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) |

# ビルドの概要

ビルドとは、実行可能なソフトウェアのこと、あるいは実行可能なソフトウェアを構築する作業のことでした。前者の意味でのビルドは、2章で詳細に説明しました。本章では、後者について詳細な手順と、必要なインプットと生成されるアウトプットについて解説します。まず、ビルドの概略を再確認しましょう。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生成

このように、複数のソースファイル (一次ファイル) をコンパイルして統合し、最終的に実行可能なソフトウェアを構築する作業のことをビルドといいます。この作業を支援するのがビルドツールです。現在、よく使われるビルドツールにAntがあります。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。Antについては後述します。

ビルドツールを使わなくても、手作業でコンパイラなどの外部ツールを起動し、ビルドを構築することはできます。しかし、ビルドには再現可能であることが強く求められます。再現可能とは、いつ誰がビルドを作っても、同じソースファイルからは同じビルドを構築できるということです。不適切なビルド作業により製品にバグが混入することは避けなければいけません。

しかし、手作業でビルドをすると、厳密にいつも同じ手順に従ってビルドすることが難しいのです。というのは、ソフトウェアの規模によっては一次ファイルの数が数百から数千になることもあり、しかもこれら一次ファイルとそ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。複数の一次ファイルを正しい順序でコンパイルし、生成された中間ファイルを正しい順序で統合しなければなりません。さらには、できあがったビルドに対して自動テストを実行したり、リリース場所 (ファイルサーバなど) にビルドをコピーしてテストチームにリリースをアナウンスしたりするなど、ビルドには決まりきった手順の作業がたくさん含まれています。

ソフトウェア開発においては何度もくりかえしビルドを作る必要があります。このような繰り返しの処理は、コンピュータが最も得意とするところです。つまり、ビルドの処理はコンピュータで自動化すべきなのです。この自動化を受け持つのが、ビルドツールというソフトウェアです。

# ファイル間の依存関係

ビルドにはさまざまな作業が含まれますが、まずは素朴なビルド作業とは何をするものかを説明しましょう。図4 - 2を見てください。これは、複数のCプログラムをコンパイル・リンクしてビルドする例です。Cは知らないがJavaには馴染みがあるという方は、.hと.cを.javaに、.objを.classに、.exeを.jarに置き換えて読み進めてください[[1]](#footnote-1)。



図4 - 2 Cプログラムファイルの依存関係の例

この例では、3つの一次ファイル (プログラムファイル) があり、main.cはcommon.hで定義されたデータ構造を使うとします。このとき、「main.cはcommon.hに依存する」といいます。もしcommon.hを修正してそのデータ構造を変更したら、common.hに依存するソースファイル全て をコンパイルし直す必要があります。その場合は、たとえmain.c自体は変更していなくても、main.cをコンパイルし直さなければならないのです。sub.cも同様です。さもないと、product.exeは正しくビルドされず、意図した通りに動きません。

sub.cだけを修正したときは、main.cはコンパイルし直す必要はありません。sub.cをコンパイルしてsub.objを作り直し、main.objは前回のビルドの成果物を再利用できます。このmain.objとsub.objをリンクすれば、新しいproduct.exeをビルドできます。このように、再コンパイルが必要なファイルだけをコンパイルしてビルドすることをインクリメンタルビルドといいます。一次ファイルの数が非常に多いとき、インクリメンタルビルドはビルドに要する時間 (ビルド時間) を劇的に短縮します。

以上のように、ビルドツールの基本的な役割は2つあります。

* 再現可能なビルドを実現すること
* 安全なインクリメンタルビルドを実現すること

それぞれ、次節で説明しましょう。

# 再現可能なビルドの実現

ビルド作業のインプット (一次ファイル群) が同じなら、何回ビルドし直しても同じアウトプット (ビルド) が得られなければいけません。これを「再現可能」であるといいます。ビルドを再現可能にするためには、ビルドツールが次のことを知らなくてはなりません。

* ビルドの入力となる一次ファイル群の名前と場所
* ビルドの出力となる実行可能ファイルの名前と場所
* 各ファイルの依存関係と統合する順序
* 各ファイルを変換 (コンパイルなどを)する外部ツールの名前
* 各ファイルを変換する条件 (コンパイルオプションやマクロ定義など)

一般に、以上のことをビルドツールが勝手に知ることはできません。例えば、一次ファイル群の名前だけをビルドツールに与えても、ビルドツールがそれらプログラムファイルの内部構造を点検し、ファイル間の依存関係を自動的に調べることはできません[[2]](#footnote-2)。そこで、そのプログラムを書いた開発者が、どのファイルからどのファイルをどの外部ツールを使って生成するのか、ビルドツールに指示する必要があります。この、開発者がファイル間の依存関係や、ビルド作業に含まれる複数のタスクの依存関係、必要な外部ツールの呼び出し方法などをビルドツールに指示するためのファイルをビルドスクリプトといいます。表4 - 1に、Makeというビルドツールのビルドスクリプト (メイクファイル) の例を示します。図4 - 2に示したファイル間の依存関係が、そのまま文書化されているのがわかるでしょう。

表4 - 1 単純なMakefileの例[[3]](#footnote-3)

|  |
| --- |
| # product.exeはmain.objとsub.objからlink.exeで作る  product.exe: main.obj sub.obj  link.exe main.obj sub.obj /out:product.exe  # main.objはcommon.hとmain.cからcl.exeで作る  main.obj: common.h main.c  cl.exe /c main.c  # sub.objはcommon.hとsub.cからcl.exeで作る  sub.obj: common.h sub.c  cl.exe /c sub.c |

#で始まる行は、コメント行です。次の行で、アウトプットとなるファイルと、そのインプットとなるファイルをコロンで区切って並べます。その次の行は、前の行で指定したアウトプットとなるファイルを生成する外部コマンド (レシピ) を記述します。ビルドスクリプトは、わかりやすく簡潔に書きましょう。それが、清潔なビルド作業や、手間のかからないビルドスクリプトの保守作業につながります。

# 安全なインクリメンタルビルドの実現

インクリメンタルビルドは、前回のビルド時に生成した中間ファイルで再利用できるものは再利用し、コンパイルが必要なファイルだけをコンパイルし直して高速にビルドを行います。これを実現するには、前回のビルドをした後にどの一次ファイルが変更されたかをビルドツールが知る必要があります。しかし、これについては、開発者がビルドツールに指示する必要はありません。ビルドツールは、各ファイルのタイムスタンプを使って自動的にこれを知るからです[[4]](#footnote-4)。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。WindowsやUnixなどの一般的なファイルシステムは、タイムスタンプをすべてのファイルに自動的に付加します。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、ビルドツールはインプットとなるファイル群とそのアウトプットとなるファイルのタイムスタンプを比較し、アウトプットとなるファイルが古ければ、これを再生成する必要があると判断します。図4 - 2と合わせて、図4 - 4を見てください。ビルドツールはmain.cとmain.objのタイムスタンプを比較して、もしmain.cの方が新しければmain.objを作り直す必要があると判断します。あるいは、common.hとmain.objのタイムスタンプを比較して、common.hの方が新しければ、やはりmain.objを作り直す必要があると判断します。main.objファイルが存在しない場合も、main.objを作り直す必要があると判断します。



図4 - 4 インクリメンタルビルド[[5]](#footnote-5)

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[[6]](#footnote-6)。これは、取得した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ができません。



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

インクリメンタルビルドに対し、ソースファイルを全てコンパイルし直してビルドすることをフルビルドといいます。ソフトウェアの規模や自動化されたテストの量にもよりますが、フルビルドに要する時間は数分から数時間にもおよびます。フルビルドをするには、中間ファイルをすべて削除してからビルドします。この、中間ファイルをすべて削除する操作のことをクリーンといいます。

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、次回のビルドではそのファイルから生成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。

フルビルドとインクリメンタルビルドには、それぞれ一長一短があります。フルビルドならビルドの再現性を確保できますが、ビルドに時間がかかります。逆に、インクリメンタルビルドは高速にビルドできますが、ビルドの再現性に若干の不安があります。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。

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。サンドボックスでは、開発作業のために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をします。

しかし、サンドボックス上であっても、ときどきは (少なくとも数日に一度くらいは) フルビルドを行うべきです。退社前にビルドを開始しておけば、次の日の朝にはビルドが終わっているはずです。ただ、朝に出社してみたらビルドエラーが出ていてビルドが終わっていなかったらショッキングです。こうなると、エラーを取り除いてリビルド (フルビルド) が終わるまで、開発作業ができません。これは、★3-13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。

表4 - 2 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 | 遅い | 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 | 若干の 不安あり | 早い | 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。主に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場合です。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、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かった場合です。例えば、変更したファイルが再コンパイルされていなかったり、古いバージョンのライブラリがリンクされてしまったりすると、静かなビルドブレークが起こります。ビルドシステムやビルドスクリプト中のバグが原因ですが、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) が意図通りに動作しない場合、ふつうはプログラムの記述ミスが先に疑われ、ビルドシステムが疑われるのはその次です。ビルドシステムが正しく動作していることは、プロジェクトチームにとって当たり前であり、開発作業の前提となっているからです。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、開発者の工数が無駄になってしまいます。

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のです。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明示的にフルビルドをする、コンパイラなどのツールが出力する警告をエラーと同じように扱ってビルドを中断する、assert文を活用したプログラミングをする、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、このようなライト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。あんどんに、異常のほか、ビルドの進捗状態を表示させることもできます。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[[7]](#footnote-7)。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るでしょう。

# 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 |

# MakeとAnt

以前は、ビルドツールとしてMakeがよく使われていました。ソースファイル間の依存関係 (生成順序) を記述できるMakeを使えば、たいがいのビルド環境は構築できます。事実、Makeは非常に長い間、標準のビルドツールとして不動の地位に君臨していました。C/C++ のプログラムはもちろん、Fortranも、Visual Basicも、Javaも、TeX のドキュメントでさえMakeでビルドできます。

ところが、プログラミング言語Javaが広く使われるようになると、MakeはJavaのプログラムをビルドするには不便だと考えられるようになりました。Javaには方言がないのに、Make にはさまざまな方言があるからです。例えば、Makeには GNU make, BSD make, Microsoft nmake, Borland bmake などの種類があり、メイクファイルの文法が少しずつ違います。また、Makeは正しい順序で外部のツールを呼び出すだけなので、外部ツールの名前がプラットフォームによって違うと面倒なことになりました。例えば、ビルドの途中でファイルをコピーしたいとき、Windows環境ではcopyコマンドを使いますが、Unix環境ではcpコマンドを使います。これでは、プラットフォームごとに別のメイクファイルを準備するか、メイクファイルに仕掛けをするなどの工夫をしなければなりません。

Javaプログラムは (Java仮想マシンが用意されていれば) どんなプラットフォーム上でも動作するのに、なぜビルド用のスクリプトはプラットフォームごとに違うのを使わなければならないのでしょうか。そこで、ジェームズ・デビッドソン氏がAntという新しいビルドツールをJavaで開発しました[[8]](#footnote-8)。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。

AntはJavaで開発されているため、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、ファイルのコピーは<copy>タグで行います。<copy>タグ以外にも、さまざまな作業をこなすためのタグ (タスク) が多く用意されています。

基本的な操作はすべてこのような組み込みのタスクで実行できるようになっているので、基本的には外部コマンドを使う必要はありません。しかし、Execタスク (<exec> タグ) を使えば、外部コマンドも呼び出せます[[9]](#footnote-9)。

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。

もちろん、ソフトウェアのドメインによっては今でもMakeが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NAntやMSBuildなどのビルドツールがあります。みなさんも、みなさんの開発環境に適切なビルドツールを選択して下さい。ビルドツールは構成管理ツールやバグ追跡システムに比べると選択肢が少なく、またNAntとMSBuildはAntによく似ていることもありますので、以後、本書ではAntの使用を前提として説明をすすめます。

# 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)もビルドボタンを装備していましたが、ビルドツールと分離されていませんでした。固有のビルド機能を内部に持ち、独自のビルド構成ダイアログを装備します。確かにビルドを構成するのは比較的簡単でしたが、この設定 (コンパイラオプションやリンクするモジュール一覧など) は外部のビルドツールからは利用できない形式なので、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。この操作は簡単には自動化できないので、とても不便です[[10]](#footnote-10)。実際には、開発環境用のビルド構成ファイルとは別に、ビルドマシンでのビルド作業を自動化するためのビルドスクリプトを別に用意することがふつうでした。が、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、問題が出ることもありました。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です。

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。これにより、ビルドマシンとサンドボックス上のIDEで、全く同じビルドスクリプトによりビルドできるようになっています。Microsoft Visual Studioも、少し前のバージョンからビルドツールが分離されて使いやすくなりました[[11]](#footnote-11)。



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[[12]](#footnote-12)。

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べきです。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ました (その代わり、CIサーバを管理する作業が増えました)。

## 「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。

# 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ている人です。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。 |

# ビルド手順の概要

ここまで、素朴なビルドというのが一次ファイルを統合して実行可能ファイルを生成するプロセスであることを説明しました。実際のビルドには、より多くの作業が含まれます。図4 - 7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、各ターゲット中にも複数のタスクを含むものがあります。サンドボックス上でプライベートビルドをするときは、図4 - 7の5番目に示したターゲット「実行可能なソフトウェアをビルド」に含まれるタスクの手順と同じものを使います。

また、図4 - 7に示したビルド手順は、完全に自動化することはできません。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。

処理の流れを記述して保守することはとても大変です。処理の流れというのは変わりやすく、何らかの前提条件などで、処理の流れがあっちに行ったりこっちに飛んだりすると、ビルドスクリプトは出来の悪いプログラムと同じく、スパゲッティコード[[13]](#footnote-13)になってしまうからです。

そうなるのを防ぐには、処理の流れを可能な限り (手続き的でなく) 宣言的に記述することです[[14]](#footnote-14)。Antのビルドスクリプトは、タスクを宣言的に記述できるようになっています。つまり、ビルドの手順を「ビルド番号を増やしてコミット」や「実行可能なソフトウェアをビルド」などのターゲットに分割し、各ターゲット間の依存関係を宣言することで、処理の流れを記述します。そして、各ターゲットの中に複数のタスクを配置し、具体的な作業を行わせます。Makeはファイル間の依存関係を記述するものでしたが、Antはターゲット間の依存関係を記述する、ということです。Antでは、ファイル間の依存関係は基本的にタスクが面倒を見ることになっています[[15]](#footnote-15)。

# ビルドスクリプト (build.xml)

では、Antのビルドスクリプトの基本的な構造を見てみましょう。AntのビルドスクリプトはXML形式で記述します。Antのタスクは、XMLのタグに小文字で記述します。

表4 - 3 単純なbuild.xmlの断片

|  |
| --- |
| <?xml version="1.0" encoding="Shift-JIS"?>  <project default="inc">  <target name="full" description=”**フルビルド。**”  depends="update, inc\_build#, clean, compile,  test, doc, jar, release2dev" />  <target name="inc" description=”**インクリメンタルビルド。**”  depends="update, inc\_build#, compile, test, doc, zip, copy2dev" />  <target name="release2dev" description=”**DEVにリリース。**”  depends="copy2dev, mail2dev" />  <target name="tagging" description=”**ビルドに成功したらタグ付け。**”  depends="commit\_build#, add\_tag" />  <target name="release2tester" description=”**テストチームにリリース。**”  depends="copy2tester, mail2tester" />  <target name=”update” description=”**リポジトリからソースを取得。**”>  …  </target>  <target name="clean" description=”**前回の中間ファイルを削除。**”>  …  </target>  <target name="inc\_build#" description=”**ビルド番号を増やす。**”>  …  </target>  <target name="compile" description=”**コンパイル。**”>  …  </target>  …  </project> |

Build.xmlの中に複数のターゲットを書き、このターゲット間の依存性を、depends属性で記述します (depends～とは、～に依存するという意味です) 。図4 - 7に示したビルドの手順と比べてみてください。ビルドの手順とビルドスクリプトが、うまく対応することがわかるでしょう。表4 - 3では省略しましたが、<target>タグの中に、あらかじめ用意されているさまざまなタスク・タグを配置して、ビルドスクリプトを構成します。Antのタスクを包括的に解説することは本書の範囲を超えますので控えますが、Antに標準で装備されている代表的なタスクを表4 - 4に列挙しておきます。このほかにも多くのタスクが標準で装備されています。詳しくはAntのマニュアルを参照してください[[16]](#footnote-16)。

表4 - 4 Antの代表的なタスク一覧



# Antの独自タスクを書く

表4 - 4に示したように、Antは標準で多くの機能をタスクとしてもっています。しかし、標準で用意されていない機能もあります。例えば、Subversionのような比較的新しいSCMソフトウェアを操作するタグは、標準では用意されていません。標準のタグではこなせないタスクを実現するには、3つの選択肢があります。

1. Execタスクで、外部コマンドを呼び出す  
   いちばん簡単な方法です。例えば、Subversionのコマンドsvnを<exec>タグを使って呼び出します。ただし、ビルドスクリプトが読みにくくなるため、このメンテナンスが少々面倒になることがあります。
2. したいことができる非標準のタスクを探す  
   例えば、あなたがSubversionのコマンドをAntから呼び出したいなら、SvnAntタスクが利用可能です。また、Mercurialにはant4hg[[17]](#footnote-17)が利用可能です。しかし、大事なビルドをブラックボックスに託すことになるので、枯れた[[18]](#footnote-18)タスクであったり、ソースコードが公開されているタスクでない限り、あまりお勧めできません。
3. 独自のタスクを自分で書く

Antには、高機能なワイルドカードでファイルの集合をエレガントに指定できるファイルセットという機能があります。独自タスクを書けば、このファイルセットの機能を使って処理対象のファイルを簡潔に外部コマンド (例えばsvnコマンド) に指示できるようにできます。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。

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、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。タグのプロパティ値は、対応するsetterメソッドを書けば取得できます。

# 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 - 4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。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 - 4に示したように、CVSやMicrosoft Visual SourceSafe, PerforceなどのSCMソフトウェアには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。例えば、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ソフトウェアをお使いなら、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4-11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、リポジトリにコミットします。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。ただし、自動化する場合には、ビルド番号を更新するターゲットを他のターゲットと分離して実行できるようにしておく必要があります。ビルドするたびに、勝手に週次のビルド番号が増えてしまうと、計画通りのビルド番号が付加されたビルドを作ることができないからです。

継続的インテグレーションのプラクティスを実践し、一日にビルドマシンで何回もビルドするなら、(日次の) ビルド番号を増やす作業を自動化しておく必要があります。これには、BuildNumberタスクを使います。このタスクは、build.numberという名前のJavaプロパティファイルに書かれたbuild.numberプロパティの数値をひとつ増やします。BuildNumberタスクを使ってビルド番号リソースをマニフェストファイルに埋め込む例を表4 - 5と表4 - 6に、マニフェストファイルからビルド番号を読んで表示するプログラムを表4 - 7に、その実行結果を表4 - 8に示します。Antを実行するたびに、ビルド番号が自動的に増えていくのがわかるでしょう。

表4 - 5 build.numberプロパティファイルの例

|  |
| --- |
| C:\idd\doc\1perw.pngmajor.number=1  minor.number=0  C:\idd\doc\1perb.pngweekly.number=8  build.number=314 |

表4 - 6 BuildNumberタスクを含むbuild.xmlの例

|  |
| --- |
| <project default="all">  <target name="all" depends="compile, jar" />  <target name="compile">  <**javac** srcdir="." /> <!—Javaソースファイルをコンパイル -->  </target>  <target name="jar">  <**buildnumber** /> <!-- build.numberプロパティをひとつ増やす -->  <property file="build.number" /> <!-- build.number以外のプロパティもロード -->  <**manifest** file="MANIFEST.MF"> <!-- マニフェストファイルを作成 -->  <attribute name="Main-Class" value="main/ShowVersion" />  <attribute name="Implementation-Version"  value="${major.number}.${minor.number}.${weekly.number}.${build.number}" />  </manifest>  <!-- .classファイルとマニフェストファイルをアーカイブ -->  <**jar** destfile="ShowVersion.jar"  basedir="."  includes="\*\*/\*.class"  manifest="MANIFEST.MF" />  </target>  </project> |

表4 - 7 バージョン情報を表示するJavaプログラムの例

|  |
| --- |
| // main/ShowVersion.java  package main;  public class ShowVersion {  public static void main(String[] args){  System.out.println("My version is: " +  ShowVersion.class.getPackage().getImplementationVersion());  }  } |

表4 - 8 ShowVersionプログラムの実行結果

|  |
| --- |
| C:\idd\doc\comment2.pngC:\Project\Mainline\src>ant  Buildfile: C:\Project\Mainline\src\build.xml  ...  BUILD SUCCESSFUL  Total time: 0 seconds  C:\Project\Mainline\src>java -jar ShowVersion.jar  My version is: 1.0.8.315  C:\Project\Mainline\src>ant  Buildfile: C:\Project\Mainline\src\build.xml  ...  BUILD SUCCESSFUL  Total time: 0 seconds  C:\Project\Mainline\src>java -jar ShowVersion.jar  My version is: 1.0.8.316 |

もし、ビルド番号を含むバージョンリソースファイルがJavaのではなくWindows形式のリソースファイルであったり、また複数のモジュールをビルドする必要があったりする（複数のバージョンリソースファイルをメンテナンスする必要がある）のであれば、バージョンリソースファイルを自動生成するAntの独自タスクを作成しておきたいところです。

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埋め込むとか、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埋め込んでもいいでしょう。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埋め込むことを強く勧めます。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を削除します。これには、Deleteタスクを使います。もちろん、インクリメンタルビルドをしたいときには、前回のビルドの成果物を削除しないようにします。

## 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。開発言語がMicrosoft Visual BasicやBorland Delphiなどであれば、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で解消する必要があります[[19]](#footnote-19)。

## 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を使うなら、JUnit / JUnitReportタスクを使ってテストコードを実行できます。自動化されたテストの実行に時間がかかるのであれば、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めに設定して十分な時間を確保し、リリース前に自動化したテストが実施し終えることができるようにします。

あまりにも結果が芳しくなければ、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。もちろん、このリビルドに際しては、ビルド番号を再度インクリメントする必要はありません。

## 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。みなさんがJavaをお使いなら、JavadocタスクでJavadocを呼び出してドキュメントを生成します。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  
  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[[20]](#footnote-20)するのも便利でしょう。ただし、デプロイ手順もテストの対象とすべきですから、ときどきは手作業でもデプロイを試しましょう。

## 開発チームに候補ビルドのリリースをアナウンス

開発者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。より頻繁にビルドをリリースしながら開発するときは、自動化しておくのもいいでしょう。

## 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前に、テストチームでのテストが実施できる品質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、先に実施した自動化されたテストも含まれます。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。継続的インテグレーションでは、自動化されたテストだけでBVTを実施することもあります。

## 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Javaのプロパティファイルをコミットすることになります。リポジトリに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簡単になります。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。このようなタグを生成するには、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 (マップファイルなど) 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ことについては、★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。継続的インテグレーションによるビルドであれば、ビルドサーバから自動的に送信されるビルド完了メールで代えても問題ありません。

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。

# 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、descriptionに日本語を記述することもできます。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 - 5に示したbuild.xmlに対して、ant –projecthelpを実行した結果を表4 - 9に示します。

表4 - 9 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 前回の生成ファイルを削除。  inc\_build# ビルド番号を増やす。  compile コンパイル。  Default target: all |

ただし、ビルドに際しては、そもそもどのターゲットを最初に実行したらいいのか分からない、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といいます。複数のブランチから、それぞれ定刻に列車が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。ただし、リリーストレインという語を使うときは、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強調されます。



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。駆け込み乗車は危険ですから、なるべく避けましょう。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した方がいいかもしれません。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。

また、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、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、これを最終的な目的地まで全速力で走らせましょう！

# 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かもしれません。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くるものではありませんから、必要以上に頻繁にビルドをリリースすることは避けるべきです。特に、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、テストチームのためのものではないと心得ましょう。

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、1週間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書いても、それはすでに次のビルドで修正済みかもしれません。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[[21]](#footnote-21)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[[22]](#footnote-22)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が以前からあり、その中でもスティーブ・マコネル氏[[23]](#footnote-23)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[[24]](#footnote-24)。

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すことができます。



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[[25]](#footnote-25)。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報告してくれるでしょう。

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[[26]](#footnote-26)。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あるのを思い出してください。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。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、そのビルド時間は無駄になってしまうからです。

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ようにします。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。

## 長いビルド時間

長いビルド時間そのものも問題です。1番目のコミットに続けて2番目のコミットをしたら、2番目のコミットへのフィードバックが得られるのは、やはり2時間後です。これでは、継続的インテグレーションの価値は半減してしまいます。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人の年棒と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。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買い替えるとか、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) とよばれることもあります。

ビルド時間を短くするためにビルドマシン上でインクリメンタルビルドをするのは、あまりお勧めできません。

# 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。ビルドサーバは、ビルドマスターの代理人 (エージェント) であると捉えることができます。

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 (その代わり、ビルドサーバのお守りをするという仕事が増えます)。ビルドサーバの多くは、自動的にビルド番号を管理して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[[27]](#footnote-27)などがあります。また、有償の製品にはMicrosoft Team Foundation Serverなどの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ます。ビルドサーバを使ってウィークリービルドをするには、週次のビルド番号を更新してコミットしたら、後は待つだけです[[28]](#footnote-28)。

# 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

<http://www.martinfowler.com/articles/continuousIntegration.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/>

## 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. .classファイルはそのままでも実行可能なので、リンクという作業は必要ありません。しかし、アーカイブという操作により複数の.classファイルをひとつの.jarファイルにまとめておくと便利です。 [↑](#footnote-ref-1)
2. Perforce Jam のように、C/C++プログラムファイル間の依存性を自動で解析するビルドツールもあります。 [↑](#footnote-ref-2)
3. cl.exeはMicrosoftのC/C++コンパイラ、link.exeは同じくリンカです。 [↑](#footnote-ref-3)
4. このほか、各一次ファイルのMD5ハッシュ値を使って、変更されたファイルを特定するビルドツールもあります。 [↑](#footnote-ref-4)
5. この図ではcommon.hを省略して簡潔に示していますが、実際には依存関係にあるすべてのファイルのタイムスタンプを考慮してインクリメンタルビルドされます。 [↑](#footnote-ref-5)
6. SCMツールの設定により、更新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こともできます。 [↑](#footnote-ref-6)
7. バックドアとは、ならず者が簡単に侵入するための「裏口」のことです。 [↑](#footnote-ref-7)
8. デビッドソン氏は、オープンなWebアプリケーションサーバTomcatを開発したことでも有名です。 [↑](#footnote-ref-8)
9. ただし、Execタスクを使ってしまうと、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コマンドに依存してしまう問題が生じます。 [↑](#footnote-ref-9)
10. 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、という笑い話も聞いたことがあります。 [↑](#footnote-ref-10)
11. Microsoft Visual Studio 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ました。また、新しいバージョンのDelphiも、msbuildと統合されています。 [↑](#footnote-ref-11)
12. 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-12)
13. 処理の流れがごちゃごちゃに絡んで、保守ができなくなってしまったプログラムをスパゲッティコードといいます。 [↑](#footnote-ref-13)
14. これは、通常のプログラムコードを記述するときにも適用できる考え方です。 [↑](#footnote-ref-14)
15. 例えば、Javaプログラムファイル間の依存関係はJavaCタスクが面倒を見ます。つまり、JavaCタスク (Javaコンパイラ) は、指定した以外のJavaプログラムファイル (.javaと.class) のタイムスタンプもチェックして、必要と判断すればそれらも自動でコンパイルします。C/C++コンパイラにはそのような機能はありません。 [↑](#footnote-ref-15)
16. http://ant.apache.org/manual/index.html [↑](#footnote-ref-16)
17. 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-17)
18. 枯れたとは、良い意味でこれ以上成長する見込みがない、進化する必要がないソフトウェアのことをいいます。つまり、ほとんどのバグが修正済みで、安定しており、信頼性が高いということです。 [↑](#footnote-ref-18)
19. 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-19)
20. 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-20)
21. 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-21)
22. 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-22)
23. 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-23)
24. 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-24)
25. 書籍「継続的インテグレーション入門」では、「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-25)
26. 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-26)
27. 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-27)
28. BVTをどのように実施すべきか検討しましょう。前のビルドでBVTを通して、問題なければビルド番号をコミットすることもできます。もちろん、自動化できれば、それが好ましいです。 [↑](#footnote-ref-28)