**Blade用户手册**

**Blade是什么**

软件项目用各种工具来构建代码，最常用的恐怕是GNU Make。但是 GNU Make 虽然本身功能比较强，但是要直接使用的话，也是比较难的。

很多人还在手工编写 Makefile，又没有去写正确的依赖，导致每次不得不先 make clean 才放心，Make的意义大打折扣。这在几个文件的小项目下是没什么问题的，对于大的项目就很不方便了。

Autotools 号称auto，但是还是需要人工写很多东西，运行一系列命令，用起来还是比较复杂，开发人员的学习和使用的门槛很高。

Blade 就是针对这些问题，为腾讯公司基础架构部的[<http://storage.it168.com/a2011/1203/1283/000001283196.shtml> “台风”云计算平台]项目而开发的新一代构建工具，希望能成为开发者手中的“瑞士军刀”。我们现在把它开源出来，希望能让更多的人得到方便。

**Blade 解决的问题**

* 源文件更新导致需要重新构建。这个 gnu make 都能解决得很好。
* 头文件更新，所以以来这个头文件的源文件都需要重新构建。这个 gnu make 不直接支持，需要搭配 gcc 来生成和更新依赖。
* 库文件更新，所依赖的库文件更新后，程序应该重新连接，GNU Make 可以做到。
* 即使我只构建自己的目标，如果库的源代码变了，库应该重新生成，GNU Make 用递归 Make 无法做到。
* 库文件之间的依赖自动传递，一个库依赖另一个库，库的最终用户不需要关心。
* 构建过程中的警告和错误应该醒目地显示出来。
* 能自动支持台风系统大量使用的 proto buffer，以及方便扩充以支持外来可能引入的新工具。
* 应该能集成自动测试，代码检查等开发常用的功能。

**Blade运行条件**

Blade 运行时需要以下条件：

* SCons v2.0 or later (required)
* Python v2.6 or later (required)
* ccache v3.1 or later (optional)

Blade 编译项目时可能需要到：

* swig v2.0 or later (required for swig\_library)
* flex v2.5 or later (required for lex\_yacc)
* bison v2.1 or later (required for lex\_yacc)

**源代码树的组织**

Blade要求项目源代码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，这样有几点好处：

* 有效地避免头文件重名造成的问题。
* 有效地避免库文件的重名。
* 更容易找到需要的文件。
* 提高构建速度。

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。

目前源代码目录需要自己拉取，将来我们会集成到 Blade 中。BLADE\_ROOT 文件也需要用户自己创建。方法： $ touch BLADE\_ROOT

一个源代码树的根目录看起来的样子如下： BLADE\_ROOT common thirdparty xfs xcube torca your\_project ...

**BUILD文件**

Blade 通过一系列的名字为 "BUILD" 的文件（文件名全大写），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了一个目标的源文件，所依赖的其他目标，以及其他一些属性。

**BUILD文件的示例**

构建脚本很简单：

范例：common/base/string/BUILD

cc\_library(

name = 'string',

srcs = [

'algorithm.cpp',

'string\_number.cpp',

'string\_piece.cpp',

'format.cpp',

'concat.cpp'

],

deps = ['//common/base:int']

)

也是说明式的，只需要列出目标名，源文件名和依赖名（可以没有）即可。

**风格建议**

* 四空格缩进，不要用tab字符
* 总是用单引号
* 目标名用小写
* src 里的文件名按字母顺序排列
* deps 里先写本目录内的依赖（:target），后写其他目录内的（//dir:name），分别按字母顺序排列。
* 不同目标之间空一行，前面可以加注释
* 注释的 # 后面空一格，比如 # This is a comment

**描述目标**

Blade用一组target函数来定义目标，这些target的通用属性有：

* 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的允许的格式：

* "//path/to/dir/:name" 其他目录下的target，path为从BLADE\_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\_\* 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 依赖列表
* incs 头文件路径列表
* defs 宏定义列表
* warning 警告设置
* optimize 优化设置
* 注：thirdparty是我们代码库里的一个特殊目录，里面的代码都是一些第三方库，按照台风系统的代码规范，只允许对这里的代码用incs, defs和warnings，自己开发的代码要按照规范组织。Blade会对这个目录之外的代码使用这些参数发出警告。

|| *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 || 用户定义的宏加入编译中 || defs=['\_MT'] || 如果用户定义C++关键字，报warning || || incs || 用户定义的include || 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=1的cc\_binary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

)

参数：

* 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。 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，不是使用库时的属性。Blade是为大型项目设计的，基于以下因素，我们提倡任何模块都应该有自己的 cc\_library，用户程序都应该在deps里写全直接依赖，不提倡创建像boost那样的全头文件的库。
  + 编译速度
  + 将来未知的改变，比如某库一开始只需要头文件就能使用，不依赖标准库之外的任何库，但是后来依赖了MD5，所有使用这个库的代码都要加上新产生的依赖，这与我们设计Blade的初衷是违背的。

要强制用户这样使用，可以在编写代码时，总是编写 .h 对应的 .cpp 文件，并把一部分必然要用到的符号（函数，静态变量）的实现写在里面，即使对于模板库，可以引入一个非模板的基类，或者把非模板部分的实现放到 .cpp 里。

* always\_optimize 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 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\_binary**

定义C++可执行文件目标

cc\_binary(

name='prstr',

srcs=['./src/mystr\_main/mystring.cpp'],

deps=['#pthread',':lowercase',':uppercase','#dl'],

)

* 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。 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 常规情况下，so中只引用所依赖的so中的符号，但是对于应用特殊的场合，需要在so中引用宿主可执行文件中的符号，就需要这个选项。 这个选项告诉连接器在可执行文件的动态符号表中加入所有的符号，而不只是用到的其他动态库中的符号。这样就使得在dlopen方式加载的so中可以调用可执行文件中的这些符号。 详情请参考 man ld(1) 中查找 --export-dynamic 的说明。

**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子命令，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
  + 'file' 在测试程序中使用这个名字本身的形式来访问
  + '//your\_proj/path/file' 在测试程序中用"your\_proj/path/file"来访问。
  + ('//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')

]

)

**proto\_library**

用于定义protobuf目标 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"。

**thrift\_library**

用于定义thrift库目标 deps 为import所涉及的其他thrift\_library 自动依赖thrift，使用者不需要再显式指定。 构建时自动调用thrift命令生成cpp和h，并且编译成对应的cc\_library

thrift\_library(

name = 'shared\_thrift',

srcs = 'shared.thrift',

)

thrift\_library(

name = 'tutorial\_thrift',

srcs = 'tutorial.thrift',

deps = ':shared\_thrift'

)

C++中使用生成的头文件时，规则类似proto，需要带上相对BLADE\_ROOT的目录前缀。

* thrift 0.9版（之前版本未测）有个[<https://issues.apache.org/jira/browse/THRIFT-1859> bug]，需要修正才能使用，此bug已经在开发版本中[<https://builds.apache.org/job/Thrift/633/changes#detail13> 修正]

**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

)

* recursive=True 生成可重入的C scanner.

**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才会被执行 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

)

* warning 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',

]

)

* prebuilt=True 主要应用在已经编译打包好的java jar 包。

**Blade的输出**

构建过程是彩色高亮的 出错信息是彩色的，方便定位错误。

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 ... 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 即可。

**Blade Cache**

blade 支持 cache，可以大幅度加快构建速度。 blade 支持两种cache

* 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 没有安装，则使用scons cache, 配置细节如下

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cache。
* 空的BLADE\_CACHE\_DIR变量或者不带参数值的--cache-dir=, 则会禁止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。

**测试支持**

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：

* tests 的任何依赖变化导致其重新生成。
* 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。
* tests 所在环境变量发生改变。
* test arguments 改变。
* 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 -t, --test-jobs N 设置并发测试的并发数，Blade会让N个测试进程并行执行

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

)

**命令行参考**

blade `[`action`]` `[`options`]` `[`targets`]`

action是一个动作，目前有

* 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

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

参数列表：

* -m32,-m64 指定构建目标位数，默认为自动检测
* -p PROFILE 指定debug/release，默认release
* -k, --keep-going 构建过程中遇到错误继续执行（如果是致命错误不能继续）
* -j N,--jobs=N N路并行编译，多CPU机器上适用
* -t N,--test-jobs=N N路并行测试，多CPU机器上适用
* --cache-dir=DIR 指定一个cache目录
* --cache-size=SZ 指定cache大小，以G为单位
* --verbose 完整输出所运行的每条命令行
* –h, --help 显示帮助
* --color=yes/no/auto 是否开启彩色
* --generate-dynamic 强制生成动态库
* --generate-java 为proto\_library 和 swig\_library 生成java文件
* --generate-php 为proto\_library 和 swig\_library 生成php文件
* --gprof 支持 GNU gprof
* --gcov 支持 GNU gcov 做覆盖率测试

**配置**

Blade 支持三个配置文件

* blade.zip 同一个目录下的 blade.conf，这是全局配置。
* ~/.bladerc 用户 HOME 目录下的 .bladerc 文件，这是用户级的配置。
* BLADE\_ROOT 其实也是个配置文件，写在这里的是项目级配置。

后面描述的所有多个参数的配置的每个配置参数都有默认值，并不需要全部写出，也没有顺序要求。

**cc\_config**

所有c/c++目标的公共配置

cc\_config(

extra\_incs = ['thirdparty'], # 额外的 -I，比如 thirdparty

warnings = ['-Wall', '-Wextra'...], # C/C++公用警告

c\_warnings = ['-Wall', '-Wextra'...], # C专用警告

cxx\_warnings = ['-Wall', '-Wextra'...], # C++专用警告

optimize = '-O2', # 优化级别

)

所有选项均为可选，如果不存在，则保持先前值。发布带的blade.conf中的警告选项均经过精心挑选，建议保持。

**cc\_test\_config**

构建和运行测试所需的配置

cc\_test\_config(

dynamic\_link=True, # 测试程序是否默认动态链接，可以减少磁盘开销，默认为 False

heap\_check='strict', # 开启 gperftools 的 HEAPCHECK，具体取值请参考 gperftools 的文档

gperftools\_libs='//thirdparty/perftools:tcmalloc', # tcmclloc 库，blade deps 格式

gperftools\_debug\_libs='//thirdparty/perftools:tcmalloc\_debug', # tcmalloc\_debug 库，blade deps 格式

gtest\_libs='//thirdparty/gtest:gtest', # gtest 的库，blade deps 格式

gtest\_main\_libs='//thirdparty/gtest:gtest\_main' # gtest\_main 的库路径，blade deps 格式

)

所有的 config 的列表类型的选项均支持追加模式，用法如下：

cc\_config(

append = config\_items(

warnings = [...]

)

)

注意:

* gtest 1.6开始，去掉了 make install，但是可以绕过[<http://blog.csdn.net/chengwenyao18/article/details/7181514>gtest1.6.0安装方法]。
* gtest 库还依赖 pthread，因此gtest\_libs需要写成 ['#gtest', '#pthread']
* 或者把源码纳入你的源码树，比如thirdparty下，就可以写成gtest\_libs='//thirdparty/gtest:gtest'。

**proto\_library\_config**

编译protobuf需要的配置

proto\_library\_config(

protoc='protoc', # protoc编译器的路径

protobuf\_libs='//thirdparty/protobuf:protobuf', # protobuf库的路径，Blade deps 格式

protobuf\_path='thirdparty', # import 时的 proto 搜索路径，相对于 BLADE\_ROOT

protobuf\_include\_path = 'thirdparty', # 编译 pb.cc 时额外的 -I 路径

)

**thrift\_library\_config**

编译thrift需要的配置

thrift\_library\_config(

thrift='thrift', # protoc编译器的路径

thrift\_libs='//thirdparty/thrift:thrift', # thrift库的路径，Blade deps 格式

thrift\_path='thirdparty', # thrift中include时的thrift文件的搜索路径，相对于 BLADE\_ROOT

thrift\_incs = 'thirdparty', # 编译 thrift生成的.cpp 时额外的 -I 路径

)

所有这些配置项都有默认值，如果不需要覆盖就无需列入相应的参数。默认值都是假设安装到系统目录下，如果你的项目中把这些库放进进了自己的代码中（比如我们内部），请修改相应的配置。

**环境变量**

Blade还支持以下环境变量：

* 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

使用clang

CPP='clang -E' CC=clang CXX=clang++ ld=clang++ blade

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。

环境变量的支持将来考虑淘汰，改为配置编译器版本的方式，因此建议暂时不要使用。

**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。

我们编写了 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 hquickfix 特性）。

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 tools/.vimrc 中。

**alt**

在源代码目录和构建目标目录之间跳转

**安装**

执行install脚本即可安装到~/bin下，目前因还在开发阶段，变化还比较快，以软链方式安装，install后不能删除checkout出来的原始目录。 目前blade生成scons脚本，因此还需要安装scons 2.0以上版本。 Blade 需要支持 Python 2.4-2.7.x，不支持 python3。

install使得可以在任何目录下直接执行

$ blade

命令。 如果不行，确保~/bin在你的PATH环境变量里，否则修改 ~/.profile，加入

export PATH=~/bin:$PATH

然后重新登录即可。

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。