特征

- **High-level build language.**

  阅读友好

  面向库 目标文件 脚本和数据集 而非工具/命令

- **Bazel is fast and reliable.**

  保留历史文件内容和build指令——并行地、产生恰如其分地生产增量

- **Bazel is multi-platform.**

  产生多平台目标文件

- **Bazel scales.**

  适用于大规模代码和开发者

- **Bazel is extensible.**

  多语言支持

Bazel的解决方案

- 隔离：工作空间的概念隔离了不同版本的库
- 用户定义任何粒度的依赖
- 我们把这种依赖称为“对制品的依赖”，区别于对源代码的操控、复杂的脚本和工具使用
- 从用户手里收回定义构建过程的权利 由bazel实现增量式构建

在centOS7上安装

```
# cd /etc/yum.repos.d
# wget https://copr.fedorainfracloud.org/coprs/vbatts/bazel/repo/epel-7/vbatts-bazel-epel-7.repo
# yum install bazel
```

在ide中安装

下载示例

```
# git clone https://github.com/bazelbuild/examples
```

入门

使用Bazel的项目的结构

入门阶段，仅会做简单介绍。更多关于引用不同位置的内容、严谨的语法要求等内容，见 *进阶：核心概念*

```
|-- WORKSPACE
`-- main
    |-- BUILD
    `-- hello-world.cc
```

worksapce概念

在项目文件夹中创建一个名为WORKSPACE的文件，这标志着该WORKSPACE文件所在的文件夹（成为workspace）是一个使用Bazel的项目。

储存库概念

一个工作空间是储存库。一个别的文件夹（可能包含项目的代码）也是储存库。

可能会使用到不同的储存库中的内容，以 @储存库名字 这种格式来指明是哪一个储存库。

package概念

workspace中可以有包（package）,package即包含一个BUILD文件（或BUILD.bazel）的文件夹。

target概念

target可能是源文件、生成文件和指导生成过程的规则。

规则的输入可以是源文件或生成文件，而规则的输出只能是生成文件。

我们把生成文件称为**制品**。

lable概念

如何指明target？使用label。标签是目标的名字

Bazel如何知道怎样构建项目

Bazel根据BUILD文件中的内容来构建项目

例如，下面的代码内容依据一定的规则编写：

```
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
)
```

name = 标识生成的目标文件的名字 srcs = 标识源代码的文件名

而`cc_binary`标识着这是使用`cc_binary`规则的内容，称为`cc_binary`**的实例**

这段内容的意思是：

从源码hello-world.cc构造一个可执行二进制文件，不需要包含其他的文件，不需要依赖其他的组件

BUILD文件由若干个规则的实例构成，由于每一个实例都指出了该实例为了生成什么和相应的指导规则

我们称这样的一个实例为**规则块**或**构建目标（target）**，而name、srcs等需要指明的内容称为**属性（attribute）**

一阶：使用BUILD构建一个目标文件

```
cpp-tutorial/stage1
|-- README.md
|-- WORKSPACE
`-- main
    |-- BUILD
    `-- hello-world.cc
```

进入stage1目录（即此项目工作空间的根目录）执行：

```
[root@VM-186-82-centos ~/bazel_/examples/cpp-tutorial/stage1]# bazel build
//main:hello-world
```

`//main:`指出了`BUILD`文件此时的相对位置，即在main目录下

`hello-world`是我们在`BUILD`中的定义的一个构建目标（target）的名字

现在，构建的结果已经在工作空间中的`bazel-bin`目录下了，用以下命令来运行：

```
[root@VM-186-82-centos ~/bazel_/examples/cpp-tutorial/stage1]# bazel-
bin/main/hello-world
```

在一个构建过程用到的所有依赖应该都在BUILD文件中清晰地指出

Bazel会根据这些构建目标所指出的依赖关系构建一幅**依赖图（dependency graph）**，在依赖图的指导下，Bazel知道如何准确地、恰如其分地增量构建

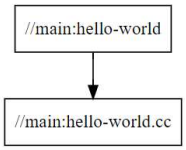
这样查询依赖图：

```
bazel query --notool_deps --noimplicit_deps "deps(//main:hello-world)" --output
graph
```

得到：

```
digraph mygraph {
  node [shape=box];
  "//main:hello-world"
  "//main:hello-world" -> "//main:hello-world.cc"
  "//main:hello-world.cc"
}
```

将以上结果借助http://www.webgraphviz.com/转换为一幅图



在一阶项目中，我们构建了hello-world可执行文件，没有使用任何外部依赖

二阶：多个规则块，但代码仍在一个package下

```
cpp-tutorial/stage2
|-- README.md
|-- WORKSPACE
`-- main
    |-- BUILD
    |-- hello-greet.cc
    |-- hello-greet.h
    `-- hello-world.cc
```

对小项目来说，构建单个制品就足够了。 但在大的项目中，拆分出多个目标文件、分出多个package盛放代码会让项目结构更加清晰，也使得**快速增量构建 （fast incremental builds）**成为可能。——i)只重新构建发生变化的制品 ii)如果制品之间没有依赖关系则可以并行构建

在stage2项目中，BUILD文件内容如下所示：

```
cc_library(
    name = "hello-greet",
    srcs = ["hello-greet.cc"],
    hdrs = ["hello-greet.h"],
)

cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = [
        ":hello-greet",
    ],
)
```

BUILD首先定义了生成 `hello-greet` 制品的规则。`cc_library` 指出这是一个使用 `cc_library` 规则的规则块（target）。

关于hdrs属性

`cc_library` 规则块中的hdrs属性指明#include行为可以用到的头文件

在某规则块A的hdrs属性中指明的头文件可以被以下文件包含：

i）块A的hdrs中指明的其他文件

ii）块A的srcs中指明的文件

iii）块A所依赖的制品，生成其的规则块中hdrs属性指明的文件

iv）块A所依赖的制品，生成其的规则块中srcs属性指明的文件

头文件也可以写在srcs属性中，假设头文件在某规则块A的srcs属性中被指明，能够#include其的文件仅限以下范围：

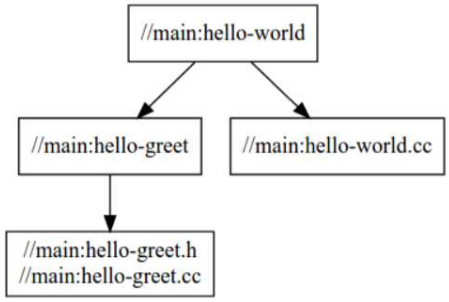
i）块A的hdrs中指明的其他文件

ii）块A的srcs中指明的文件

这类似于public和private机制。

然后生 `hello-world` 可执行文件。`deps` 属性指出：生成 `hello-world` 制品需要依赖 `hello-greet` 制品。

此时的依赖图：



执行命令：

```
bazel build //main:hello-world
bazel-bin/main/hello-world
```

如果修改了 `hello-greet.cc` 后重新构建项目，Bazel只会重新编译 `hello-greet.cc`，而不会重新编译 `hello-world.cc`。

Bazel默默安排好这一切，而程序员无需操心（显然，与依赖图有关）。

三阶：多package，多target的项目

```
cpp-tutorial/stage3
|-- README.md
|-- WORKSPACE
|-- lib
|   |-- BUILD
|   |-- hello-time.cc
|   `-- hello-time.h
`-- main
    |-- BUILD
    |-- hello-greet.cc
    |-- hello-greet.h
    `-- hello-world.cc
```

如上所示，stage3目录下有两个子文件夹，每个子文件夹下都包含一个BUILD文件。如前所述（见:使用Bazel的项目的结构）,现在工作空间中有两个package，分别是 `lib` 和 `main`。

`lib/BUILD` 中的内容：

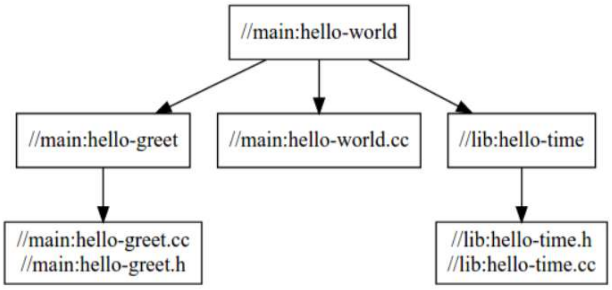
```
cc_library(
    name = "hello-time",
    srcs = ["hello-time.cc"],
    hdrs = ["hello-time.h"],
    visibility = ["//main:__pkg__"],
)
```

`main/BUILD` 中的内容：

```
cc_library(
    name = "hello-greet",
    srcs = ["hello-greet.cc"],
    hdrs = ["hello-greet.h"],
)

cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = [
        ":hello-greet",
        "//lib:hello-time",
    ],
)
```

此时的依赖关系如下



注意到`visibility`是一个新的属性，`visibility = ["//main:__pkg__"]`，指出了`hello-time`可以被main中的其他制品依赖。

如果没有用`visibility`属性直接指出，那么只有`lib/BUILD`中的制品可以依赖`hello-time`。

构建、运行stage3项目的过程如法炮制。

怎样标注依赖

我们使用**标签(label)**来指明依赖制品的位置。

```
//path/to/package:target-name
```

例如`//main:hello-world`和`//lib:hello-time`

- 如果依赖制品从规则块生成，则`path/to/package`是那个规则块的BUILD文件所在包的路径，`target-name`是由那个规则块的name属性指明的制品名称。
  - 特别地，若使用依赖的规则块和被依赖的规则块位于同一个BUILD文件中，`//`可以被省略，仅写作：`:target-name`（见stage2）
  - 特别地，若生成依赖制品的规则块所在的BUILD文件不在某一包中，而在工作空间的根目录中，写作：`//:target-name`即可
- 如果依赖制品不需由规则块生成，而是已经存在的文件，则`path/to/package`代表到制品所在文件夹的路径，而`target-name`是制品文件名（包含全路径）。
  - 同BUILD和根目录情况对`path/to/package`的处理同上

进阶

核心概念

worksapce概念

在项目文件夹中创建一个名为WORKSPACE的文件，这标志着该WORKSPACE文件所在的文件夹（成为workspace）是一个使用Bazel的项目。

WORKSPACE文件中可以包含对外部依赖的引用。

包含名为的文件`WORKSPACE`的目录被视为工作空间的根。因此，Bazel会忽略工作空间中以包含`WORKSPACE`文件的子目录为根的任何目录树（因为它们形成另一个工作空间）。

Bazel还支持将`WORKSPACE.bazel`file作为文件的别名`WORKSPACE`。如果两个文件都存在，`WORKSPACE.bazel`将具有优先权。

储存库概念

一个工作空间是储存库。一个别的文件夹（可能包含项目的代码）也是储存库。

可能会使用到不同的储存库中的内容，以 @储存库名字 这种格式来指明是哪一个储存库。

package概念

workspace中可以有包（package）,package即包含一个BUILD文件（或BUILD.bazel）的文件夹。

包是相关文件的集合以及它们之间的依赖性的规范。

target概念

target可能是源文件、生成文件和指导生成过程的规则。

规则的输入可以是源文件或生成文件，而规则的输出只能是生成文件。

我们把生成文件称为**制品**。

它们的关系是：

生成规则和输出始终在一个包下。

输入可以来自和规则所在的同一个/其他的包。

高级：package_group(包组)的概念

lable概念

如何指明target？使用label。标签是目标的名字

label常常配合储存库名字使用。

- 如何引用规则生成的制品？

规范形式的典型标签如下所示：

```
@ myrepo // my / app / main：app_binary
```

标签的第一部分是存储库名称`@myrepo//`。在同一储存库，存储库标识符可以缩写为`//`。因此，在`@myrepo`此标签内通常写为

```
// my / app / main：app_binary
```

标签的第二部分是相对于存储库根目录的程序包路径。当标签指向使用的相同包装时，可以省略包装名称（以及可选的冒号）。

在包`@myrepo//my/app/main`内引用该目标，可以使用以下两种方式之一编写此标签：

```
app_binary
：app_binary
```

- 如何引用文件？

文件input.txt位于`my/app/main/testdata`存储库的子目录中，main是一个包名：

```
//my/app/main:testdata/input.txt
```

在一个包中引用另一个包中的文件，例如，如果工作空间中同时包含包`my/app`和包`my/app/testdata`（即，这两个包中的每一个都有其自己的BUILD文件）。后者包含一个名为的文件`testdepot.zip`。这里有两种方法（一种错误，一种正确）来引用此文件`//my/app:BUILD`：

```
testdata / testdepot.zip    #错误：testdata是一个不同的软件包。
//my/app/testdata:testdepot.zip#正确。
```

如果错误地`testdepot.zip`使用了错误的标签（例如`//my/app:testdata/testdepot.zip`或）进行引用`//my:app/testdata/testdepot.zip`，则会从构建工具中收到一条错误消息，指出标签“跨越了包装边界”。您应该通过将冒号放在包含最里面的BUILD文件的目录之后来更正标签`//my/app/testdata:testdepot.zip`。

命名语法

目标命名

规则块的名字（name）的合法字符集

```
a-z A-Z 0-9 !%-@^_` "#$&'()*-+,;<=>?[]{|}~/.
```

（以上字符集包含一个空格）

但不能以斜杠开头和结尾，也不能包括多个斜杠。

引用其他包中的内容时不可以使用../这种方式

包命名

集合如下

`A-Z`, `a-z`, `0-9`, '`/`', '`-`', '`.`', and '`_`', 并且不能以斜杠开头

BUILD文件的规定

不同的构建规则做什么

`*_binary`规则以给定的语言构建可执行程序`cc_binary`则将c++编写的文件生成可执行程序。

同时生成一个.runfiles为后缀的文件目录。包括构建目标和运行时所需要的文件。（使用data属性标注）

具体示例在*使用外部项目*中

`*_test`规则是一种特殊化的`*_binary`，用于自动化测试

cc_import： 允许用户导入预编译的`C/C++`库，包括动态库、静态库

cc_library： 生成动/静态库

- cc_proto_library： 从`.proto`文件生成`C++`代码
- fdo_prefetch_hints： 表示位于工作区中或位于指定绝对路径的 FDO 预取提示配置文件
- fdo_profile： 表示工作区中或位于指定绝对路径的 FDO 配置文件
- cc_test： 测试`C/C++`样例
- cc_toolchain： 表示一个`C++`工具链
- cc_toolchain_suite： 表示`C++`工具链的集合

总是声明直接依赖

避免潜在错误

https://docs.bazel.build/versions/master/build-ref.html

只有直接依赖需要被写出

假设头文件 `sandwich.h` 中有`#include "bread.h"`，`bread.h` 中有`#include "flour.h"`

`sandwich.h` 中没有 `#include "flour.h"` `BUILD` 文件应如下所示：

```
cc_library(
    name = "sandwich",
    srcs = ["sandwich.cc"],
    hdrs = ["sandwich.h"],
    deps = [":bread"],
)

cc_library(
    name = "bread",
    srcs = ["bread.cc"],
    hdrs = ["bread.h"],
    deps = [":flour"],
)

cc_library(
    name = "flour",
    srcs = ["flour.cc"],
    hdrs = ["flour.h"],
)
```

通用属性说明

srcs：直接使用的源文件

deps：要依赖的被单独编译的模块

data：运行时使用的文件

在代码中，使用运行时相对路径访问文件

指代一个目录下的所有文件时

查看我们的`BUILD`文件时，您可能会注意到一些`data`标签指向目录。这些标签以`/.`或`/`类似结尾：

```
data = ["//data/regression:unittest/."]  # don't use this
```

或像这样：

```
data = ["testdata/."]  # don't use this
```

或像这样：

```
data = ["testdata/"]  # don't use this
```

这似乎很方便，特别是对于测试（因为它允许测试使用目录中的所有数据文件）。

但是，请不要这样做!!

为了确保更改后正确的增量重建（和测试的重新执行），构建系统必须知道作为构建（或测试）输入的完整文件集。当您指定目录时，仅当目录本身更改时（由于添加或删除文件），构建系统才会执行重建，但是由于这些更改不会影响封闭目录，因此无法检测到对单个文件的编辑。

显示地使用`glob()`函数枚举其中包含的文件集，增量构建时将会检测目录中的文件是否发生了变化。（`**`用于强制递归。）

```
data = glob（[“ testdata / **”]）＃改用它
```

如果必须使用目录标签，请记住，您不能使用相对`../`路径来引用父包；相反，请使用“`//data/regression:unittest/.`”之类的绝对路径。

请注意，目录标签仅对数据依赖项有效。如果您尝试将目录用作以外的参数中的标签，则该目录`data`将失败，并且您将收到（可能是隐秘的）错误消息。

```
cc_library(
    name = "build-all-the-files",
    srcs = glob(["*.cc"]),
    hdrs = glob(["*.h"]),
)
```

使用这种写法，Bazel会指定BUILD所在包中的所有`.cc`文件和`.h`文件。（递归地分析子文件夹）

包含头文件的路径问题

假设some_lib.cc和some_lib.h原先在同一个目录下，那么在some_lib.cc中#include "some_lib.h"是没有问题的

但也许你不愿意把源文件和包含头文件放在同一个目录下。当目录层级变成这样：

```
└── my-project
    ├── legacy
    |   └── some_lib
    |       ├── BUILD
    |       ├── include
    |       |   └── some_lib.h
    |       └── some_lib.cc
    └── WORKSPACE
```

在some_lib.cc中#include "some_lib.h"将不再生效，g++ some_lib.cc将会报错。此时，将语句改为#include "../include/some_lib.h"才能通过编译

如此一来，移动头文件的相对位置将造成源代码的频繁修改。Bazel的解决方案是：使用**copts**和**-I**标注被包含头文件存在的位置，而不必在源代码中解决相对路径的问题。

如下方所示：

```
cc_library(
    name = "some_lib",
    srcs = ["some_lib.cc"],
    hdrs = ["include/some_lib.h"],
    copts = ["-Ilegacy/some_lib/include"],
)
```

copts含义

将这些选项添加到C ++编译命令。

visibility:可见性范围

制品本身控制其他的规则块是否可以使用它。可以用来区分实现细节和公共API

使用 `--check_visibility=false` 禁用可见性检查。

- `"//visibility:public"`: 任何人可用
- `"//visibility:private"`: 仅同一包中的规则可用
- `"//foo/bar:__pkg__"`: 仅 `//foo/bar` 包可用（但不包含其子包） `__pkg__` 代表一个包中的所有目标
- `"//foo/bar:__subpackages__"`: `//foo/bar` 包、所有直接子包、间接子包可用
- `"//foo/bar:my_package_group"`: my_package_group是一个包的群组

  （不太懂）

  - Package groups do not support the special `__pkg__` and `__subpackages__` syntax. Within a package group, `"//foo/bar"` is equivalent to `"//foo/bar:__pkg__"` and `"//foo/bar/..."` is equivalent to `"//foo/bar:__subpackages__"`.

As a special case, `package_group` targets themselves do not have a `visibility` attribute; they are always publicly visible.

默认的visibility

如果规则块未设置该 `visibility` 属性，则其可见性由所在BUILD文件 `default_visibility` 的 `package` 语句中指定的可见性给出 。如果没有这样的 `default_visibility` 声明，则可见性为 `//visibility:private` 。

文件可见性

访问另外一个包中的文件时，需要在被使用的文件所在的包中的BUILD文件中用exports_files（）注明

高级：

生成动态链接库

使用到的规则是cc_binary（）

Create a shared library. To enable this attribute, include `linkshared=True` in your rule. By default this option is off.

The presence of this flag means that linking occurs with the `-shared` flag to `gcc`, and the resulting shared library is suitable for loading into for example a Java program. However, for build purposes it will never be linked into the dependent binary, 我们假设使用cc_binary生成的.so只作为被其他的程序显示地链接使用！！！不应该认为这是cc_library rule的替代！！！！. For sake of scalability we recommend avoiding this approach altogether and simply letting `java_library` depend on `cc_library` rules instead.

If you specify both `linkopts=['-static']` and `linkshared=True`, you get a single completely self-contained unit. If you specify both `linkstatic=1` and `linkshared=True`, you get a single, mostly self-contained unit.

选项

Attributes	
`linkopts`	将这些标志添加到C ++链接器命令。
`linkshared`	`linkshared=True` 代表生成共享库 等价于gcc的-shared
`linkstatic`	在可能使用 `.a` 的情况下不链接 `.so` 用户库 而系统库shiyong.so `linkstatic` 如果用于 `cc_library()` 规则，则 该属性具有不同的含义 。对于C ++库，`linkstatic=True` 表示仅允许静态链接，因此不会 `.so` 产生。linkstatic = False 不会阻止创建静态库。该属性用于控制动态库的创建。

练习

现为以下代码生成动态链接库：

```
#include <string>

std::string make_string(){
return "test";
}
```

对应的BUILD文件中的规则块:

```
cc_binary(
    name = "libmakestring.so",
    srcs = ["make_string.cpp"],
copts = ["-g"],
linkopts = ["-lstdc++"],
linkshared = True,
linkstatic = True,
)
```

执行：

```
 bazel build //yaml_:libmakestring.so
```

将生成的.so文件复制到/usr/local/lib

测试代码：

```
#include <iostream>
std::string make_string();
int main(){
std::cout<<make_string()<<std::endl;
return 0;
}
```

编译

```
g++ useso.cpp /usr/local/lib/libmakestring.so
```

运行及结果

```
[root@VM-186-82-centos ~/bazel_/use_so]# ./a.out
test
```

生成静态链接库

使用cc_library

```
cc_library(
    name = "makestring",
    srcs = ["make_string.cpp"],
linkstatic = True,
)
```

执行

```
bazel build //yaml_:makestring
```

得到：

INFO: Analyzed target //yaml:*makestring (1 packages loaded, 2 targets configured).*
INFO: Found 1 target...
Target //yaml:makestring up-to-date:
  bazel-bin/yaml_/libmakestring.a
INFO: Elapsed time: 0.134s, Critical Path: 0.07s
INFO: 2 processes: 2 processwrapper-sandbox.
INFO: Build completed successfully, 4 total actions

包含外部的库：以gtest为例

建议首先参考官方文档

下载gtest代码

方式一：配置WORKSPACE

方式二：手动下载

把含有src include的那个google文件夹改名为gtest放在workspace下

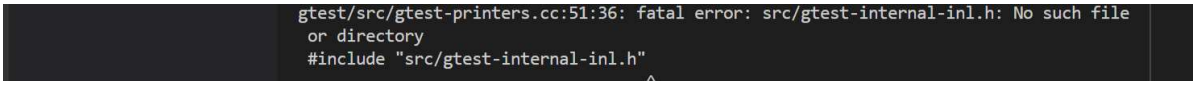
然后，在workspace/gtest中写入BUILD

BUILD内容：

```
cc_library(
    name = "main",
    srcs = glob(
        ["src/*.cc"],
        exclude = ["src/gtest-all.cc"]
    ),
    hdrs = glob([
        "include/**/*.h",
        "src/*.h"
    ]),
    copts = ["-Igtest/include","-Igtest/",],
    linkopts = ["-pthread"],
    visibility = ["//visibility:public"],
)
```

`copts = ["-Igtest/include","-Igtest/",]`,配置的规则：如果报错，要参考代码中引用头文件时的写法

例如：



说明-I后的相对路径应该写到src的上级为止，且应该从WORKSPACE文件所在的层级写起

使用bazel和gtest

现在，使用gtest测试在stage项目中使用过的函数，我们的workspace现在是这样的：（hello-time没啥用）

```
.....gtest和WORKSPACE之类的
|-- lib
|   |-- BUILD
|   |-- hello-time.cc
|   `-- hello-time.h
|-- main
|   |-- BUILD
|   |-- hello-greet.cc
|   |-- hello-greet.h
|   `-- hello-world.cc
`-- test
    |-- BUILD
    `-- hello-test.cc
```

以下是`hello-test.cc`源码和产生`hello-test`制品的规则块

```
[root@VM-186-82-centos ~/bazel_/use_gtest/test]# ls
BUILD  hello-test.cc
[root@VM-186-82-centos ~/bazel_/use_gtest/test]# cat hello-test.cc
#include "gtest/gtest.h"
#include "main/hello-greet.h"

TEST(HelloTest, GetGreet) {
  EXPECT_EQ(get_greet("Bazel"), "Hello Bazel");
}
[root@VM-186-82-centos ~/bazel_/use_gtest/test]# cat BUILD
cc_test(
    name = "hello-test",
    srcs = ["hello-test.cc"],
    copts = ["-Igtest/include"],
    deps = [
        "//gtest:main",
        "//main:hello-greet",
    ],
)
```

`"gtest/gtest.h"`头文件在`use_gtest/gtest/include`目录下

`#include "main/hello-greet.h"`写法在Bazel中是可行的，因为Bazel会在构建时到workspace中去寻找文件。

这和我们使用g++编译不同，使用g++编译时，须写作`#include "../main/hello-greet.h"`，这种相对路径的写法在Bazel中也是行得通的。

下图为使用g++编译出错：

```
[root@VM-186-82-centos ~/bazel_/use_gtest]# g++ test/hello-test.cc -o out
test/hello-test.cc:2:30: fatal error: main/hello-greet.h: No such file or
 directory
 #include "main/hello-greet.h"
```

别忘了修改可见度！`hello-test`制品的生成需要依赖`gtest`包和`main`包中的制品，在`gtest/BUILD`中，我们注明了

`visibility = ["//visibility:public"]`，相应地，应修改`hello-greet`制品的规则块，如下：

```
cc_library(
    name = "hello-greet",
    srcs = ["hello-greet.cc"],
    hdrs = ["hello-greet.h"],
visibility = ["//test:__pkg__"],
)
```

现在，在工作空间的根目录执行

```
bazel test test:hello-test
```

现在可以使用`bazel test`来运行刚刚生成的test类制品了！

```
bazel test test:hello-test
```

如果修改源代码使得测试结果为错误，重新构建项目，在bazel test的指令下运行测试制品，将会得到输出的错误信息在日志中的提示

```
INFO: Build completed, 1 test FAILED, 2 total actions
//test:hello-test
FAILED in 0.0s
  /root/.cache/bazel/_bazel_root/ab86d363d7d126be8506d26b63b758fb/execroo
t/__main__/bazel-out/k8-fastbuild/testlogs/test/hello-test/test.log
```

查看这个日志的内容就会发现这正是gtest测试报错内容。

对预编译依赖的指定

如果你想生成一个只编译一次的库制品，比如头文件和.so文件，使用cc_library规则编写：

```
cc_library(
    name = "mylib",
    srcs = ["mylib.so"],
    hdrs = ["mylib.h"],
)
```

工作空间中的其他C++制品可以依赖它。

链接和依赖外部项目

链接.a文件

在包mylib下存在libyaml-cpp.a

为了**使其他包中的制品可以使用它**，应在`mylib/BUILD`中编写生成制品的规则块，如下：

```
cc_library(
    name = "yaml",
    srcs = ["libyaml-cpp.a"],
 visibility = ["//yaml:__pkg__"],
)
```

`visibility`标注其对yaml包中的制品可见

yaml包中的代码如下：

```
#include "gtest/gtest.h"
#include <yaml-cpp/yaml.h>
#include <string>
using namespace std;
TEST(HelloTest, GetGreet) {
        YAML::Node config;
    try
    {
        config = YAML::LoadFile("config.yml");
    }
    catch (YAML::BadFile &e)
    {
        std::cout << "read error!" << std::endl;
        exit(0);
    }
 EXPECT_EQ(config["test"].as<string>(),"test");
}
```

其BUILD文件对应如下:

```
cc_binary(
    name = "hello-test",
    srcs = ["hello-test.cc"],
    copts = ["-Igtest/include",],
    deps = [
        "//gtest:main",
        "//mylib:yaml",
    ],
)
```

依赖另一个Bazel项目的制品

用以下目录层级举例：

```
/
  home/
    user/
      project1/
        WORKSPACE
        BUILD
        srcs/
          ...
      project2/
        WORKSPACE
        BUILD
        my-libs/
```

如果在 `project1` 依赖制品 `:foo` （定义在 `/home/user/project2/BUILD` 中），那么在 `/home/user/project1/BUILD` 中以 `@project2//:foo` 指出这个制品。

而使用 `@project2` 需要在 `project1` 的 `WORKSPACE` 中这样写：

```
local_repository(
    name = "coworkers_project",
    path = "/path/to/coworkers-project",
)
```

External project names must be valid workspace names, so `_` (valid) is used to replace `-` (invalid) in the name `coworkers_project`.

练习：使用外部Bazel项目

功能概述：

`use_yaml` 项目提供读取yaml配置文件并返回内容的功能单元

`use_gtest` 调用 `use_yaml` 的功能单元并使用gtest进行测试

依赖关系：

`use_yaml` 使用的静态链接库在该项目的一个包中，通过cc_library规则注册，调用者使用deps属性指明。

`use_gtest` 使用的gtest源代码在 `use_gtest` 的包中。

`use_gtest` 通过配置其WORKSPACE文件使用 `use_yaml` 项目中的制品，而无需配置对yaml的依赖。

项目结构：

```
use_yaml/
|-- WORKSPACE
|-- mylib
|   |-- BUILD
|   `-- libyaml-cpp.a
`-- yaml
    |-- BUILD
    `-- yaml_unit.cc
use_gtest/
|-- WORKSPACE
|-- config
|   |-- BUILD
|   `-- test_config.yml
|-- gtest
|   |-- BUILD
    ……
|-- test
|   |-- BUILD
|   `-- use_another.cc
```

依赖配置：

use_yaml/mylib/BUILD：

```
cc_library(
    name = "yaml",
    srcs = ["libyaml-cpp.a"],
 visibility = ["//yaml:__pkg__"],   #指明本workspace中的yaml包可以使用制品yaml
)
```

use_yaml/yaml/BUILD：

```
cc_library(
    name = "use_yaml",
    srcs = ["yaml_unit.cc"],
    deps = [
        "//mylib:yaml", #使用mylib包中的yaml制品
    ],
visibility = ["//visibility:public"],   #可以被任何包使用，也可以被其他项目使用
)
```

构建use_yaml项目：

```
[root@VM-186-82-centos ~/bazel_/use_yaml]# bazel build //yaml:use_yaml
INFO: Analyzed target //yaml:use_yaml (0 packages loaded, 0 targets configured).
INFO: Found 1 target...
Target //yaml:use_yaml up-to-date:
  bazel-bin/yaml/libuse_yaml.a
  bazel-bin/yaml/libuse_yaml.so
INFO: Elapsed time: 0.058s, Critical Path: 0.00s
INFO: 0 processes.
INFO: Build completed successfully, 1 total action
```

use_gtest/gtest/BUILD:

```
cc_library(
    name = "main",
    srcs = glob(
        ["src/*.cc"], #这样写 src是以BUILD文件所属的包为根目录
        exclude = ["src/gtest-all.cc"] #因为gtest-all.cc包含其他的源文件
    ),
    hdrs = glob([
        "include/**/*.h",
        "src/*.h"
    ]),
    copts = ["-Igtest/include","-Igtest/",],
    linkopts = ["-pthread"],
    visibility = ["//visibility:public"],
)
```

构建gtest的main制品:

```
[root@VM-186-82-centos ~/bazel_/use_gtest]# bazel build //gtest:main   INFO:
Analyzed target //gtest:main (13 packages loaded, 77 targets configured).
INFO: Found 1 target...
Target //gtest:main up-to-date:
  bazel-bin/gtest/libmain.a
  bazel-bin/gtest/libmain.so
INFO: Elapsed time: 2.544s, Critical Path: 2.27s
INFO: 11 processes: 11 processwrapper-sandbox.
INFO: Build completed successfully, 14 total actions
```

源代码use_gtest/test/use_another.cc:

```
#include "gtest/gtest.h" #需要在BUILD文件中使用copts选项处理 因为gtest/gtest.h在项目空间的gtest/include目录下
#include <string>
using namespace std;
string use_yaml();
TEST(HelloTest, GetGreet) {
  EXPECT_EQ(use_yaml(),"test");
}
```

use_gtest/test/BUILD:

```
cc_binary(
    name = "read_yaml",
    srcs = ["use_another.cc",],
    copts = ["-Igtest/include",],
    deps = [
        "//gtest:main",
        "@use_yaml//yaml:use_yaml",#使用到了另一个项目的制品 还需配置本项目的WORKSPACE
    ],
)
```

use_gtest/WORKSPACE:

```
local_repository(
    name = "use_yaml",
    path = "/root/bazel_/use_yaml", #使用了绝对路径
)
```

构建在use_gtest中构建read_yaml制品：

```
[root@VM-186-82-centos ~/bazel_/use_gtest]# bazel build //test:read_yaml
INFO: Analyzed target //test:read_yaml (3 packages loaded, 7 targets
configured).
INFO: Found 1 target...
Target //test:read_yaml up-to-date:
  bazel-bin/test/read_yaml
INFO: Elapsed time: 0.623s, Critical Path: 0.52s
INFO: 3 processes: 3 processwrapper-sandbox.
INFO: Build completed successfully, 7 total actions
```

运行read_yaml制品：

```
[root@VM-186-82-centos ~/bazel_/use_gtest]# ./bazel-bin/test/read_yaml
Running main() from gtest/src/gtest_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from HelloTest
[ RUN      ] HelloTest.GetGreet
[       OK ] HelloTest.GetGreet (0 ms)
[----------] 1 test from HelloTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

依赖不使用Bazel的项目（eg.外部.so文件）

在WOERKSPACE文件中使用new开头的规则。

现以引用make的外部项目`my-project/`中的.so文件为例，bazel项目为`coworkers-project/`，

在`my_project/WORKSPACE`中写一条new_规则

```
new_local_repository(
    name = "coworkers_project",
    path = "/path/to/coworkers-project",
    build_file = "coworker.BUILD",
)
```

`build_file` specifies a BUILD file to overlay on the existing project, for example:

```
cc_library(
    name = "some-lib",
    srcs = glob(["**"]),
    visibility = ["//visibility:public"],
)
```

You can then depend on `@coworkers_project//:some-lib` from your project's BUILD files.

练习：使用非Bazel的外部项目

功能概述：

`no_bazel_yaml` 作为不使用Bazel的外部项目

`use_gtest` 项目的yaml_包use_yaml模块：读取yaml配置文件

test包的read_yaml模块：调用yaml_包use_yaml模块并使用gtest进行测试

和前一个练习的区别之处在于，使用的外部项目是非Bazel的，所以`use_gtest`的WORKSPACE需要修改：

使用`new_`来引入外部非Bazel项目

使用`build_file`来为这个外部项目指定一个位于本项目工作空间中的BUILD文件

```
new_local_repository(
    name = "no_bazel_yaml",
    path = "/root/bazel_/no_bazel_yaml",
    build_file = "external_project_no_bazel_yaml.BUILD",
)
```

例如，`external_project_no_bazel_yaml.BUILD`的内容如下

```
cc_library(
    name = "yaml",
    srcs = ["libyaml-cpp.a"],
    visibility = ["//visibility:public"],
)
```

因为这是一个为外部项目所写的BUILD文件，所以`srcs = ["libyaml-cpp.a"]`指定的文件是相对于外部项目的路径。

此时，test包BUILD新增规则块：

```
cc_binary(
    name = "lib_no_bazel_yaml",
    srcs = ["use_another.cc",],
    copts = ["-Igtest/include",],
    deps = [
        "//gtest:main",
"//yaml_:use_yaml", #这一行是yaml_包内制品的依赖
"@no_bazel_yaml//:yaml", #这一行就是对引入的外部项目的依赖 其实依据依赖的传递性 不必写出
    ],
)
```

yaml_包和之前使用bazel的外部项目的yaml包没什么不同，其BUILD文件如下：

```
cc_library(
    name = "use_yaml",
    srcs = ["yaml_unit.cc"],
    deps = [
        "@no_bazel_yaml//:yaml", #这一行就是对引入的外部项目的依赖
    ],
visibility = ["//visibility:public"],
)
```

构建lib_no_bazel_yaml

```
[root@VM-186-82-centos ~/bazel_/use_gtest]# bazel build //test:lib_no_bazel_yaml
```

使用c/c++库 cc_import()

使用现有的.a或.so文件只需将其写到srcs属性中

引入预编译的库

1.引入静态库

```
cc_import(
  name = "mylib",
  hdrs = ["mylib.h"],
  static_library = "libmylib.a",
  # alwayslink 位指示是否强制为所有依赖该制品的制品连接静态库
  # alwayslink = 1,
)
```

2.连接

```
cc_import(
  name = "mylib",
  hdrs = ["mylib.h"],
  shared_library = "libmylib.so",
)
```

3.

```
cc_import(
  name = "mylib",
  hdrs = ["mylib.h"],
  static_library = "libmylib.a",
  shared_library = "libmylib.so",
)

# first will link to libmylib.a
cc_binary(
  name = "first",
  srcs = ["first.cc"],
  deps = [":mylib"],
  linkstatic = 1, # default value
)

# second will link to libmylib.so
```

```
cc_binary(
  name = "second",
  srcs = ["second.cc"],
  deps = [":mylib"],
  linkstatic = 0,
)
```