4. 再現可能なビルド

|  |
| --- |
| Go to the ant, thou sluggard; consider her ways and be wise.  なまけ者よ、アリのところへ行き、そのすることを見て、知恵を得よ。  (旧約聖書『箴言』第6章6節より) |

# ソフトウェアをビルドする

本章は、ビルド作業を扱います。まず、この概略を確認しましょう。図4 - 1を見てください。



図4 - 1 ビルドツールによるビルドの構築

「ビルドをする」とは、複数のソースファイル (一次ファイル) を複数のツールで加工して統合し、最終的に実行可能なソフトウェアを生成することです。これが素朴で基本的なビルドの働きです。Javaプログラムなら、JavaコンパイラとJarアーカイバというツールでソースファイルを実行可能ファイルに変換します。C/C++プログラムなら、C/C++コンパイラとリンカを使ってビルドします[[1]](#footnote-1)。これだけの作業でも、ソースファイルの数が多いと、手作業でビルドするのは難しいものです。というのは、複数のソースファイルと、そこから生成される複数の中間ファイルの間には複雑な依存関係があるからです。複数の外部ツールを、適切なパラメータを伴って、正しい順序で呼び出さなければなりません。ビルドするたびに違うビルドができるのでは困ります。ビルド作業の不備により、その成果物にバグが入り込むこともあります。次節では、これを避けるために必要なことを整理します。

# 再現可能なビルド

再現可能なビルドとは、いちどビルドしたソフトウェアを、あとでもう一度作れることを指していいます。誰がいつビルドしても、同じビルドを作れるということです。一方で、再現性のないビルドとは、ビルドするたびに振る舞いが変わってしまったり、そもそもビルドし直すことさえできないビルドのことです。そんなビルドには何の商品価値もないことは、すぐにわかります。仮に、たまたま安定したビルドを構築できたとしても、それが再現できないものなら、ユーザーにリリースなどできません(保守をする気がないなら、別ですが)。

プライベートビルドのようにリリースの範囲が狭いビルドは、再現ができなくても問題はありません。しかし、ビルド番号が管理されてテストチームやユーザーにリリースされるビルドのように、リリースの範囲が広くなるほど、そのビルドには再現可能性が強く求められます。

そこで、再現可能なビルドを実現するために必要なものを列挙してみます。

## ビルドの材料。(ソースファイル一式)

まず、再現したいビルドのソースファイル一式が必要です。当たり前ですが、もしソースファイルを紛失したら、同じビルドは再現できません。紛失までしなくても、当該のビルドをそのソースファイル一式のどのリビジョンでビルドしたのかを忘れると、やはり同じビルドは再現できません。

## ビルドのレシピ。(ビルド手順)

ビルドを構築するには、その構築手順が必要です。開発の段階が進むにつれ、ビルドの手順は少しずつ変化していきます。しかし、以前のビルドを再現するためには、最新のビルド手順ではなく、その当時のビルド手順を使わねばなりません。そこで、ビルド手順書やビルドスクリプトなども、ソースファイルと一緒にSCMリポジトリで管理します。

## ビルドの厨房と調理器具。(ビルド環境とビルドツール)

同じ材料と手順でホットケーキを焼いても、厚みの違うフライパンや火力の違うコンロで調理すると、違う焼き色がつきます。環境や道具が違えば、同じホットケーキは再現できないのです。ソフトウェアのビルドも同様です。同じ材料と手順に加えて、同じ環境と道具(同じバージョンのコンパイラやビルドツールなど)が必要です[[2]](#footnote-2)。

「料理」という語と同じく、「ビルド」という語には名詞と動詞の両方の意味があります。2章では、名詞としてのビルドの種別を紹介しました。また、3章ではビルドの材料であるソースファイル一式を管理する方法を説明しました。本章では、動詞としてのビルド、つまりビルド作業を扱います。ビルドの手順と、ビルド環境とビルドツールについて説明しようというわけです。まず、基本的なビルドツールを紹介します。次に、具体的なビルド手順を説明し、最後にCIサーバを使ったより高度なビルド環境を紹介します。

# 動詞としてのビルドの種別

ビルドツールとビルドの手順を説明する前に、用語の確認をしておきましょう。名詞としてのビルドの種別を2章で紹介したように、動詞としてのビルドにもいくつか種類があります。

## Full build (フルビルド)

すべてのソースファイルを使ってビルドします。Complete build(完全ビルド)ということもあります。

一般に、ビルドマシンで統合ビルドをするときは、必ずフルビルドをすべきです。特に、ユーザーにリリースするビルドは必ずフルビルドで構築しなければなりません。これは、フルビルドが最も良い再現性を得られるビルド方法だからです。

その一方で、フルビルドに必要なビルド時間はとても長くなります[[3]](#footnote-3)。規模が大きい製品や、自動化されたテストが多く準備された製品では、ビルド時間が半日からまる一日にもなることがあります。

## Incremental build (インクリメンタルビルド)

フルビルドに対して、「すべてのソースファイルを使わないビルド」のことをインクリメンタルビルドといいます。Complete buildに対して、Partial build(部分ビルド)ということもあります。インクリメンタルビルドは、前回のビルド時点から修正されたソースファイルだけを処理し、これに依存する中間ファイルだけを生成し直します。それ以外のファイルについては、前回のビルドで生成された中間ファイルを再利用することで、ビルド時間を劇的に短くします。もちろん、インクリメンタルビルドを指示しても、これが初めてのビルド作業なら、再利用できる中間ファイルはありませんから、フルビルドと同じことが起こります(さもなければ、ビルドスクリプトのバグです)[[4]](#footnote-4)。

一般に、サンドボックスで開発作業中は、一日に何度もビルドしなければいけませんから、ビルド時間を節約するためにインクリメンタルビルドが多用されます。

その一方で、インクリメンタルビルドばかりを続けていると、ビルドの再現性が不安定になります。中間ファイルのそれぞれがいつのビルドで生成されたのか、追跡できないからです。特に、同じ環境で長いことフルビルドをせず、何度もインクリメンタルビルドを繰り返していると、ビルドの再現性が悪くなり、静かなビルドブレークを引き起こす原因となります。サンドボックスでも、せめて数日に1回くらいはフルビルドをしましょう。退社前にビルドを開始しておけば、明朝までにはビルドが終わっているはずです。ただ、次の日に出社してみたらビルドエラーが出ていてビルドが完了していなかったら悲しいことになります。エラーを取り除いてビルドを再開し、それが終わるまでは開発作業を開始できません。朝からいきなりコーヒーブレークというわけです。これは、★3-14節で紹介したように、サンドボックスを2つ作っておくことで対処できます。

## Clean build (クリーンビルド)

前回のビルドで生成された中間ファイルを、すべて削除する操作を「クリーン」といいます。ビルド環境に含まれている中間ファイルをきれいにする、ということです。クリーンの操作をしてからビルドを開始することを、クリーンビルドといいます。クリーンビルドは、フルビルドと同じ意味であると考えて差し支えありません。クリーンした直後にインクリメンタルビルドを開始するのは、フルビルドをするのと同じことです。サンドボックスで開発中に、うかつにクリーンの操作をすると、やはり悲しいことが起きます。気をつけましょう。

## Rebuild (再ビルド)

もういちど、ビルドをやり直すことです。それ以上の意味はありません。しかし、IDEによっては、[ビルド]メニューを選択するとインクリメンタルビルドを、[リビルド]メニューを選択するとフルビルドをするものもあります。お使いの開発ツールのドキュメントを確認してください。

## Parallel build (並行ビルド)

マルチスレッドを活用したビルドをすることで、ビルド時間を短くします。Antを含む複数のビルドツールで、並行ビルドがサポートされています[[5]](#footnote-5)。ただし、並行ビルドをするにはマルチコアCPUの環境が必要です。シングルコアの環境でマルチスレッドを使っても、ビルド時間は逆に遅くなります。並行ビルド時のスレッド数は、最大でもCPUコアの数と同じに設定しましょう。また、並行ビルドをするときは、ファイルの生成順にも注意を払う必要があります。

## Distributed build (分散ビルド)

複数のマシンで、複数のビルドを並行して走らせることで、見かけ上のビルド時間を短くします。また、複数のマシンを使って並行ビルドをすることも分散ビルドといいます。分散ビルドについては後述します。

表4 - 1 フルビルドとインクリメンタルビルドの比較

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 再現性 | ビルド時間 | どこですべき？ | ビルド後に実行すべきテスト |
| フルビルド | 安心 | 長い | ビルドマシンで | 自動化された リグレッション テスト |
| インクリメンタル ビルド | ちょっと 不安 | 短い | サンドボックスで | 自動化された スモークテスト |

# ビルドブレーク

ビルドブレークとは、文字通りビルドが壊れるという意味です。ふたつに分類できます。

## 賑やかなビルドブレーク

自動化されたビルドプロセスが、何らかのエラーで中断することです。例えば、あるプログラムファイルがプログラミング言語の文法エラーでコンパイルできないときは、コンパイラとビルドツールがエラーを報告してビルドプロセスは異常終了します。この結果、実行可能なソフトウェアは出てきません。この場合は、少なくともビルドに失敗したことはすぐわかりますから、ビルドエラーを取り除くのもそれほど難しくありません。

## 静かなビルドブレーク

ビルドシステムがビルドに成功したと報告し、実際にビルドが生成されたのに関わらず、それが意図通りのビルドにならないことです。例えば、ビルドシステムやビルドスクリプトのバグによりインクリメンタルビルドが正しく行われなかったり、古いバージョンのライブラリをリンクしてしまうなどの原因で起こります。これを取り除くのはなかなか厄介です。ビルドに失敗したことが、すぐにはわからないからです。ビルド (実行可能なソフトウェア) が意図通りに動かない場合、ふつうはプログラムの記述ミスが先に疑われ、ビルドシステム(ビルドマシン上のビルド環境)が疑われるのはその次になることが多いものです。開発者が長い時間を費やして調べたあげく、プログラムには何も問題がなく、実はビルドスクリプトがいけなかった、ということになれば、開発者の工数を無駄にしてしまいます。

ですから、ビルドブレークするなら、静かなビルドブレークよりも賑やかなビルドブレークの方が好ましいといえます。静かなビルドブレークを避けるには、ビルドマシンでビルドをする時には必ずフルビルドをする、コンパイラなどのツールが警告を出力したときは、これをエラーと同じように扱ってビルドを中断する、などの工夫が有効です。

ビルドマシンでビルドブレークしたときに、より賑やかさを演出する方法として、自動車工場でラインが止まったときに使われるようなパトランプを光らせるという方法があります。トヨタ生産方式では、このようなランプを指してあんどんといいます。ソフトウェア開発においても、これを「ソフトウェアあんどん」と呼んで楽しく使う人が増えているそうです。異常のほか、ビルドの進捗状態を色分けしてあんどんに表示させることもできるでしょう。また、このような装置を指してXFD (Extreme Feedback Device) ということもあります。開発者に、そのコミットした内容について、過激にフィードバックを伝える装置という意味です。

このほか、悪意あるビルドブレークというのもあります。悪意の固まりみたいな人 (社外の開発者など) が、一見正しく動作するが、実はバックドアを備えたセキュリティ上問題のあるビルドを意図的に作ってしまうなどです[[6]](#footnote-6)。滅多に起きることではありませんが、ソフトウェア会社にとってビルドシステムが非常に重要であり、厳重に保護されるべきものであることがわかります。

# ファイル間の依存関係を管理する

C言語のプログラムをビルドすることを例にとりあげ、ファイルの依存関係について見てみましょう。これは、C言語で記述されたソースファイル間の依存関係が程よく複雑なので、説明するのに都合がいいからです。C言語の文法に関する知識は一切不要なので安心してください。



図4 - 2 Cプログラムファイルの依存関係の例

C言語のプログラムは、拡張子が.cのソースファイルに記述します。また、複数の.cファイルから利用する共通のデータ構造は、拡張子が.hのソースファイル(ヘッダファイル)に記述します。このとき、.cファイルは.hファイルに依存する、といいます。.hファイルを変更したら、これに依存する.cファイルは、それ自体は修正されていなくても再コンパイルする必要が生じます。

Cコンパイラは、.cのファイルをコンパイルして、拡張子が.objの中間ファイルを生成します。さらに、ライブラリアンというツールが複数の.objを結合して.libファイル(ライブラリ)を生成し[[7]](#footnote-7)、リンカが複数の.objファイルと.libファイルを結合して拡張子が.exeの実行可能ファイルを生成します[[8]](#footnote-8)。これらの中間ファイルの間にも、依存関係があります。.objファイルを更新したら、それに依存する.libファイルや.exeファイルも生成し直す必要があります。

図4 - 2には、4つのソースファイルがあります。この例では、コンパイルを3回、リンクを2回するとproduct.exeが生成されます(矢印の数に注目してください)。そこで、あるソースファイルを修正してインクリメンタルビルドを指示したとき、これがどのように動作するかを検討しましょう。

* sub.cを修正した場合

1. sub.cをコンパイルして、これに依存するsub.objを再生成します。
2. sub.objと、前回のビルドで生成済みのlibrary.objをリンクして、これらに依存するlibrary.libを再生成します。
3. library.libと、前回のビルドで生成済みのproduct.objをリンクして、これらに依存するproduct.exeを再生成します。

このシナリオでは、コンパイルが1回、リンクが2回実行されることで新しいproduct.exeがビルドされることがわかります。

* library.hを修正した場合

1. library.hに依存するproduct.cとlibrary.cとsub.cをすべてコンパイルして、product.objとlibrary.objとsub.objを再生成します。
2. 同様に、すべての中間ファイルを生成し直す必要が生じます。最終的に、product.exeが再生成されます。

このシナリオでは、コンパイルを3回、リンクを2回、つまりフルビルドによってproduct.exeがビルドされることがわかります[[9]](#footnote-9)。

このように、ソフトウェアをビルドするときは、複雑な依存関係を管理しなければなりません。例えば、先にproduct.objとlibrary.libをリンクしてからsub.cをコンパイルしたのでは、正しくビルドできません。これは静かなビルドブレークを引き起こすシナリオのひとつです。

このほかのありがちなビルドブレークのシナリオとしては、構成管理の不備に起因するものがあります。例えば、外部から購入したライブラリ製品のソースコードには、アクセスできないことがあります。図4 - 2でいえば、library.hとlibrary.libは手に入るが、library.cとsub.cは手元にないような状況です。このようなとき、ライブラリのバージョンを上げるためにlibrary.libを更新したのに、誤ってlibrary.hを古いままにしてしまうと、やはり静かなビルドブレークが起きます。

もちろん、インクリメンタルビルドだけでなく、フルビルドのときにもファイル間の依存関係を正しく処理しなければなりません。そこで、これをビルドスクリプトに記述しておき、ビルドのたびにビルドツールに渡します。ビルドツールは、ビルドスクリプトを解釈して、適切なビルドを自動的に行います。

# Make

Makeは昔からある伝統的なビルドツールで、さまざまな言語で記述されたプログラムをビルドするのに使われます。Makeを実行すると、カレントディレクトリにあるメイクファイル(Makeのビルドスクリプト; Makefileという名前のファイル)を自動的にロードし、ビルドが開始されます。図4 - 2に示したファイルの依存関係を、メイクファイルに記述したものを見てみましょう。

表4 - 2 単純なMakefileの例

|  |
| --- |
| # Makefileのサンプル  # ターゲット: 材料ファイル1 材料ファイル2 …  # ターゲットのレシピ  all: product.exe  product.exe: product.obj library.lib  link.exe product.obj library.lib /out:product.exe  product.obj: product.c library.h  cl.exe /c product.c  library.lib: library.obj sub.obj  lib.exe library.obj sub.obj /out:library.lib  library.obj: library.c library.h  cl.exe /c library.c  sub.obj: sub.c library.h  cl.exe /c sub.c  clean:  del \*.obj \*.lib  del product.exe |

このMakefileは、例えば次のように実行します。nmake.exeは、MicrosoftのMakeツールです。

c:\project\_root> nmake all

Makeは、現在のディレクトリにあるMakefileという名前のファイルをロードし、コマンドライン引数で指定されたターゲットをビルドします[[10]](#footnote-10)。Makefileの中には複数のターゲットが含まれ、各ターゲットの中には複数行のレシピが含まれます。

#で始まる行は、コメント行です。本体の最初の行には、生成したいターゲットの名前と、その依存先となる材料ファイル(インプットファイル)をコロンで区切って並べます。次の行には、インプットファイルからターゲットを生成するためのレシピ(外部ツールの呼び出し)を記述します[[11]](#footnote-11)。この組み合わせを、ターゲットの数だけ繰り返して記述します。cl.exeはMicrosoftのC/C++コンパイラ、lib.exeは同じくライブラリアン、link.exeはリンカです。

# ファイルのタイムスタンプとビルド

Makeは、前回のビルドから変更されたファイルがどれとどれなのか、どうやって知るのでしょうか。これには、各ファイルのタイムスタンプを使います。タイムスタンプとは、そのファイルの更新日時のことです。みなさんがWindowsをお使いなら、スタートメニューのアクセサリの中にあるコマンドプロンプトを開いてdirコマンドを起動してみましょう。各ファイルのタイムスタンプが表示されます。あるいは、エクスプローラで任意のファイルを選択し、Alt+Enterキーを押してみましょう。ファイルのプロパティが表示され、更新日時の項目にタイムスタンプが表示されます。

★★★★

図4 - 3 ファイルのプロパティの例

つまり、Makeはターゲットファイルのタイムスタンプと、インプットファイルのタイムスタンプを比べます。ターゲットファイルが存在しなければ、指定された外部ツールを起動してターゲットファイルを生成します。ターゲットファイルが存在したとき、これがインプットファイルよりも古ければ、それは前回のビルドの後にインプットファイルが更新されたと考えられるので、同様にターゲットファイルを生成し直します。



図4 - 4 タイムスタンプを使って  
インクリメンタルビルドする

話題は前後しますが、SCMリポジトリからどのリビジョンのファイルを取り出しても、そのタイムスタンプには (それをコミットした過去の日時でなく) 現在の日時が自動的に設定されます[[12]](#footnote-12)。これは、取得した(過去の)ファイルをビルド対象とし、正しくインクリメンタルビルドができるようにするためです。リポジトリから取得したファイルのタイムスタンプが過去の日時になっていると、正しくインクリメンタルビルドできません。



図4 - 5 リポジトリから取得したファイルのタイムスタンプ

このほか、ファイルのタイムスタンプを現在の日時に変更する操作をタッチといいます。タッチとは、文字通りファイルに「触る」ということです。Unixのtouchコマンドや、AntのTouchタスクを使うと、ファイルの中身を変更することなしにファイルに「触る」ことができます。これにより、次回のインクリメンタルビルドではそのファイルに依存する中間ファイルを作り直す必要があることをビルドツールに指示できます。

# Makeの問題点

ファイルの依存関係を簡単に記述できるMakeはとても便利です。しかし、数多くのソースファイルから構成されるソフトウェアをビルドするとき、Makeのメイクファイルはすぐにごちゃごちゃと読みにくくなってしまいます。さらに、メイクファイルの可読性を下げる要因として次のようなものがあります。

* メイクファイルの方言  
  Makeツールには、GNU make, BSD make, Microsoft nmake, Borland bmake など、非常に多くの種類があります。これらは、ファイル間の依存関係をシンプルに記述するための固有な文法を含むさまざまな方言をもつため、メイクファイルの互換性はほとんどありません。また、このような文法が、逆にメイクファイルの可読性を下げ、ツールごとにメイクファイルの読み書きに習熟する必要を生じさせています。
* メイクファイルは複雑な処理が苦手  
  Makeは、より複雑な条件に従ってビルドの手順を変えるといったことが苦手です。このため、大規模なソフトウェア開発では、MakeとPerlスクリプトなどを組み合わせてビルドシステムを作ることもよくあります。これも、メイクファイルの可読性や保守性を下げる一因となります。
* 外部ツールへの依存  
  Makeを使ってファイルをコピーしたいとき、Unix環境ではcpコマンドを、Windows環境ではcopyコマンドを呼び出さねばなりません。このため、マルチプラットフォームに対応するソフトウェアをビルドするときは、必要以上に複雑なメイクファイルを書くか、プラットフォームごとに別のメイクファイルを用意しなければなりません。

このような問題を解決すべく、C/C++プログラムの依存関係を調べてメイクファイルを自動生成するツールなどもいくつか開発されています[[13]](#footnote-13)。大規模なソフトウェアをC/C++で開発するときには便利ですが、それらのツールを使いこなすにもある程度の習熟が必要なのは変わりません。

# Antの登場

前節で紹介した問題にかかわらず、MakeはC/C++のプログラムはもちろん、それ以外のソフトウェアのビルドにも多く使われます。ファイル間の依存関係を簡単に記述できるMakeは、コマンドライン (シェル) から利用できる外部ツールさえあれば、なんでもかんでもビルドできるからです[[14]](#footnote-14)。当時は、JavaのプログラムもMakeでビルドしていたものです。しかし、Javaには方言がありません。Java仮想マシンさえあれば、どんなプラットフォーム上でも動作するJavaプログラムをビルドするのに、プラットフォームごとに違うビルドスクリプトを使わねばならないのは不合理です。

そこで、オープンなWebアプリケーションサーバTomcatを開発していたジェームズ・デビッドソン氏が、これをビルドするための専用のビルドツールをJavaで開発しました。それがAntです。英語でAntといえばアリンコという意味ですが、これをAnother Neat Tool (手際の良い新しい道具) の略とする人もいます。折しも、XMLというファイル形式が世間を席巻し始めた頃でした。Antは、XMLとJavaのリフレクションという機能を組み合わせた、非常に洗練されたビルドツールになりました。Tomcatと一緒に配布されたAntは、Makeの問題点を解決する優れたビルドツールであることが世間に認知され、Tomcatから分離されて、バージョン1.1としてリリースされました。2000年7月のことです[[15]](#footnote-15)。

AntはJava仮想マシン上で実行されるので、どのプラットフォーム上でも同じビルドスクリプトを同じように実行できます。例えば、ファイルのコピーをするときは、外部ツールを呼び出す代わりにCopyタスク(<copy>タグ)を使います。★Copyタスク以外にも、さまざまな作業をこなすための標準タスクが多く用意されています。これらのタスクはJavaで実装されているため、どのプラットフォーム上でも同じように動作できるのです。このため、ビルドスクリプトから外部ツールを呼び出す必要は少なくなっていますが、Execタスク (<exec> タグ) を使えば、外部ツールを呼び出すこともできます[[16]](#footnote-16)。

また、メイクファイルからJavaコンパイラを呼び出すと、そのたびにJava仮想マシンを起動しますが、Antでは何度Javaコンパイラを呼び出しても同じJava仮想マシンを使いまわすので、CPUリソースの消費が少なくて済むというメリットもあります。

さらに、みなさんがJavaを使って独自のタグを作成してAntの機能を拡張し、自分のビルド環境に専用のタグを準備することもできます。AntはJavaに特別なビルドツールですが、その扱いやすさから、Java以外の言語で記述されたコードをビルドする時にもAntを使う人が増えています。Antの技アリでMakeのマケ、というわけです。

といっても、ファイルの依存関係を簡単に記述できるMakeは、ソフトウェアのドメインによっては今も広く使われています。このほか、RubyのプログラムをビルドするRakeや、Microsoft .NET 環境に特化されたMSBuildやNAntなどのビルドツールがあります。みなさんの開発環境に適切なビルドツールを選択してください。次節以降では、Antの使用を前提として説明をすすめます。

# ターゲット間の依存性を管理する

Antのビルドスクリプト(build.xml)の基本的な構造を見てみましょう。

表4 - 3 単純なbuild.xmlの例[[17]](#footnote-17)

|  |
| --- |
| <project name="MyProject" default="dist" basedir=".">  <target name="init" description="ビルドプロセスを初期化する">  <tstamp/>  <mkdir dir="build"/>  </target>  <target name="compile" depends="init" description="コンパイルする">  <javac srcdir="src" destdir="build"/>  </target>  <target name="dist" depends="compile" description="アーカイブする">  <mkdir dir="dist/lib"/>  <jar jarfile="dist/lib/MyProject-${DSTAMP}.jar" basedir="build"/>  </target>  <target name="clean" description="クリーンする">  <delete dir="build"/>  <delete dir="dist"/>  </target>  </project> |

このbuild.xmlは、例えば次のように実行します。

c:\project\_root> ant dist

Antは、現在のディレクトリにあるbuild.xmlという名前のファイルをロードし、コマンドライン引数で指定されたターゲットをビルドします[[18]](#footnote-18)。build.xmlの中には複数のターゲットが含まれ、各ターゲットの中には複数のタスクが含まれます。

これを見てすぐに気づくのは、build.xmlにはファイル間の依存性が記述されていないことです。実は、build.xmlの枠組みの中には、ファイル間の依存関係を直接記述する仕組みは用意されていません。Antは、ファイル間の依存関係は各タスクが解決するというデザインがされています。

例えば、Javaコンパイラを呼び出すJavaCタスクは、指定されたディレクトリに含まれる.javaソースファイルの依存関係を調べて、これを自動的に適切な順序でコンパイルします。また、Jarアーカイバを呼び出すJarタスクは、変更されたファイルだけを.jarの中で差し替えます。このほか、依存関係にあるファイルのタイムスタンプを調べて、古くなったターゲットファイルを削除するDependSetタスクなども用意されていますが、やはりファイル間の依存関係はタスクの中に閉じ込められています。

では、build.xmlには何を記述するのかといえば、ターゲット間の依存関係を記述します。各ターゲットのdepends属性に注目してください。depends～とは、～に依存するという意味です。この例ではdistターゲットはcompileターゲットに依存しますから、distターゲットを実行する前には、自動的にcompileターゲットが実行されます。同様に、compileターゲットを実行する前には、initターゲットが実行されます。Antは、どのターゲットが実行済みなのかを追跡するので、一回のビルド中に同じターゲットが複数回実行されることはありません。

<target>タグのdepends属性の値には、カンマで区切って複数のターゲットを指定できます。そこで、ビルドに必要な作業を適切な単位で複数のターゲットに分割し、それらに適切な依存関係を与えて、build.xmlを記述します。このbuild.xmlをAntで実行するときに、最終的に実行したいターゲット名をコマンド引数で指定します。

このほか、ターゲットのif属性やunless属性でその実行の可否を制御したり、extensionOf属性で依存関係にあるターゲット間に別のターゲットを差し込んだりすることもできます。

# Antの標準タスク

Antに標準で装備されている代表的なタスクを表4 - 5に列挙しておきます。例えば、Dependタスクは.classファイル間の依存を調べて、古くなった無効な.classファイルを削除します(.javaファイルがなくても処理できます)。また、Mailタスクはメールを送信します。このほかにも、多くの標準タスクが用意されています。

このほか、タスクの作業対象としたいデータ集合を指定するためのデータ型も用意されています。例えば、条件を処理するConditionや、正規表現[[19]](#footnote-19)を処理するRegexp、Unixのパイプのように連結してファイルを処理できるさまざまなフィルタなどがあります。また、DirSetやFileSetでは指定したパターンに合致する名前のディレクトリやファイルを選択できます。さらに、FileSetではさまざまなセレクタを利用できます。<contains>セレクタは指定した文字列を含むテキストファイルを、<depend>セレクタは対応するファイルよりも新しいファイルを選択します。<modified>セレクタは前回のビルド時から実際に中身が変更されたファイルを選びだします[[20]](#footnote-20)。これらで指定したデータは、パラメータとして各タスクに渡します。

このほか、スクリプト言語を直接build.xmlの中に記述する仕組みも用意されています。詳しくはAntのマニュアルを参照してください。

表4 - 5 Antの代表的な標準タスク一覧



# Antの独自タスク

標準タスクではこなせない仕事を、Antにさせたいこともあります。例えば、SubversionやMercurialのような比較的新しいSCMツールを操作するタスクは、標準では用意されていません。いくつか選択肢があります。

1. 標準のExecタスクを使う  
   いちばん簡単な方法です。例えば、Subversionのコマンドsvn.exeを<exec>タグで呼び出します。
2. 独自タスクを探す  
   一般的と思われる作業については、誰かがすでにAntの独自タスクを書いているはずですから、それを探してインストールします★。例えば、Ant のビルドスクリプトでSubversionを使いたいなら、SvnAntが使えます。また、MercurialならANT4HG[[21]](#footnote-21)が使えます。
3. 独自タスクを書く

外部ツールを呼び出す代わりに、独自のタスクタグをJavaで実装します。あなたのプロジェクトをビルドするための固有な作業については、有力な選択肢です。例えば、私はJavaでないプログラムをビルドするとき、ビルド番号が記述されたリソースファイルを更新するための独自タスクを書く機会が何度かありました[[22]](#footnote-22)。

独自タスクを書けば、外部ツールに依存しない、マルチプラットフォームに対応したbuild.xmlを書けるようになります(ただし、当該のbuild.xmlはその独自タスクに依存するようになるので、少し微妙ですけど)。また、より親切なエラーメッセージを出力したり、Antのほかのタスクのログ出力と、あなたの独自タスクのログ出力をきれいに統合したりできます。

さらに、前節で紹介したデータ型を、独自タスクのパラメータで受け取るように実装し、操作対象としたいデータをbuild.xmlで柔軟に指定できるようにもできます(ただし、操作対象としたいファイルをファイルセットでタスクに指定しても、それらを正しい順序で処理するのは各タスクの責任です)。

そこで、Antのファイルセットやログのメリットを享受するために独自タスクを作って、その実装は外部ツールを呼び出すようにするのもいいでしょう。ファイルセットに含まれるファイル名をAnt APIで展開し、適切なコマンドラインパラメータにして外部ツールに渡します。例えば、ANT4HGタスクの実装は外部のhgコマンドを呼び出すようになっています。

Antの独自タスクを書くのは非常に簡単なので、ぜひ一度挑戦してみてください。Javaで、org.apache.tools.ant.Taskクラスを継承してexecuteメソッドをオーバーライドするだけです。タグの属性値は、対応するsetterメソッドを書けば取得できます。このほか、独自フィルタや独自セレクタを書くこともできます。

# コラム ソフトウェア開発のメタファ④自動車工場

|  |
| --- |
| **コラム ソフトウェア開発のメタファ④ 自動車工場**  ソフトウェアの開発プロセスの改善に取り組んでいる人の中には、ときどきソフトウェア開発について誤解している方がおられるようです。きちんとプロセスを定義し、このプロセスに則って厳格かつ機械的にプロジェクトを運用すれば、ソフトウェア開発プロジェクトに失敗するはずがない、というものです。このような誤解の元には、高度に自動化された自動車工場のように、工業製品としてソフトウェアも生産できるはずだという期待を見ることができます。残念ながら、これはほとんど不可能です。  車を生産する前には、研究開発をして剛性や形の設計、試作品の製作といった段階がありますが、これには属人的なスキルやノウハウがどうしても必要です。それが完了してはじめて、工場での大量生産が可能になります。  ソフトウェアも同じです。特に、プログラミングという作業そのものが自動車の設計作業にあたります。ですから、ソフトウェア開発におけるプログラマは自動車開発における設計士に相当します。その仕事はツールなどで支援することはできても、機械的に作業をすすめて均質的な製品を製造したり、まして完全に自動化したりすることはできません。  ひと昔前はコンピュータ資源が不十分だったため、頻繁にコンパイルとビルドをしながらソフトウェアを開発することが難しいことがありました。そのため、上級エンジニアが紙と鉛筆でフローチャート図を書いて処理の流れを検討し、コーダーといわれる下級エンジニアがそのフローチャート図をそのままコンピュータ言語に翻訳する、という手順で開発が行われることもあったのです。このため、単純作業をするコーダー (=プログラマ) の地位は低いものと考えられてきました。  現在、実行可能な処理を詳細にフローチャート図で定義し、その後でコーディングするという手順でソフトウェアを開発することはほとんどありません。それは、現在のプログラミング言語が、フローチャートよりもはるかに高い表現能力をもっており、また現在のコンピュータ資源が潤沢だからです。上級エンジニア自身が直接プログラムを書き、頻繁にコンパイルとビルドを繰り返しながらソフトウェアをデザインすることができます。オブジェクト指向言語を活用し、高度なスレッドやネットワークなどの技術を駆使してソフトウェアをデザインするプログラマは、まさしく職人です。  では、自動車工場で自動化できる作業は、ソフトウェア開発のどの作業に対応するのかといえば、それはビルド (組み立て) 作業です。パッケージ製品であれば、CDやDVDをプレスして箱詰めする作業も含まれるでしょうが、それは本書の範囲を超えます。  つまり、ソフトウェア開発におけるビルド作業は、自動車工場の流れ作業に対応すると捉えることができます。ビルドの最適化は、自動車組み立て工場の最適化と同じく、ソフトウェア開発において非常に重要な位置を占めるのです。  ウォーカー・ロイス氏は、著書「反復型開発のエコノミクス」で「統合およびテストを実施する組織の大半が、統合のために60%の時間を費やしている」と述べています。つまり、統合ビルドとテストの作業を自動化できれば、ソフトウェア開発のコストを大きく削減できるのです。ビルドの自動化が、とても重要なタスクであることがわかります。  C:\idd\doc\flowchart.png  図3 - 1 フローチャート図の凡例 |

# コラム 開発中の催し④ ワンオンワンミーティング

|  |
| --- |
| **コラム 開発中の催し④ ワンオンワンミーティング**  ワンオンワンミーティング (one-on-one meeting) とは、1対1で行うミーティングのことです。メンターとメンティー同士で、週次で5分から30分程度の時間を使って行うのが一般的です。メンター (mentor) とは指導者や庇護者、師匠という意味で、メンティー (mentee) とはその指導を受ける人です。この語源は、トロイ戦争に赴くオデッセウスが、息子テレマカスの教育を任せた教師メントールの名前に由来します。組織上のレポートラインとは別に、メンターとメンティーの組み合わせを決めておけばOJT (On the Job Training) がしやすく、また定期的にワンオンワンを実施することで、いわゆる社内メンター制度も文化として定着しやすくなるでしょう。  開発チームの規模が大きくなると、どうしても組織のすみずみまで目が行き届かなくなります。そこで、上司がレポートライン下の部下全員と毎週もしくは隔週といった頻度でワンオンワンを実施する組織もあります。上司は部下に期待することを伝え、作業の障害となっていることがあれば吸い上げます。組織が大きすぎて全員とワンオンワンを実施するのが難しいときは、ワンオンワンの実施頻度をひと月に一度やマイルストーンごとに一度などに減らします。  また、上司は、そのまた上司とワンオンワンを計画することもあります。これにより、比較的大きな組織でも、組織全体をうまく方向づけることができます。 |

# ビルド手順の概要

ここまで、駆け足でAntを紹介しました。ビルドをするための道具が揃ったところで、いよいよ実際のビルド手順を説明しましょう。素朴なビルドとは一次ファイルを変換・統合して実行可能ファイルを生成するプロセスですが、実際のビルドにはより多くの作業が含まれます。図4 - 8にビルドマシン上で統合ビルドをするときの手順の一例を示します。ウィークリービルドを行っている開発プロジェクトでは、これはビルドマスターが週に一度行う定時の作業となります。もちろん、ソフトウェアのドメインによってビルド手順の詳細はかなり異なります。ここに示したのは、多くのソフトウェアのビルドに共通したものです。



図4 - 8 週次の統合ビルドの手順 (例)

この例では、ビルドの手順を16個のターゲットに分割しましたが、みなさんが作るソフトウェアに合わせて、より適切な粒度でターゲットを分割してください。

また、ここでは各ターゲットを実行順に単純に並べましたが、これらをAntのビルドスクリプトには、各ターゲット間の依存関係を記述します。例えば、★「オートメーションを実行する」ターゲットは「オートメーションコードをビルドする」ターゲットに依存します。

また、ビルドの種類によって依存するターゲットも異なります。例えば、インクリメンタルビルドをするときは、クリーンの操作をするターゲットが実行されないようにします。また、サンドボックス上でプライベートビルドをするときは「実行可能なソフトウェアをビルド」するターゲットだけを実行すればいいかもしれません。あるいは、短時間で終わるスモークテスト用のオートメーションを実行するターゲットを別途用意すべきかもしれません。

また、図4 - 8に例示したビルド手順は、手作業で実施するターゲットを含むため、完全には自動化できません。この中で自動化できる部分だけをビルドスクリプトに記述することになります。運用次第では、手作業で実施するターゲットをすべて取り除き、ビルドプロセスを完全に自動化することもできるでしょう。

# ビルド手順の詳細

2章の図2-11を再掲し、これと対応づけながら、図4 - 8に紹介したターゲットを順に説明します。表4 - 5に示したAntの代表的なタスクも参照しながら読み進めてください。



図2 - 11 ビルド中のやりとり (再掲)

## 開発チームにビルド開始を通告

ウィークリービルドを行っているソフトウェア開発プロジェクトでは、週に1度のビルド曜日がやって来ると、ビルドマスターの出番となります。ビルドを開始するときは、ビルドマスターはその旨を開発チームに通知します。このビルドで完了する予定の作業項目を完了できていない開発者は、ビルドマスターを拝み倒してビルドの開始を少しだけ待ってもらうことができます。ビルドマスターを拝んだ開発者は、急いでソースコードの修正を完了し、これを構成管理ツールにコミットしなければいけません[[23]](#footnote-23)。それが済めば、ビルドマスターはビルド作業を開始します。

## リポジトリから最新のソースコードを取得

リポジトリのタグを付けた時点からソースファイルを取得し、ビルドマシンの環境を更新します。表4 - 5に示したように、CVSやPerforceなどのSCMツールにはAntに標準のタスクが用意されているので、Antからこれらを操作するのは簡単です。リポジトリにCVSを使っていればCvsタスクを使って、Perforceを使っていればP4Syncタスク[[24]](#footnote-24)を使って、リポジトリから最新のソースコードを取得できます。それ以外のSCMツールをお使いなら、ExecタスクでSCMコマンドを呼び出すので十分でしょう。詳しくは★4-12節を参照してください。

## 週次のビルド番号を増やす

ビルド番号が記述されたリソースファイルを更新して、ビルド番号をひとつ増やします。週次のビルド番号を増やす作業なら手作業で行っても十分ですが、もちろん自動化しても便利です。

Javaのリソースファイルでビルド番号を管理するなら、標準タスクBuildNumberが使えます。★興味がある読者は、Manifestタスクとjava.lang.PackageクラスのgetImplementationVersion() メソッドを調べてみましょう。

ただし、計画通りのビルド番号をビルドに付加したいなら、ビルド番号を更新するターゲットは他のターゲットと分離して実行できるようにしておくと良いでしょう。ビルドマシン上でビルドに失敗するたびにビルド番号が増えてしまうのでは、困ってしまいます。

より簡便にビルドに番号をつけるには、作ったビルドのファイル名にビルド番号をくっつけておくとか、作ったビルドを保存するディレクトリ名にビルド番号をくっつけておくことができます。しかし、最終的にユーザーにリリースするビルドには、その中にビルド番号を直接埋め込んでおくべきです。どうせやるなら、最初からやっておいた方が楽ちんです。

## 前回のビルドで生成されたファイルを削除

フルビルドをするために、前回のビルドで生成されたファイルの削除(クリーンの操作) をします。これには、Deleteタスクを使います。実行可能なソフトウェアをビルドするターゲットを、このターゲットに依存させることで、フルビルドを行わせることができます。

## 実行可能なソフトウェアをビルド

Javaのソースコードをコンパイルするなら、JavaCタスクを使ってJavaコンパイラを起動します。簡単ですね。それ以外の開発言語をお使いなら、Execタスクでコマンドラインコンパイラを呼び出すか、コマンドラインコンパイラを呼び出す独自のAntタスクを書いてビルド環境を構築します。もしこのターゲットを実行中にビルドブレークしたら、ビルドマスターは開発者と協力してこれをASAPで解消します[[25]](#footnote-25)。

## テスト用のコードをビルド

プログラミング言語に応じて、さまざまな自動テストツールがあります。単体テストのツールでは、xUnitが有名で、広く使われています。このほか、統合テストを自動化する商用のテストツールも多くあります。

これらの自動テストツールは、テストスクリプト (テストコード) を準備し、テストツールから実行するといった形で利用するものがほとんどです。JavaプログラムをテストするテストコードはJavaで、CプログラムをテストするテストコードはCで記述されるのがふつうです。これらのコードが記述されていれば、ここで一緒にビルドします。ただし、テスト用のコードはリリース用の配布パッケージに含めないようにしてください。

## 自動化されたテストの実施

ビルドしたテストコードを、ここで実行します。Java用の単体テストツールであるJUnitをお使いなら、JUnit / JUnitReportタスクを使ってテストコードを実行できます[[26]](#footnote-26)。このターゲットの実行時間も長くなりやすいですから、それだけテストチームにリリースする予定時刻よりもビルド開始時刻を早く設定しておく必要があります。テストの結果があまりにも芳しくなければ、このビルドはテストチームにリリースする候補にはなりません。修正すべきところを修正して、リビルドします。継続的インテグレーションを活用しているなら、ひとつ前のビルドをテストチームにリリースできるかもしれません。

## ソースコードからドキュメントを生成

ソースファイル中に直接記述されたコメントから、ドキュメントを生成するツールがあります。このようなツールを使って、ビルド時に最新のドキュメントを生成します。プログラミング言語によってコメントの形式が違うため、それぞれ利用可能なツールも異なります。

JavaにはJavadocという標準のドキュメント生成ツールがJDKに含まれています。Java APIのドキュメントも、Javadocで生成されています。みなさんがJavaをお使いなら、ぜひJavadocタスクを活用してください。Java以外の開発言語にも、Sandcastle, RDoc, Doxygenなど、多くのドキュメント生成ツールがあります。独自のIDEを持つ高機能なものもありますが、そのほとんどがコマンドラインから利用できるツールを含んでいます。Execタスクか、もしくはAntの独自タスクを書いてこれを呼び出してください。

コメントは実行可能なファイル (プログラムファイル) 中に直接記述されるドキュメントであるため、次のようなメリットを期待できます。

* 外部のドキュメントより正しいことを期待できる[[27]](#footnote-27)  
  コメントは、プログラム本体を変更したときに同時に修正する機会を作りやすいため、外部のMicrosoft Wordなどで書かれたドキュメントよりも、最新の状態を反映した正しい記述になっていることが期待できます。
* 開発者にコメントの記述を促すのが容易

必要な部分にコメントが記述されていないとき、ドキュメント生成ツールに警告メッセージを出力させることができ、開発者にコメントの記述を求めることができます。

* 書式に制約をつけることが容易

ドキュメント生成ツールが、正しくドキュメントを生成するための書式ルールに従ってコメントを書く必要があり、この制約はより正しいドキュメントを漏れなく残す手段として非常に有力です。

プログラム内部の詳細な仕様は、このようなツールを使って保守すると便利です。

## 配布パッケージを作成

クラスファイルをアーカイブしてjarファイルを生成したり、インストーラを作成することで、配布パッケージを作成します。この配布パッケージをインストールイメージということもあります。また、この作業を指してパッケージングとかキッティングといいます。開発言語がJavaであれば、JarタスクもしくはZipタスクなどを使います。あるいは、外部のインストーラ生成ツールを呼び出す必要があるかもしれません。このターゲットも、ソフトウェアのドメインによって具体的な処理が大きく異なります。

## 開発チームのファイルサーバに候補ビルドをコピー

キッティングした候補ビルドをしかるべき場所 (ファイルサーバなど) にコピーします。もちろん、テストチームに見える場所ではいけません。テストチームにリリース前のビルドが、テストチームに漏れてしまうことになるからです。このターゲットを自動化するならCopyタスクやFTPタスクが役に立ちます。また、Webアプリケーションであれば、自動でWebサーバ上にデプロイ[[28]](#footnote-28)するターゲットを用意すると便利です。

## 開発チームに候補ビルドのリリースをアナウンス

開発チームに候補ビルドができたことをアナウンスします。これを自動化するなら、MailタスクやSoundタスクです。景気よくパンパカパーンといきましょう。

## 開発チーム内でスモークテスト

各開発者は、候補ビルドのリリースを受けて、これを自分の環境にインストールし、自分が新しく実装した部分が適切に実装されているか、修正したバグが正しく修正されているかどうかを確認します。もちろん、テストチームによるテスト作業をブロックするような問題が見つかれば、これを修正して候補ビルドをリビルドしなければなりません。

このように、ビルドをテストチームにリリースする前に、テストチームによるテストが実施できる品質に達しているかを判定するテストをBVT (Build Verification Test) といいます。ビルド受け入れテスト (Build Acceptance Test) ということもあります。これには、先に実施したオートメーションも含まれます。基本的なテストシナリオについて十分なカバレッジをもつオートメーションを用意できれば、手作業によるBVTは省略して構いません(むしろ、そうすべきです)。BVTは、テストチームの作業時間を無駄にしないために重要ですが、短時間で行えるように工夫する必要があります。

## 増やした週次のビルド番号をコミット

ビルドの成功を確認したら、増やしたビルド番号が記述されているリソースファイルをリポジトリにコミットします。先述の通り、BuildNumberタスクを使ってビルド番号を管理しているなら、build.numberという名前のファイルをリポジトリにコミットすることになります。SCMにCVSを使っているならCvsタスク、PerforceならP4Submitタスクを使って自動化できます。

## ビルドのインプットとなったソースにタグ付け

ビルド番号のコミットに続けて、今回のビルドのインプットとなったリポジトリ中のソースファイルにタグを付けます。タグを付けておくことで、このビルドとまったく同じビルドを後日に再現することが容易になります。タグの名前には、<製品名>\_<日付>\_<ビルド番号> のようなものが良いでしょう。このような名前を生成するには、TStampタスクが役に立ちます。CVSならCvsタスク、PerforceならP4Labelタスクを使って自動化できます。

## テストチームのファイルサーバにビルドをコピー

上記の手順により構築した成果物を、しかるべき場所に保存します。開発言語によっては、バグの原因を調査しやすくするために、ビルドの過程で出力された中間ファイル (.mapファイルや.pdbファイルなど) も保存しておくことがあります。

ただし、ビルドの成果物は一般にソースコードのサイズよりも大きくなるので、ビルドの成果物をすべて保存すると、ディスク容量を圧迫することになります。古いビルドは、日がたつと、新しいビルドのリリースに伴って不要になっていきますから、ファイルサーバ上に保存したビルドは古いものから削除していく、あるいは外部メディアに移動するといった管理をすると良いでしょう。古いビルドを保存しておく価値については、★3-14節を参照してください。

## テストチームにビルドのリリースをアナウンス

ビルドマスターは、開発者が記述したリリースノートを取りまとめてビルドに添付し、テストチームにリリースをアナウンスします。テストチームは、そのビルドのテストを開始します。ビルドマスターは、ビルドの作業をどこまで自動化すべきか、よく検討してみてください。

# IDEとビルドツールの分離

みなさんがサンドボックスで開発中のソフトウェアをビルドするときは、IDEについているビルドボタンを押してビルドすることと思います。IDEとはIntegrated Development Environment (統合開発環境) の略で、コンパイラやデバッガ、エディタなどのツールが統合された開発環境のことです。現在、人気のあるIDEにはEclipseやIntelliJ IDEA, Microsoft Visual Studioなどがあります。これらのIDEのほどんどが、外部のビルドツールを呼び出してビルドを開始できる「ビルドボタン」を装備しています。



図4 - 6 Eclipseのビルドボタン  
(Project -> Build Automaticallyのチェックを外すと表示される)

ひと昔前のIDE (Microsoft Visual Studio 6.0やBorland Delphiなど)もビルドボタンが付いていましたが、IDEとビルドツールが分離されていませんでした。固有のビルド機能をIDE内部にもち、独自のビルド構成ダイアログを装備していました。これは確かにビルドを構成するのは比較的簡単でしたが、ビルドマシン上でビルドするときもIDEを起動してビルドボタンを押さなければなりませんでした。この操作は簡単に自動化できないので、とても不便です[[29]](#footnote-29)。そこで、ビルドマシン専用のメイクファイルと、サンドボックス用のIDEのプロジェクトファイル(ビルド構成やIDEの環境設定などを記録したファイル)を別に用意することがふつうでした。しかし、これではビルドマシンとサンドボックスでビルドの手順が異なるために、サンドボックスでは問題なくビルドできるのに、ビルドマシンではビルドブレークする、などの問題が出ることもありました。また、IDEのプロジェクトファイルはバイナリ形式のものもあり、それらはSCMリポジトリで管理しても差分が確認できず、最終的にコンパイラなどの外部ツールに渡されるパラメータが何だかわからないものもありました。

このような問題を解決するために、現在のIDEのほとんどがビルドツールと分離されています。JavaのIDEも、ビルドツールとしてAntを使うものがほとんどです。これにより、ビルドマシン上のビルドシステムとサンドボックス上のIDEで、全く同じビルドスクリプトを使ってビルドできるようになりました。また、違う種類のIDE間でもビルドスクリプトを共有できますから、各開発者が好みのIDEを使うこともできます[[30]](#footnote-30)。Microsoft Visual Studioも、2005からビルドツールがMSBuildという外部コマンドに分離されて使いやすくなりました。また、新しいバージョンのDelphiのIDEも、MSBuildと連携するようになっています。



図4 - 7 ビルドマシンとサンドボックスで  
同じビルドツールとビルドスクリプトを使う

# 誰がビルドマスターを務める？

ビルドマスターとは、ビルドマシン上でのビルド作業に責任をもつ担当者のことで、ビルド担当者、ビルドマネージャ、ビルドエンジニアなどということもあります。ビルドマスターは、多くの場合、開発者が兼務します[[31]](#footnote-31)。

ビルドマスターは、各開発者が作成したソースコードをビルドして統合し、インストール可能な形にキッティングしてテストチームにリリースするまでの責務を負います。このため、ビルドマスターはインストール手順書を作成したり、インストーラが必要なソフトウェアの場合にはインストーラの開発も担当することがあります。また、ビルドマスターは、開発中のソフトウェアをビルドする手順を文書化する責務も負っています。

1章にも紹介しましたが、一般に、ビルドマスターはプロジェクトの新人さんに担当してもらうと良い結果を産むことが多いようです。その理由を列挙します。

## ソフトウェアのビルドプロセスを学習しやすい

当然のことですが、ビルドマスターはビルドプロセスをよく理解していなければ務まりません。ビルドのプロセスは、ソフトウェア開発プロセスの中でも非常に基本的かつ重要なものです。ソフトウェアエンジニアなら、一度はビルドマスターというロールを経験しておくべきでしょう。

## 多くの開発者とコミュニケーションする機会が得られる

ビルドマシンでコンパイルエラーなどが発生してビルドが完了できないとき、そのエラーを解消すべく、適切な開発者を探して一緒にトラブルシュートしなければなりません。その過程で、複数の先輩の開発者と話をしたり、作業のコツを伝授してもらえたりできます。

## ビルド作業そのものは、それほど難しくないことが多い

ビルド手順を確立してビルドを自動化する作業は、上級エンジニアが責任を持つ方がいいでしょう。しかし、毎週のビルド作業は手順を間違えなければ良く、それほど技術的に難しいものではありません。新人さんが十分に責任をもって果たせるものです。もっとも、現在は優れたCIサーバがビルド作業のほとんどを自動化するので、ビルドマスターの責務は軽減されています。

## 「ビルドマスター」の称号が、責任感を育む

学校を卒業してソフトウェアの会社に就職した後、すぐに「あなたをビルドマスターに任命します」と言われたら、かなり誇らしい気持ちになります (私はそうでした)。「ビルドマスター」(ビルドの達人) というジョブタイトルは、プロジェクトチームに一人しかいない、責任ある偉い人の称号だからです。もし、いきなりビルドの達人と呼ばれるのが面映ゆいと新人さんがいうなら、「ビルドエンジニア」に任命してあげましょう (呼称が違うだけで、役割は変わらないですが)。

もちろん、上級エンジニアがビルドマスターを担当しても悪いことは全くありません。ビルドマスターとは、それだけの責任を伴う重要な任務だからです。

# ビルド手順書の記述

ビルドマスターは、自分が不慮の事故 (あるいは、故意の転職) などで突然いなくなっても、ほかの人が作業を引き継ぐことができるように準備しておかなくてはいけません。ビルドマスターが風邪をひいて休んだからビルドができなかったのでは話になりません。さすがに、ビルド手順書を記述する作業を自動化するというわけにはいきませんが、やはりAntによる支援を受けることはできます。

build.xmlにおける各ターゲットにdescription要素を用意し、各ターゲットが自動化する作業についての説明をbuild.xmlに記述しておきます。こうしておくと、ant -projecthelpを実行することで、実行されるターゲットのdescription (説明) を順に表示させることができ、なかなか便利です。build.xmlにエンコーディングを指定しておけば、日本語でもdescriptionを書けます。開発するソフトウェアの規模が小さければ、これをそのままビルド手順書の代用としてもいいかもしれません。参考までに、 ★★表4 - 7に示したbuild.xmlに対して、ant -projecthelpを実行した結果を表4 - 10に示します。

表4 - 10 ant –projecthelpの実行結果の例

|  |
| --- |
| C:\Project\Mainline\src>ant -projecthelp  Buildfile: C:\Project\Mainline\src\build.xml  Main targets:  clean クリーンする  compile コンパイルする  dist アーカイブする  init ビルドプロセスを初期化する  Default target: dist |

ただし、ビルドするときに、Antのコマンドライン引数で指定すべきターゲットがどれだか分からない、なんてことでは困ります。また、ビルド手順書はビルド環境の構築手順 (ビルドに必要な各ツールのインストール手順など) に加えて、各開発者のマシンにおける開発環境の構築手順についても説明するべきです。Antスクリプトのdescriptionの記述に加えて、別にきちんとしたビルド手順書 (ビルド仕様書) を用意することをお勧めしておきます。

ビルド環境は、ビルドのリリースを繰り返すたびに少しずつ洗練され、自動化された部分やビルドすべきモジュールなども増えていくはずです。それにつれて、ビルドの手順も変わってきます。ビルドの手順書を常に最新の状態にメンテナンスする作業も、ビルドマスターの責務です。

# リリース列車を定刻通りに発車させる

ビルドを定期的かつ定刻にリリースすることを、列車の発着時刻になぞらえてリリーストレイン(Release Train; リリース列車)といいます[[32]](#footnote-32)。複数のブランチから、それぞれ列車が定刻に発車するようなイメージです。これらの列車は同時刻に発車させるべきものもあれば、そうでないものもあるでしょう。リリーストレインという語には、複数のブランチでのビルド作業の同期を取り、その成果物を同時に (同じ貨物列車に載せて) リリースするという意味があります。また、複数の新機能を別の列車に載せて、少しずつデリバリするという意味もあります。このとき、いつも同じ数の新機能を貨物列車に載せる必要はなく、計画通りの日時に列車を発車させることが重要です。



図4 - 9 リリーストレインを同期させる

この喩えは、ビルド開始と終了の時刻をどのように扱うべきかを良く示唆しています。例えば、ある開発者のコミットが少々間に合わなくて、ビルド開始を待ってもらうというのは、列車の発車を少し待ってもらって飛び乗るようなものです。駆け込み乗車は、大変危険ですのでお止めください。コミットしなければ、後続 (テストなど) の作業をブロックしてしまうものでないなら、その修正は次の列車を待ってコミットすべきです。チームで共有した時刻表を厳守しましょう。また、複数のブランチでのビルドで同期を取りたいなら、それぞれの (ビルド開始時刻ではなく) ビルド終了時刻を考慮して同期させることが必要です。

テストチームはウィークリートレインに乗るべきか、バイウィークリートレインに乗るべきか、列車に乗り込む頻度をよく検討しましょう。品質の良いテスト結果を得るには、なるべく頻繁に列車を乗り換えるべきですが、頻繁すぎるとテストし切れませんし、乗り換えの手間や待ち時間などが発生するので、テストに使える時間も少なくなります。リリース列車を上手に連結して、遅延も事故も起こさず、最終的な目的地まで定刻通りに走らせましょう！

# 適切なビルドの頻度を探す

本書では、ビルドの頻度として1週間を勧めていますが、それはさまざまな状況で使いやすく、リズムを作りやすい単位だからです。例えば、バグの修正に必要な時間を1日単位で見積もるのは難しいですが、1週間の単位でなら、正しい見積もりをしやすいでしょう。

しかし、ソフトウェアの開発時期や規模によっては、適切なリリースの間隔が1週間ではないことがあります。定期的なビルドは、テストチームにリリースされてテストされますから、ビルドの頻度はテストチームの要求に合うように設計すべきです。例えば、1時間おきにリリースしても、テストチームはそれをテストし切れません。前のビルドをアンインストールし、新しいビルドをインストールするだけで1時間を使ってしまうでしょう。どのみち、テスターにとって意味のある機能追加やバグ修正は、それほどの頻度でビルドに入ってはきませんから、必要以上に頻繁にビルドをテストチームにリリースしても仕方ありません。継続的インテグレーションは開発者のために実施するものであり、テストチームにリリースするためのものではありません。

しかし、できるだけ新しいビルドでテストをした方が、より有益な情報を開発者にフィードバックできます。例えば、一か月おきにテストチームにリリースしたとしましょう。せっかくテスターがバグを発見して報告票を起票しても、次のビルドではUIが変更になったりするなどで、報告票に価値がなくなっているかもしれません。そんなことが頻繁に起これば、テスターのやる気を削ぐことになかねません。これは、リリースの頻度が少なすぎます。この観点からは、ビルドはできる限り頻繁にテストチームにリリースした方がよいのです。

ここにトレードオフがあります。テストチームにリリースするビルドの適切な頻度を探しましょう。テストチームが最新のビルドを必要とするときに、それを引き取れるような頻度でビルドをリリースするのが理想的です。

# 継続的インテグレーション

継続的インテグレーション (CI; Continuous Integration) とは、継続的にインテグレーションビルド (統合ビルド) をすべし、というXP[[33]](#footnote-33)のプラクティスです。マーチン・ファウラー氏の「Continuous Integration」という記事が有名ですが、この記事のオリジナル版[[34]](#footnote-34)には「このプラクティスは、XPを仕事に使おうとは考えない大勢の人たちにより、以前から使われてきた」とあります。実際に、同様のプラクティスを説明する文章は以前からあり、その中でもスティーブ・マコネル氏[[35]](#footnote-35)の1996年の記事「Daily Build and Smoke Test」が有名です。また、グラディ・ブーチ氏は、1995年の著書「オブジェクトソリューション」で「定期的な ’継続的インテグレーション’ のプロセスが、リリースのたびに機能を成長させ、実行可能なリリースを生み出す」と書いています[[36]](#footnote-36)。

XPの継続的インテグレーションは、これらの考え方をより発展させたものです。日次よりもずっと頻繁にビルドをすることで、より早く開発者にコミットに対するフィードバックを返します。



図4 - 10 CIによるフィードバックのサイクル

# 継続的インテグレーションとビルド時間

もし、あなたのソフトウェアをビルドするのに5分しかかからないなら、5分おきに自動ビルドすることで、文字通り「継続的」な統合ビルドを実現できます[[37]](#footnote-37)。そうすれば、もし開発者が何か変なものをコミットしても、遅くても10分後にはビルドシステムがビルドエラーやテスト失敗を検知し、メールもしくはソフトウェアあんどんなどで開発者に報告するでしょう。

しかし、開発が進み、ソフトウェアの規模や自動化されたテストの数が大きくなると、ビルド時間も増えていきます。もしビルドに1時間かかるようになったらどうなるでしょうか。2つの問題があります。

## ビルド開始のタイミング

1時間おきに、決まった時刻にビルドを始めるのはうまくいきません。ビルド開始直後に開発者が何かをコミットしたら、そのフィードバックが得られるのは (1時間後ではなく) 2時間後になってしまいます[[38]](#footnote-38)。継続的なインテグレーションの目的は、フィードバックをより早く開発者に返すことでした。このため、ビルド時間が長くなってきたら、単純にビルドを繰り返し実行するのは継続的インテグレーションを実践する方法としては良くありません。それに、ソースファイルがまったく変更されていないのにリビルドをしたら、そのビルド時間はやはり無駄になってしまいます。

そこで、ソースファイルが変更されたとき、つまり誰かが任意の変更をリポジトリにコミットしたときにビルドを開始することができます。これは、単純にビルドを繰り返すよりも良い方法です。多くのSCMソフトウェアはブランチを監視し、任意のコミットがされたときに外部コマンド (ビルドツールなど) を起動する機能があります。このほか、多くのビルドサーバは、定期的にリポジトリをポーリングし、コミットを検出したらビルドを開始する機能を備えています。

しかし、これも最善とはいえません。コミットした後で、それに問題があったことを教えてもらっても面倒だからです。早急にそのコミットを取り消したり、追加の修正をする必要が生じます。そこで、よりモダンなビルドサーバは、コミットする前に、その差分をきれいなサンドボックスに適用してプライベートビルド(バディビルド)をする機能をもつものがあります[[39]](#footnote-39)。これは、ビルドサーバでCIビルドの開始を指示するタイミングとしては非常に優れています。そのバディビルドをテストして、問題がなければ安心してコミットできますし、問題があればコミットせずに済むからです。さらに、この作業まで自動化するビルドサーバもあります。つまり、自動化されたテストを実行し、問題がなければ自動的にコミットまでをビルドサーバが行います。逆にいえば、コミットしたいチェンジセットに問題があれば、コミットを失敗させることができるということです。

## 長いビルド時間

長いビルド時間そのものも問題です。ビルドを開始した直後に、別のビルドの開始を指示しても、そのビルドを実際に開始できるのは最初のビルドが完了してからですから、そのビルドが完了するのは2時間後になってしまいます。これでは、継続的インテグレーションの価値は半減します。ビルド時間がもっと長くなったらどうなるか、想像してみましょう。

そこで、分散ビルドというテクノロジーが利用されるようになりました。分散ビルドは、複数のビルドマシンを制御し、並行してビルドを走らせることで、見かけ上のビルド時間を短くします。これにより、前のビルドが完了するのを待たずに、並行して次のビルドを開始できます。例えば、20台のPCを使ってビルド・ラボを構築すれば、1回のビルドに1時間かかるとしても、平均して3分おきにビルドを開始できます。開発者1人の年収とPC1台の値段を比べれば、これがどれだけ大きな投資対効果を得られるかがわかるでしょう。

現在は、以前より大量のテストが自動化されるようになってきていますから、計算機の処理速度が速くなったとはいえ、やはり1回のビルドに数時間かかることもまれではありません。このため、ビルド時間の管理は今でもビルド作業における重要な関心事のひとつです。念のため繰り返しますが、ビルドマシン上でのインクリメンタルビルドは避けるべきです。

ビルド時間を短くするには、分散ビルドのほか、ビルドマシンを速いマシンに替えるとか、自動テストのような時間のかかるターゲットはナイトリービルドでだけ実行し、ほかのビルドではこのターゲットを動かさないようにするなどの工夫があります。継続的に実行するビルドターゲットを「コミットビルド」とか「軽量ビルド」、時間がかかるビルドターゲットを「2次ビルド」とか「重量ビルド」などといいます。コミットビルドが成功したときに限り2次ビルドを実行するアプローチを、マーチン・ファウラー氏は「段階的ビルド」(staged build)とよんでいます。このほか、ビルドパイプライン (build pipeline) を計画することもあります。複数のビルドマシン(パイプライン)に、どのようにターゲットを流し込むのか計画する、という意味です。

# 継続的インテグレーションとビルドサーバ

実際に分刻みのビルドスケジュールを組み、分割したビルドターゲットを管理し、分散ビルドをしてより多くのビルドを効率的に流すには、ビルドサーバというソフトウェアを使います。ビルドサーバを指してCIサーバということもあります。また、ビルドサーバ(またはその機能)を指して、ビルドエージェントということもあります。ビルドマスターのエージェント (代理人)という意味です。

本章では、ビルドマスターが手動で起動するウィークリービルドの手順を示しました。しかし、ビルドサーバを活用するなら、この手順も大きく簡素化されます。ビルドマスターの仕事は、ビルドサーバのお守りをすることになります。ビルドサーバの多くは、自動的にビルド番号を管理し、できあがったビルドがどこにいったか追跡したり、リポジトリにタグをつけたりする機能があります。

無償で利用できる、分散ビルドをサポートするビルドサーバにはHudsonやTeamCity[[40]](#footnote-40)などがあります。また、有償の製品にはMicrosoft Team Foundation Serverなど、より高機能なビルドシステムもあります。このほかにも、無償・有償を問わず、大変多くのビルドサーバが利用可能になっています。

# 継続的インテグレーションのまとめ

継続的インテグレーションの背景には、PCの性能が向上し、値段も安くなったことがあります。ひと昔前は、規模の大きいソフトウェアシステムのビルドには半日からまる一日かかることも少なくありませんでした。そんな状況では、開発者が何かをコミットするたびに自動でビルドが始まっても困ります。このため、ナイトリービルドのアプローチがよく使われました 。

しかし、PCリソースが潤沢になり、分散ビルドのテクノロジーも利用しやすくなった現在、ビルドを1日に1回しかしてはいけない理由はありません。日次よりもずっと頻繁にビルドすることで、何かをコミットするたびにビルドブレークしないことを自動的に確認し、自動テストの結果も得られます。リグレッションをより早く検知することで、リポジトリを清潔に保つことができるのです。

ただし、ビルドサーバを活用した継続的インテグレーションをするときも、ナイトリービルドとウィークリービルドのプラクティスを併用して活用してほしいと思います。ナイトリービルドにより、時間がかかる自動テストを継続的なインテグレーションから分離できます。また、ウィークリービルドは1週間というタイムボックスをチームメンバーに強く意識させ、開発チーム向けのビルドとテストチーム向けのビルドを明確に区別します。

# まとめ

ソフトウェアの開発プロセスの中でも、ビルドのプロセスは非常に重要なものです。定期的なビルドこそが、ソフトウェア開発を前に進める心臓の脈動のようなものだからです。本章では、ビルドの基本的な手順を示し、再現可能なビルドをすることがソフトウェア開発に特に重要であることを強調しました。そのためには、ビルドツールを上手に活用すること、ファイル間の依存関係やコンパイルオプションなどのビルドの構成を正しく管理すること、ビルド番号を直接実行可能なソフトウェアに埋め込むことなどを説明しました。過去にユーザーにリリースした製品を社内で再現できず、ユーザーから報告を受けたバグの再現もできないし治しようがない、ということがないようにしましょう。もし、みなさんがすでにソフトウェア製品をリリースした経験をお持ちなら、その製品を再現できるかどうか確認してみましょう。

また、頻繁なビルドと定期的なビルドを組み合わせて活用することに価値があることを示しました。ぜひ、ビルドサーバを導入して、真に継続的な統合ができる環境を実現してください。

# 参考文献

## Daily Build and Smoke Test

Steve McConnell

<http://www.stevemcconnell.com/ieeesoftware/bp04.htm>

## オブジェクトソリューション

Grady Booch (著), 石川 克己 (翻訳)

<http://www.amazon.co.jp/dp/479529724X>

## Continuous Integration

Martin Fowler  
<http://www.martinfowler.com/articles/continuousIntegration.html>

## OutputBuildTarget

Martin Fowler  
http://martinfowler.com/bliki/OutputBuildTarget.html

## make (software)

<http://en.wikipedia.org/wiki/Make_(software)>

## GNU `make'

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

## Ant

<http://ant.apache.org/>

## SvnAnt

<http://subclipse.tigris.org/svnant.html>

## ANT4HG

<http://ant4hg.free.fr/>

## MSBuild

<http://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx>

## NAnt

<http://nant.sourceforge.net/>

## Jam

<http://www.perforce.com/jam/jam.html>

## SCons

<http://www.scons.org/>

## Hudson

<http://wiki.hudson-ci.org/display/HUDSON/Meet+Hudson>

## TeamCity

<http://www.jetbrains.com/teamcity/index.html>

## CruiseControl

<http://cruisecontrol.sourceforge.net/>

## Continuum

<http://continuum.apache.org/>

## Anthill

<http://www.anthillpro.com/html/default.html>

## Software Build Analysis

<http://www.coverity.com/library/pdf/Coverity-Software-Build-Analysis.pdf>

## Build automation

<http://en.wikipedia.org/wiki/Build_automation>

## Build Verification Test

<http://en.wikipedia.org/wiki/Build_Verification_Test>

## Mentor

<http://en.wikipedia.org/wiki/Mentor>

1. リンカが統合されているC/C++コンパイラもあります。そのようなコンパイラを使うときは、明示的にリンカを起動する必要はありません。 [↑](#footnote-ref-1)
2. これらはSCMリポジトリで管理する必要はありません。SCMツールは構成管理の有力な手段ですが、通常のファイルシステムや外部メディアを使っても構成管理はできるということです。どのバージョンのコンパイラやビルドツールを使うのか、ビルド手順書にきちんと書いておきましょう。 [↑](#footnote-ref-2)
3. ビルドを開始してから完了するまでにかかる時間をビルド時間といいます。 [↑](#footnote-ref-3)
4. Rational ClearMakeなどのように、別のサンドボックスでのビルドの成果物を利用してインクリメンタルビルドできるビルドツールもあります。 [↑](#footnote-ref-4)
5. AntではParallelタスクを使います。 [↑](#footnote-ref-5)
6. バックドアとは、ならず者が簡単に勝手に侵入するための「裏口」もしくは「勝手口」のことです。 [↑](#footnote-ref-6)
7. ライブラリとは、ソフトウェア開発の文脈では共通部品のことですが、本来の語義は図書館という意味です。また、ライブラリアンとは司書 (ライブラリの管理をする人) という意味です。 [↑](#footnote-ref-7)
8. ファイルの拡張子は、OSのプラットフォームやコンパイラにより異なります。本文中に示したのは、Windows環境でよく使われる拡張子です。 [↑](#footnote-ref-8)
9. 本文中では、簡潔に示すためにインプットファイルから依存を辿りました。実際には、ビルドツールは最終的に生成したいファイルから依存関係を辿ります。例えば、product.exeはproduct.objとlibrary.libに依存していて、それらはどれに依存していて、という具合です。 [↑](#footnote-ref-9)
10. コマンドライン引数でターゲットを指定しなかったときは、Makefileの先頭のターゲットがビルドされます。また、-fオプションでMakefileファイルの名前を指定することもできます。 [↑](#footnote-ref-10)
11. 1つのターゲットに対して、複数のレシピ行を書けます。また、各レシピ行の先頭は、空白文字ではなくタブ文字で始めます。これは、Makeを初めて使う誰しもがはまる落とし穴です。 [↑](#footnote-ref-11)
12. SCMツールによっては、更新の操作で取得するファイルのタイムスタンプを、それをコミットした日時にする設定もできます。 [↑](#footnote-ref-12)
13. Perforce JamやRational ClearMakeなどがあります。 [↑](#footnote-ref-13)
14. コマンドラインコンパイラを利用できないコンパイラ言語は、私が知る限りありません。Visual BasicやDelphiにも、コマンドラインコンパイラがあります。TeXドキュメントも、Makeでビルドできます。 [↑](#footnote-ref-14)
15. JDK 1.0は、1996年1月にリリースされました。 [↑](#footnote-ref-15)
16. ただし、Execタスクを使うと、メイクファイルと同じように、ビルドスクリプトが各プラットフォーム上の外部ツールに依存してしまう問題が生じます。 [↑](#footnote-ref-16)
17. この例のようにbuild.xml中に日本語を書くときは、build.xmlをUTF-8で保存するか、<project>タグのencoding属性にエンコードを指定します。initはinitialize(初期化)の略、distはdistribute(配布)の略です。${DSTAMP}プロパティは、<tstamp>タグにより現在の日時に設定されます。 [↑](#footnote-ref-17)
18. コマンドライン引数でターゲットを指定しなかったときは、<project>タグのdefault属性で指定されたターゲットがビルドされます。また、-fオプションでbuild.xmlファイルの名前を指定することもできます。 [↑](#footnote-ref-18)
19. 正規表現とは、文字列の検索や置換に使う★パターンです。私は通勤電車の中でこの書籍を執筆しましたが、置換するときは正規表現が便利です。(私は品位ある人間を目指しています。) [↑](#footnote-ref-19)
20. これは、毎ビルド時に指定されたファイルのダイジェストを計算して保存します。ダイジェストとは、任意の長さの元データから固有のアルゴリズムで取得した短いデータです。元データをちょっぴり変更しただけでもそのダイジェストが変わる、あるダイジェストが与えられたときにそれと同じダイジェストをもつ元データを恣意的に生成するのは困難、などの性質があります。計算機への負荷低減のため、暗号化や電子署名などにも使われるテクノロジーです。よく使われるダイジェストアルゴリズムにはMD5や★SHA-1があります。 [↑](#footnote-ref-20)
21. ANT4HGの4はバージョン番号ではなく、hg (Mercurialのコマンド) のためのAntタスクという意味です。ant **for** hgを、ant **four** hgと短く表記しているわけです。このような4の用例は多く見つかります。同様に、2を**to**という意味で使うことがあります (native2asciiなど)。 [↑](#footnote-ref-21)
22. Javaのリソースファイルでビルド番号を管理するなら、標準タスクBuildNumberが使えます。 [↑](#footnote-ref-22)
23. と説明しておいて何ですが、私自身はこの十年以上もの間、こうした景色を見たことがありません。おそらく、自律して動く成熟した組織では、このようなことは起こらないのでしょう。 [↑](#footnote-ref-23)
24. Perforceのコマンドラインツールはp4.exeです。つまり、P4の4はPer**for**ceを短く表記したもので、ANT4HGの4とはまた別のものです。 [↑](#footnote-ref-24)
25. ASAPはas soon as possibleの略語で、エイサップと読みます。「なるはや」の訳語です。 [↑](#footnote-ref-25)
26. ただし、単体テスト(unit test)は、一般に開発者がサンドボックス上で動かします。ビルドマシン上で動かすことはあまりないと思います。この話題は6章で取り上げます。 [↑](#footnote-ref-26)
27. といっても、外部のドキュメントは間違っているかも、などという前提を持ち込むべきではありませんが。。 [↑](#footnote-ref-27)
28. 一般に、クライアントソフトウェアをPCに導入することをインストール、サーバソフトウェアをPCに導入することをデプロイ (配置) といいます。 [↑](#footnote-ref-28)
29. ビルドマシン上でIDEを使ったビルドを自動化するために、「IDEを起動してビルドボタンを押す」という処理をするプログラムを開発した、という笑い話もありました。 [↑](#footnote-ref-29)
30. プロジェクトの管理コストを抑えるために、開発に使うIDEをプロジェクト内で統一することは今でもよくあることです。そもそも、開発言語によっては利用可能なIDEの選択肢が1つしかないこともよくあります。 [↑](#footnote-ref-30)
31. ビルドは開発者が責任をもってリリースすべきものですから、一般にテスターがビルドマスターを兼務することはありません。 [↑](#footnote-ref-31)
32. このほか、製品をどのようにユーザーに届けるかを指してリリースビークル(Release Vehicle; リリースの乗り物)という語が使われることもあります。 [↑](#footnote-ref-32)
33. Extreme Programmingというソフトウェア開発手法のことです。7章で簡単に紹介します。 [↑](#footnote-ref-33)
34. この記事は2006年に大きく書き直されていますが、オリジナル版もwebで閲覧できます。 [↑](#footnote-ref-34)
35. マコネル氏は「Code Complete」や「ラピッドデベロップメント」などの著作でも知られています。そのほとんどが邦訳で読めます。 [↑](#footnote-ref-35)
36. 1998年に出版された邦訳本では、この原文に含まれる'continuous integration'という語が'継続的インテグレーション'とは訳出されていなかったので、この邦訳本の訳文を参考に、筆者が訳出し直しました。 [↑](#footnote-ref-36)
37. 書籍「継続的インテグレーション入門」では、継続的インテグレーションとはこういうものではない、といっています。つまり、「継続的(continuous)」という語を「断続的(continual)」に近い意味で使うとしています。 [↑](#footnote-ref-37)
38. 現在のビルドが完了するまでに1時間、さらにコミットを反映したビルドを開始して完了するまでに1時間かかります。 [↑](#footnote-ref-38)
39. ビルドサーバによるバディビルドは★3-12節でも紹介しました。 [↑](#footnote-ref-39)
40. TeamCityは商用のビルドサーバですが、規模の小さいソフトウェア開発には無償で利用できます。使う前にライセンス条項を確認してください。 [↑](#footnote-ref-40)