



LINKS

[ABOUT](#) | [ARTICLES](#) | [ECE](#) | [SHOWCASE](#) | [GUESTBOOK](#) | [f FACEBOOK](#)

맞춤검색

Home > ECE

✓뷰어로 보기

일반

[Make 튜토리얼] Makefile 예제와 작성 방법 및 기본 패턴

Posted 2017. 02. 05 Updated 2018. 11. 21 Views 29680 Replies 2

ABOUT

ARTICLES

일반 (38)
건강 (9)
여행 (91)
독서 (5)
영화 (3)
박람회 (5)

ECE

일반 (60)
PSpice (6)
AVR (32)
Android (8)
Nginx (2)
Apache (9)
Linux (49)
XE (11)
Python (11)
Security (3)

SHOWCASE

GUESTBOOK

모든 게시물에 대하여 '링크' 방식의 퍼가기만 허용합니다.



한양대학교

Be Bold of Robotics &
Advanced Micro Intelligence
바라미
Since 1994

826

1104916





리눅스 환경에서 소스코드를 다운받아서 수동으로 프로그램 설치를 해 보신 분들은 다음 세 줄의 명령어에 매우 익숙할 것입니다.

```
./configure
make
sudo make install
```

근래에는 make 외에 다른 빌드툴들이 많이 나와서 다른 방법을 사용하기도 하지만, 대개 이 세 줄의 명령어만 알고 있으면 리눅스에서 웬만한 프로그램은 수동으로 설치할 수 있습니다. 주로 리눅스 환경에서 사용하지만, Windows에서도 [GNU make for Windows](#) 를 설치하면 make를 사용해 빌드를 할 수 있습니다

이 중 빌드 사전작업을 수행하는 첫 번째 명령(./configure)을 제외한 나머지 두 명령은 프로그램의 소스코드 디렉토리에 포함되어 있는 [Makefile\(또는 makefile\)](#)이라는 이름의 스크립트 파일을 읽어서 지정된 순서대로 빌드를 수행하는 것입니다.

Makefile을 텍스트 에디터로 열어 보면 뭐라뭐라 ~~알다가도 모를 내용~~들이 매우 길게 써져 있는 것을 볼 수 있을 텐데, 이는 빌드 대상 소스

파일 갯수가 많기 때문일수도 있지만, 실제 빌드 작업 전/중/후(Pre-build/Pre-link/Post-build)에 해야 할 잡일(?)들이 많아서일 가능성이 높습니다.

이 때문에 Makefile은 외계인이 만든거니 나는 건들지 말아야겠다(...) 라고 생각하고 돌아서는 분들이 계실 텐데, **Makefile의 기본적인 골격은 매우매우 심플**하기 때문에 이를 한 번 익혀 두면 두고두고 편리하게 써먹을 수 있습니다. 하다못해 작업 도중 간단한 실험용 프로그램을 작성해야 할 경우에라도 Makefile을 빠르게 작성해서 사용하면 매우 요긴합니다.

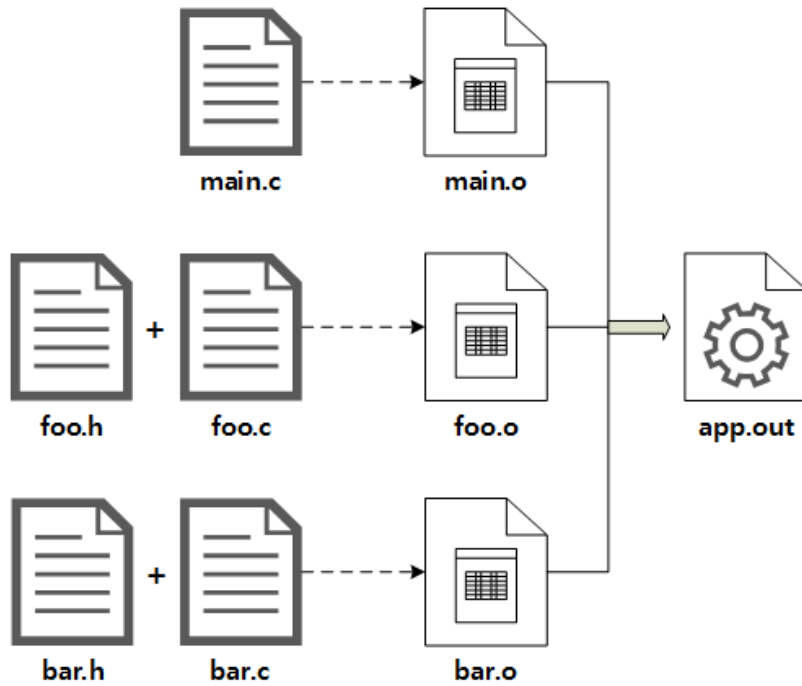
후술하겠지만, 참고로 요즈음에는 위에서 언급한 장편(?)의 Makefile은 일일이 수동으로 만들지 않고, 보다 상위 레벨의 추상화된 빌드 툴(CMake라던가...)을 통해 **Makefile을 자동으로 생성**하여 사용합니다.

여기서 "그럼 굳이 Makefile 작성법은 알 필요가 없지 않은가?" 라는 의문을 가질 수도 있는데, 저는 둘 다 알고 있는 편이 좋다고 생각합니다. 호미로 막을거 가래로 막을 필요 없고, 망치로 박을 수 있는 못을 오함마로 박는건 낭비가 아닐까요? ㅎㅎ 경험상 실제로, CMake를 쓰기보다는 Makefile을 직접 만들어서 쓰는 경우가 더욱 편했던 적이 종종 있었습니다. 대개 소스파일이 몇 개 되지 않고 빌드 옵션이 고정되어 바뀔 여지가 제로인 작은 프로젝트들이 여기에 해당합니다.

▶ 이 글에서는 빌드 예제를 통해 Makefile을 쓰면 좋은 점에 대해 기술하고, 실질적인 작성 방법에 대해 설명합니다.

■ 빌드 예제

다음 그림은 예시에서 수행하려는 빌드 작업을 나타낸 도표입니다.



세 개의 소스파일을 각각 컴파일하여 Object파일(*.o)을 생성하고, 이들을 한 데 묶는 링크 과정을 통해서 실행 파일인 app.out을 생성합니다. 여기서 foo와 bar에 정의된 함수를 main에서 호출하는 의존성이 존재합니다.

■ Make를 쓰지 않고 불편+지저분하게 빌드하기

다음 세 줄의 명령어를 통해 컴파일(위 도표에서 점선에 해당하는 과정)을 합니다. 헤더파일들은 각자의 소스파일에 포함되어 있으므로 명시해주지 않아도 됩니다.

```
gcc -c -o main.o main.c
gcc -c -o foo.o foo.c
gcc -c -o bar.o bar.c
```

여기서 **-c** 옵션은 링크를 하지 않고 컴파일만 하겠다는 의미입니다. 이 옵션을 생략하면 main 함수를 찾을 수 없다는 오류가 출력됩니다.

그리고, 다음 명령으로 Object파일들을 한데 묶는 링크 과정을 수행합니다. 명령은 gcc지만, gcc 내부적으로 링커(ld)를 실행해서 실행 파일(app.out)을 생성합니다.

```
gcc -o app.out main.o foo.o bar.o
```

이상 평범한 C 프로그램의 빌드 절차였습니다. IDE를 사용할 수 있는 프로젝트라면 빌드 버튼 원클릭으로 실행 파일을 영접할 수 있지만, 그렇지 않은 경우 이 명령들을 모두 일일이 실행해 줘야 합니다. 그것도, 한 번만 하면 끝나는 것이 아니라 소스코드를 수정할 때마다 매번 반복해야 하므로 불편함이 꽃피게 됩니다.

혹 이러한 불편함을 감수하더라도, 다음과 같은 치명적인 실수를 할 가능성도 있습니다.

※ 경고: 다음 내용은 다소 충격적일 수 있습니다.

Q. 다음 명령을 실행해서 생성된 바이너리를 실행했는데, 원하는 결과가 안 나와서 소스파일을 다시 열었습니다. 이 때, 개발자가 느낀 심정을 서술하시오. (10점)

```
gcc -o main.c main.c
```

... A. 개발자님이 멘탈붕괴하셨습니다.

이 명령의 의미는 main.c를 빌드해서 같은 이름인 main.c라는 실행 파일로 저장하라는 의미입니다. 즉, 이 명령을 실행하는 순간 소스파일은 사라지고 빌드된 바이너리만 남게 됩니다.

다소 극단적으로 보이는 이 예시는 제가 학부시절 수업을 같이 들었던 친구중 하나가 실제로 저질렀던 실수입니다. 오타 한 글자로 인해 밤새 작업했던 프로그래밍 과제의 소스파일을 날려먹고 멘탈붕괴했다는 안타까운 사연이었습니다...

명령어를 손으로 치다 보면 누구나 언제든지 위와 같은 실수를 저지룰 수 있습니다.

그러면 쉘 스크립트에 빌드 명령들을 다 구겨넣고 실행하면 되지...!?
...라는 해결책을 제시할 수도 있습니다.

쉘 스크립트를 사용하면 실수를 방지한다는 측면에서는 해결책이 될 수 있지만, 이 경우 Makefile이 제공하는 강력한 기능 중 하나인 Incremental build를 사용할 수 없게 됩니다.

Incremental build란 반복적인 빌드 과정에서 변경된 소스코드에 의존성(Dependency)이 있는 대상들만 추려서 다시 빌드하는 기능입니다. 예를 들어, 위의 빌드 예제에서 main.c의 한 줄만 바꾸고 다시 빌드를 할 때, main.o 컴파일(gcc -c -o main.o main.c)과 app.out링크(gcc -o app.out main.o foo.o bar.o)만 수행하는 경우가 이에 해당합니다.

'뭐 귀찮게 그렇게 바뀐것만 골라서 빌드하나 - 그냥 깔끔하게 전부 다시 빌드하면 되지 ㅇㅇ'라고 생각할 수도 있지만, 대규모 프로젝트에서는 빌드 대상 소스파일의 갯수와 복잡도가 어마어마하므로 Incremental build 기능이 거의 필수적이라고 할 수 있습니다. 소스코드를 수정하고 빌드하던 도중 오류로 인해 중단되면, 디버깅 후 중단된 시점부터 다시 빌드를 진행할 수 있으므로 작업 효율이 향상됩니다.

Makefile에서 빌드 대상(Target)별로 의존성을 명시해 주면 이에 따라 자동으로 Incremental build를 수행해 주므로 매우 편리합니다. 또한, 자주 사용되는 빌드 규칙은 기술하지 않아도 내부적으로 자동으로 처리해 주기 때문에 쉘 스크립트에 비해 편리하고 깔끔합니다.

■ Make로 깔끔하게 빌드하기: Makefile 작성법 튜토리얼

Makefile는 Bash 쉘 스크립트와 문법이 비슷하기 때문에 쉘 스크립트에 익숙한 분은 매우 쉽게 배워서 작성하실 수 있습니다. 실상 Makefile은 빌드 대상(Target) 이름으로 된 Label별로 구분된 쉘 스크립트라고 볼 수 있습니다. 빌드 대상별로 구분된 쉘 스크립트를 make 프로그램에서 읽어서 필요한 영역의 명령들만 실행하는 것입니다.

다음은 위의 예제를 빌드하기 위한 Makefile입니다. 좀 길어 보이지만 우선은 설명을 위해 모든 빌드 규칙들을 풀어서 써 놓은 것입니다. 우선 이를 설명하고, 차차 고급 기법(?)들을 적용해서 줄이고 깔끔하게 바꿔 나갈 것입니다.

Makefile

```

1  app.out: main.o foo.o bar.o
2      gcc -o app.out main.o foo.o bar.o
3
4  main.o: foo.h bar.h main.c
5      gcc -c -o main.o main.c
6
7  foo.o: foo.h foo.c
8      gcc -c -o foo.o foo.c
9
10 bar.o: bar.h bar.c
11     gcc -c -o bar.o bar.c

```

소스코드가 위치하는 디렉토리에 위와 같이 Makefile을 작성하고,

make

명령을 치면 한큐에 실행 파일(app.out)을 만들어 낼 수 있습니다. 이때, 중간과정 부산물인 Object 파일들(main.o, foo.o, bar.o)도 함께 생성됩니다.

make 명령 뒤에 Target을 명시하면 해당 Target만 선별적으로 빌드합니다. 예를 들어, 다음 명령을 치면 'gcc -c -o foo.o foo.c'만 실행되어 foo.o만 생성됩니다.

```
make foo.o
```

빌드 규칙(Rule) 블록

Makefile에서 반복되는 구조인 Rule block의 구조는 다음과 같습니다.

```
<Target>: <Dependencies>
      <Recipe>
```

위의 명칭은 GNU make 공식 매뉴얼에서 그대로 들고 온 것인데, 쉽게 설명해서 다음과 같은 의미입니다.

- **Target:** 빌드 대상 이름. 통상 이 Rule에서 최종적으로 생성해내는 파일명을 써 줍니다.
- **Dependencies:** 빌드 대상이 의존하는 Target이나 파일 목록. 여기에 나열된 대상들을 먼저 만들고 빌드 대상을 생성합니다.
- **Recipe:** 빌드 대상을 생성하는 명령. 여러 줄로 작성할 수 있으며, **각 줄 시작에 반드시 Tab문자로 된 Indent가 있어야 합니다.**

내장 규칙(Built-in Rule)

Make에서는 자주 사용되는 빌드 규칙들은 내장을 해서 굳이 기술하지 않아도 자동으로 처리해 줍니다. **소스 파일(*.c)을 컴파일해서 Object 파일(*.o)로 만들어 주는 규칙**이 여기에 해당합니다. 즉, Makefile에 다음 두 줄만 써도 app.out을 빌드할 수 있습니다.

```
Makefile
1 | app.out: main.o foo.o bar.o
2 |     gcc -o app.out main.o foo.o bar.o
```

헌데, 이렇게만 작성하면 Incremental build를 위한 의존성 검사에서 **헤더 파일의 변경을 감지하지 못하는 문제가 발생합니다.** Make는 소

스 파일의 마지막 변경 시점만 확인할 뿐, 소스코드를 일일이 들여다 보고 어떤 헤더 파일이 포함되었는지 추적하지는 않기 때문입니다.

따라서, 아쉽지만 다음과 같이 각 Target에 대한 Dependencies까지는 명시해 줘야 합니다. 이렇게 하면 헤더 파일만 변경된 경우에도 의존성이 올바르게 탐지됩니다. 물론, 해당 소스파일에 헤더 파일을 추가(#include)할 때마다 이 부분을 업데이트 해 줘야 합니다.(혹은 그냥 방치하다가 원인 불명의 문제로 빌드가 안 되면 Clean build를 하는 방법을 쓸 수도 있습니다.)

Makefile

```
1 app.out: main.o foo.o bar.o
2     gcc -o app.out main.o foo.o bar.o
3
4 main.o: foo.h bar.h main.c
5     foo.o: foo.h foo.c
6     bar.o: bar.h bar.c
```

마지막 세 줄에 있는 Target의 Recipe는 모두 생략되어 있지만, Make 내부 Rule에 의해 컴파일이 수행됩니다.

변수 사용하기

변수를 사용하면 Makefile을 보다 깔끔하고 확장성 있게 작성할 수 있습니다. 변수 선언 및 사용법은 Bash 셸 스크립트에서와 같습니다. 변수들 중에는 Make 내부에서도 함께 사용하는 내장 변수나(CFLAGS 등), 확장성을 용이하게 해 주는 자동 변수(\$@, \$< 등)도 있습니다.

다음은 위의 예제를 빌드하기 위한 같은 목적의 Makefile입니다. (다만, 예시를 위해 컴파일 플래그 몇 개를 추가해 주었습니다.)

Makefile

```
1 CC=gcc
2 CFLAGS=-g -Wall
3 OBJS=main.o foo.o bar.o
4 TARGET=app.out
5
6 $(TARGET): $(OBJS)
7     $(CC) -o $@ $(OBJS)
8
9 main.o: foo.h bar.h main.c
10 foo.o: foo.h foo.c
11 bar.o: bar.h bar.c
```

이제 좀 제대로 된 Makefile 같아진 듯 합니다. 여기서 정의한 각 변수의 의미는 다음과 같습니다. 이 네 개의 변수들은 웬만한 Makefile에

항상 등장하는 것들이므로 눈여겨 봐 두는 것이 좋습니다. 참고로, 이들 중 **CC**와 **CFLAGS**는 Make 내장 변수들입니다.

- **CC**: 컴파일러
- **CFLAGS**: 컴파일 옵션
- **OBJS**: 중간 산물 Object 파일 목록
- **TARGET**: 빌드 대상(실행 파일) 이름

그 외에 자주 사용되는 내장 변수는 다음과 같습니다.

- **LDLFLAGS**: 링커 옵션
- **LDLIBS**: 링크 라이브러리

참고로, Make에서 내부적으로 정의되어 있는 변수들은 다음 명령으로 확인할 수 있습니다. 주석이 같이 달려서 출력되기 때문에 필요한 기능을 바로 찾기 편리합니다.

```
make -p
```

자동 변수(Automatic variables)

위 예제 Makefile의 7번째 줄을 보면 Recipe 중간에 정의한 적이 없는 변수인 **\$@**이 포함되어 있는 것을 알 수 있습니다. **\$@**은 현재 빌드 규칙 블록의 Target 이름을 나타내는 자동 변수입니다.

자동 변수는 위치한 곳의 맥락에 맞도록 치환됩니다. 즉, 7번째 줄의 **\$@**는 Recipe를 실행할 때 **\$(TARGET)**값으로 치환됩니다. 이렇게 하면 Target 이름을 수정할 때 Recipe까지 일일이 찾아서 수정하는 수고를 덜 수 있습니다.

Make에서 지원하는 자동 변수들 중 자주 사용하는 것들은 다음과 같습니다.

- **\$@**: 현재 Target 이름
- **^**: 현재 Target이 의존하는 대상들의 전체 목록
- **?**: 현재 Target이 의존하는 대상들 중 변경된 것들의 목록

사용 가능한 자동 변수들의 전체 목록과 설명은 다음 사이트에서 확인하실 수 있습니다.



http://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

TIP: 빌드 결과물과 부산물을 삭제하는 'Clean Rule' 추가하기

Makefile의 Rule에는 빌드 대상물 뿐만 아니라, 프로젝트에서 요긴하게 사용할 수 있는 매크로를 지정해서 사용할 수도 있습니다. 이렇게 쓰는 대표적이고, 또 널리 사용하는 매크로가 바로 clean입니다.

Clean 매크로는 빌드 결과물(e.g. app.out)과 중간 부산물들(*.o)을 모두 삭제하여 '깨끗한' 상태에서 다시 빌드할 수 있는 환경을 만듭니다. 이렇게 하고 수행하는 빌드를 **클린 빌드(Clean build)**라고 합니다. 아마 어디선가 많이 들어봤고, 또 많이 해 보셨을 것입니다.

Makefile에 다음 빌드 규칙을 추가하면 Clean 매크로를 사용할 수 있습니다. ~~(최소 Clean 매크로까지는 넣어 줘야 Makefile의 완성~!^^)~~

```
clean:
    rm -f *.o
    rm -f $(TARGET)
```

사용법은 뭐.. (다들 잘 아시겠지만) 다음과 같습니다.

```
make clean
```

앞서서도 언급했듯이, 클린 빌드는 잘못된 의존성 검사로 인해 변경사항이 반영되지 않아서 빌드 오류가 발생하는 것으로 추정될 때(;) 수행합니다.(흔히 '**빌드가 꼬였다**'라고 말합니다.) 빌드 도중에 도통 원인을 알 수 없는 문제로 인해 중단되는 경우 '**make clean; make**'를 실행하면 만파식적**이** 불었는자 Error와 Warning이 죄다 사라지고 빌드가 성공하는 신비한 경험을 할 수 **도** 있습니다.

물론 이렇게 빌드가 꼬이는 원인은 Make 자체의 문제가 아니라, 소스 코드를 수정하면서 의존성이 변동할 때(e.g. 헤더 파일을 추가한 경우), Makefile을 제때 업데이트해 주지 않아서 입니다. 결국 개발자 잘못..이라는 이야기인데, 현실적으로 소스 코드를 수정할 때마다 매번 의존성을 검사해서 Makefile에 일일이 기술해주기는 어려우므로 중간중간 클린 빌드를 수행하는 것으로 타협을 하는 것입니다.

첨언하자면, '**make install**'도 이와 같은 방법으로 구현한 것입니다. Install이라고 하니 뭔가 거창한 것처럼 보이지만, 사실 빌드된 바이너리(+라이브러리, man entry, 기타 잡템 등등...)를 파일 시스템의 적당한 위치(/usr/bin 이라던가..)로 복사하는 cp명령들이 나열되어 있을 뿐입니다. Root 소유 디렉토리로 복사해야 하니 항상 sudo와 함께 써야 하는 것이구요..

■ Makefile 기본 패턴

다음은 통상적으로 널리 사용되는 Makefile의 기본 패턴입니다.

새로운 프로젝트를 시작할 때 처음 작성하는 Makefile은 이것을 복사해서 수정해서 사용하면 편리합니다. 소스파일이 추가될 때마다 끝부분에 각 Object 파일들의 의존성을 기술하는 Rule들만 추가해 나가면 됩니다.

Makefile

```

1  CC=<컴파일러>
2  CFLAGS=<컴파일 옵션>
3  LDFLAGS=<링크 옵션>
4  LDLIBS=<링크 라이브러리 목록>
5  OBJS=<Object 파일 목록>
6  TARGET=<빌드 대상 이름>
7
8  all: $(TARGET)
9
10 clean:
11     rm -f *.o
12     rm -f $(TARGET)
13
14 $(TARGET): $(OBJS)
15 $(CC) -o $@ $(OBJS)
```

참고로, Makefile을 작성하는 방법을 기술한 공식 전체 매뉴얼은 다음 사이트에서 확인하실 수 있습니다.

▶ <http://www.gnu.org/software/make/manual/make.html>

■ CMake: Makefile을 쉽고 간편하게 만들고 관리하기

프로젝트의 규모가 커지고 복잡해질수록 Makefile을 유지/보수하는 작업도 점차 버거워지기 시작합니다. 대규모 프로젝트에서는 빌드 대상 소스 파일들을 관리하는 것부터 시작 해서, 빌드 전/중/후 수행하는 작업과 패키지를 구성하는 부속품들을 생성해 내는 작업 등등 여러 가지 Build Step들이 복잡하게 뒤엉키게 됩니다. 예를 들면 다음과 같은 작업들입니다.

- 프로그램의 버전을 명시하는 등의 이유로 빌드에 전 특정 헤더 파일들을 자동 생성하는 경우

- 실행 파일 외에 공유 라이브러리들을 몽땅 함께 생성하는 등 빌드 대상물이 한두개가 아닌 경우
- 프로젝트에 포함된 서브모듈(써드파티 프로그램들)들이 다단계로 존재하는 경우
- 빌드 전 프로그램에 사용되는 리소스 파일들은 한 데 묶어서 가상 파일시스템을 만들어야 하는 경우
- 빌드 완료된 실행 파일로부터 임베디드 프로세서에 퓨징하기 위한 바이너리를 생성해야 하는 경우

...등과 같이 Makefile의 크기가 매우 방대해질 김새가 보인다면, 보다 추상화 된 상위 레벨의 빌드 시스템을 사용하는 편이 정신건강에 좋습니다.

이 중 대표적인 것이 바로 **CMake**입니다. CMake는 Makefile보다 추상화된 기술 문법으로 Build Step을 기술하면, 이로부터 Makefile을 자동으로 생성해 줍니다. 즉, CMake는 Makefile을 기술하는 상위 레벨 추상 계층인 Meta-Makefile이라고 할 수 있습니다.

특히, CMake의 Build Step 기술 파일인 CMakefile은 Eclipse나 CLion 등과 같은 IDE에서 인식하므로, 하나의 프로젝트 소스를 팀원 각자가 선호하는 IDE로 작업할 수 있도록 해 주는 장점도 있습니다.

▶ CMake에 대한 튜토리얼은 다음 글에서 바로 이어서 다루도록 하겠습니다.

연관글



[CMake 튜토리얼] 3. CMakeLists.txt 기본 패턴 (11278) *4

[CMake 튜토리얼] 2. CMakeLists.txt 주요 명령과 변수 정리 (29178) *1

[CMake 튜토리얼] 1. CMake 소개와 예제, 내부 동작 원리 (30065)

PSpice 기초와 활용

₩40,000



전기전자전공을 위한
기초 수학

₩35,000

기초영어공장! 스피킹
자동생산 프로젝트
패키지

₩108,000

TAG

Make, Makefile, CMake

< Prev

[CMake 튜토리얼] 1. CMake 소개와 예제, 내부 동작 원리


[적외선 통신] IR 리모컨 신호 분석

Next >

FacebookTwitterGooglePinterest




이용중인 SNS 버튼을 클릭하여 로그인 해주세요.
SNS 계정을 통해 로그인하면 회원가입 없이 댓글을 남길 수 있습니다.



등록

Comment '2'



Dongmin Kim

Dongmin Kim

2017.02.06 22:03:32

gcc -o main.c main.c 로 소스를 날려버렸다구요? git으로 복구하세요! (짱긏&도망)

댓글 1

일반PSpiceAVRAndroidNginxApacheLinux

XEPythonSecurity

번호	분류	제목	글쓴이	최근 수정일	조회 수
»	일반	[Make 튜토리얼] Makefile 예제와 작성 방법 및 기본 패턴 2 	 TUW	2018.11.21	29680
174	일반	[적외선 통신] IR 리모컨 신호 분석 	 TUW	2017.06.02	7918
173	일반	[적외선 통신] IR 송수신 소자, IR 송수신 회로 	 TUW	2017.06.02	7998
172	일반	GitLab 코드리뷰 페이지 탭 크기(Tab Size) 4칸으로 바꾸기 	 TUW	2019.05.31	1740
171	일반	Linux에서 Code Composer Studio (CCS) - Ti ARM 개발환경 구축하기 	 TUW	2017.06.02	1968
170	Nginx	Nginx에서 자동 Redirection(301 Permanently moved) 설정하기	 TUW	2016.06.25	2321
169	Nginx	Nginx에서 SSL(HTTPS) 보안 서버 설정하기 (+약간의 이론)	 TUW	2016.06.25	3205
168	Security	SSL Handshake 과정	 TUW	2016.06.21	3939
167	Linux	[Ubuntu] 원격 Shell에서 로그인 사용자 디스플레이에 GUI 프로그램 실행하기	 TUW	2016.03.06	2564
166	Linux	Root권한 없이 Wireshark 사용하기 	 TUW	2017.06.02	1507
165	Linux	fstab과 sshfs fuse를 활용한 원격 디렉토리 자동 마운트하기	 TUW	2016.01.11	9211
164	Python	[Django Tutorial] 9. Production - uWSGI를 통해 Nginx 웹 서버와 연동하기 1	 TUW	2018.06.17	7839
163	Python	[Django Tutorial] 8. Production - setting.py설정, Static파일 모으기	 TUW	2017.06.16	2070
162	Python	[Django Tutorial] 7. 백엔드 콘솔에 Custom Command 추가하기 	 TUW	2017.06.16	1932
161	Python	[Django Tutorial] 6. Database 연동하기 - Model설계, Migration 	 TUW	2017.06.16	11453
160	Python	[Django Tutorial] 5. Static 파일 사용하고 관리하기 	 TUW	2017.06.16	4134

≡ 목록

검색

Powered by Xpress Engine / Designed by Sketchbook