试较大幅度地提高了测试效率。

升级到blade构建

- 为了避免重名和预处理带来的问题，blade生成的gcc命令并没有太多的-I 和-D参数
- 所有的 #include 若非是当前目录，需要从BLADE_ROOT开始的完全路径。提供了转换工具以减轻成本
- lex和yacc的扩展名为.ll和.yy

安装和配置

- blade在common/builder/blade/下
- svn co common的代码，进入此目录
- 执行install脚本即可安装到~/bin下，确保~/bin在你的$PATH里，否则修改~/.profile修正。
- install后，只要能连上svn，blade就会自动升级自己，不需要用户关心。
- 目前blade生成scons脚本，因此还需要安装scons，2.0以上版本。

辅助工具

- lsrc
  - 列出当前目录下指定的源文件，以blade的源文件列表格式输出。
- alt
  - 在源代码目录和构建结果目录之间来回跳转
- fix-include-path.sh
  - 自动修正Include路径，以满足从根开始的需要。

VIM集成

- 语法高亮
  - 编写了vim的blade语法文件，高亮显示blade关键字，安装后即可生效，看到五彩斑斓的世界。
- 命令和快捷键
  - 把common/tools/.vimrc拷到~下（或者跟你的合并，如果熟悉vim的话）
  - 在冒号模式下，执行:Blade命令即可使用，参数跟blade一样
  - 按<F3>跳转到上一个构建/测试错误，<F4>到下一个

未来的特性

- 依照BUILD的依赖关系，按需checkout代码
- 自动二进制发布
- 打包机制
- 对闭源项目的平滑构建支持
- 使用 cpplint 检查代码
- 分布式编译支持
- 集中式 cache 支持

联系我们

- 在线文档
  http://code.tencent.com/
- Email：
  - chen3feng@gmail.com
  - lapsangx@gmail.com

Q&A

Thanks!