

Haxe 実践マクロ

shohei909

Table of Contents

1. はじめに	1
1.1. Haxeとは	1
1.2. この本を読むうえで	2
1.3. Haxeのバージョンについて	2
2. マクロの基本事項	3
2.1. ハロー、マクロ	3
2.2. Haxeのコンパイルの過程	3
2.3. マクロの種類	4
3. 初期化マクロ	6
3.1. ビルド日時の埋め込み	6
3.1.1. Tips: マクロとnekoパッケージ	7
3.2. フィールドの削除、型の変更、タグ付け	7
3.2.1. Tips: パッチファイル	9
3.3. include	9
3.3.1. Tips: @:expose	10
3.3.2. Tips: ファイル単体でのinclude	11
3.4. exclude	11
3.4.1. Tips: --gen-hx-classes	12
3.4.2. Tips: 初期化マクロとhaxe.macro.Compiler	12
4. 式マクロ	13
4.1. 処理を2回繰り返す	13
4.1.1. Tips: 引数に使える型	14
4.1.2. Tips: レイフィケーション	14
4.2. 時間計測	15
4.2.1. Tips: 式のデバッグ方法	17
4.2.2. Tips: staticでないマクロ	17
4.3. ローカル変数のデバッグトレース	17
4.3.1. Tips: 出力ターゲット側のデバッグ機能	19
4.3.2. Tips: エラーの記述	19
5. ビルドマクロ	20
5.1. 定数を自動生成する	20
5.1.1. Tips: マクロとドキュメント生成	21
5.1.2. Tips: #if display	22
5.2. 関数の呼び出しをトレースする	22
5.2.1. Tips: コンパイルにかかった時間を計測する	24
6. イベントハンドラ	25

6.1. Linterを作る(onGenerate)	25
6.1.1. Tips: Typeとhaxe.macro.Type	27
6.2. 出力にライセンス情報を追加する(onAfterGenerate)	28

Chapter 1. はじめに

1.1. Haxeとは

Haxeは静的型付けのプログラミング言語です。特徴的なのはその出力ターゲットの豊富さで、以下の3種類のバイトコードと8種類ソースコードへコンパイルすることが可能です。

- バイトコード
 - Flash
 - Neko (Haxe Foundationが開発しているVM用)
 - hl (Haxe Foundationが開発している中間言語)
- ソースコード
 - JavaScript
 - ActionScript3
 - C#
 - Java
 - Python
 - C++
 - PHP
 - Lua

標準ライブラリはすべての出力ターゲット(または複数のターゲット)で利用可能なパッケージと、それぞれの出力ターゲット側の標準ライブラリの両方が提供されています。このためHaxeではクロスプラットフォームで動作するようなプログラムと、特定のターゲットに強く依存するようなプログラムの両方を書くことができます。

HaxeはもともとはActionScriptの代替として開発されていたもので、文法もActionScript3によく似ています。ただ、ActionScript3に似ているというだけではなく、以下のようなActionScript3には無い機能を多く備えています。

- 型推論
- 型パラメータ (ジェネリクス)

- enum（一般化代数データ型、GADT）
- パターンマッチング
- typedef（型エイリアス）
- 構造的部分型付け
- 配列内包表記、マップ内包表記
- 文字列内での変数展開
- マクロ

この本ではマクロの機能について、くわしくあつかいます。

1.2. この本を読むうえで

この本はHaxeの入門書ではないので、Haxeの基本文法やインストール手順などに触れません。ただHaxeを知らないプログラマでも読めるように、具体的にマクロがどのような問題を解決するのかに焦点を当てています。

マクロはあつかいの難しい機能ですが、同時にとても魅力的な機能です。Haxeを知らない方も、この本を読んでいただいてその世界を感じていただけたらと思います。

1.3. Haxeのバージョンについて

この本では、Haxe 3.3.0 RC1のバージョンを前提に書かれています。

Chapter 2. マクロの基本事項

2.1. ハロー、マクロ

Haxeのマクロは、Haxeのコンパイラの振る舞いをHaxeのコードで操作できる機能です。これがどういうことか理解するために、以下のコードを見てください。

```
import haxe.macro.Context;

class HelloMacro {
    public static function hello():Void {
        Context.error("Something Wrong!?", Context.makePosition({min:0, max:2,
        file:"HelloMacro.hx"}));
    }
}
```

これをHelloMacro.hxという名前で保存して、以下のコマンドでコンパイルをします。

```
haxe --macro HelloMacro.hello()
```

この引数はコンパイルの初期化段階でHelloMacro.hello()関数を呼び出す指定をしています。そして、この結果はコンパイルエラーです。

```
HelloMacro.hx:1: characters 0-2 : Something Wrong!?
```

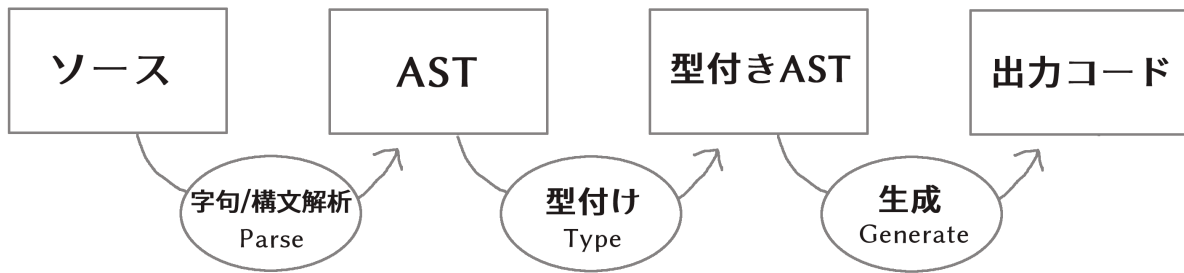
しかし、これで問題ありません。これが意図した動作です。上のサンプルは、コンパイルエラーを起こすように命令をしています。そして、エラーの発生位置としてHelloMacro.hxファイルの0から2文字目を指定しています。

これがまさに「Haxeのコンパイラの振る舞いをHaxeのコードで操作する」ということです。もちろんマクロでできるのはそれだけではありません。Haxeの抽象構文木(AST)にアクセスして書き換えたり、クラスに変数を追加したり、リソースの埋め込みを行ったり、コンパイルのさまざまな過程に介入することができます。

2.2. Haxeのコンパイルの過程

マクロとコンパイルには深い関係がありますから、まずHaxeのコンパイルがどのように進むのか知っておくとよいでしょう。

以下は、コンパイルの過程を簡単な図にしたものです。



マクロを使用した場合、この内、ソースから型付きASTまでの過程は2重に動作します。

つまり、1つ目はマクロのコードを読み込むため、2つ目は実際のコードを出力するためのものです。マクロとして読み込まれたコードは、実際出力のためのコンパイルの各過程を操作するのに使用されます。

同一のファイル内で、マクロ用の読み込みと、実際出力の読み込みで、別々のコードを読みこみさせたい場合、`macro`コンパイル条件フラグで分岐をさせます。

例を見てみます。

```
#if macro
class Macro {}
#else
class Main {}
#end
```

このように記述した場合、マクロの読み込み時には`class Macro {}`として解釈されて、実際の出力用には`class Main {}`として解釈されます。

2.3. マクロの種類

Haxeのマクロはいくつかの種類があります。この本では以下の4種類に分けてあつかいます。カッコ内は、出力用のコンパイルがどの段階のときに実行されるかです。

- 初期化マクロ（初期化段階）
- 式マクロ（構文解析の途中）
- ビルドマクロ（構文解析の途中）
- イベントハンドラ(`onGenerate`は生成前、`onAfterGenerate`は生成後)

次の章から、それぞれが具体的にどういうものなのか実用例と共にみていきます。

Chapter 3. 初期化マクロ

初期化マクロはもうすでに見ています。最初のコンパイルエラーの例がそうでした。コンパイラオプションで関数を指定するとコンパイルの初期化段階で実行されます。

3.1. ビルド日時の埋め込み

例えばスマートフォンアプリの開発をしていると、いま端末に入っているアプリがいつビルドしたバージョンなのかわからなくなってしまうことがあります。こういった場合、ビルドした日時を開発版のアプリに埋め込んで、画面に表示してしまうといったのバージョンなのかが一目でわかるようになります。

以下は初期化マクロを使って日時を埋め込んで、出力するサンプルです。

```
import haxe.Resource;
import haxe.io.Bytes;

#if macro
import haxe.macro.Context;
#end

class EmbeddingDate {
    public static var DATE_RESOURCE_KEY = "dateResource";

    #if macro
    public static function initialize():Void {
        // 初期化マクロのエントリーポイント

        // 現在時刻を取得して文字列に
        var dateString = Date.now().toString();

        // 文字列をリソースとして埋め込み
        Context.addResource(DATE_RESOURCE_KEY, Bytes.ofString(dateString));
    }
    #end

    public static function main():Void {
        // アプリの実行時のエントリーポイント

        // リソースからビルド日時を取り出して出力
        trace(Resource.getString(DATE_RESOURCE_KEY));
    }
}
```

```
}
```

これを以下のオプションで、Nekoのバイトコードにコンパイルします。

```
haxe --macro EmbeddingDate.initialize() -main EmbeddingDate -neko EmbeddingDate.n
```

そして出力されたファイルを実行します。

```
neko EmbeddingDate.n
```

すると以下のようにビルド日時の出力がされます。

```
EmbeddingDate.hx:30: 2016-04-01 00:00:09
```

今回はマクロから実行時へ情報をわたすのに、`Context.addResource`関数で情報を埋め込んで、実行時に`Resource`でそれを取り出す方法をとりました。これはマクロでよく使うパターンです。

時刻以外にも以下のような情報を見れるようにすると、ビルドした状況が確認できて便利です。

- `Sys.systemName()` : OS
- `Context.defines()` : コンパイラフラグ

Haxe公式サイトCookbookでは、gitのコマンドを呼び出して、Gitのコミットハッシュ値を埋め込む方法も紹介されています。

3.1.1. Tips: マクロとnekoパッケージ

マクロの実行時の標準ライブラリはhaxe.macroパッケージやsysパッケージだけでなくnekoパッケージも利用可能です。

3.2. フィールドの削除、型の変更、タグ付け

HaxeではJavaScriptのライブラリなど出力ターゲット側のライブラリを使いたい場合は、多くの場合、型定義ファイル(extern)を用意します。

使いたいライブラリが有名なものであれば、多くの場合externをすでに作って公開している人がいるのでそれを使えばよいのですが、このときに問題がある場合があ

ります。それは、使いたいライブラリのバージョンとexternのバージョンが合っていない場合です。

こういった場合はexternを直接編集してしまいたくなりますが、そうすると元のexternが更新されたときなどに面倒です。

サードパーティのexternだとちゃんとメンテナンスされないことも多いので、externを自分で編集してしまうのは実際悪くない選択肢です。ただし必要な変更がフィールドの削除や、メタデータタグ、型の変更で済むのであれば、初期化マクロの出番です。

```
#if macro
import haxe.macro.Compiler;
#end

// externクラス
extern class SampleExtern {
    public static function test():Void;
    public static function test2():Void;
    public static function test3():Void;
}

class PatchExtern {
    #if macro
    public static function initialize():Void {
        // SampleExtern.testに非推奨のタグ付け
        Compiler.addMetadata("@:deprecated", "SampleExtern", "test", true);

        // SampleExtern.test2を削除
        Compiler.removeField("SampleExtern", "test2", true);

        // SampleExtern.test3の戻り値をStringに変更
        Compiler.setFieldType("SampleExtern", "test3", "String", true);
    }
    #else
    public static function main():Void {
        // コンパイル時に非推奨の警告が表示される
        SampleExtern.test();

        // アクセスしようとするとエラー
        // SampleExtern.test2();

        // 戻り値がStringに変更されているので、traceの引数に使える
        trace(SampleExtern.test3());
    }
}
```

```
#end  
}
```

こうしてマクロで修正をしておくことで、元のexternが更新された場合にも比較的ラクに追従することができます。もちろん、このようなフィールドに対する編集はexternでないクラスに対しても同様に可能です。

3.2.1. Tips: パッチファイル

変更が複数必要であれば、パッチファイルを使うと良いです。先ほどの例と、同じ意味になるパッチは以下の通りです。

```
@:deprecated static SampleExtern.test  
-static SampleExtern.test2  
static SampleExtern.test3 : String
```

これをsample.patchというファイル名で保存して、マクロからCompiler.patchTypesで適用します。

```
public static function initialize():Void {  
    Compiler.patchTypes("sample.patch");  
}
```

変更するフィールドがstaticでない場合は、単純にパッチファイルの各static を消せば動作します。

3.3. include

通常Haxeでは基本的にコンパイルオプションの-mainでmain関数を持つクラスを指定してコンパイルを行いますが、実はこの指定をしなくてもコンパイルは可能です。ここでは初期化マクロからコンパイル対象を指定する方法を紹介します。

IncludeMacro.hx

```
import haxe.macro.Compiler;  
  
class IncludeMacro {  
    public static function initialize():Void {  
        // libパッケージ以下のすべての型をコンパイル対象に指定  
        Compiler.include("lib", true);  
    }  
}
```

```
}
```

```
lib/IncludeSample.hx
```

```
package lib;

class IncludeSample {
    public function new() {
        trace(Math.random());
    }
}
```

以上の2つのファイルを使って、以下のコマンドでJavaScriptにコンパイルします。

```
haxe -js lib.js --macro IncludeMacro.initialize()
```

すると、以下のJavaScriptが生成されます。

```
(function (console) { "use strict";
var lib_IncludeSample = function() {
    console.log(Math.random());
};
})(typeof console !== "undefined" ? console : {log:function(){} });
```

メインクラスを指定しなくてもコンパイルが成功しており、lib.IncludeSampleクラスが出力結果に含まれているのが分かります。

このようなコンパイル対象の指定方法はHaxeでJavaScriptのライブラリを作成した場合に便利です。Haxeはmain関数から到達できないコードを出力コードから省くデッドコード削除機能を備えていますが、上記のような指定を行った場合パッケージ全体を出力に含めた上でそこから使用されていないコードを削除してくれます。

3.3.1. Tips: @:expose

HaxeからJavaScriptに出力したクラスや関数は、デフォルトではJavaScriptからのアクセスができません。JavaScriptからアクセスしたいクラスや関数には以下のように@:exposeのタグを付けてください。

```
@:expose
class IncludeSample {
```

こうするとJavaScriptから、`new lib.IncludeSample()`や`IncludeSample`のフィールドが呼び出せるようになります。

3.3.2. Tips: ファイル単体でのinclude

パッケージまるごとでは無くファイル1つ1つをincludeしたい場合、単純にコマンドライン上でそのファイルのパスを指定します

```
haxe lib.IncludeSample lib.IncludeSample2
```

3.4. exclude

JavaScriptターゲットで外部ライブラリを使いたい場合は、JavaScriptで直接書かれたライブラリを使うかHaxeで書かれたライブラリをそのまま使うことが多いですが、まれにHaxeからJavaScriptに出力したコードをまたHaxeから使いたいということがあります。

例えば、ライブラリ本体とそれに対するプラグインの両方をHaxeで書きたいという場合です。この場合、本体のコードに依存しているプラグインを単純にコンパイルすると、本体側のコードがプラグインに含まれてしまいます。

このような場合に、初期化マクロで`exclude`を行うと本体側のコードを出力から削除できます。以下は、先ほどの`lib.IncludeSample`に依存するようなコードで`exclude`を行っているサンプルです。

```
import lib.IncludeSample;

#if macro
import haxe.macro.Compiler;
#end

class ExcludeSample {
    public function new() {
        new IncludeSample();
    }

    #if macro
    public static function initialize():Void {
        // libパッケージ以下を、出力結果に含めない
        Compiler.exclude("lib");
    }
#end
```

```
}
```

これをコンパイルします。

```
haxe ExcludeSample -js exclude_test.js --macro ExcludeSample.initialize()
```

すると、以下が出力されます。

```
(function (console) { "use strict";  
var ExcludeSample = function() { };  
ExcludeSample.main = function() {  
    new lib.IncludeSample();  
};  
})(typeof console !== "undefined" ? console : {log:function(){}});
```

確かに、`lib.IncludeSample`の呼び出しを行っていますが、`lib.IncludeSample`自体の実装は含まないようなコードが生成できました。

3.4.1. Tips: --gen-hx-classes

この本体とプラグインの関係を実現できる機能としては、`--gen-hx-classes`もあります。`--gen-hx-classes`のオプションをつけてHaxeのコンパイラを実行すると、ソースコードからそのexternを生成することができます。

この機能ではjarやswcなどターゲットのライブラリからexternを生成することもできるのでその用途で利用されることも多いです。

3.4.2. Tips: 初期化マクロとhaxe.macro.Compiler

初期化マクロで指定する関数は自作の関数でなくても、標準ライブラリの関数を直接指定することが可能です。つまり、`exclude`の例は以下のコマンドでも同じ結果になります。

```
haxe ExcludeSample -js exclude_test.js --macro haxe.macro.Compiler.exclude('lib')
```

さらに、`haxe.macro.Compiler`クラスの関数を使う場合クラス名が省略可能です。

```
haxe ExcludeSample -js exclude_test.js --macro exclude('lib')
```

Chapter 4. 式マクロ

式マクロは関数呼び出しのように使えるマクロです。Haxeの式を受け取って別の式へと変換します。

4.1. 処理を2回繰り返す

式マクロがどのようなものか理解するために、同じ処理を2回繰り返すマクロを書いてみます。

```
import haxe.macro.Context;
import haxe.macro.Expr;

class ExprMacro {
    public static macro function twice(expr:Expr):Expr {
        return {
            expr: ExprDef.EBlock([expr, expr]),
            pos: Context.currentPos(),
        }
    }
}
```

普通の関数定義のようですが、`macro`の修飾子がこの関数が式マクロであることを表しています。引数と戻り値に使われている`haxe.macro.Expr`は、Haxeの抽象構文木(AST)を表す構造体です。要素の種類を表すenumと、その要素がコードのどの位置から来たかの情報で構成されます。このマクロではもらった式を2度繰り返すブロック式を生成して返しています。`Context.currentPos()`はこの関数の呼び出し箇所の位置情報で、生成したブロック式の位置情報としてこれを割り当てています。

このマクロを実際につかってみます。

```
class ExprMacroSample {
    static function main() {
        var i = 0;
        ExprMacro.twice(i += 4);
        trace(i); // 8
    }
}
```


コンパイル時に`ExprMacro.twice`関数に`i += 4`の式の構文木が渡されて、それを繰り返すブロック式を生成します。つまり、コンパイルの過程で`main`関数は以下の意味に書き換えがされます。

```
static function main() {
    var i = 0;
    {
        i += 4;
        i += 4;
    }
    trace(i); // 8
}
```

4.1.1. Tips: 引数に使える型

マクロの関数の引数としては`Expr`型の他に、基本型、文字列型、それらの配列が使用できます。これらの型を指定した場合、そのリテラルを記述して渡すとその値を受け取ることができます。また最後の引数に`Array<Expr>`を指定した場合、`Expr`を可変長引数で受け取ることができます。

4.1.2. Tips: レイフィケーション

ブロック式一つ作るにも`ExprDef.EBlock`だとか`Context.currentPos`だとかを書かないといけないのは面倒です。Haxeのマクロではこのような`haxe.macro.Expr`の構造体をもっと簡単に書くための構文が用意されています。それがレイフィケーション(Reification)です。

さきほどの`twice`をレイフィケーションを使って書き換えてみます。

```
public static macro function twice(expr:Expr):Expr {
    return macro {
        $expr;
        $expr;
    }
}
```

元のコードよりも簡単に、もらった式を2回繰り返すブロック式を表現できています。レイフィケーションは`macro` 式の形で使用できます。`macro`に続けてHaxeのコードをそのまま記述するとそれを表す`haxe.macro.Expr`を返します。`$`はエスケープの記号で`$expr`はその位置で`expr`変数に格納されている式を使用することを指定しています。

使用できるエスケープには以下の種類があります。

	型	説明
<code>\${}</code> 、 <code>\$e{}</code>	<code>Expr->Expr</code>	<code>{}</code> の中身进行评估して、その位置に展開
<code>\$a{}</code>	<code>Array<Expr>->Array<Expr></code> または <code>Array<Expr>->Expr</code>	<code>Array<Expr></code> を期待する位置に記述すると、値をその位置に展開。 <code>Expr</code> を期待する位置では、配列の宣言の式に変換して展開。
<code>\$b{}</code>	<code>Array<Expr>->Expr</code>	ブロック式。
<code>\$i{}</code>	<code>String->Expr</code>	文字列から識別子を生成。
<code>\$p{}</code>	<code>Array<String>->Expr</code>	フィールドアクセス式。
<code>\$v{}</code>	<code>Dynamic->Expr</code>	その値のリテラルの式を生成。基本型、enumのインスタンス、それらの配列で動作する。
<code>object.\$name</code>	<code>String->Expr</code>	フィールドアクセス。
<code>var \$name = 1;</code>	<code>String->Expr</code>	変数宣言。
<code>function \$name () {}</code>	<code>String->Expr</code>	関数宣言。
<code>{ \$name : 1 }</code>	<code>String->Expr</code>	オブジェクトのリテラル。
<code>try e() catch(\$name:Dynamic) {}</code>	<code>String->Expr</code>	try-catch
<code>new \$typePath()</code>	<code>TypePath->Expr</code>	インスタンス化。
<code>@:pos(p)</code>	<code>Position</code> を引数に取るタグ	その式の位置情報を`p`に差し替え。

4.2. 時間計測

式マクロの振る舞いや仕様については確認できたので、この節からは式マクロが現実でどう役に立つのかを見ていきます。

プログラムの一部をカジュアルに時間計測したいという場合、ローカル変数に時刻を記録して処理が終わった後の時刻の差分をとるというコードをよく書きます。

```
class BenchmarkSample {
    static function main() {
        var time = Date.now().getTime();

        // 何か時間のかかる処理
        for (i in 0...100000) {}

        trace((time - Date.now().getTime()) + "ms");
    }
}
```

しかし、何度も書くには長くて面倒です。そこで次のようなマクロを定義しておくと、簡単に時間の計測が行えるようになります。

```
import haxe.macro.Expr;

class ExprMacro {
    public static macro function bench(target:Expr):Expr {
        return macro {
            var time = Date.now().getTime();
            $target;
            trace((time - Date.now().getTime()) + "ms");
        }
    }
}
```

これにより元の時間計測のコードを、以下の関数呼び出しの形式で書き換えることができます。

```
static function main():Void {
    ExprMacro.bench(
        for (i in 0...100000) {}
    );
}
```

面倒な記述はなくなり簡単に時間計測ができるようになりました。

4.2.1. Tips: 式のデバッグ方法

自分が書いた式マクロが正しい式を生成できているのか確認するには、`haxe.macro.Printer`が便利です。`haxe.macro.Printer`は式や型のインスタンスをHaxeのコードの文字列に変換するモジュールです。

4.2.2. Tips: staticでないマクロ

HaxeのマニュアルやGithubなどで見つけれられるほとんどの式マクロはstaticとして定義されているので、式マクロはstaticな関数としてのみ定義できると勘違いされがちですが、実際はそうではありません。

以下のようにstaticでない式マクロを定義することもできます。

```
import haxe.macro.Expr;

class NonStaticSample {
    public function new() {}

    #if !macro
    public static function main() {
        var array = new NonStaticSample().test();
    }
    #end

    private macro function test(self:Expr):Expr {
        return macro [$self, $self];
    }
}
```

この場合、上記の例のように、staticでない式マクロを定義されている引数より1つ少なくして呼び出します。こうすると、`.test()`の左側の式が第一引数として受け取られます。つまり、`new NonStaticSample().test()`は、`[new NonStaticSample(), new NonStaticSample()]`に変換されています。

4.3. ローカル変数のデバッグトレース

バグについての調査を行うとき、ある時点での変数の状態をまとめて知りたいことがあります。このような場合、マクロを使ってローカル変数をまとめてトレースできるようにしておくと便利です。

Haxeではマクロの呼び出し箇所で定義されているローカル変数の一覧を`Context.getLocalTVars()`関数で取得できます。これを使って以下のようなマクロを定義しておきます。

```
import haxe.macro.Context;
import haxe.macro.Expr;

class DebugMacro {
    public static macro function debug() {
        var exprs:Array<Expr> = [];
        for (tvar in Context.getLocalTVars()) {
            // 変数strに"変数の名前 : 変数の中身"の文字列を追加する式を生成
            var expr = macro str += ${tvar.name} + " : " + ${tvar.value} + "\n";
            exprs.push(expr);
        }

        // 呼び出し元の関数名を取得
        var methodName = Context.getLocalMethod();

        // 変数strを定義して、用意した式の配列をブロック式化する
        return macro {
            var str = "Called from " + ${methodName} + "\n";
            $b{exprs}
            trace(str + "-----\n");
        };
    }
}
```

そして、このdebug関数を次のように呼び出してみます。

```
class DebugMacroSample {
    public static function main() {
        test(100);
    }

    public static function test(hoge:Int) {
        var fuga = "ok";
        DebugMacro.debug();
    }
}
```

結果は、次の通りです

```
DebugMacroSample.hx:20: Called from test
```

```
fuga : ok  
hoge : 100  
-----
```

呼び出し元であるtest関数のローカル変数の一覧を表示することができました。これらに合わせてthisインスタンスのフィールドについてもあわせて出力するようにすれば、バグ発生時の状況を調べるための強力なツールになります。

4.3.1. Tips: 出力ターゲット側のデバッグ機能

Haxeではターゲット側のデバッグ機能もサポートされているものが多いので、そちらも使うとバグの調査がはかどります。例えば、Flashターゲットの場合はFlashDevelopではステップ実行やブレークポイントがサポートされています。JavaScriptの場合は、`js.Lib.debug()`関数でブレークポイント(debuggerステートメント)が使えるたり、ソースマップで実行エラーなどの発生行がHaxeのソースコード上の位置でわかったりします。

4.3.2. Tips: エラーの記述

式マクロの記述をする場合は、引数で与えられた式についてなるべく丁寧にエラー処理を記述するのが重要です。式マクロでは、エラーになるべき式がエラーになっていないとデバッグがとても辛くなります。冒頭のサンプルで紹介した通り、マクロからは警告やエラーが発生させられますので積極的に使うといいです。

ただし、HaxeのコンパイラはUTF-8の文字列の出力に対応しておらず、日本語でエラーを出力をすると(少なくともWindowsでは)文字化けを起こすので注意が必要です。

Chapter 5. ビルドマクロ

ビルドマクロはクラスへの変数や関数の追加や削除を行うマクロです。クラスにメタデータタグを付けて呼び出すことができます。

5.1. 定数を自動生成する

ビルドマクロの典型的な使用例として、定数フィールドの自動生成があります。以下は、コンパイル時にフォルダ内のファイルを検索して、そのファイル名を定数として定義するサンプルです。

```
import haxe.macro.Context;
import haxe.macro.Expr;
import haxe.macro.Printer;
import sys.FileSystem;

class BuildMacro {
    public static function addFileNames(directory:String):Array<Field> {
        var fields:Array<Field> = [];

        // ディレクトリ内のファイルに対してループ処理
        for (fileName in FileSystem.readDirectory(directory)) {
            // ファイル名を表す定数の式を作成
            var expr = macro ${fileName};

            // フィールドを定義して追加。
            // public static inline var 大文字ファイル名 = "ファイル名";
            // の意味になる
            fields.push({
                name : StringTools.replace(fileName, ".", "_").toUpperCase(),
                access : [Access.APublic, Access.AStatic, Access.AInline],
                // 型にnullを指定すると推論をさせる。値はファイル名を表す定数の式
                kind : FieldType.FVar(null, expr),
                // 位置情報は関数の呼び出し元のものを使う
                pos : Context.currentPos(),
                // ドキュメントコメントの追加
                doc : new Printer().printExpr(expr),
            });
        }

        return fields;
    }
}
```

これをクラスに`@:build`のメタデータをつけて呼び出します。

```
@:build(BuildMacro.addFileNames("./assets"))
class Constants {}
```

これにより、コンパイル時のワーキングディレクトリから`./assets`の位置にあるディレクトリを探索して、その直下にあるファイル名の定数が`Constants`の`static`フィールドとして生成されます。これは次のように利用できます。

```
class ConstantsSample {
  public static function main() {
    trace(Constants.SAMPLE_TXT); // ConstantsSample.hx:3: sample.txt
  }
}
```

これは、単純に`"sample.txt"`を文字列リテラルで使うのよりも手間がかかっているように見えるかもしれませんが、定数化には2つのメリットがあります。

1つ目は「存在しないファイル名を指定しようとする」とコンパイルエラーになるということです。これによりタイポが防げますし、ファイル名を変更した場合にもコード側でどこを修正すれば良いかすぐにわかります。

2つ目は「エディタ上でのコード補完が効く」ようになることです。これはHaxeコンパイラ自体がエディタの補完用の機能を提供していて、多くのIDEやエディタはそれを使っているためです。つまり、マクロによるフィールドの追加が行われた上で補完がされます。このため長いファイル名を入力しなければならない場合でも、わざわざ目で確認したりコピペしたりせずに簡単に入力ができるようになります。

このような`@:build`で定数を自動で生成する方法はファイル名だけでなく、JSON、CSV、HTML、CSSのデータを元に生成したりなどさまざま利用方法があります。

5.1.1. Tips: マクロとドキュメント生成

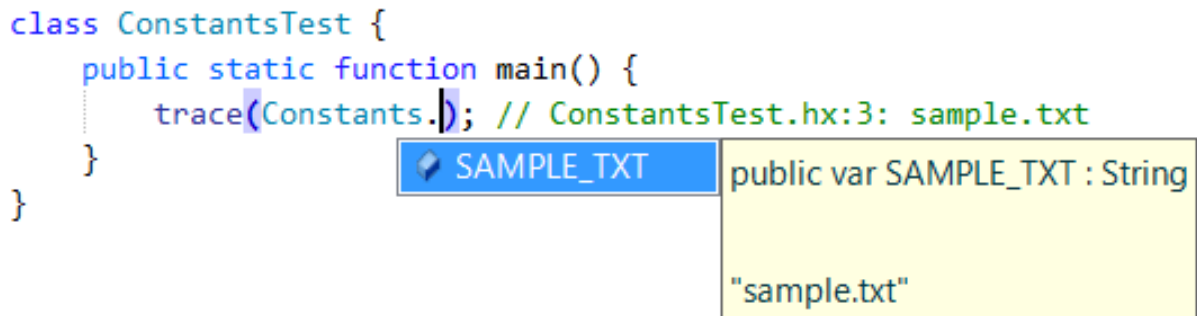
JavaのJavadoc、JavaScriptのJSDocに当たる、いわゆるドキュメント生成ツールとして`haxedoc`や`dox`があります。これらのツールでドキュメント生成を行った場合、ビルドマクロを使って追加したフィールドもちゃんと出力に含まれます。これは、ドキュメント生成用のxml出力もHaxeのコンパイラが持っている機能が使われるためです。

ですから多くのフィールドをマクロで生成して、それらを一覧で確認したいような場合は、doxなどのドキュメント生成を使うのが良いかもしれません。

また、ドキュメントコメントをビルドマクロから差し込むこともできます。複雑な式を生成した場合、生成したExprインスタンスをhaxe.macro.Printerで文字列に変換してそのままドキュメントコメントとして使ってしまうと、実際にどのような式が生成されているかを可視化できて便利です。

これは、先ほどの定数生成でもやっています。

このようにして追加したドキュメントコメントは、ただドキュメント生成で使えるだけでなく、コンパイラの補完機能を利用しているIDE上でも表示されます。



```
class ConstantsTest {
    public static function main() {
        trace(Constants.); // ConstantsTest.hx:3: sample.txt
    }
}
```

SAMPLE_TXT public var SAMPLE_TXT : String
"sample.txt"

5.1.2. Tips: #if display

入力補完にマクロの実行結果が反映されるということは、重たい処理をマクロで行うとそれだけ入力補完が遅くなるということです。補完が遅くなるのを防ぎたい場合、displayの条件フラグが役に立ちます。

重たいマクロのコードは#if !display~#endで囲んでおくと、Haxeの補助機能ではその範囲のコードが無視されます。

5.2. 関数の呼び出しをトレースする

もう一つビルドマクロの実用例として、クラスの間数すべての先頭に関数名と引数の内容のtrace呼び出しを追加するマクロを紹介します。このようなマクロを定義しておくと、関数がどの順番で呼び出されているのかを簡単に追いかけることができます。

```
import haxe.macro.Context;
import haxe.macro.Expr.Field;
import haxe.macro.Expr.FieldType;
import haxe.macro.Type.FieldKind;
```

```
class BuildMacro {
public static function methodTrace():Array<Field> {
    // すでに定義されているフィールドを取得
    var fields = Context.getBuildFields();

    for (field in fields) {
        switch (field.kind) {
            case FieldType.FFun(func):
                // trace用の式を準備
                var traceArg = macro "auto trace: " + $v{field.name} + "(";

                // trace用に引数も追加
                var first = true;
                for (arg in func.args) {
                    if (!first) {
                        traceArg = macro ${traceArg} + ",";
                    }
                    traceArg = macro ${traceArg} + $i{arg.name};
                    first = false;
                }

                traceArg = macro ${traceArg} + ")";

                // 元の式の実行前にtrace文を差し込む
                func.expr = macro {
                    trace(${traceArg});
                    ${func.expr};
                }

            case _:
                // 関数以外には何もしない。
        }
    }

    return fields;
}
}
```

これを以下のように使います。

```
@:build(BuildMacro.methodTrace())
class TraceSample {
public static function main():Void {
    for (i in 0...2) {
```

```
    for (j in 0...3) {
        test(i, j);
    }
}

public static function test(i:Int, j:Int):Void {}
}
```

実行結果は以下の通りです。

```
BuildMacro.hx:31: auto trace: main()
BuildMacro.hx:31: auto trace: test(0,0)
BuildMacro.hx:31: auto trace: test(0,1)
BuildMacro.hx:31: auto trace: test(0,2)
BuildMacro.hx:31: auto trace: test(1,0)
BuildMacro.hx:31: auto trace: test(1,1)
BuildMacro.hx:31: auto trace: test(1,2)
```

この例ではただ単に関数名を出力しているだけですが、より詳細な記録をすれば、呼び出し関数の多いクラスを調べたり、実行時間の長い関数を発見したりなど、さまざまなプロファイリングに応用できます。

また、このようなビルドマクロは、初期化マクロから `haxe.macro.Compiler` の `addGlobalMetadata` 関数で、パッケージ内のクラスに対して一括でビルドマクロの適用を行うことができます。

5.2.1. Tips: コンパイルにかかった時間を計測する

マクロの処理にかかっている時間を知りたい場合、`--times` のコンパイラ引数をつけるとコンパイルの各処理にかかった時間が出力されるようになります。さらに `-D macro_times` のオプションをマクロの各処理の時間についての内訳が表示されるようになります。

Chapter 6. イベントハンドラ

初期化マクロ、式マクロ、ビルドマクロからイベントハンドラの登録をすることで、より後のタイミングでの処理をさせることができます。

`onGenerate`はすべての型の構文解析と型付けが終わった後に実行されます。ここではすべての型の情報（型付け済みの抽象構文木を含む）を配列でうけとることができます。`onAfterGenerate`はさらに後に実行されて、出力後のファイルにアクセスできます。

6.1. Linterを作る(`onGenerate`)

`onGenerate`で登録したハンドラには、コンパイル対象に含まれたすべての型が引数として渡されます。この型から、すべての型付け済みのASTにアクセスすることができますが、このASTに対する変更はメタデータタグの変更に限られています。

`onGenerate`のタイミングでできることとしては、以下のような例が挙げられます。

- メタデータタグや`Context.addResource`で、文字列やバイナリを埋め込む。
- `Type`の情報を解析して、コンパイラ警告やエラーを出力する。

ここでは`Type`の情報をもとにコンパイラ警告を発生させる。いわゆるLinterの作成方法を紹介します。

以下は、変数名がローワーキャメルケースであることをチェックするLinterです

```
import haxe.macro.Context;
import haxe.macro.Type;

class Linter {
    // 初期化マクロとして呼び出す用
    public static function initialize():Void {
        Context.onGenerate(lint);
    }

    private static function lint(types:Array<Type>):Void {
        for (type in types) {
            switch (type) {
                case Type.TInst(ref, _):
                    var classType = ref.get();
                    lintFields(classType.statics.get());
                    lintFields(classType.fields.get());
            }
        }
    }
}
```

```
case Type.TAbstract(ref, _):
    var abstractType = ref.get();
    lintFields(abstractType.array);

case _:
}
}
}

// フィールドに対するチェック
private static function lintFields(fields:Array<ClassField>):Void {
    for (field in fields) {
        switch (field.kind) {
            case FieldKind.FVar(VarAccess.AccInline, _):
                // インライン変数をチェックから除外

            case _:
                // フィールド名のケースがおかしくないか判定。
                if (!isValidFieldName(field.name)) {
                    Context.warning("should be lower camlcase", field.pos);
                }
        }
    }
}

// 変数名がローワーキャメルケースであることのチェック
private static function isValidFieldName(name:String):Bool {
    if (StringTools.startsWith(name, "get_") || StringTools.startsWith(name, "set_")) {
        // getter、setter用のサフィックスを除外
        name = name.substr(4);
    } else {
        // 先頭の_は使用可
        while (name.substr(0, 1) == "_") {
            name = name.substr(1);
        }
    }

    if (name.length == 0) { return false; }

    // スネークケースでないことのチェック
    if (name.indexOf("_") != -1) { return false; }

    // 小文字始まりであることのチェック
    var charCode = name.charCodeAt(0);
    if (charCode < 97 || 122 < charCode) { return false; }
```

```
    return true;
  }
}
```

これを例えば、以下のようなクラスと合わせて使います。

```
class LintSample {
  public static function main():Void {
    Test();
    test_test();
  }

  // 大文字始まり
  public static function Test():Void {}

  // スネークケース
  public static function test_test():Void {}
}
```

これに対して、以下のような警告が発生します。

```
LintSample.hx:10: characters 15-38 : Warning : should be lower camlcase
LintSample.hx:13: characters 15-43 : Warning : should be lower camlcase
```

実際にはこのコードだと`Math.NaN`などの標準ライブラリに対しても警告を出してしまうので、対象パッケージの限定などの工夫が必要になりますが、この方法を応用していけば循環的複雑度の検査などさまざまな静的コード解析を行うことができます。

6.1.1. Tips: Typeとhaxe.macro.Type

これまで、`haxe.macro.Type`というモジュールの`import`を使っていますが、これとは別にHaxeのライブラリにはトップレベルに`Type`というモジュールがあります。この両方を使用する場合、単純に`haxe.macro.Type`を`import`してしまうと、トップレベル`Type`は使えなくなってしまいます。これを回避する方法は、2通りあります。

- `haxe.macro.Type`を`import`せずに毎回フルパス指定で使う。
- `import haxe.macro.Type in MacroType`というように別名での`import`を使う。

6.2. 出力にライセンス情報を追加する (onAfterGenerate)

onAfterGenerateが動作するのはすでに出力が終わったあとです。ですから、これまでのコンパイルに介入するということとはできませんが、その代わりに出力ファイルを直接読み込んだり、書きこんだりができます。

onAfterGenerateが役に立つ例としては、出力したファイルへのライセンス情報を記述があります。

以下はjsターゲットの出力ファイルの先頭にライセンスについてのコメントを書き込むサンプルです。

```
import haxe.macro.Compiler;
import haxe.macro.Context;
import sys.io.File;

class LicenseWriter {
    // 初期化マクロとして呼び出す用
    public static function initialize():Void {
        Context.onAfterGenerate(write);
    }

    private static function write():Void {
        var fileName = Compiler.getOutput();
        var comment = "/*This is MIT License.*\n";

        File.saveContent(fileName, comment + File.getContent(fileName));
    }
}
```
