彻底理解链接器：六，大型项目是如何被构建出来的

Original 码农的荒岛求生 码农的荒岛求生 2018-09-26 21:00

收录于话题
#链接器
6个 >

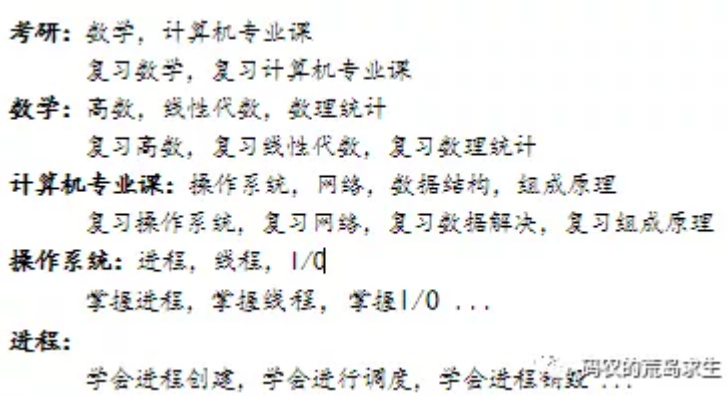
在讲解大型项目如何被构建之前，我们首先来讨论一个问题，有句话说的很好，梦想总是要有的，万一实现了呢，那么问题来了，要怎么实现呢，这里就涉及到了如何实现目标，

目标是如何实现的

其实很简单，本质上只有两点：

- 知道最后想要的是什么
- 为此需要做些什么

有时我们的目标可能不是简单的诸如每天跑五公里之类，比如像通过一门考试，学会一项技能这样的系统性工程。这时我们可能一下子不知道要做些什么，那么这就需要进行任务分解了，即这里的规则就是，把一个大的目标分解为一个个小的目标，如果对于其中一个小的目标还是不够具体，那么就继续将小目标进行分解，直到将每个小目标分解为如每天读懂两个章节，做完十个练习题之类很具体可以马上实施的任务为止。到这时，对于如何实现这个大的目标就很清晰了，只需要严格按照计划去实施就好了。比如对于考研，我们就可以列出如下的计划：



在考研这个例子中我们就按照上述规则将目标进行了分解，每个目标都按如下格式列出：

> *目标(target): 依赖什么*
> *要怎么做*

如果"要怎么做"还不是一个具体的目标就继续分解，直到分解为类似进程这样的目标，因为像进程这样的目标已经有了具体的实现步骤。最后我们将各个已经实现的小目标汇集起来整个大的目标就实现了。

本质上，一个大型项目的构建过程与此类似。

Make

再大的项目最后生成的都是一个可执行文件，只要是可执行文件就需要依赖各种目标文件，动态库，静态库；静态库同样需要依赖其它目标文件，静态库；而动态库可能又依赖其它目标文件，动态库，静态库，知道了这些又该如何构建呢，我们可以利用上面目标划分的方法规划好构建最终的可执行文件需要哪些原材料，这些原材料又是如何获取的。有了这些规划后，我们就可以依次编译出一些小的目标文件，将这些目标文件链接成静态库，动态库以方便使用。然后再一步步连接目标文件以及各种库从而形成更大的库，最后将几个必要库以及目标文件进行链接从而生成最终的可执行文件。

程序员先驱们确实就是使用这种现在看起来非常原始非常古老的方法进行程序编写的，每个目标文件以及库都是自己手动编译链接出来，然后再将它们链接成更大的库，直到最后生成可执行文件。

这种方法看上去非常简单，但是缺点也很明显，那就是非常繁琐，一旦某个源文件进行了改动，所有依赖此文件的库都需要重新编译链接，手工来完成这项工作是极其枯燥且容易出错的。为解决这个问题，天才的程序员们想出了一个小工具，没错就是make，从此编译链接这个过程就被make自动化了，程序员得以从繁琐的编译链接中解放出来，使用make时我们只需要编写规则，也就是告诉make最终的可执行文件依赖什么，为此需要做些什么，这些规则类似于上面的目标分解，当编写好这些规则后，然后简单的执行一个命令也就是make就可以了。如果某个源文件被修改了，也只需要简单的重新执行一下make命令，因为整个过程的规则并没有改变，而make也会很聪明的只编译链接那些需要更新的目标文件，库，并重新进行可执行文件的生成。对于那些没有改动的源文件，make不会重新编译它们。

make中每一条规则与前面的目标划分非常相似，make的规则是这样的：

> ***target: prerequisites***
> ***recipe***

target也就类似于我们的一个目标；而prerequisites，即先决条件，也就是依赖什么；recipe，这个就更形象了，即菜谱，也就是上面的要怎么做。make中的规则保存在了叫做Makefile的文件当中(没错，这个文件的名字就叫做Makefile)，当运行make命令时，make程序会自动找到当前路径下的Makefile，然后开始执行里面的规则。

有些同学可能为此感到疑惑，这里的Makefile其实就是脚本，而make读取这个脚本然后根据里面的内容来执行命令，而对于make大家也不要觉得很神奇，make也是一个普通程序，和我们平时使用的程序没什么区别。确定好了make需要执行的脚本的名字，这样在运行make命令时就少打了几个单词，假如用户可以自定义make的执行脚本名字，比如用户创建了一个脚本叫做foo，那么执行make的时候就需要多打一个单词“make foo”，所以干脆就直接确定好了脚本的名字就叫Makefile，这样在运行命令时只需要打一个单词make就可以了。

这里举个简单的例子，比如我们写了一个helloworld程序，将源文件命名为了helloworld.c，我们想把该源文件编程成一个叫做hw的可执行文件，那么一个最简单的Makefile就可以写成这样：

```
hw: helloworld.o
	gcc helloworld.o -o hw
helloworld.o : helloworld.c
	gcc -c helloworld.c
```

在这里最终的可执行文件hw依赖目标文件helloworld.o，那么假设我们现在已经有helloworld.o了就可以利用命令gcc helloworld.o -o hw生成我们需要的可执行文件了。那么helloworld.o又该如何获得呢？我们看第二条规则，helloworld.o依赖helloworld.c，因为helloworld.c已经写好了，所以可以直接用命令gcc -c helloworld.c来生成。这样整个目标就达成了。

本质上现在我们使用的各种集成开发环境(IDE)，其自动化编译工具背后的原理和make是一样的，比如我们在使用Visual Studio时从来没有关心过每个文件是如何被编译链接的，这些IDE都为我们代劳了。但是在比如Linux环境下进行开发时，这个过程依然是需要程序员了解的。

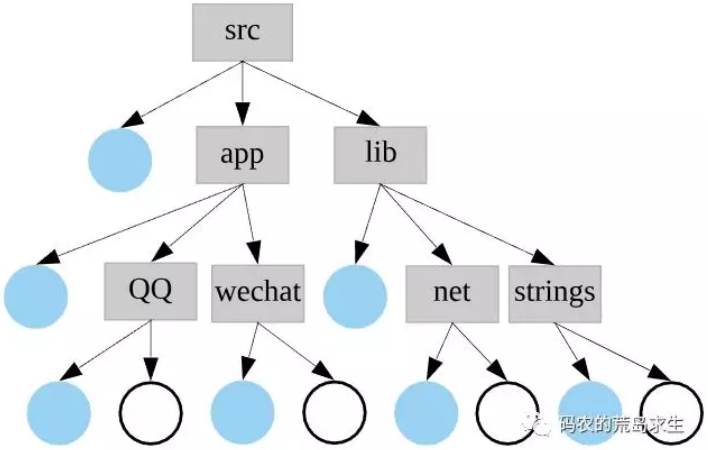
现在让我们来回答本节提出的问题，也就是大型项目是如何被构建的。

构建大型项目

大型项目中通常会有成百上千甚至上万个源文件，这些源文件统一放在了一个文件夹中方便管理。典型的项目如图所示，圆形代表源文件，其它为文件夹。注意这里仅仅为说明问题，各个公司团队都有自己的代码组织以及命名方式，而且真实项目要比该图复杂的多，但是本质上这里的讨论适用于其它情况。

源码组织方式

通常项目的组织方式如下图所示：



项目源码会被放置在src当中，这个例子当中src下有两个文件夹，lib以及app，lib用于存放一些工具性的代码，比如这里列举的网络通信以及字符串处理模块，通常lib下的代码会被编译成各种库，方便app使用。app中就是各种需要可执行文件(程序)的代码了。通常像这里的lib以及app都会有专门的团队来负责。更大一些的项目，每个lib下的子目录比如这里的net，strings都会有专门的团队来负责以方便项目的模块化管理。

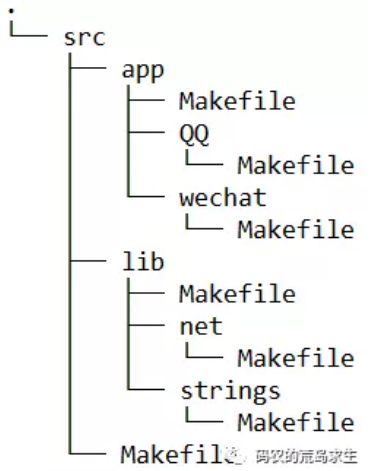
从这里可以看出一般项目通常会按模块将源文件放入相应的文件夹下进行分类，我们在上一节中简单介绍了make的用法，但是那里仅仅需要编译一个源文件helloworld.c。对于如上图所示的项目，像make这一类的编译工具又该如何处理呢？

make的嵌套执行能力可以解决这个问题。比如对于模块net，你可以为net模块写一个单独的Makefile，该Makefile只用于编译net下的源文件，具体的脚本如下所示，只需要简单的两行。

```
network:
	cd net && make
```

这句话的意思是告诉make，要想编译网络模块(network)需要进到net文件夹并且执行make命令，当make进入到net文件夹开始执行make时，net下的Makefile就开始被执行了。通过这样一个简单的命令就可以实现make的嵌套执行了。make的这项特性使得每个模块都可以当做独立项目进行维护。

编译工具的这项功能，方便了项目的模块化管理。使得项目中每个模块都可以有独立的编译脚本，比如使用make进行编译的话，那么每个模块中都会有单独的Makefile，比如在文件夹net，strings中都有自己的Makefile。如上图中蓝色部分，其中白色部分为源文件，更清晰的关于Makefile的组织方式如下图所示：



这些脚本中定义了如何编译该模块，以及编译该模块需要依赖什么。这些模块的父目录也就是lib文件夹下同样也有自己的Makefile，lib下的Makefile会收集各个子模块的编译结果，然后将其链接成各种库。而对于app下面的子目录来说，这些子目录中就是各个可执行文件的源码了，比如这里的wechat文件下就是可执行程序微信的源码了，微信中可能会用到lib下提供的功能，那么对于wechat中的Makefile来说，只需要简单的加入对lib中所需要的库的依赖就可以了。wechat的父目录app中同样也有Makefile，这里的Makefile就相对简单了，只需要依次执行QQ，wechat中的Makefile就可以了，因此在src目录下简单的运行make命令，所有app比如QQ和wechat就都被编译出来了。

接下来我们详细的讲解一下这个过程。请注意一点，接下来讲解的make执行过程仅仅是可能的一种实现方式，但是这个示例已足够说明项目的构建过程。

make的执行过程

在上面的示例中src下的Makefile是整个编译过程的入口，因此我们进入src文件夹开始执行make命令。

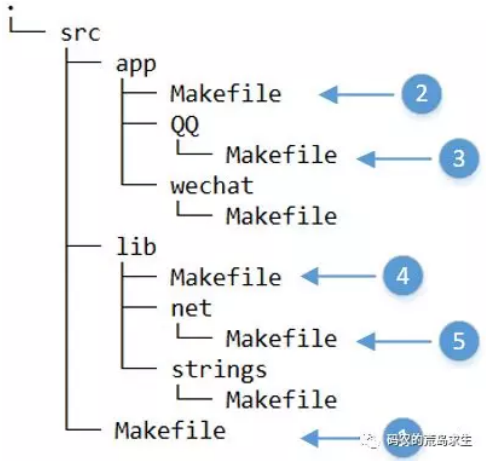
（1）在src目录下，make首先读取src下的Makefile，./src/Makefile非常简单，该文件仅仅告诉make需要去app目录下执行make命令。

（2）make来./src/app目录下，开始读取该目录下的Makefile，该文件定义了编译出QQ，微信的规则，make首先执行编译QQ的规则，该规则告诉make编译QQ则需要到./src/app/QQ目录并执行make命令。

（3）make来到./src/app/QQ目录下，开始读取该目录下的Makefile，该文件定义了编译QQ程序 的规则，make开始执行这些规则，其中一项规则需要依赖网络模块的库，同时该规则告诉了make如果想得到该网络库则需要进入到./src/lib下执行make命令。

（4）make来到./src/lib目录下，开始读取该目录下的Makefile，该文件定义了编译出网络库，字符串处理库的规则，make首先执行编译网络库的规则，该规则告诉make如果想得到该网络库则需要进入到./src/lib/net下执行make命令。

（5）make来到./src/lib/net目录下，开始读取该目录下的Makefile，该文件定义了编译网络库的规则，编译网络库不再依赖任何其它库，make终于可以安心的开始工作不用再跳来跳去了，make开始执行该目录下的Makefile，将一个个源文件编译成目标文件，最后将这些目标文件链接成了静态库(当然也可以是动态库，依赖编译规则)。make在./src/lib/net完成任务后跳转回./src/lib，因为make会记住自己是从哪个目录跳转到当前目录的。

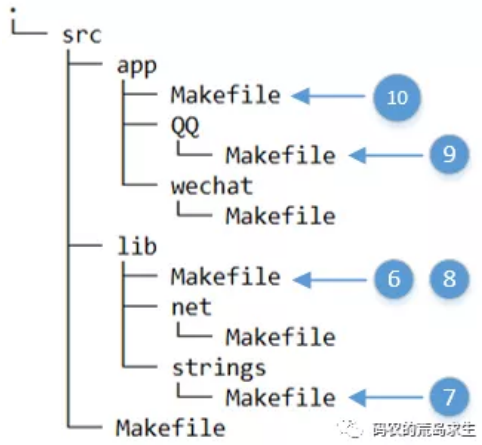


（6）make再次回到./src/lib下，因为make执行完了网络库的编译规则，因此继续往下执行，也就是字符串库的编译规则，该规则告诉make如果想得到字符串库则需要进入到src/lib/strings下执行make命令。

（7）make来到./src/lib/strings目录项，开始读取该目录下的Makefile，该文件定义了编译字符串库的规则，同样，编译字符串库不需要依赖任何其它库，make开始执行该目录下的Makefile，将一个个源文件编译成目标文件，最后将这些目标文件链接成了静态库(当然也可以是动态库)。make在./src/lib/strings下完成任务后跳转回./src/lib，因为make就是从这个目录跳转到./src/lib/strings的。

（8）make回到./src/lib，如果该目录下的Makefile还有其它编译规则，则继续上面的过程，如果没有其它规则，则该目录下的编译任务执行完成，make返回到./src/app/QQ。

（9）make回到./src/app/QQ下继续执行被中断的规则，这时QQ所依赖的库都已经编译完成，因此make可以直接进行链接了，QQ程序编译链接完成。make返回到./src/app。



（10）make来到./src/app下继续执行被中断的规则，make开始执行微信程序的编译规则，这里和QQ的编译是一样的，唯一一点即如果微信也需要依赖网络库和字符串库，那么当make调转到./src/lib下会发现这些库已经生成了，因此直接返回。当make执行完./src/app下的编译规则后，QQ和微信程序就都编译完成了。make返回到./src后，发现该目录下的Makefile执行完毕，因此make程序退出，整个编译过程完成。

如果你对这个过程还不是很清楚的话，我们用一个游戏的类比来加深你对整个过程的理解。

相信很多同学都玩过RPG(角色扮演)游戏，比如仙剑奇侠传，阴阳师。你可以把大型项目的编译过程想象成玩RPG游戏，这类的游戏通常都会有一个主线，若干支线，通常主线的每一关都需要你去某个支线完成任务，例如拿到宝物之类，当你完成支线任务拿到宝物后，你才能回到主线进入到下一关。

在这里，make程序就好比玩家，游戏里的任务就好比编译脚本Makefile，主线任务就好比app下的Makefile，支线任务就好比编译app所依赖的库或者目标文件，比如这里的lib下的Makefile。

首先玩家make进入主线，也就是app下，读取主线需要完成的任务(app下的Makefile)，主线任务告诉玩家make通过其中某一关(比如编译出可执行文件app1)依赖一个支线任务，拿到宝物(app1所依赖的lib下的某个库)，这时玩家make开始去支线场景(进入lib文件夹)，然后读取支线任务(读取lib下的Makefile)，make开始在lib下打怪升级(开始编译链接lib下源文件并生成相应的库)，当make完成支线任务拿到宝物(lib中编译出来的库)回到主线任务(回到app下Makefile因跳转到lib被中断的接下来的编译脚本)后，才可以继续接下来的通关。

有的同学可能已经发现了，像上面的这种编译实现方式其实是比较混乱的，既然我们make给了我们可以将每个模块当做独立项目进行编译的能力，那么对于非应用程序的代码比如这里的src/lib，我们可以提前编译出来，最后再来编译src/app下的代码，这样当依赖某个库时无需再去将该库编译出来。使用上面的编译顺序是为了说明make的构建方式是多样的，实际上使用make这一类的工具你可以使用任何你想要的编译顺序进行项目构建，本质上写Makefile就是写程序，这些程序告诉make该如何构建出最后的可执行文件，至于构建程序该以什么样的顺序构建出可执行文件，一切由你做主。这就是make这类编译工具的灵活以及强大之处。

还有一点需要注意的就是，真实的项目中会有很多模块是相互独立的，即这些模块互不依赖，为加快编译速度，make支持并行编译以充分利用多核的处理能力。

关于大型项目的构建到这里就讲解的差不多了，我们可以看到大型项目的构建其实和我们平时完成一个目标是类似的，先有一个大的目标并将其分解为一个个比较容易实现的小目标，当所有的小目标完成后我们的目的也就是实现了。本质上大型项目的构建与此类似。

至此，彻底理解链接器这一系列的文章就讲解完毕了，如果你喜欢该文章也欢迎关注我的微信公共账号，码农的荒岛求生，获取更多内容。

收录于话题 #链接器 6

< 上一篇 · 彻底理解链接器：五，重定位

People who liked this content also liked

一年赚了100多万，美金！

码农的荒岛求生