Blade 用户手册

Michael michaelpeng@tencent.com

目录

1. 关于本手册 2
2. Blade 运行条件与安装 2
3. 源代码树组织 2
4. 构建描述文件：BUILD 文件 3
5. Blade 命令行参考 10
6. Blade 配置文件 12
7. Blade 环境变量 13
8. Blade 测试支持 13
9. Blade cache 14
10. Blade 输出 14
11. Blade 辅助命令 15
12. 联系我们 16

1.关于本手册

Blade 是在腾讯公司台风云计算平台开发中诞生的新一代构建系统，本手册是旨在介绍如何使用 Blade。

2.Blade 运行条件与安装

Blade 目前支持的平台只有 Linux，以后将会打算支持更多的平台（如 Windows，Solaris 等）。运行时需要安装：

1. SCons v2.0 or newer (required)
2. Python v2.6 or newer (required, python v3.0 or newer excluded)
3. ccache v3.1 or newer (optional)
4. distcc v3.1 or newer (optional)

如果需要构建其他类型的代码，还可能需要用到：

1. swig v2.0 or newer (required for swig_library)
2. flex v2.5 or newer (required for lex_yacc)
3. bison v2.1 or newer (required for lex_yacc)

从 svn checkout blade 后，执行 install 脚本即可安装到~/bin 下，目前以软链方式安装，install 后不能删除 checkout 出来的原始目录。 确保~/bin 在 PATH 环境变量里后，可以在任何目录下直接执行：

```
blade
或者
blade --help 验证安装是否成功
```

3.源代码树组织

台风云计算平台的项目目录组织，要求整个源代码树有一个明确的根目录，C++ 中的 #include 的路径也需要从这个目录开始写起，这样能有效地避免头文件重名造成的问题，而且还可以减少编译器预处理时搜索头文件时间。 Blade 并不从某个配置文件或者环境变量读取这个信息，因为开发人员往往需要同时有多个目录树并存。 Blade 获取源代码根的方法是，无论当前从哪一级子目录运行，都从当前目录开始向上查找 BLADE_ROOT 文件，有这个文件的目录即为源代码树的根。一个源代码树的根目录看起来是这样子的：

```
BLADE_ROOT
common
poppy
thirdparty
xfs
xcube
torca
your_project_name
```

你要需要做的就是在根目录 touch 一个 BLADE_ROOT 文件，让 Blade 知道这个目录是项目根目录。

4.构建描述文件：BUILD 文件

Blade 通过一系列名字为 "BUILD" 的构建描述文件（文件名全大写）进行项目构建，这些文件需要开发者去编写。每个 BUILD 文件通过一组目标描述函数描述一个目标的源文件，所依赖的其他目标，以及其它一些属性等。BUILD 文件很简单，如 common/base/string/BUILD：

```
cc_library(
   name = 'string',      # 目标名
   srcs = [
      'algorithm.cpp',
      'string_number.cpp',
      'string_piece.cpp',
      'format.cpp',
      'concat.cpp'
   ],                            # 源代码集合
   deps = ['//common/base:int'] # 直接依赖集合，没有可以不写
)
```

这些描述都是自解释的，只需列出目标名，源文件名和依赖名即可。在写完这个 BUILD 文件后，用户即可以通过命令行构建这个 library 目标。

```
blade build common/base/string:string
```

当然你需要在 common/base 目录描述目标 int，这里就省略了。Blade 会自动分析这里的依赖关系，先构建被依赖的库//common/base:int，再构建 string 库。

Blade 支持的构建目标有 11 种：

1. cc_library
2. cc_binary
3. cc_test
4. proto_library
5. lex_yacc_library
6. gen_rule
7. swig_library
8. cc_plugin
9. resource_library
10. java_jar
11. py_binary

Blade 用一组 target 函数来定义目标，这些 target 的通用属性有：

1) 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。
2) 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。
3) deps: 列表或字符串，该对象所依赖的其它 targets。

deps 的允许的格式：

1) “//path/to/dir/:name” 某目录下的 target，path 为从 BLADE_ROOT 出发的路径，name 为被依赖的目标名。
2) “:name” 当前目录下的 target， path 可以省略。

3) “#pthread”系统库。直接用#拼接名字即可。

cc_* 目标 包括 cc_test, cc_binary, cc_library，CC 目标均支持的参数为：

srcs 源文件列表

deps 依赖目标列表

incs 头文件路径列表，仅用于 thirdparty

defs 宏定义列表，仅用于 thirdparty

warning 警告设置，仅用于 thirdparty

optimize 优化设置

字段	解释	举例	备注
warning	是否屏蔽 warning	warning=’no’	默认不屏蔽 warning='yes' ，默认不用写，已开启
defs	编译时加入用户定义宏	defs=[‘_MT’]	如果用户定义 C++关键字，报 warning
incs	编译时计入用户定义的 include 目录	incs=['poppy/myinc']	用户通常不要使用
optimize	用户定义的 optimize flags	optimize=['O3']	适用于 cc_library cc_binary cc_test proto_library swig_library cc_plugin resource_library

cc_library

用于描述 C++库目标。 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=True 的 cc_binary(或 cc_test)依赖时或者命令行指定 --generate-dynamic 才生成动态链接库。

举例：

```
cc_library(
    name='lowercase',
    srcs=['./src/lower/plowercase.cpp'],
    deps=['#pthread'],
    link_all_symbols=False
)
```

目标属性：

link_all_symbols(True or False)

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。

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。

需要注意的是，link_all_symbols 是库自身的属性，不是使用这个库时的属性。

always_optimize(True or False)

True：不论 debug 版本还是 release 版本总是被优化。False：debug 版本不作优化。默认为 False。目前只对 cc_library 有效。

prebuilt(True or False)

True：主要应用于没有源代码从其他构建系统生成的库，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_binary

定义生成 C/C++可执行文件

```
cc_binary(
    name='prstr',
    srcs=['./src/mystr_main/mystring.cpp'],
    deps=['#pthread',':lowercase',':uppercase','#dl'],
)
```

目标属性：

dynamic_link(True or False)

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以供链接。需要注意的是，dynamic_link 只适用于可执行文件，不适用于库。

export_dynamic(True or False)

Pass the flag '-export-dynamic' to the ELF linker, on targets that support it. This instructs the linker to add all symbols, not only used ones, to the dynamic symbol table. This option is needed for some uses of dlopen or to allow obtaining backtraces from within a program.

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。

testdata=[]

在 name.runfiles 里建立 symbolic link 指向工程目录的文件，目前支持 以下几种形式

1) 'file'， 在测试程序中使用这个名字本身的形式来访问

2) '//your_proj/path/file'， 在测试程序中用"your_proj/path/file"来访问。

3) ('//your_proj/path/file', "new_name")， 在测试程序中用"new_name"来访问

可以根据需要自行选择，这些路径都也可以是目录。

```
cc_test(
    name = 'textfile_test',
    srcs = 'textfile_test.cpp',
    deps = ':io',
    testdata = [
        'test_dos.txt',
        '//your_proj/path/file',
        ('//your_proj/path/file', 'new_name')
    ]
)
```

目标属性：

export_dynamic(True or False)

Pass the flag '-export-dynamic' to the ELF linker, on targets that support it. This instructs the linker to add all symbols, not only used ones, to the dynamic symbol table. This option is needed for some uses of dlopen or to allow obtaining backtraces from within a program.

heap_check('minimal', 'normal', 'strict', 'draconian', 'as-is', 'local')

使用 gperftool 进行内存检查时 heap_check 的值，请参考 gperftools 文档中对 HEAPCHECK 环境变量的说明（http://google-perftools.googlecode.com/svn/trunk/doc/heap_checker.html）。

heap_check_debug(True or False)

是否使用 gperftools 的 debug 版本。Debug 不但能检测内存泄露，还能检测越界，double free 等其他问题。

proto_library

用于产生 protobuf 库。 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。

```
proto_library(
   name = 'rpc_meta_info_proto',
   srcs = 'rpc_meta_info.proto',
   deps = ':rpc_option_proto',
)
```

Blade 支持 proto_library，使得在项目中使用 protobuf 十分方便。

要引用某 proto 文件生成的头文件，需要从 BLADE_ROOT 的目录开始，只是把 proto 扩展名改为 pb.h 扩展名。比如 //common/base/string_test.proto 生成的头文件，路径为 "common/base/string_test.pb.h"。

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。

```
lex_yacc_library(
    name = 'parser',
    srcs = [
        'line_parser.ll',
        'line_parser.yy'
    ],
    deps = [
        ":xcubetools",
    ],
    recursive=True
)
```

目标属性：

recursive(True or False)

生成可重入的 C scanner. flex has the ability to generate a reentrant C scanner. This

is accomplished by specifying %option reentrant (`-R'). 例如：
可以同时扫描两个文件并且在 token level 比较而不是单个字符比较

```
/* Example of maintaining more than one active scanner. */
do {
   int tok1, tok2;
   tok1 = yylex( scanner_1 );
   tok2 = yylex( scanner_2 )
   if( tok1 != tok2 )
      printf("Files are different.");
} while ( tok1 && tok2 );
```

prefix 字符串

用于指定生成的代码的函数名的前缀，取代默认的前缀。

gen_rule

用于定制自己的目标。outs = []，表示输出的文件列表，需要填写这个域 gen_rule 才会执行 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 $BUILD_DIR -- 可被替换为 build[64,32]_[release,debug] 输出目录。

```
gen_rule(
   name='test_gen_target',
   cmd='echo what_a_nice_day;touch test2.c',
   deps=[':test_gen'],                # 可以有 deps , 也可以被别的 target 依赖
   outs=['test2.c']
)
```

很多用户使用 gen_rule 动态生成代码文件然后和某个 cc_library 或者 cc_binary 一起编译， 需要注意应该尽量在输出目录生成代码文件，如 build64_debug 下，并且文件的路径名要写对， 如 outs = ['websearch2/project_example/module_1/file_2.cc'], 这样使用 gen_rule 生成的文件和库在一起编译时就不会发生找不到动态生成的代码文件问题了。

swig_library

根据.i 文件生成相应的 python, java 和 php cxx 模块代码，并且生成对应语言的代码。

```
swig_library(
   name = 'poppy_client',
   srcs = [
      'poppy_client.i'
   ],
   deps = [
      ':poppy_swig_wrap'
   ],
   warning='yes',
   java_package='com.soso.poppy.swig',   # 生成的 java 文件的所在 package 名称
   java_lib_packed=1, # 表示把生成的 libpoppy_client_java.so 打包到依赖者的 jar 包里，如 java_jar 依赖
                      # 这个 swig_library
```

```
    optimize=['O3']    # 编译优化选项
)
```

目标属性：

warning(‘yes’ or ‘no’)

这里的 warning 仅仅指 swig 编译参数 cpperraswarn 是否被指定了，swig_library 默认使用非标准编译告警级别（没有那么严格）。

cc_plugin

支持生成 target 所依赖的库都是静态库.a 的 so 库，即 plugin。

```
cc_plugin(
    name='mystring',
    srcs=['./src/mystr/mystring.cpp'],
    deps=['#pthread',':lowercase',':uppercase','#dl'],
    warning='no',
    defs=['_MT'],
    optimize=['O3']
)
```

cc_plugin 是为 JNI，python 扩展等需要动态库的场合设计的，不应该用于其他目的。

resource_library

编译静态资源。

大家都遇到过部署一个可执行程序，还要附带一堆辅助文件才能运行起来的情况吧？ blade 通过 resource_library，支持把程序运行所需要的数据文件也打包到可执行文件里， 比如 poppy 下的 BUILD 文件里用的静态资源：

```
resource_library(
    name = 'static_resource',
    srcs = [
        'static/favicon.ico',
        'static/forms.html',
        'static/forms.js',
        'static/jquery-1.4.2.min.js',
        'static/jquery.json-2.2.min.js',
        'static/methods.html',
        'static/poppy.html'
    ]
)
```

生成 和 libstatic_resource.a 或者 libstatic_resource.so。 就像一样 protobuf 那样，编译后后生成一个库 libstatic_resource.a，和一个相应的头文件 static_resource.h，带路径包含进来即可使用。

在程序中需要包含 static_resource.h（带上相对于 BLADE_ROOT 的路径）和 "common/base/static_resource.hpp"， 用 STATIC_RESOURCE 宏来引用数据：

```
StringPiece data = STATIC_RESOURCE(poppy_static_favicon_ico);
```

STATIC_RESOURCE 的参数是从 BLADE_ROOT 目录开始的数据文件的文件名，把所有非字母数字和下划线的字符都替换为_。

得到的 data 在程序运行期间一直存在，只可读取，不可写入。

用 static resource 在某些情况下也有一点不方便：就是不能在运行期间更新，因此是否使用，需要根据具体场景自己权衡。

java_jar

编译 java 源代码。

```
java_jar(
   name = 'poppy_java_client',
   srcs = [
      'src/com/soso/poppy'     # 这里只需要指定 java 文件所在目录，不要写上具体 java 文件列表
   ],
   deps = [
      '//poppy:rpc_meta_info_proto',   # 可以依赖 proto_library，生成的 java 文件一起编译打包
      '//poppy:rpc_option_proto',
      '//poppy:rpc_message_proto',
      '//poppy:poppy_client',          # 可以依赖 swig_library，生成的 java 文件一起编译打包
      './lib:protobuf-java',           # 可以依赖别的 jar 包
      './lib:junit',
   ]
)
```

目标属性：

prebuilt(True or False)

主要应用在已经编译打包好的 java jar 包。

py_binary

把 py_binary 中 name 指定的目录里的 python 文件及其依赖打包成 python egg。用户在当前 BUILD 文件可以不写 setup.py。

比如以下例子，当前 BUILD 里有这个 py_binary 目标，如果不存在 setup.py， blade 会为用户自动建立并且给出告警，poppy_python_client 里需要有__init__.py（必须存在）指示这是个将要打包的 package：

```
py_binary(
   name='poppy_python_client',
   srcs='poppy/poppy_python_client',
   deps=[
      ':rpc_channel_option_info_proto'
   ]
)
```

运行 blade build :poppy_python_client 后会在相应目录的子目录 dist 输出一个 egg 文件。

5.Blade 命令行参考

Blade 命令行书写方式是：
blade [action] [options] [targets]

action 是一个动作，目前有 5 种：
build 表示构建项目
test 表示构建并且跑单元测试
clean 表示清除目标的构建结果
query 查询目标的依赖项与被依赖项
run 构建并 run 一个单一目标
每个动作支持的 options 都不一样，具体的 options 可以通过 blade [action] --help 获取具体含义。

targets 是一个列表，支持的格式：
- 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。

每个动作对应的选项的具体含义如下：
build

```
-h, --help          帮助
-m M                生成 32 位或者 64 位 package
-p PROFILE, --profile PROFILE
                    debug 或者 release 版本，默认为 release
--generate-scons-only
                    只生成构建文件，不构建
-j JOBS, --jobs JOBS  并行构建参数，不指定的话，blade 会自动计算
-k, --keep-going      如果出错则继续构建
--verbose             verbose 模式显示所有构建细节，如全部执行的命令行
--no-test             不去构建 cc_test 目标
--color COLOR         色彩设定: yes, no or auto, 默认是 auto.
--cache-dir CACHE_DIR 设定 scons cache 目录
--cache-size CACHE_SIZE 设定 scons cache 大小，以 G 计算，unlimited 表示无限制
--generate-dynamic    生成动态库
--generate-java       为 proto_library 和 swig_library 目标生成 java 文件
--generate-php        为 proto_library 和 swig_library 目标生成 php 文件
--gprof               支持 GNU gprof
--gcov                支持 GNU gcov 生成覆盖率信息文件，但不统计覆盖率
```

test

```
-h, --help          帮助
```

```
-m M                生成 32 位或者 64 位 package
-p PROFILE, --profile PROFILE
                    debug 或者 release 版本，默认为 release
--generate-scons-only
                    只生成构建文件，不构建
-j JOBS, --jobs JOBS  并行构建参数，不指定的话，blade 会自动计算
-k, --keep-going      如果出错则继续构建
--verbose           verbose 模式显示所有构建细节，如全部执行的命令行
--color COLOR       色彩设定: yes, no or auto, 默认是 auto.
--cache-dir CACHE_DIR 设定 scons cache 目录
--cache-size CACHE_SIZE 设定 scons cache 大小，以 G 计算，unlimited 表示无限制
--generate-dynamic    生成动态库
--generate-java       为 proto_library 和 swig_library 目标生成 java 文件
--generate-php        为 proto_library 和 swig_library 目标生成 php 文件
--gprof             支持 GNU gprof
--gcov              支持 GNU gcov 生成覆盖率信息文件，但不统计覆盖率
--testargs TESTARGS   传给 test 程序的命令行参数
--full-test         全量测试
-t TEST_JOBS, --test-jobs TEST_JOBS 并行测试时同时运行的程序个数
--show-details        测试完成后提供详细的测试总结信息
```

query

```
-h, --help          帮助
--deps              显示当前查询目标的所有依赖项
--depended          显示当前查询目标被哪些目标依赖
```

run

```
--runargs RUNARGS   传给运行程序的命令行参数
其它参数同 test，此处略
```

clean

```
-h, --help          帮助
-m M                清理 32 位或者 64 位的目标
-p PROFILE, --profile PROFILE 清理 debug 或者 release 版本
--generate-dynamic    清理动态库
```

我们举一些简单例子，比如对于这个目标 common/base/string:string：

```
构建这个目标的动态库:
blade build common/base/string:string –-generate-dynamic
构建 32 位的静态库 debug 版本：
blade build common/base/string:string –m32 –pdebug
构建 common/base/string 目录下的所有目标并且跑单元测试，产生覆盖率信息
blade test common/base/string… --gcov
清理 common/base/string
```

```
blade clean common/base/string…
```

6.Blade 配置文件

Blade 支持从用户配置文件读入用户配置，有三个配置文件：与 blade.zip 同级的 blade.conf，~/.bladerc 和 BLADE_ROOT。blade.conf 是全局配置，~/.bladerc 是用户配置选项，而 BLADE_ROOT 因为是不同项目的根目录标识文件，里面的配置一般是针对项目而设定，如果这三个文件同时存在， blade.conf, ~/.bladerc 和 BLADE_ROOT 按顺序后者覆盖前者。blade.conf 每个配置解释如下：

```
# 测试配置
cc_test_config(
    dynamic_link=True,   # 依赖目标是否生成动态库并且链接动态库
    heap_check='strict', # gerftools 内存堆检查选项，有 ‘minimal’, ‘normal’, ‘strict’, ‘draconian’,
                         # ‘as-is’, ‘local’ 五种模式，如果BUILD文件里的cc_test没有指定heap_check的
                         # 值，则从这里读取
    gperftools_lib='thirdparty/perftools:tcmalloc',  # cc_test 默认链接gperftool时使用的目标
    gperftools_debug_lib='thirdparty/perftools:tcmalloc_debug', # 如果BUILD文件里的cc_test指定
                                                                # heap_check_debug=True时
                                                                # 则从这里读取需要链接的
                                                                #gperftools debug 库
    gtest_lib='thirdparty/gtest:gtest', # cc_test 默认链接gtest时使用的目标
    gtest_main_lib='thirdparty/gtest:gtest_main' # cc_test默认链接gtest main库时使用到的目标
)

# distcc 配置
distcc_config(
    enabled=False  # True 时使用distcc编译源代码文件
)

# 链接配置
link_config(
    link_on_tmp=True,  # 为True时在tmpfs链接产生目标
    enable_dccc=False  # 为True时进行分布式编译和链接
)

# protobuf library 配置选项
proto_library_config(
    protoc='thirdparty/protobuf/bin/protoc', # protoc程序位置
    protobuf_lib='thirdparty/protobuf:protobuf', # protobuf library库目标
    protobuf_path='thirdparty', # 调用protoc时protobuf_path的值
    protobuf_include_path = 'thirdparty', # 调用protoc时-I参数后的值，可以用空格分开多个值
    protobuf_php_path='thirdparty/Protobuf-PHP/library', # 调用protoc时php路径
    protoc_php_plugin='thirdparty/Protobuf-PHP/protoc-gen-php.php' # 调用protoc时php plugin路径
```

```
)

# cc配置选项
cc_config(
    extra_incs='thirdparty' # 编译c++时 -I 参数，可以用空格分开
)

# java编译配置选项
java_config(
    source_version='1.6', # 编译java时source参数
    target_version='1.6'   # 编译java时target参数
)
```

BLADE_ROOT 里的配置选项和~/.bladerc 里的选项是一样的，只不过两个都存在时，前者的值会覆盖后者的同段同名值，也就是说，项目级的配置优先级高于全局的。

7.Blade 环境变量

Blade 支持以下环境变量：

- TOOLCHAIN_DIR，默认为空
- CPP，默认为 cpp
- CXX，默认为 c++
- CC，默认为 gcc
- LD，默认为 c++

TOOLCHAIN_DIR 和 CPP 等组合起来，构成调用工具的完整路径，例如：

调用/usr/bin 下的 gcc（开发机上的原版 gcc）

```
TOOLCHAIN_DIR=/usr/bin blade build [targets]
```

使用 clang

```
CPP='clang -E'  CC=clang  CXX=clang++  ld=clang++  blade build [targets]
```

如同所有的环境变量设置规则，放在命令行前的环境变量，只对这一次调用起作用，如果要后续起作用，用 export，要持久生效，放入 ~/.profile 中。

8.Blade 测试支持

Blade 对测试的集成力求全面高效，比如具有生成覆盖率信息，增量测试，并行测试等特性。

Blade test 支持增量测试 ，可以加快 tests 的执行。 已经 pass 的 tests 在下一次构建和测试时不需要再跑，除非：

1. tests 的任何依赖变化导致其重新生成。

2. tests 依赖的测试数据改变，这种依赖为显式依赖，用户需要使用 BUILD 文件指定，如 testdata。
3. tests 所在环境变量发生改变。
4. test arguments 改变。
5. Fail 的 test cases ，每次都重跑。

如果需要使用全量测试，使用--full-test option, 如 blade test common/... --full-test ，全部测试都需要跑。另外，cc_test 支持了 always_run 属性，用于在增量测试时，不管上次的执行结果，每次总是要跑。

```
cc_test(
    name = 'zookeeper_test',
    srcs = 'zookeeper_test.cc',
    always_run = True
)
```

Blade test 支持并行测试， 并行测试把这一次构建后需要跑的 test cases 并发地 run。 blade test [targets] --test-jobs N （或者-tN）， --test-jobs N 设置并发测试的并发数，Blade 会让 N 个测试进程并行地执行。

对于某些因为可能相互干扰而不能并行跑的测试，可以加上 exclusive 属性：

```
cc_test(
    name = 'zookeeper_test',
    srcs = 'zookeeper_test.cc',
    exclusive = True
)
```

9.Blade cache

Blade 支持 cache，可以大幅度加快构建速度。现在支持两种 cache:

1) ccache , cache 配置使用 ccache 的配置, 如通过配置 CCACHE_DIR 环境变量指定 ccache 目录，ccache 如果安装了，默认优先使用 ccache，推荐结合使用 ccache。

2) 如果 ccache 没有安装，则使用 scons cache。

Scons cache 需要一个目录，依次按以下顺序检测：

命令行参数--cache-dir

环境变量 BLADE_CACHE_DIR

如果均未配置，则不启用 scons cache。

空的 BLADE_CACHE_DIR 变量或者不带参数值的--cache-dir=, 则会禁止 scons cache。

--cache-size 如不指定，则默认为 2G，如指定，则使用用户指定的以 Gigabyte 为单位的大小的 cache。

如 --cache-dir='~/user_cache' --cache-size=16 (16 G)大小 cache。 用户可以根据需要配置大小，超出大小 blade 会执行清理工作，限制 cache 大小在用户指定的 cache 大小， 请谨慎设置这个大小，因为涉及到构建速度和机器磁盘空间的占用。

10. Blade 输出

Blade 构建过程是彩色高亮的， 出错信息也是彩色的，方便定位错误。如图所示：

```
Compiling poppy/rpc_channel_test.cc
In file included from ./thirdparty/protobuf-2.3.0/src/google/protobuf/service.h:106:0,
                 from ./thirdparty/protobuf/service.h:12,
                 from ./poppy/rpc_channel_interface.h:10,
                 from ./poppy/rpc_channel.h:11,
                 from poppy/rpc_channel_test.cc:8:
./common/system/net/socket_address.hpp:1:2: warning: #warning "This file has been renamed to socket_address.h"
cc1plus: warning: unrecognized command line option "-Wno-unused-but-set-variable"
Creating Static Library build64_release/thirdparty/protobuf-2.3.0/src/google/protobuf/libprotobuf.a
Compiling poppy/rpc_client_test.cc
Ranlib Library build64_release/thirdparty/protobuf-2.3.0/src/google/protobuf/libprotobuf.a
Linking Program build64_release/poppy/rpc_channel_test
Linking Program build64_release/poppy/rpc_client_test
In file included from ./thirdparty/protobuf-2.3.0/src/google/protobuf/service.h:106:0,
                 from ./thirdparty/protobuf/service.h:12,
                 from ./poppy/rpc_channel_interface.h:10,
                 from ./poppy/rpc_channel.h:11,
                 from ./poppy/rpc_mock.h:67,
                 from poppy/rpc_mock.cc:5:
./common/system/net/socket_address.hpp:1:2: warning: #warning "This file has been renamed to socket_address.h"
cc1plus: warning: unrecognized command line option "-Wno-unused-but-set-variable"
Linking Shared Library build64_release/poppy/_poppy_client.so
In file included from ./thirdparty/protobuf-2.3.0/src/google/protobuf/service.h:106:0,
                 from ./thirdparty/protobuf/service.h:12,
                 from ./poppy/rpc_channel_interface.h:10,
                 from ./poppy/rpc_mock_channel.h:26,
                 from poppy/rpc_mock_channel.cc:4:
./common/system/net/socket_address.hpp:1:2: warning: #warning "This file has been renamed to socket_address.h"
cc1plus: warning: unrecognized command line option "-Wno-unused-but-set-variable"
Creating Static Library build64_release/poppy/libpoppy_mock.a
Ranlib Library build64_release/poppy/libpoppy_mock.a
Compiling poppy/rpc_controller_test.cc
Compiling poppy/rpc_form_test.cc
In file included from ./thirdparty/protobuf-2.3.0/src/google/protobuf/service.h:106:0,
                 from ./thirdparty/protobuf/service.h:12,
                 from ./poppy/rpc_controller.h:21,
                 from poppy/rpc_controller_test.cc:8:
./common/system/net/socket_address.hpp:1:2: warning: #warning "This file has been renamed to socket_address.h"
cc1plus: warning: unrecognized command line option "-Wno-unused-but-set-variable"
Linking Program build64_release/poppy/rpc_controller_test
Linking Program build64_release/poppy/rpc_form_test
```

默认生成 `native arch` 的可执行文件，指定生成 `32/64` 位结果也很简单，加上 `-m32/64` 即可。 默认生成 `release` 版本的结果，如果生成 `debug` 版的，加上 `-p debug` 即可。 默认构建当前目录，如果当前目录依赖的外面的模块需要重新构建，也会被连带构建起来（`Make` 很难做到）。如果要从当前目录构建所有子目录的目标，也很简单：`blade build ...` 即可。

不同构建选项的结果放在不同的目录下，生成的文件一律按层次也放在这个目录里，不会污染源代码目录。

要清除构建结果（一般不需要），`blade clean [targets]` 即可。

11. Blade 辅助命令

install

blade 命令的符号链接会被安装下面的命令到~/bin 下。

lsrc

列出当前目录下指定的源文件，以 blade 的 srcs 列表格式输出。

genlibbuild

自动生成以目录名为库名的 cc_library，以测试文件的名为名的 cc_test，proto 的 BUILD 文件，并假设这些测试都依赖这个库。

vim 集成

我们编写了 vim 的 blade 语法文件，高亮显示 blade 关键字，install 后就会自动生效。

同时编写了 Blade 命令，使得可以在 vim 中直接执行 blade，并快速跳转到出错行（得益于 vim 的 quickfix 特性）。

```
function! Blade(...)
    let l:old_makeprg = &makeprg
    setlocal makeprg=blade
    execute "make " . join(a:000)
    let &makeprg=old_makeprg
endfunction
command! -complete=dir -nargs=* Blade call Blade('<args>')
```

使用时直接在 vim 的 : 模式输入（可带参数）

Blade

即可构建。

这个命令的源代码在 .vimrc 中。

alt

在源代码目录和构建目标目录之间跳转。

12. 联系我们

在 Blade 使用过程中如果遇到什么问题，请联系我们:

Michael michaelpeng@tencent.com

Phongchen phongchen@tencent.com

欢迎使用!