# nginx架构详解(50%)

nginx的下篇将会更加深入的介绍nginx的实现原理。上一章，我们了解到了如何设计一个高性能服务器，那这一章将会开始讲解，nginx是如何一步一步实现高性能服务器的。

## nginx的源码目录结构(100%)

nginx的优秀除了体现在程序结构以及代码风格上，nginx的源码组织也同样简洁明了，目录结构层次结构清晰，值得我们去学习。nginx的源码目录与nginx的模块化以及功能的划分是紧密结合，这也使得我们可以很方便地找到相关功能的代码。这节先介绍nginx源码的目录结构，先对nginx的源码有一个大致的认识，下节会讲解nginx如何编译。

下面是nginx源码的目录结构： :

.  
├── auto 自动检测系统环境以及编译相关的脚本  
│ ├── cc 关于编译器相关的编译选项的检测脚本  
│ ├── lib nginx编译所需要的一些库的检测脚本  
│ ├── os 与平台相关的一些系统参数与系统调用相关的检测  
│ └── types 与数据类型相关的一些辅助脚本  
├── conf 存放默认配置文件，在make install后，会拷贝到安装目录中去  
├── contrib 存放一些实用工具，如geo配置生成工具（geo2nginx.pl）  
├── html 存放默认的网页文件，在make install后，会拷贝到安装目录中去  
├── man nginx的man手册  
└── src 存放nginx的源代码  
 ├── core nginx的核心源代码，包括常用数据结构的定义，以及nginx初始化运行的核心代码如main函数  
 ├── event 对系统事件处理机制的封装，以及定时器的实现相关代码  
 │ └── modules 不同事件处理方式的模块化，如select、poll、epoll、kqueue等  
 ├── http nginx作为http服务器相关的代码  
 │ └── modules 包含http的各种功能模块  
 ├── mail nginx作为邮件代理服务器相关的代码  
 ├── misc 一些辅助代码，测试c++头的兼容性，以及对google\_perftools的支持  
 └── os 主要是对各种不同体系统结构所提供的系统函数的封装，对外提供统一的系统调用接口

## nginx的configure原理(100%)

nginx的编译旅程将从configure开始，configure脚本将根据我们输入的选项、系统环境参与来生成所需的文件（包含源文件与Makefile文件）。configure会调用一系列auto脚本来实现编译环境的初始化。

### auto脚本

auto脚本由一系列脚本组成，他们有一些是实现一些通用功能由其它脚本来调用（如have），有一些则是完成一些特定的功能（如option）。脚本之间的主要执行顺序及调用关系如下图所示（由上到下，表示主流程的执行）：

![](data:text/html; charset=utf-8;base64,)

接下来，我们结合代码来分析下configure的原理:

1) 初始化

. auto/options  
. auto/init  
. auto/sources

这是configure源码开始执行的前三行，依次交由auto目录下面的option、init、sources来处理。

2) auto/options主要是处理用户输入的configure选项，以及输出帮助信息等。读者可以结合nginx的源码来阅读本章内容。由于篇幅关系，这里大致列出此文件的结构：

##1. 设置选项对应的shell变量以及他们的初始值  
help=no  
NGX\_PREFIX=  
NGX\_SBIN\_PATH=  
NGX\_CONF\_PREFIX=  
NGX\_CONF\_PATH=  
NGX\_ERROR\_LOG\_PATH=  
NGX\_PID\_PATH=  
NGX\_LOCK\_PATH=  
NGX\_USER=  
NGX\_GROUP=  
  
...  
  
  
## 2, 处理每一个选项值，并设置到对应的全局变量中  
for option  
do  
 opt="$opt `echo $option | sed -e \"s/\(--[^=]\*=\)\(.\* .\*\)/\1'\2'/\"`"  
  
 # 得到此选项的value部分  
 case "$option" in  
 -\*=\*) value=`echo "$option" | sed -e 's/[-\_a-zA-Z0-9]\*=//'` ;;  
 \*) value="" ;;  
 esac  
  
 # 根据option内容进行匹配，并设置相应的选项  
 case "$option" in  
 --help) help=yes ;;  
 --prefix=) NGX\_PREFIX="!" ;;  
 --prefix=\*) NGX\_PREFIX="$value" ;;  
 --sbin-path=\*) NGX\_SBIN\_PATH="$value" ;;  
 --conf-path=\*) NGX\_CONF\_PATH="$value" ;;  
 --error-log-path=\*) NGX\_ERROR\_LOG\_PATH="$value";;  
 --pid-path=\*) NGX\_PID\_PATH="$value" ;;  
 --lock-path=\*) NGX\_LOCK\_PATH="$value" ;;  
 --user=\*) NGX\_USER="$value" ;;  
 --group=\*) NGX\_GROUP="$value" ;;  
  
 ...  
  
 \*)  
 # 没有找到的对应选项  
 echo "$0: error: invalid option \"$option\""  
 exit 1  
 ;;  
 esac  
done  
  
## 3. 对选项进行处理  
  
# 如果有--help，则输出帮助信息  
if [ $help = yes ]; then  
  
 cat << END  
  
 --help print this message  
  
 --prefix=PATH set installation prefix  
 --sbin-path=PATH set nginx binary pathname  
 --conf-path=PATH set nginx.conf pathname  
 --error-log-path=PATH set error log pathname  
 --pid-path=PATH set nginx.pid pathname  
 --lock-path=PATH set nginx.lock pathname  
  
 --user=USER set non-privileged user for  
 worker processes  
 --group=GROUP set non-privileged group for  
 worker processes  
END  
  
 exit 1  
fi  
  
# 默认文件路径  
NGX\_CONF\_PATH=${NGX\_CONF\_PATH:-conf/nginx.conf}  
NGX\_CONF\_PREFIX=`dirname $NGX\_CONF\_PATH`  
NGX\_PID\_PATH=${NGX\_PID\_PATH:-logs/nginx.pid}  
NGX\_LOCK\_PATH=${NGX\_LOCK\_PATH:-logs/nginx.lock}  
  
...

上面的代码中，我们选用了文件中的部分代码进行了说明。大家可结合源码再进行分析。auto/options的目的主要是处理用户选项，并由选项生成一些全局变量的值，这些值在其它文件中会用到。该文件也会输出configure的帮助信息。

3) auto/init

该文件的目的在于初始化一些临时文件的路径，检查echo的兼容性，并创建Makefile。

# 生成最终执行编译的makefile文件路径  
NGX\_MAKEFILE=$NGX\_OBJS/Makefile  
# 动态生成nginx模块列表的路径，由于nginx的的一些模块是可以选择编译的，而且可以添加自己的模块，所以模块列表是动态生成的  
NGX\_MODULES\_C=$NGX\_OBJS/ngx\_modules.c  
  
NGX\_AUTO\_HEADERS\_H=$NGX\_OBJS/ngx\_auto\_headers.h  
NGX\_AUTO\_CONFIG\_H=$NGX\_OBJS/ngx\_auto\_config.h  
  
# 自动测试目录与日志输出文件  
NGX\_AUTOTEST=$NGX\_OBJS/autotest  
# 如果configure出错，可用来查找出错的原因  
NGX\_AUTOCONF\_ERR=$NGX\_OBJS/autoconf.err  
  
NGX\_ERR=$NGX\_OBJS/autoconf.err  
MAKEFILE=$NGX\_OBJS/Makefile  
  
  
NGX\_PCH=  
NGX\_USE\_PCH=  
  
  
# 检查echo是否支持-n或\c  
  
# check the echo's "-n" option and "\c" capability  
  
if echo "test\c" | grep c >/dev/null; then  
  
 # 不支持-c的方式，检查是否支持-n的方式  
  
 if echo -n test | grep n >/dev/null; then  
 ngx\_n=  
 ngx\_c=  
  
 else  
 ngx\_n=-n  
 ngx\_c=  
 fi  
  
else  
 ngx\_n=  
 ngx\_c='\c'  
fi  
  
# 创建最初始的makefile文件  
# default表示目前编译对象  
# clean表示执行clean工作时，需要删除makefile文件以及objs目录  
# 整个过程中只会生成makefile文件以及objs目录，其它所有临时文件都在objs目录之下，所以执行clean后，整个目录还原到初始状态  
# 要再次执行编译，需要重新执行configure命令  
  
# create Makefile  
  
cat << END > Makefile  
  
default: build  
  
clean:  
 rm -rf Makefile $NGX\_OBJS  
END

4) auto/sources

该文件从文件名中就可以看出，它的主要功能是跟源文件相关的。它的主要作用是定义不同功能或系统所需要的文件的变量。根据功能，分为CORE/REGEX/EVENT/UNIX/FREEBSD/HTTP等。每一个功能将会由四个变量组成，\"\_MODULES\"表示此功能相关的模块，最终会输出到ngx\_modules.c文件中，即动态生成需要编译到nginx中的模块；\"INCS\"表示此功能依赖的源码目录，查找头文件的时候会用到，在编译选项中，会出现在\"-I\"中；"DEPS\"显示指明在Makefile中需要依赖的文件名，即编译时，需要检查这些文件的更新时间；\"SRCS\"表示需要此功能编译需要的源文件。

拿core来说：

CORE\_MODULES="ngx\_core\_module ngx\_errlog\_module ngx\_conf\_module ngx\_emp\_server\_module ngx\_emp\_server\_core\_module"  
  
CORE\_INCS="src/core"  
  
CORE\_DEPS="src/core/nginx.h \  
 src/core/ngx\_config.h \  
 src/core/ngx\_core.h \  
 src/core/ngx\_log.h \  
 src/core/ngx\_palloc.h \  
 src/core/ngx\_array.h \  
 src/core/ngx\_list.h \  
 src/core/ngx\_hash.h \  
 src/core/ngx\_buf.h \  
 src/core/ngx\_queue.h \  
 src/core/ngx\_string.h \  
 src/core/ngx\_parse.h \  
 src/core/ngx\_inet.h \  
 src/core/ngx\_file.h \  
 src/core/ngx\_crc.h \  
 src/core/ngx\_crc32.h \  
 src/core/ngx\_murmurhash.h \  
 src/core/ngx\_md5.h \  
 src/core/ngx\_sha1.h \  
 src/core/ngx\_rbtree.h \  
 src/core/ngx\_radix\_tree.h \  
 src/core/ngx\_slab.h \  
 src/core/ngx\_times.h \  
 src/core/ngx\_shmtx.h \  
 src/core/ngx\_connection.h \  
 src/core/ngx\_cycle.h \  
 src/core/ngx\_conf\_file.h \  
 src/core/ngx\_resolver.h \  
 src/core/ngx\_open\_file\_cache.h \  
 src/core/nginx\_emp\_server.h \  
 src/core/emp\_server.h \  
 src/core/task\_thread.h \  
 src/core/standard.h \  
 src/core/dprint.h \  
 src/core/ngx\_crypt.h"  
  
CORE\_SRCS="src/core/nginx.c \  
 src/core/ngx\_log.c \  
 src/core/ngx\_palloc.c \  
 src/core/ngx\_array.c \  
 src/core/ngx\_list.c \  
 src/core/ngx\_hash.c \  
 src/core/ngx\_buf.c \  
 src/core/ngx\_queue.c \  
 src/core/ngx\_output\_chain.c \  
 src/core/ngx\_string.c \  
 src/core/ngx\_parse.c \  
 src/core/ngx\_inet.c \  
 src/core/ngx\_file.c \  
 src/core/ngx\_crc32.c \  
 src/core/ngx\_murmurhash.c \  
 src/core/ngx\_md5.c \  
 src/core/ngx\_rbtree.c \  
 src/core/ngx\_radix\_tree.c \  
 src/core/ngx\_slab.c \  
 src/core/ngx\_times.c \  
 src/core/ngx\_shmtx.c \  
 src/core/ngx\_connection.c \  
 src/core/ngx\_cycle.c \  
 src/core/ngx\_spinlock.c \  
 src/core/ngx\_cpuinfo.c \  
 src/core/ngx\_conf\_file.c \  
 src/core/ngx\_resolver.c \  
 src/core/ngx\_open\_file\_cache.c \  
 src/core/nginx\_emp\_server.c \  
 src/core/emp\_server.c \  
 src/core/standard.c \  
 src/core/task\_thread.c \  
 src/core/dprint.c \  
 src/core/ngx\_crypt.c"

如果我们自己写一个第三方模块，我们可能会引用到这些变量的值，或对这些变量进行修改，比如添加我们自己的模块，或添加自己的一个头文件查找目录(在第三方模块的config中)，在后面，我们会看到它是如何加框第三方模块的。 在继续分析执行流程之前，我们先介绍一些工具脚本。

5) auto/have

cat << END >> $NGX\_AUTO\_CONFIG\_H  
  
#ifndef $have  
#define $have 1  
#endif  
  
END

从代码中，我们可以看到，这个工具的作用是，将$have变量的值，宏定义为1，并输出到auto\_config文件中。通常我们通过这个工具来控制是否打开某个特性。这个工具在使用前，需要先定义宏的名称 ，即$have变量。

6) 再回到configure文件中来：

# NGX\_DEBUG是在auto/options文件中处理的，如果有--with-debug选项，则其值是YES  
if [ $NGX\_DEBUG = YES ]; then  
 # 当有debug选项时，会定义NGX\_DEBUG宏  
 have=NGX\_DEBUG . auto/have  
fi

这段代码中，可以看出，configure是如何定义一个特性的：通过宏定义，输出到config头文件中，然后在程序中可以判断这个宏是否有定义，来实现不同的特性。

configure文件中继续向下：

# 编译器选项  
. auto/cc/conf  
  
# 头文件支持宏定义  
if [ "$NGX\_PLATFORM" != win32 ]; then  
 . auto/headers  
fi  
  
# 操作系统相关的配置的检测  
. auto/os/conf  
  
# unix体系下的通用配置检测  
if [ "$NGX\_PLATFORM" != win32 ]; then  
 . auto/unix  
fi

configure会依次调用其它几个文件，来进行环境的检测，包括编译器、操作系统相关。

7) auto/feature

nginx的configure会自动检测不同平台的特性，神奇之处就是auto/feature的实现，在继续向下分析之前，我们先来看看这个工具的实现原理。此工具的核心思想是，输出一小段代表性c程序，然后设置好编译选项，再进行编译连接运行，再对结果进行分析。例如，如果想检测某个库是否存在，就在小段c程序里面调用库里面的某个函数，再进行编译链接，如果出错，则表示库的环境不正常，如果编译成功，且运行正常，则库的环境检测正常。我们在写nginx第三方模块时，也常使用此工具来进行环境的检测，所以，此工具的作用贯穿整个configure过程。

先看一小段使用例子：

ngx\_feature="poll()"  
ngx\_feature\_name=  
ngx\_feature\_run=no  
ngx\_feature\_incs="#include <poll.h>"  
ngx\_feature\_path=  
ngx\_feature\_libs=  
ngx\_feature\_test="int n; struct pollfd pl;  
 pl.fd = 0;  
 pl.events = 0;  
 pl.revents = 0;  
 n = poll(&pl, 1, 0);  
 if (n == -1) return 1"  
. auto/feature  
  
if [ $ngx\_found = no ]; then  
 # 如果没有找到poll，就设置变量的值  
 EVENT\_POLL=NONE  
fi

这段代码在auto/unix里面实现，用来检测当前操作系统是否支持poll函数调用。在调用auto/feature之前，需要先设置几个输入参数变量的值，然后结果会存在$ngx\_found变量里面, 并输出宏定义以表示支持此特性:

$ngx\_feature 特性名称  
$ngx\_feature\_name 特性的宏定义名称，如果特性测试成功，则会定义该宏定义  
$ngx\_feature\_path 编译时要查找头文件目录  
$ngx\_feature\_test 要执行的测试代码  
$ngx\_feature\_incs 在代码中要include的头文件  
$ngx\_feature\_libs 编译时需要link的库文件选项  
$ngx\_feature\_run 编译成功后，对二进制文件需要做的动作，可以是yes value bug 其它  
  
#ngx\_found 如果找到，并测试成功，其值为yes，否则其值为no

看看auto/feature的关键代码：

# 初始化输出结果为no  
ngx\_found=no  
  
#将特性名称小写转换成大写  
if test -n "$ngx\_feature\_name"; then  
 # 小写转大写  
 ngx\_have\_feature=`echo $ngx\_feature\_name \  
 | tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ`  
fi  
  
# 将所有include目录转换成编译选项  
if test -n "$ngx\_feature\_path"; then  
 for ngx\_temp in $ngx\_feature\_path; do  
 ngx\_feature\_inc\_path="$ngx\_feature\_inc\_path -I $ngx\_temp"  
 done  
fi  
  
  
# 生成临时的小段c程序代码。  
# $ngx\_feature\_incs变量是程序需要include的头文件  
# $ngx\_feature\_test是测试代码  
cat << END > $NGX\_AUTOTEST.c  
  
#include <sys/types.h>  
$NGX\_INCLUDE\_UNISTD\_H  
$ngx\_feature\_incs  
  
int main() {  
 $ngx\_feature\_test;  
 return 0;  
}  
  
END  
  
# 编译命令  
# 编译之后的目标文件是 $NGX\_AUTOTEST，后面会判断这个文件是否存在来判断是否编译成功  
ngx\_test="$CC $CC\_TEST\_FLAGS $CC\_AUX\_FLAGS $ngx\_feature\_inc\_path \  
 -o $NGX\_AUTOTEST $NGX\_AUTOTEST.c $NGX\_TEST\_LD\_OPT $ngx\_feature\_libs"  
  
# 执行编译过程  
# 编译成功后，会生成$NGX\_AUTOTEST命名的文件  
eval "/bin/sh -c \"$ngx\_test\" >> $NGX\_AUTOCONF\_ERR 2>&1"  
  
# 如果文件存在，则编译成功  
if [ -x $NGX\_AUTOTEST ]; then  
  
 case "$ngx\_feature\_run" in  
  
 # 需要运行来判断是否支持特性  
 # 测试程序能否正常执行（即程序退出后的状态码是否是0），如果正常退出，则特性测试成功，设置ngx\_found为yes，并添加名为ngx\_feature\_name的宏定义，宏的值为1  
 yes)  
 # 如果程序正常退出，退出码为0，则程序执行成功，我们可以在测试代码里面手动返回非0来表示程序出错  
 # /bin/sh is used to intercept "Killed" or "Abort trap" messages  
 if /bin/sh -c $NGX\_AUTOTEST >> $NGX\_AUTOCONF\_ERR 2>&1; then  
 echo " found"  
 ngx\_found=yes  
  
 # 添加宏定义，宏的值为1  
 if test -n "$ngx\_feature\_name"; then  
 have=$ngx\_have\_feature . auto/have  
 fi  
  
 else  
 echo " found but is not working"  
 fi  
 ;;  
  
 # 需要运行程序来判断是否支持特性，如果支持，将程序标准输出的结果作为宏的值  
 value)  
 # /bin/sh is used to intercept "Killed" or "Abort trap" messages  
 if /bin/sh -c $NGX\_AUTOTEST >> $NGX\_AUTOCONF\_ERR 2>&1; then  
 echo " found"  
 ngx\_found=yes  
  
 # 与yes不一样的是，value会将程序从标准输出里面打印出来的值，设置为ngx\_feature\_name宏变量的值  
 # 在此种情况下，程序需要设置ngx\_feature\_name变量名  
 cat << END >> $NGX\_AUTO\_CONFIG\_H  
  
#ifndef $ngx\_feature\_name  
#define $ngx\_feature\_name `$NGX\_AUTOTEST`  
#endif  
  
END  
 else  
 echo " found but is not working"  
 fi  
 ;;  
  
 # 与yes正好相反  
 bug)  
 # /bin/sh is used to intercept "Killed" or "Abort trap" messages  
 if /bin/sh -c $NGX\_AUTOTEST >> $NGX\_AUTOCONF\_ERR 2>&1; then  
 echo " not found"  
  
 else  
 echo " found"  
 ngx\_found=yes  
  
 if test -n "$ngx\_feature\_name"; then  
 have=$ngx\_have\_feature . auto/have  
 fi  
 fi  
 ;;  
  
 # 不需要运行程序，最后定义宏变量  
 \*)  
 echo " found"  
 ngx\_found=yes  
  
 if test -n "$ngx\_feature\_name"; then  
 have=$ngx\_have\_feature . auto/have  
 fi  
 ;;  
  
 esac  
else  
 # 编译失败  
 echo " not found"  
  
 # 编译失败，会保存信息到日志文件中  
 echo "----------" >> $NGX\_AUTOCONF\_ERR  
 # 保留编译文件的内容  
 cat $NGX\_AUTOTEST.c >> $NGX\_AUTOCONF\_ERR  
 echo "----------" >> $NGX\_AUTOCONF\_ERR  
 # 保留编译文件的选项  
 echo $ngx\_test >> $NGX\_AUTOCONF\_ERR  
 echo "----------" >> $NGX\_AUTOCONF\_ERR  
fi  
  
# 最后删除生成的临时文件  
rm $NGX\_AUTOTEST\*

8) auto/cc/conf

在了解了工具auto/feature后，继续我们的主流程，auto/cc/conf的代码就很好理解了，这一步主要是检测编译器，并设置编译器相关的选项。它先调用auto/cc/name来得到编译器的名称，然后根据编译器选择执行不同的编译器相关的文件如gcc执行auto/cc/gcc来设置编译器相关的一些选项。

9) auto/include

这个工具用来检测是头文件是否支持。需要检测的头文件放在$ngx\_include里面，如果支持，则$ngx\_found变量的值为yes，并且会产生NGX\_HAVE{ngx\_include}的宏定义。

10) auto/headers

生成头文件的宏定义。生成的定义放在objs/ngx\_auto\_headers.h里面：

#ifndef NGX\_HAVE\_UNISTD\_H  
#define NGX\_HAVE\_UNISTD\_H 1  
#endif  
  
  
#ifndef NGX\_HAVE\_INTTYPES\_H  
#define NGX\_HAVE\_INTTYPES\_H 1  
#endif  
  
  
#ifndef NGX\_HAVE\_LIMITS\_H  
#define NGX\_HAVE\_LIMITS\_H 1  
#endif  
  
  
#ifndef NGX\_HAVE\_SYS\_FILIO\_H  
#define NGX\_HAVE\_SYS\_FILIO\_H 1  
#endif  
  
  
#ifndef NGX\_HAVE\_SYS\_PARAM\_H  
#define NGX\_HAVE\_SYS\_PARAM\_H 1  
#endif

11) auto/os/conf

针对不同的操作系统平台特性的检测，并针对不同的操作系统，设置不同的CORE\_INCS、CORE\_DEPS、CORE\_SRCS变量。nginx跨平台的支持就是在这个地方体现出来的。

12) auto/unix

针对unix体系的通用配置或系统调用的检测，如poll等事件处理系统调用的检测等。

13) 回到configure里面

# 生成模块列表  
. auto/modules  
# 配置库的依赖  
. auto/lib/conf

14) auto/modules

该脚本根据不同的条件，输出不同的模块列表，最后输出的模块列表的文件在objs/ngx\_modules.c：

#include <ngx\_config.h>  
#include <ngx\_core.h>  
  
  
extern ngx\_module\_t ngx\_core\_module;  
extern ngx\_module\_t ngx\_errlog\_module;  
extern ngx\_module\_t ngx\_conf\_module;  
extern ngx\_module\_t ngx\_emp\_server\_module;  
  
...  
  
  
ngx\_module\_t \*ngx\_modules[] = {  
 &ngx\_core\_module,  
 &ngx\_errlog\_module,  
 &ngx\_conf\_module,  
 &ngx\_emp\_server\_module,  
 ...  
 NULL  
};

这个文件会决定所有模块的顺序，这会直接影响到最后的功能，下一小节我们将讨论模块间的顺序。这个文件会加载我们的第三方模块，这也是我们值得关注的地方：

if test -n "$NGX\_ADDONS"; then  
  
 echo configuring additional modules  
  
 for ngx\_addon\_dir in $NGX\_ADDONS  
 do  
 echo "adding module in $ngx\_addon\_dir"  
  
 if test -f $ngx\_addon\_dir/config; then  
 # 执行第三方模块的配置  
 . $ngx\_addon\_dir/config  
  
 echo " + $ngx\_addon\_name was configured"  
  
 else  
 echo "$0: error: no $ngx\_addon\_dir/config was found"  
 exit 1  
 fi  
 done  
fi

这段代码比较简单，确实现了nginx很强大的扩展性，加载第三方模块。$ngx\_addon\_dir变量是在configure执行时，命令行参数--add-module加入的，它是一个目录列表，每一个目录，表示一个第三方模块。从代码中，我们可以看到，它就是针对每一个第三方模块执行其目录下的config文件。于是我们可以在config文件里面执行我们自己的检测逻辑，比如检测库依赖，添加编译选项等。

15) auto/lib/conf

该文件会针对nginx编译所需要的基础库的检测，比如rewrite模块需要的PCRE库的检测支持。

16) configure接下来定义一些宏常量，主要是是文件路径方面的：

case ".$NGX\_PREFIX" in  
 .)  
 NGX\_PREFIX=${NGX\_PREFIX:-/usr/local/nginx}  
 have=NGX\_PREFIX value="\"$NGX\_PREFIX/\"" . auto/define  
 ;;  
  
 .!)  
 NGX\_PREFIX=  
 ;;  
  
 \*)  
 have=NGX\_PREFIX value="\"$NGX\_PREFIX/\"" . auto/define  
 ;;  
esac  
  
if [ ".$NGX\_CONF\_PREFIX" != "." ]; then  
 have=NGX\_CONF\_PREFIX value="\"$NGX\_CONF\_PREFIX/\"" . auto/define  
fi  
  
have=NGX\_SBIN\_PATH value="\"$NGX\_SBIN\_PATH\"" . auto/define  
have=NGX\_CONF\_PATH value="\"$NGX\_CONF\_PATH\"" . auto/define  
have=NGX\_PID\_PATH value="\"$NGX\_PID\_PATH\"" . auto/define  
have=NGX\_LOCK\_PATH value="\"$NGX\_LOCK\_PATH\"" . auto/define  
have=NGX\_ERROR\_LOG\_PATH value="\"$NGX\_ERROR\_LOG\_PATH\"" . auto/define  
  
have=NGX\_HTTP\_LOG\_PATH value="\"$NGX\_HTTP\_LOG\_PATH\"" . auto/define  
have=NGX\_HTTP\_CLIENT\_TEMP\_PATH value="\"$NGX\_HTTP\_CLIENT\_TEMP\_PATH\""  
. auto/define  
have=NGX\_HTTP\_PROXY\_TEMP\_PATH value="\"$NGX\_HTTP\_PROXY\_TEMP\_PATH\""  
. auto/define  
have=NGX\_HTTP\_FASTCGI\_TEMP\_PATH value="\"$NGX\_HTTP\_FASTCGI\_TEMP\_PATH\""  
. auto/define  
have=NGX\_HTTP\_UWSGI\_TEMP\_PATH value="\"$NGX\_HTTP\_UWSGI\_TEMP\_PATH\""  
. auto/define  
have=NGX\_HTTP\_SCGI\_TEMP\_PATH value="\"$NGX\_HTTP\_SCGI\_TEMP\_PATH\""  
. auto/define

17) configure最后的工作，生成编译安装的makefile

# 生成objs/makefile文件  
. auto/make  
  
# 生成关于库的编译选项到makefile文件  
. auto/lib/make  
# 生成与安装相关的makefile文件内容，并生成最外层的makefile文件  
. auto/install  
  
# STUB  
. auto/stubs  
  
have=NGX\_USER value="\"$NGX\_USER\"" . auto/define  
have=NGX\_GROUP value="\"$NGX\_GROUP\"" . auto/define  
  
# 编译的最后阶段，汇总信息  
. auto/summary

### 模块编译顺序

上一节中，提到过，nginx模块的顺序很重要，会直接影响到程序的功能。而且，nginx和部分模块，也有着自己特定的顺序要求，比如ngx\_http\_write\_filter\_module模块一定要在filter模块的最后一步执行。想查看模块的执行顺序，可以在objs/ngx\_modules.c这个文件中找到，这个文件在configure之后生成，上一节中，我们看过这个文件里面的内容。

下面是一个ngx\_modules.c文件的示例：

ngx\_module\_t \*ngx\_modules[] = {  
 // 全局core模块  
 &ngx\_core\_module,  
 &ngx\_errlog\_module,  
 &ngx\_conf\_module,  
 &ngx\_emp\_server\_module,  
 &ngx\_emp\_server\_core\_module,  
  
 // event模块  
 &ngx\_events\_module,  
 &ngx\_event\_core\_module,  
 &ngx\_kqueue\_module,  
  
 // 正则模块  
 &ngx\_regex\_module,  
  
 // http模块  
 &ngx\_http\_module,  
 &ngx\_http\_core\_module,  
 &ngx\_http\_log\_module,  
 &ngx\_http\_upstream\_module,  
  
 // http handler模块  
 &ngx\_http\_static\_module,  
 &ngx\_http\_autoindex\_module,  
 &ngx\_http\_index\_module,  
 &ngx\_http\_auth\_basic\_module,  
 &ngx\_http\_access\_module,  
 &ngx\_http\_limit\_conn\_module,  
 &ngx\_http\_limit\_req\_module,  
 &ngx\_http\_geo\_module,  
 &ngx\_http\_map\_module,  
 &ngx\_http\_split\_clients\_module,  
 &ngx\_http\_referer\_module,  
 &ngx\_http\_rewrite\_module,  
 &ngx\_http\_proxy\_module,  
 &ngx\_http\_fastcgi\_module,  
 &ngx\_http\_uwsgi\_module,  
 &ngx\_http\_scgi\_module,  
 &ngx\_http\_memcached\_module,  
 &ngx\_http\_empty\_gif\_module,  
 &ngx\_http\_browser\_module,  
 &ngx\_http\_upstream\_ip\_hash\_module,  
 &ngx\_http\_upstream\_keepalive\_module,  
 //此处是第三方handler模块  
  
 // http filter模块  
 &ngx\_http\_write\_filter\_module,  
 &ngx\_http\_header\_filter\_module,  
 &ngx\_http\_chunked\_filter\_module,  
 &ngx\_http\_range\_header\_filter\_module,  
 &ngx\_http\_gzip\_filter\_module,  
 &ngx\_http\_postpone\_filter\_module,  
 &ngx\_http\_ssi\_filter\_module,  
 &ngx\_http\_charset\_filter\_module,  
 &ngx\_http\_userid\_filter\_module,  
 &ngx\_http\_headers\_filter\_module,  
 // 第三方filter模块  
 &ngx\_http\_copy\_filter\_module,  
 &ngx\_http\_range\_body\_filter\_module,  
 &ngx\_http\_not\_modified\_filter\_module,  
 NULL  
};

http handler模块与http filter模块的顺序很重要，这里我们主要关注一下这两类模块。

http handler模块，在后面的章节里面会讲到多阶段请求的处理链。对于content phase之前的handler，同一个阶段的handler，模块是顺序执行的。比如上面的示例代码中，ngx\_http\_auth\_basic\_module与ngx\_http\_access\_module这两个模块都是在access phase阶段，由于ngx\_http\_auth\_basic\_module在前面，所以会先执行。由于content phase只会有一个执行，所以不存在顺序问题。另外，我们加载的第三方handler模块永远是在最后执行。

http filter模块，filter模块会将所有的filter handler排成一个倒序链，所以在最前面的最后执行。上面的例子中，&ngx\_http\_write\_filter\_module最后执行，ngx\_http\_not\_modified\_filter\_module最先执行。注意，我们加载的第三方filter模块是在copy\_filter模块之后，headers\_filter模块之前执行。

## nginx的事件机制

### event框架及非阻塞模型

### 定时器实现

### 信号处理

### 惊群问题

## nginx的进程机制

### master进程

### worker进程

### 进程间通讯