情報システム工学 II(システム開発工学)

オープンソース

Learning Programming with OSS: Open Source Software

山田泰司

taiji@aihara.co.jp

株式会社あいはら 研究開発チーム

オープンソースソフトウェアとは

オープンソースソフトウェアとは、ソフトウェアを機械実行形式ではなく、その設計書であるソースコードならびに機械実行形式に再構築する手順書を、広く公開しているソフトウェアを指し、利用者には、その複製・改変・再頒布の自由が保証されている。広く認められている定義については以下を参照：

- OSI: Open Source Initiative, "**オープンソースの定義**," `http://sourceforge.jp/projects/opensource/wiki/Open_Source_Definition`, **これは** Debian **のフリーソフトウェアガイドラインを起源とする文書**

それに対して、プロプライエタリソフトウェアとはその対極のソフトウェアを指す。多くの商用ソフトウェア (MS Windows, MS Office, Adobe Acrobat, Apple iTunes, Skype, etc) **がプロプライエタリソフトウェアである。**

オープンソースの利点とは：

- **フィードバック、開発者となる人材が得られやすい。**BSD, Linux, Fedora, Debian GNU/Linux, etc
- **ソフトウェアへの信頼が得られやすい。**GCC, Apache HTTP Server, OpenSSL, OpenSSH, MySQL, etc
- **情報処理技術の向上が大いに期待できる。**Gnome, KDE, Python, Ruby, Octave, Maxima, etc
- **継続的な開発や利用が大いに期待できる。**TeX, X Window System, Perl, Tcl/Tk, etc

ソースが公開されているので、これらの情報処理技術の宝庫・様々なノウハウに直に接することが出来る。

「そこから学ばない手はない」のであるが、実は学び方に王道がある。それは「開発者と同じ土俵に立つ」ことである。

ソフトウェアライセンスについて

オープンソースソフトウェアの潮流は、私企業がシステムソフトウェアを独占しようとした危機意識から始まった。それは、ソースコードを隠匿したことで利用者がソフトウェアから断絶されてしまうことへの不快感だけではない。そこでは、プロプライエタリなソフトウェアは、利便性を向上させようとする利用者の権利を奪い、情報処理技術の発展を著しく阻害するものとして捉えられている。

そして、そうした開発者及び利用者の権利を守るため、オープンソースソフトウェアにはソフトウェアライセンスが明記されている。大別すると、以下のようなライセンス形態がある。

- BSD ライセンス … 無保証・著作権保持・宣伝条項つき
- 修正 BSD ライセンス、X11 ライセンス … 無保証・著作権保持
- GNU GPL: General Public License … 無保証・著作権保持・同一ライセンス継承条項
- GNU LGPL: Lesser General Public License … 同上だが、リンクについては同一ライセンス継承条項なし
- GNU Affero GPL … GPL に加えて、ウェブサービスは私的利用ではない旨の条項が付記
- Apache ライセンス、Mozilla パブリックライセンス、その他 … 無保証・著作権保持、しかし、それぞれ特許や商標などに関する特別な条項が付記

その他多くのライセンスの見極めについて、「OSI が承認したライセンスか」「GPL と矛盾しないライセンスか」「X11 ライセンスとどこが異なるのか」「GNU GPLv3 とは」などを手掛かりに理解を努めると良い。

- OSI, "OSI 承認ライセンス," `http://www.gnu.org/licenses/license-list.ja.html`.
- FSF, "さまざまなライセンスとそれらについての解説," `http://sourceforge.jp/projects/opensource/wiki/licenses`.
- Open Source Group Japan, "GNU GPLv3 情報ページ," `http://sourceforge.jp/projects/opensource/wiki/GPLv3_Info`.

OSS の利用方法

オープンソースソフトウェアの利用方法としては：

1. **ソースコードから利用する**
   - ▷ **ソースコードツリーからビルド＆インストールする**
   - ▷ **パッケージ管理ツールを利用する** … SRPM(Linux), ports(FreeBSD, OpenBSD), pkgsrc(NetBSD), portage(Gentoo Linux), MacPorts(Mac OS X), etc
2. **コンパイル済みバイナリパッケージを利用する**
   - ▷ Windows **での各種インストーラ**
   - ▷ Mac OS X **でのインストーラ、**`/Application` **配下へのコピー**
   - ▷ Linux **での** rpm, yum、dpkg, apt-get、etc
   - ▷ Solaris **での** pkgadd, etc
   - ▷ Cygwin **での** setup.exe, etc

このようにソースコードツリーを利用する以外の方法は、オペレーティングシステムや環境によってまちまちであるが、なんの苦労もなくオープンソースソフトウェアをインストールできる (はずである)。
「なんの苦労もない」ということは、利用したいオープンソースソフトウェアがどのように構築されているか「学ぶチャンスを自ら逃している」ことを知るべきである。パッケージ管理ツールの利用やバイナリパッケージのインストールを否定するわけではないが、可能な限り、ソースコードツリーからビルド＆インストールすることを推奨する。
**　自らビルドする作業を惜しまなければ「そのオープンソースソフトウェアがどの程度の規模のものか」「他のどういったオープンソースソフトウェアに依存して構築されているのか」を肌で感じることができる。それは、自分が開発者となってプロジェクトに最適なツールを選択しなければならない状況になった時に、極めて有効な知識と経験となり得るのである。それに、苦労したことは忘れないものである。**
**　よってここでは、「ソースコードツリーからビルド＆インストールする」際のコツを伝授する。**

タイプ別ソースコードツリーの取得・ビルド方法

ソースコードツリーのビルド方法を大別すると、以下のようになる。

1. `configure` スクリプトが用意されている → 『configure 系』
2. `Imakefile` が用意されている → 『X11 系』
3. `makefile*` が用意されている → 『makefile 系』
4. `*.c` 等、プログラム単体しか用意されていない → 『単体系』
5. その他、個別の方法が用意されている → 『独自系』

加えて、ソースコードツリーの配布形態にもいくつかある。

1. `*.tar.gz`, `*.tar.bz2` 等の tarball が配布されている
2. さらに、tarball のダイジェストもしくは電子署名が配布されている
3. CVS, Subversion, Git, Bazaar 等のバージョンコントロールシステム (VCS) のリポジトリとして配布
4. オリジナル以外に、取り込まれていないパッチも広く配布されている

まずは、ソースコードツリーの取得方法についてまとめていこう。

tarball の取得方法

tarball の取得方法であるが、もちろんウェブブラウザ上で「マウスでポチクリ」と取得しても構わないわけだが、後に、再現性を確保することと、その再利用の応用へ繋げるために、コマンドラインで取得する方法を示す。

```
$ wget -N http://curl.haxx.se/download/curl-7.19.2.tar.bz2
$ wget -N ftp://ftp.gnupg.org/gcrypt/gnupg/gnupg-1.4.9.tar.bz2
* ここで、-N オプションは、元のタイムスタンプを保持し、既に取得済みなら速やかに終えるための工夫である。

$ curl -RO -C - http://ftp.gnu.org/gnu/wget/wget-1.11.4.tar.gz
$ curl -RO -C - ftp://ftp.gnupg.org/gcrypt/gnupg/gnupg-1.4.9.tar.bz2
* ここで、-O オプションは、そのファイル名にて保存するオプションで、その際の -C - オプションはリトライを効率良く行なうオプションである。
* そして、-R オプションは、元のタイムスタンプを保持するためのオプションである。総じて、既に取得済みなら速やかに終えるための工夫となっている。
```

HTTP の場合、主に負荷分散を意図して、その URL は実体ではなく別のサーバへのリダイレクト情報が記されている場合がある。wget は自動的にリダイレクトするが、cURL はそれを許可しなければならない：

```
$ curl -ROL -C - http://downloads.sourceforge.net/boost/boost_1_37_0.tar.bz2
* ここで、-L オプションは、リダイレクトを許可するオプションである。
```

HTTPS の場合、SSL/TLS 対応の wget、curl である必要がある。さもなくば、それを理解した上で、サイト証明書を検証を省くという選択肢もあり得る。

```
$ wget --ca-certificate=/usr/share/curl/curl-ca-bundle.crt -N https://launchpad.net/bzr/1.10/1.10/+download/bzr-1.10.tar.gz
$ curl --cacert /usr/share/curl/curl-ca-bundle.crt -ROL -C - https://launchpad.net/bzr/1.10/1.10/+download/bzr-1.10.tar.gz
* ここで、--ca-certificate, --cacert オプションはともに認証局が記されたファイルを指定するオプションである。そのファイルは信頼できる方法で取得し存在している必要がある。ともにこのオプションは省略できる場合がある。

$ wget --no-check-certificate -N https://launchpad.net/bzr/1.10/1.10/+download/bzr-1.10.tar.gz
* ここで、--no-check-certificate オプションはサイト証明書の検証を省いて接続を継続するオプションである。つまり、DNS が汚染されていたりサイトが乗っ取られていたりしても、HTTP 同様、それを検証しないと言うことである。
* 但し、curl の場合の同様のオプションは不明。
```

cURL(Mac OS X Tiger 以降ではシステム標準)、wget の使い方についてはそれぞれの MANPAGE を参照のこと：

```
$ man wget
$ man curl
```

tarball の検証方法（1）

取得したファイルが破損していないか検証することは重要であるが、それ以上に、悪意ある者に改竄されていないか検証することは、セキュリティに関するソフトウェアなどにおいて特に重要である。前者にはダイジェストが、後者には電子署名が用いられる。

ダイジェストには伝統的に md5 が、昨今は sha1 が用いられる。検証には `md5sum`, `sha1sum`(GNU Core Utilities)、`md5`, `sha1`(BSD 系) などのコマンドが用いられているが、著名な `openssl` コマンドを使うと同一の方法で検証できる。

MD5:

```
$ test "`openssl md5 gnupg-1.4.9.tar.bz2`" = "MD5(gnupg-1.4.9.tar.bz2)= cc52393087480ac8d245625004a6a30c" \
&& echo OK || echo NG

OK

* 上記で、バッククォートに囲まれた箇所は以下のような実行と出力を表している：
$ openssl md5 gnupg-1.4.9.tar.bz2
MD5(gnupg-1.4.9.tar.bz2)= cc52393087480ac8d245625004a6a30c
```

SHA1:

```
$ test "`openssl sha1 gnupg-1.4.9.tar.bz2`" = "SHA1(gnupg-1.4.9.tar.bz2)= 826f4bef1effce61c3799c8f7d3cc8313b340b55" \
&& echo OK || echo NG

OK

* 上記で、バッククォートに囲まれた箇所は以下のような実行と出力を表している：
$ openssl sha1 gnupg-1.4.9.tar.bz2
SHA1(gnupg-1.4.9.tar.bz2)= 826f4bef1effce61c3799c8f7d3cc8313b340b55
```

ここで、コマンドライン行で指定している 16 進数列は、tarball を取得したのと別の信頼できる取得方法で得られたものでないと、ただ破損していないか検証できるのみで、改竄の検証にはほとんど無力であることを理解しよう。

`openssl` や `md5sum`, `sha1sum`、`md5`, `sha1` の使い方についてはそれぞれの MANPAGE を参照のこと：

```
$ man openssl
```

tarball の検証方法（2）

電子署名には、gnupg: The GNU Privacy Guard **の** gpg **が便利である。**

```
$ wget -N http://curl.haxx.se/download/curl-7.19.2.tar.bz2
$ wget -N http://curl.haxx.se/download/curl-7.19.2.tar.bz2.asc

$ gpg --verify curl-7.19.2.tar.bz2.asc

gpg: 木 11/13 22:05:07 2008 JST に DSA 鍵 ID 279D5C91 で施された署名
gpg: 署名を検査できません: 公開鍵が見つかりません

$ gpg --recv-keys 0x279D5C91 && gpg --verify curl-7.19.2.tar.bz2.asc && echo OK || echo NG
        :
gpg: 木 11/13 22:05:07 2008 JST に DSA 鍵 ID 279D5C91 で施された署名
gpg: “ Daniel Stenberg (Haxx) <daniel@haxx.se> ”からの正しい署名
gpg: 警告: この鍵は信用できる署名で証明されていません!
        :
OK
```

まず電子署名の検証を試みようとして、失敗するものの、公開鍵 ID **が判明。そして、キーサーバから** ID **に対応する公開鍵を取り込み、改めて電子署名の検証を試みている。ここで重要なのは、**gpg --recv-keys 0x279D5C91 **で指定した** ID **もしくは公開鍵は出来ることなら、別の信頼できる取得方法で得られたものを使用すべきである(ウェブサイトのドキュメント等に取得方法が書かれていたりする)。そうした実例として：**

```
$ curl -RO -C - http://jaist.dl.sourceforge.net/sourceforge/clamav/clamav-0.93.tar.gz
$ curl -RO -C - http://jaist.dl.sourceforge.net/sourceforge/clamav/clamav-0.93.tar.gz.sig

$ curl http://www.clamav.net/gpg/tkojm.gpg | gpg --import && gpg --verify clamav-0.93.tar.gz.sig && echo OK || echo NG
        :
gpg: 木  4/10 01:05:53 2008 JST に DSA 鍵 ID 985A444B で施された署名
gpg: “ Tomasz Kojm <tkojm@clamav.net> ”からの正しい署名
gpg: 警告: この鍵は信用できる署名で証明されていません!
        :
OK
```

ClamAV **のドキュメントに則って、**tarball **と電子署名ファイルとは別のサイトにある公開鍵を** gpg --import **で取り込んでいることに注目しよう。**
ちなみに、「警告: この鍵は信用できる署名で証明されていません!**」とあるが、これは手元にある取り込んだ公開鍵を自分で署名していないからであって、改竄の検証にはほぼ十分である。**

ソースコードツリーの準備：tarball と VCS

tarball が取得できたならば、それを作業ディレクトリで展開すればよい。必ず、中身を確認してから展開を行なうこと。さもないと、意図しない場所にファイルがばらまかれてしまうことが稀にあり得る。

```
$ tar tvjf curl-7.19.2.tar.bz2 | less
$ tar xvjf curl-7.19.2.tar.bz2
curl-7.19.2/
        :

$ tar tvzf wget-1.11.4.tar.gz | less
$ tar xvzf wget-1.11.4.tar.gz
wget-1.11.4/
        :
```

さて、諸般の事情で tarball ではなく、CVS, Subversion, Git, Bazaar 等のバージョンコントロールシステム (VCS) のリポジトリからソースコードツリーを準備しなければならない場合がある。プロジェクトサイトでそのやり方が書かれているはずなので、それに従う。ここでは、その典型例を紹介する。

CVS: Concurrent Versions System の場合：

```
$ cvs     -d:pserver:anonymous@gnuplot.cvs.sourceforge.net:/cvsroot/gnuplot login
CVS password:
$ cvs -z3 -d:pserver:anonymous@gnuplot.cvs.sourceforge.net:/cvsroot/gnuplot checkout -P gnuplot
```

Subversion の場合：

```
$ svn checkout svn://svn.ffmpeg.org/ffmpeg/trunk ffmpeg
```

Git - the stupid content tracker の場合：

```
$ git clone git://git.sourceforge.jp/gitroot/nkf/nkf.git
```

Bazaar - next-generation distributed version control の場合：

```
$ bzr branch lp:mailman
```

以上により `gnuplot`, `ffmpeg`, `nkf`, `mailman` というソースコードツリーが作成される。

ソースコードツリーの更新：VCS

バージョンコントロールシステムの場合、ソースコードツリーを取得した上で、さらにリポジトリに更新があった場合に、それを手元に反映させることができる。

CVS: Concurrent Versions System **の場合：**

```
$ cd gnuplot
$ cvs update -dP
```

Subversion **の場合：**

```
$ cd ffmpeg
$ svn update
```

Git - the stupid content tracker **の場合：**

```
$ cd nkf
$ git pull
```

Bazaar - next-generation distributed version control **の場合：**

```
$ cd mailman
$ bzr merge
```

以上により `gnuplot`, `ffmpeg`, `nkf`, `mailman` **というソースコードツリーが更新される。**

各種VCSコマンド対応表

各種 VCS **の主なコマンドの対応表を以下に示す。**

CVS	Subversion	Git	Bazaar
`cvs -dREPO checkout PROJ`	`svn checkout URL`	`git clone URL`	`bzr branch LOC`
`cvs diff -u | less`	`svn diff | less`	`git diff`	`bzr diff | less`
`cvs update`	`svn update && svn status`	`git pull && git status`	`bzr merge && bzr status`
`cvs log | less`	`svn log | less`	`git log`	`bzr log | less`
`cvs annotate FILE`	`svn blame FILE`	`git blame FILE`	`bzr blame FILE | less`
`cvs checkout -r TAG`	`svn checkout -r ARG`	`git checkout REV`	`bzr checkout -r ARG`
`cvs checkout -r BRANCH`	`svn switch BRANCHURL`	`git checkout BRANCH`	`bzr checkout BRANCHLOC`
`cvs update -j BRANCH`	`svn merge -r ARG URL`	`git merge BRANCH`	`bzr merge BRANCHLOC`
`cvs -dREPO init`	`svnadmin create repo`	`git init`	`bzr init`
`cvs import PROJ VTAG RTAG`	`svn import URL`	`git add . && git commit`	`bzr add && bzr commit`
`cvs add FILE`	`svn add FILE`	`git add FILE`	`bzr add FILE`
`cvs rm FILE`	`svn rm FILE`	`git rm FILE`	`bzr rm FILE`
-	`svn mv FILE`	`git mv FILE`	`bzr mv FILE`
`cvs commit`	`svn commit`	`git commit -a && git push URL`	`bzr commit && bzr push LOC`
`cvs tag TAG`	`svn copy TRUNKURL TAGURL`	`git tag TAG`	`bzr tag TAG`
`cvs tag -b BRANCH`	`svn copy TRUNKURL BRANCHURL`	`git branch BRANCH`	`bzr branch BRANCHLOC`

中央集権型 VCS **の** CVS, Subversion **と分散型** VCS **の** Git **と** Bazaar **とはそもそもの概念が異なることに注意しよう。詳しくはそれぞれのヘルプや** MANPAGE **を見て欲しい。**

```
$ man cvs

$ man svn
$ svn help | less

$ man git
$ man git-init

$ man bzr
```

特に Subversion **と** Git **がよくわかる** `http://git.or.cz/course/svn.html` **による入門をお勧めする** (**日本語訳も探せばあるはず**)。

ソースコードツリーの複製：VCS

バージョンコントロールシステムの場合、そのソースコードツリー内でビルド作業を行なっても構わないのだが、もしやり直したい時やそのスナップショットを保存しておくためにも、ソースコードツリーの複製のディレクトリでビルド作業を行なった方が望ましい。その場合、以下のようにしておけばよいだろう。

```
$ mkdir gnuplot-20090115
$ (cd gnuplot && tar cf - . | (cd ../gnuplot-20090115 && tar xf -))
$ tar cvjf gnuplot-20090115.tar.bz2 gnuplot-20090115
```

```
$ mkdir ffmpeg-20090115
$ (cd ffmpeg && tar cf - . | (cd ../ffmpeg-20090115 && tar xf -))
$ tar cvjf ffmpeg-20090115.tar.bz2 ffmpeg-20090115
```

```
$ mkdir nkf-20090115
$ (cd nkf && tar cf - . | (cd ../nkf-20090115 && tar xf -))
$ tar cvjf nkf-20090115.tar.bz2 nkf-20090115
```

```
$ mkdir mailman-20090115
$ (cd mailman && tar cf - . | (cd ../mailman-20090115 && tar xf -))
$ tar cvjf mailman-20090115.tar.bz2 mailman-20090115
```

以上により `gnuplot-20090115`, `ffmpeg-20090115`, `nkf-20090115`, `mailman-20090115` というソースコードツリーに複製され、そのバックアップがアーカイブされる。当然のことながら、この手続きは VCS の種類には無関係である。

ビルド＆インストール方法：単体系

*.c 等、プログラム単体しか用意されていない『単体系』は極めて稀だが、Perl, shell などのスクリプト系には少なからずあるかも知れない。ここでは、dvdbackup-0.1.1 を例に説明する。

ソースコードツリー（＝ビルド作業場所）の準備：

```
$ cat dvdbackup-0.1.1-sakuya-prepare.sh
curl -RO -C - http://dvd-create.sourceforge.net/dvdbackup-0.1.1.tar.gz
curl -RO -C - http://ftp.debian.org/debian/pool/main/d/dvdbackup/dvdbackup_0.1.1-3.diff.gz
tar xvzf dvdbackup-0.1.1.tar.gz
mv dvdbackup dvdbackup-0.1.1
(cd dvdbackup-0.1.1 && gzcat ../dvdbackup_0.1.1-3.diff.gz | patch -p1)

$ sh dvdbackup-0.1.1-sakuya-prepare.sh
$ cd dvdbackup-0.1.1
```

ちなみに、このように Debian GNU/Linux のサイトが有益なパッチを保持しているケースが多い。パッチを眺めてみて価値を判断できるようになれるとよいだろう。

ビルド：

```
$ cat ../dvdbackup-0.1.1-sakuya-build.sh
gcc -O -I/opt/local/include -L/opt/local/lib -o dvdbackup -ldvdread src/dvdbackup.c

$ sh ../dvdbackup-0.1.1-sakuya-build.sh
```

ちなみに、この dvdbackup-0.1.1 は libdvdread というライブラリに依存していることがわかるだろう。本題ではないので詳細は省略するが、先立って libdvdread をインストールしておく必要があることに注意しよう。

インストール：

```
$ cat ../dvdbackup-0.1.1-sakuya-install.sh
cp dvdbackup /opt/local/bin

$ sudo sh ../dvdbackup-0.1.1-sakuya-install.sh
```

ちなみに、このような類のプログラムは、ただ PATH の通っているところへコピーすればよい。

さて、ここで重要なのは準備・ビルド・インストールについて行なったオペレーションが、結果的にか事前的にかは問わないが、それぞれシェルスクリプトの形式で記述されていることである。そのファイル名の sakuya は「コードネーム」のようなもので、ここではホスト名から命名している。また、/opt/local はインストール先のプレフィックスを表しており（よく使われる /usr/local は推奨できない)、特権がない場合は、$HOME/local などに読み替えて欲しい。

また、上記の sudo は特権がない場合は不要である。sudo がないシステムは su で root ユーザになってから引数を実行するものと読み替えて欲しい。

ビルド＆インストール方法：makefile 系

configure も Imakefile もその他独自の方法も用意されておらず、makefile* が用意されている『makefile系』は少なくない（X11 に非依存で、比較的小規模かつプラットフォーム依存性が少ないオープンソースソフトウェアは configure への依存さえも嫌う傾向にあるのだろう）。付属のドキュメントや makefile* をよく読めばビルド＆インストールは比較的簡単な場合が多い。ここでは、Git リポジトリの nkf を例に説明する。

ソースコードツリーの準備：

```
$ cat nkf-20090115-sakuya-prepare.sh
if [ ! -f nkf-20090115.tar.bz2 ]; then
  if [ ! -d nkf ]; then
    git clone git://git.sourceforge.jp/gitroot/nkf/nkf.git
  else
    (cd nkf && git pull)
  fi
  [ ! -d nkf-20090115 ] && mkdir nkf-20090115
  (cd nkf && tar cf - . | (cd ../nkf-20090115 && tar xf -))
  tar cvjf nkf-20090115.tar.bz2 nkf-20090115
else
  tar xvjf nkf-20090115.tar.bz2
fi

$ sh nkf-20090115-sakuya-prepare.sh
$ cd nkf-20090115
```

ちなみに、これは先に説明した「ソースコードツリーの準備・更新・複製」をより精密にしたものである。

ビルド：

```
$ cat ../nkf-20090115-sakuya-build.sh
make

$ sh ../nkf-20090115-sakuya-build.sh
```

予め、Makefile を読んだ結果、install ターゲットもなく、かつ、インストール先に依存しない類であることがわかり、make を試みても特に問題が発生しなかった例である。ここで問題が発生した場合、さらにチャレンジしてみるかは、メリットとスキルとコストなどに依存するだろう。ちなみに、Debian, MacPorts などを探せば問題が解決することが多い。

インストール：

```
$ cat ../nkf-20090115-sakuya-install.sh
cp nkf /opt/local/bin
cp nkf.1 /opt/local/man/man1

$ sudo sh ../nkf-20090115-sakuya-install.sh
```

ビルド＆インストール方法：makefile 系（2）

Makefile のみ用意されているが、それを読むとインストール先が prefix=/usr/local の形でに固定されている場合も多い。ここでは getopt-1.1.4 を例に説明する。

ソースコードツリーの準備：

```
$ cat getopt-1.1.4-sakuya-prepare.sh
curl -RO -C - http://software.frodo.looijaard.name/getopt/files/getopt-1.1.4.tar.gz
tar xvzf getopt-1.1.4.tar.gz

$ sh getopt-1.1.4-sakuya-prepare.sh
$ cd getopt-1.1.4
```

ビルド：

```
$ cat ../getopt-1.1.4-sakuya-build.sh
make prefix=/opt/local LIBCGETOPT=0 'CC=gcc -I/opt/local/include' 'LDFLAGS=-L/opt/local/lib -lintl'

$ sh ../getopt-1.1.4-sakuya-build.sh
```

ここで注意すべきは、Makefile を書き換えないのであれば、インストール時だけでなくビルド時も make prefix=/opt/local の引数が必須であることである。ちなみに、他の引数は、ここで特有の問題への対処である（CPPFLAGS=-I/opt/local/include としたいところを、CPPFLAGS が使用済みゆえに CC="gcc -I/opt/local/include"としていることなど）。

インストール：

```
$ make -n prefix=/opt/local install install_doc 2>&1 | less     # make の -n オプションは、実際には実行しないが、実行されるコマンド群を表示するためのオプション

$ cat ../getopt-1.1.4-sakuya-install.sh
make prefix=/opt/local install install_doc

$ touch ../.getopt-1.1.4-sakuya.touch
$ sudo sh ../getopt-1.1.4-sakuya-install.sh
$ touch ../.getopt-1.1.4-sakuya.touched
$ find /opt -newer ../.getopt-1.1.4-sakuya.touch ! -newer ../.getopt-1.1.4-sakuya.touched ! -type d | tee ../getopt-1.1.4-sakuya.installed
/opt/local/bin/getopt
/opt/local/lib/getopt/getopt-parse.bash
/opt/local/lib/getopt/getopt-parse.tcsh
/opt/local/lib/getopt/getopt-test.bash
/opt/local/lib/getopt/getopt-test.tcsh
/opt/local/man/man1/getopt.1
```

Makefile に install ターゲットがあるが、make install を盲目的に行なうのは、かなり不安であろう。そこで実際には実行しない make -n install で様子を伺う。それでもそこからインストールされるアイテムを把握するのは難しいかもしれない。よって（事後的ではあるが）、find -newer 〜 ! -newer … でインストールされたアイテムをリストアップしている。

ビルド＆インストール方法：Imakefile 系

Imakefile が用意されているのは X Window System, Version 11(X11) に依存したグラフィカルなプログラムがほとんどである。ここでは、日本語の扱いに優れたターミナルである KTerm を例に説明する。

ソースコードツリーの準備：

```
$ cat kterm-6.2.0-sakuya-prepare.sh
curl -RO -C - ftp://ftp.ring.gr.jp/pub/X/opengroup/contrib/applications/kterm-6.2.0.tar.gz
curl -RO -C - http://www.st.rim.or.jp/~hanataka/kterm-6.2.0.ext03.patch.gz
tar xvzf kterm-6.2.0.tar.gz
(cd kterm-6.2.0 && gzcat ../kterm-6.2.0.ext03.patch.gz | patch -p1)

$ sh kterm-6.2.0-sakuya-prepare.sh
$ cd kterm-6.2.0
```

ここでは、KTerm にさらに「JIS X 0213:2000(第三水準、第四水準)、JIS X 0213:2004」対応パッチを当てていることに注目。

ビルド：

```
$ cat ../kterm-6.2.0-sakuya-build.sh
xmkmf -a
make

$ sh ../kterm-6.2.0-sakuya-build.sh
```

Imakefile が用意されているので、xmkmf -a コマンドで Makefile(s) 等を作成し、make する。ちなみに xmkmf -a は xmkmf && make Makefiles && make includes && make depend と等価である。ここで、インストール先を指定したいところだが、X11 に依存したプログラムの場合、所定の /usr/X11* 以外にインストールさせようとすることは不可能ではないが、ここでは素直に従うものとする。

インストール：

```
$ touch ../.kterm-6.2.0-sakuya.touch
$ sudo make install install.man
$ touch ../.kterm-6.2.0-sakuya.touched
$ find /usr/X11* /etc/X11 -newer ../.kterm-6.2.0-sakuya.touch ! -newer ../.kterm-6.2.0-sakuya.touched ! -type d | tee ../kterm-6.2.0-sakuya.installed
/usr/X11R6/bin/kterm
/usr/X11R6/lib/X11/doc/html/kterm.1.html
/usr/X11R6/man/man1/kterm.1
/etc/X11/app-defaults/KTerm
```

Imakefile から生成された Makefile(s) であれば、基本的に make install install.man でインストールできる。するとここでは、/usr/X11* 配下にインストールされるが、昨今の環境では /usr/X11R6/lib/X11/app-defaults が /etc/X11/app-defaults へのシンボリックリンクになっていることが多い。よって、以上のようにインストールされたアイテムをリストアップしている。

もしビルド＆インストールで Imakefile 系に問題があると、どう対処してよいものか困惑するかも知れない。しかし実は xmkmf スクリプトと `man imake` を見ればわかるように、Imakefile は Makefile(s) を CPP(C プリプロセッサ) を使ってシステム毎の設定 /usr/X11R6/lib/X11/config/* に応じて生成しているだけである。よってそこから定義を辿っていけば問題が解決できることが多い。

ビルド＆インストール方法：configure 系

昨今は configure スクリプトが用意されているオープンソースソフトウェアが大勢を占める。これを使えば、ほとんどのシステムで大きな問題なくビルド＆インストールできる。ここでは、gnupg-1.4.9 を例に説明する。

ソースコードツリーの準備：

```
$ cat gnupg-1.4.9-sakuya-prepare.sh
verify_digest(){        # $0 method file string
  DIGEST=`echo "$1" | tr a-z A-Z`
  [ "$DIGEST($2)= $3" = "`openssl $1 $2`" ]
}
error_out(){    # $0 message ...
  echo "$@" 1>&2; exit 1
}
wget -N ftp://ftp.gnupg.org/gcrypt/gnupg/gnupg-1.4.9.tar.bz2
wget -N ftp://ftp.gnupg.org/gcrypt/gnupg/gnupg-1.4.9.tar.bz2.sig
verify_digest sha1 gnupg-1.4.9.tar.bz2 "826f4bef1effce61c3799c8f7d3cc8313b340b55" || error_out "$0: stopped at line $LINENO"
tar xvjf gnupg-1.4.9.tar.bz2

$ sh gnupg-1.4.9-sakuya-prepare.sh
$ cd gnupg-1.4.9
```

ちなみに、これは先に説明した「tarball の取得とダイジェストの検証」をより精密にしたものである。

ビルド：

```
$ cat ../gnupg-1.4.9-sakuya-build.sh
./configure --prefix=/opt/local
make

$ sh ../gnupg-1.4.9-sakuya-build.sh 2>&1 | tee ../.gnupg-1.4.9-sakuya-build.log
```

configure スクリプトが用意されているので、それで Makefile(s) を作成し、make する。インストール先は --prefix オプションで指定できる。先立って付属のドキュメントや `configure --help` を読んでおくことが肝要である。

インストール：

```
$ touch ../.gnupg-1.4.9-sakuya.touch
$ sudo make install 2>&1 | tee ../.gnupg-1.4.9-sakuya-install.log
$ touch ../.gnupg-1.4.9-sakuya.touched
$ find /opt -newer ../.gnupg-1.4.9-sakuya.touch ! -newer ../.gnupg-1.4.9-sakuya.touched ! -type d | tee ../gnupg-1.4.9-sakuya.installed
```

configure から生成された Makefile(s) は、基本的に make install でインストールできる。するとここでは、/opt/local 配下にインストールされるので、以上のようにインストールされたアイテムをリストアップしている。また、ビルド時、インストール時ともに、その様子を tee コマンドでログファイルに保存していることにも注目しよう。これは特に confi gure系は大量の手続きを経るので、問題が発生した時にそれを逃さず確実に原因を辿れるようにするための工夫である。

もしビルド＆インストールで confi gure系に問題があるときは、手っ取り早く configure を変更しても構わない。しかし configure は configure.in から生成されたファイルなので、理想的には元のファイルにおいて問題の解決を計った方が好ましい。その場合、後述するように autotools が必要になる。

ビルド＆インストール方法：configure 系（2）

VCS で準備した confi gure系のソースコードツリーには `configure` は存在しないので autotools(autoconf, automake, libtools) を使って `configure` を生成する必要がある。ここでは、gnuplot を例に説明する。

ソースコードツリーの準備：

```
$ cat gnuplot-20090115-sakuya-prepare.sh
if [ ! -f gnuplot-20090115.tar.bz2 ]; then
  if [ ! -d gnuplot ]; then
    cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/gnuplot login
    cvs -z3 -d:pserver:anonymous@gnuplot.cvs.sourceforge.net:/cvsroot/gnuplot co -P gnuplot
  else
    (cd gnuplot && cvs update)
  fi
  [ ! -d gnuplot-20090115 ] && mkdir gnuplot-20090115
  (cd gnuplot && tar cf - . | (cd ../gnuplot-20090115 && tar xf -))
  tar cvjf gnuplot-20090115.tar.bz2 gnuplot-20090115
else
  tar xvjf gnuplot-20090115.tar.bz2
fi

$ sh gnuplot-20090115-sakuya-prepare.sh
$ cd gnuplot-20090115
```

ビルド：

```
$ cat ../gnuplot-20090115-sakuya-build.sh
./prepare
./configure CPPFLAGS=-I/opt/local/include LDFLAGS=-L/opt/local/lib --prefix=/opt/local
make

$ sh ../gnuplot-20090115-sakuya-build.sh 2>&1 | tee ../.gnuplot-20090115-sakuya-build.log
```

gnuplot は固有の工夫をしているので独自の `./prepare` が用意されているが、他の多くのソースコードツリーでは、慣習的に付属の `./bootstrap` や `./autogen.sh` を使う。但し、autotools(autoconf, automake, libtools) はそのバージョンにシビアなので、新しめのものを用意する必要があるかもしれない。うまくいかない場合、autoconf 付属の `autoreconf -i` を試してみよう。

インストール：

```
$ touch ../.gnuplot-20090115-sakuya.touch
$ sudo make install 2>&1 | tee ../.gnuplot-20090115-sakuya-install.log
$ touch ../.gnuplot-20090115-sakuya.touched
$ find /opt -newer ../.gnuplot-20090115-sakuya.touch ! -newer ../.gnuplot-20090115-sakuya.touched ! -type d | tee ../gnuplot-20090115-sakuya.installed
```

ちなみに、`./configure` 時に変数 `CPPFLAGS=-I/opt/local/include LDFLAGS=-L/opt/local/lib` を指定しているが、これは、既に `/opt/local` 配下にあるライブラリを活用せよ、という明示的な設定である。一方、慣習として `/usr/local` 配下のライブラリは autotools に限らず積極的に依存しようとしてしまうので、つまり、`/usr/local` 配下はシステムから捨てにくくなるわけである。つまり、インストール先として `/usr/local` が推奨できないのはそういった理由からであることを覚えておこう。

ビルド＆インストール方法：独自系

独自系として、Perl モジュール、Emacs Lisp スクリプトなどがあげられるが makefile に手順を書いてくれていることも多いので、独自系は稀である。ここでは、Boost Jam と Boost を例に説明する。

ソースコードツリーの準備＆ビルド＆インストール：

```
$ cat boost-jam-3.1.17-sakuya-prepare.sh
wget -N http://downloads.sourceforge.net/boost/boost-jam-3.1.17.tgz
tar xvzf boost-jam-3.1.17.tgz

$ sh boost-jam-3.1.17-sakuya-prepare.sh
$ cd boost-jam-3.1.17

$ cat ../boost-jam-3.1.17-sakuya-build.sh
./build.sh

$ sh ../boost-jam-3.1.17-sakuya-build.sh 2>&1 | tee ../.boost-jam-3.1.17-sakuya-build.log

$ cat ../boost-jam-3.1.17-sakuya-install.sh
cp bin.macosxppc/bjam /opt/local/bin/

$ sudo sh ../boost-jam-3.1.17-sakuya-install.sh
```

結局のところ、付属のドキュメントに書かれていることを、再現性のあるように、ログを取りながら進めるだけである。但し、後述するが、ビルド前にファイルをひとつ書き換える必要がある。引続き、Boost について説明する。

ソースコードツリーの準備＆ビルド＆インストール：

```
$ cat boost_1_37_0-sakuya-prepare.sh
wget -N http://downloads.sourceforge.net/boost/boost_1_37_0.tar.bz2
tar xvjf boost_1_37_0.tar.bz2

$ sh boost_1_37_0-sakuya-prepare.sh
$ cd boost_1_37_0

$ cat ../boost_1_37_0-sakuya-build.sh
bjam --prefix=/opt/local --toolset=darwin

$ sh ../boost_1_37_0-sakuya-build.sh 2>&1 | tee ../.boost_1_37_0-sakuya-build.log

$ cat ../boost_1_37_0-sakuya-install.sh
bjam --prefix=/opt/local --toolset=darwin install

$ touch ../.boost_1_37_0-sakuya.touch
$ sudo sh ../boost_1_37_0-sakuya-install.sh 2>&1 | tee ../.boost_1_37_0-sakuya-install.log
$ touch ../.boost_1_37_0-sakuya.touched
$ find /opt -newer ../.boost_1_37_0-sakuya.touch ! -newer ../.boost_1_37_0-sakuya.touched ! -type d | tee ../boost_1_37_0-sakuya.installed
```

結局のところ、付属のドキュメントに書かれていることを、再現性のあるように、ログを取りながら進めるだけである。ちなみに、`--toolset=darwin` の箇所はシステムに依存する。

パッチの作成および再現性の確保

ソースコードツリーそのままではシステムや環境に合わず、ファイルを書き換えなければならない場合は少なからずあり得る。ここでは、先の Boost Jam を例に説明する。

パッチの生成：

```
$ cd boost-jam-3.1.17
find . -name '*~' | while read f; do
  diff -ru "$f" "`dirname \"$f\"`/`basename \"$f\" ~`";
done | tee ../boost-jam-3.1.17-sakuya.patch

$ cat boost-jam-3.1.17-sakuya.patch
--- ./Jambase~  2007-12-03 12:27:54.000000000 +0900
+++ ./Jambase   2007-12-14 00:10:40.000000000 +0900
@@ -909,7 +909,7 @@
     case MACOSX :
     AR          ?= libtool -o ;
     C++         ?= c++ ;
-    MANDIR      ?= /usr/local/share/man ;
+    MANDIR      ?= /opt/local/share/man ;
     RANLIB      ?= "" ;

     case NCR :
@@ -974,7 +974,7 @@
     AS          ?= as ;
     ASFLAGS     ?= ;
     AWK         ?= awk ;
-    BINDIR      ?= /usr/local/bin ;
+    BINDIR      ?= /opt/local/bin ;
     C++         ?= cc ;
     C++FLAGS    ?= ;
     CC          ?= cc ;
@@ -992,12 +992,12 @@
        :
```

上記では、必ずオリジナルファイルが Emacs のバックアップファイル `*~` で保存されているものと仮定し、その上ですべての変更ファイルについての diff 出力を保存している。

再現性の確保：

```
$ cat >> boost-jam-3.1.17-sakuya-prepare.sh
(cd boost-jam-3.1.17 && patch -p < ../boost-jam-3.1.17-sakuya.patch)
^D

$ cat boost-jam-3.1.17-sakuya-prepare.sh
wget -N http://downloads.sourceforge.net/boost/boost-jam-3.1.17.tgz
tar xvzf boost-jam-3.1.17.tgz
(cd boost-jam-3.1.17 && patch -p < ../boost-jam-3.1.17-sakuya.patch)
```

`boost-jam-3.1.17-sakuya-prepare.sh` にパッチ適用のためのコマンドを追記「`>>`」しておけば再現性が確保でき、このソフトウェアのバージョンアップに際しての再構築にも役立つ。

ビルド＆インストール作業支援スクリプト

さて、ここまで説明した「ビルド＆インストール作業」が以下のように簡略化できると好ましいのではないだろうか。

```
$ sh clamav-0.93-sakuya-prepare.sh                                  # ソースコードツリーの準備
$ cd clamav-0.93

$ emacs etc/clamd.conf etc/freshclam.conf                           # ファイルの編集
$ make-patches.sh                                                   # ../clamav-0.93-sakuya.patch の生成
$ cat >> clamav-0.93-sakuya-prepare.sh                              # パッチの再現性の確保
(cd clamav-0.93 && patch -p < ../clamav-0.93-sakuya.patch)
^D

$ make-logging.sh ./configure CPPFLAGS=-I/opt/local/include LDFLAGS=-L/opt/local/lib --prefix=/opt/local/clamav
$ make-logging.sh -a make                                           # ../clamav-0.93-sakuya-build.sh の生成
$ make-installed.sh -s make install                                 # ../clamav-0.93-sakuya-install.sh, ../clamav-0.93-sakuya.installed の生成
```

また、インストールされたアイテムの保存やアンインストールが以下のように簡略化できると好ましいのではないだろうか。

```
$ less clamav-0.93-sakuya.installed                                 # インストールアイテムのリストの確認

$ installed-tbz2.sh clamav-0.93-sakuya.installed                    # clamav-0.93-sakuya-binaries.tar.bz2 の作成

$ sudo uninstall-tbz2.sh clamav-0.93-sakuya.installed               # clamav-0.93-sakuya-binaries.tar.bz2 の作成とアンインストール
* 必ず、リストの確認をすること！ `find -newer 〜 ! -newer …`は、常に完璧というわけではない。
```

さらに、ClamAV がバージョンアップした時に新たな「ビルド＆インストール作業」が以下のように簡略化できると好ましいのではないだろうか。

```
$ make-update.sh clamav-0.93 clamav-0.94.2
clamav-0.94.2-sakuya-prepare.sh                                     # clamav-0.93-sakuya-prepare.sh を元に新しい *-prepare.sh の作成
clamav-0.94.2-sakuya.patch                                          # clamav-0.93-sakuya.patch      を元に新しい *.patch      の作成
clamav-0.94.2-sakuya-build.sh                                       # clamav-0.93-sakuya-build.sh   を元に新しい *-build.sh   の作成
clamav-0.94.2-sakuya-install.sh                                     # clamav-0.93-sakuya-install.sh を元に新しい *-install.sh の作成

$ sh clamav-0.94.2-sakuya-prepare.sh                                # ソースコードツリーの準備
$ cd clamav-0.94.2

$ make-logging.sh                                                   # ../clamav-0.93-sakuya-build.sh の実行
$ make-installed.sh                                                 # ../clamav-0.93-sakuya-install.sh の実行, ../clamav-0.94.2-sakuya.installed の生成
```

そのような、ビルド＆インストール作業支援スクリプト（ここでは、make-logging.sh、make-installed.sh、make-patches.sh、make-update.sh、uninstall-tbz2.sh）の作成はそれほど難しくない（配布しているシェルスクリプト群 make-logging-20090120.tar.bz2 を参考にせよ）。＊「これのもっと便利なのがパッケージ管理ソフトなんじゃないの」という意見には「あくまで自分のためのビルド＆インストール作業支援ソフトが便利なわけであって、他のユーザのためにパッケージ管理しているわけではない、そんな依存さえ増やすべきではない」と反論したい。

ビルド＆インストール作業支援スクリプト（つづき）

ちなみに、作成された clamav-0.94.2-sakuya-prepare.sh、clamav-0.94.2-sakuya-build.sh、clamav-0.94.2-sakuya-install.sh **を記しておこう。**

ソースコードツリーの準備：

```
verify_signature(){     # $0 url|id ... sig
  while [ "$1" != "" ]; do
    case "$1" in
    0x*)
      gpg --recv-keys "$1" || return $?
      ;;
    http:*|ftp:*)
      curl "$1" | gpg --import || return $?
      ;;
    *)
      gpg --verify "$1"
      return $?
    esac
    shift
  done
}
error_out(){    # $0 message ...
  echo "$@" 1>&2; exit 1
}
curl -RO -C - http://jaist.dl.sourceforge.net/sourceforge/clamav/clamav-0.94.2.tar.gz
curl -RO -C - http://jaist.dl.sourceforge.net/sourceforge/clamav/clamav-0.94.2.tar.gz.sig
verify_signature http://www.clamav.net/gpg/tkojm.gpg clamav-0.94.2.tar.gz.sig || error_out "$0: stopped at line $LINENO"
tar xvzf clamav-0.94.2.tar.gz
(cd clamav-0.94.2 && patch -p0 -b -z.org < ../clamav-0.94.2-sakuya.patch)
```

ちなみに、これは先に説明した「tarball **の取得と電子署名の検証」をより精密にしたものである。**

ビルド：

```
./configure CPPFLAGS=-I/opt/local/include LDFLAGS=-L/opt/local/lib --prefix=/opt/local/clamav
make
```

インストール：

```
make install
```

tarball の検証方法（３）

ところで、極めて稀だがセキュリティ的に極めて重要な「局面」において、電子署名が多重になされている場合がある。ここでは、DNS キャッシュ汚染の脆弱性に対処された bind-9.5.0-P2 を例に説明する。

ソースコードツリーの準備：

```
$ cat bind-9.5.0-P2-sakuya-prepare.sh
verify_signature(){     # $0 url|id ... sig
  while [ "$1" != "" ]; do
    case "$1" in
    0x*)
      gpg --recv-keys "$1" || return $?
      ;;
    http:*|ftp:*)
      curl "$1" | gpg --import || return $?
      ;;
    *)
      gpg --verify "$1"
      return $?
    esac
    shift
  done
}
error_out(){    # $0 message ...
  echo "$@" 1>&2; exit 1
}
curl -RO -C - http://ftp.isc.org/isc/bind9/9.5.0-P2/bind-9.5.0-P2.tar.gz
curl -RO -C - http://ftp.isc.org/isc/bind9/9.5.0-P2/bind-9.5.0-P2.tar.gz.asc
verify_signature 0xD297AB8E http://www.isc.org/about/openpgp/pgpkey2006.txt bind-9.5.0-P2.tar.gz.asc || \
error_out "$0: stopped at line $LINENO"

tar xvzf bind-9.5.0-P2.tar.gz
```

ここで、先と同じ verify_signature 関数の引数として、公開鍵が２つと電子署名が１つを指定し、２つの公開鍵を gpg に読み込み、電子署名を検証していることに注目せよ。DNS サーバのひとつである bind の脆弱性はサイトを偽装できる可能性を孕んでいるので、このような慎重な対策がとられる。DNS サーバを立ち上げている管理者としては、この手続きは理に適った、至極当然の検証作業であり、手抜きをしないようにしたいものである。（ビルド他は省略）

オープンソース：まとめ

まずは補足から。オープンソースソフトウェアは、他のさまざまなオープンソースソフトウェアの上に成り立っている。自らビルド＆インストールするために必要なものは、使いたいソフトが依存しているソフトをひとつひとつ根気良くインストールしていくことである (それが、新たな発見に繋がる)。以下に、本文で扱ったもののうち、敢えてインストールしなければシステムに入っていないであろう依存関係を示しておく。

```
git-1.6.1: curl, expat, xmlto, asciidoc
asciidoc-8.3.3: xmlto, Python >= 2.3
xmlto-0.0.21: getopt, libxml2
bzr-1.10: Python >= 2.4
subversion-1.4.5: apr-util, apr-iconv, apr
autoconf-2.63: m4 >= 1.4.5
automake-1.10.2: autoconf
libtool-2.2.4:
ffmpeg-20090115: liba52
dvdbackup-0.1.1: libdvdread
```

大切なことは、恐がらずに、とにかくやってみることである。とは言っても、最初から Gnome、OpenOffi ce.orgのような大規模ソフトは無謀。

また、研究開発に役立ちそうなオープンソースソフトウェアのビルド＆インストールについて、以下に紹介しておくので、是非参考にして欲しい。

- Boost C++ **ライブラリ** … `http://www.aihara.co.jp/~taiji/macosx/install-log/sakuya.html#science/boost/boost_1_37_0`
- Imaxima - Maxima in Emacs … `http://www.aihara.co.jp/~taiji/macosx/install-log/sakuya.html#science/maxima/imaxima-imath-0.99`
  数式処理システム Maxima … `http://www.aihara.co.jp/~taiji/macosx/install-log/sakuya.html#science/maxima/maxima-5.14.0`
  GNU CLISP … `http://www.aihara.co.jp/~taiji/macosx/install-log/sakuya.html#science/maxima/clisp-2.43`
- GNU **科学ライブラリ** … `http://www.aihara.co.jp/~taiji/macosx/install-log/sakuya.html#libs/gsl/gsl-1.8`
- GNU Octave - Matlab **クローン** … `http://www.aihara.co.jp/~taiji/macosx/install-log/sakuya.html#science/octave/octave-3.0.3`

まとめ：

- **インストーラ、**rpm, apt-get, pkgadd **を動かす時、 誰と誰と誰と誰と〜を「信用」していることになっているのか、考えたことがあるか？**
- SRPM, ports, pkgsrc, portage, MacPorts **を使う時、 誰と誰と誰と〜 を「信用」していることになっているのか、考えたことがあるか？**
- **ソースコードツリーからビルド＆インストールするとき、誰と誰と〜 を「信用」しているのか、もうなんとなくわかったはず**
- **信用して（盲信して）人任せにするか、それとも、リスクを軽減しつつ自分でビルド＆インストールするか、(それとも自分で零から作るか、はまず無理)**
- **労力に見合うメリットがあるのだから、自分で出来ることは自分でしよう！そしてすぐに大変ではなくなるはず！**
- **そして、ライセンスやドキュメントがどこにあるのか把握しているので、ソフトそのものを適切に使うことも比較的やりやすいはず！**