Javaの イケてるコード 残念なコード

5カ条で学ぶ 性能を考慮したJavaコードを書く技術

Acroquest Technology
谷本 心

この連載では、Javaプログラミングの改善方法やテクニックについて、Javaの良いコード（イケてるコード）として「読みやすいコード」「拡張しやすいコード」を書く技術を解説してきました。良いコードが書けるようになったら次に進みましょう。次のステップは、プログラムの性能や堅牢性を考慮することが大切になります。

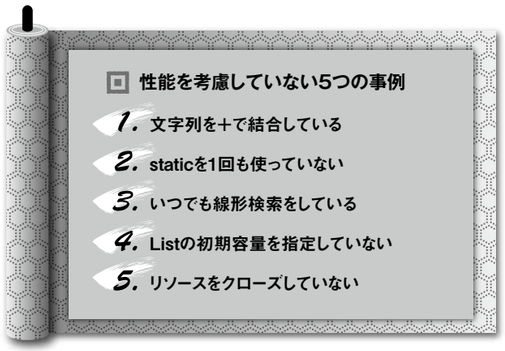
性能が悪いコードは、簡単なプログラムであれば問題になりません。しかし、本格的なプログラムでは、扱うデータ量が多くなる、プログラムの動作時間が長くなる（常駐サービスなどの終了しないプログラムの場合）、という要因から応答性が悪くなってしまったり、最悪の場合はプログラムが異常終了してしまったりします。また、堅牢性が低いコードでは、プログラムが意図しない個所で終了してしまう、問題発生時に原因を調査できない、といった障害などに耐性が低いコードになってしまいます。

そこで、今回と次回の2回に分けて「性能や堅牢性を考慮したコード」について、すぐに使えるテクニックを解説します。

あなたのコードは大丈夫?

性能低下を招く5つの例を紹介します（**図1**）。あなたのコードは大丈夫ですか。図1のコードになっていませんか。当てはまるものがあったら、これから説明する改善方法を実践してください。

図1●プログラムの性能低下を招く5つのポイント



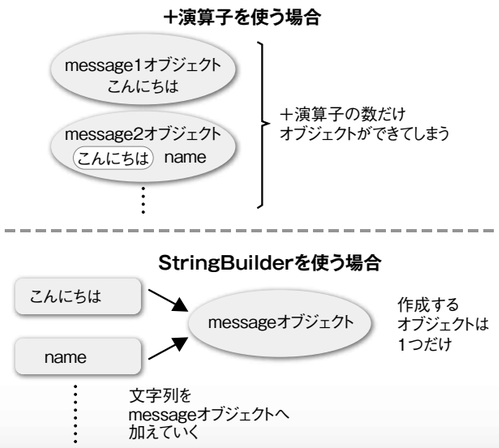
文字列を+で結合している

文字列を+で結合するというコードは、他の言語でプログラミングを学んだことがあり、Javaにあまり親しみがないプログラマがよくやってしまう間違いです。

あなたは、**リスト1**のようなコードを書いていませんか。文字列を連結するのに+演算子はとても便利です。このコードは正しくコンパイルできるため、+を使って何が悪い、と思うかもしれません。ですが、このようなコードを書いていると、プログラムの規模が大きくなったときにきっと後悔します。

この問題点は、Stringを+で結合すると、毎回Stringのオブジェクトを生成してしまうことです。オブジェクトは生成すればするほどメモリーを無駄に使ってしまいます。少し大げさに言うと、**リスト2**と同じ処理です。リスト2のように書いても同じ効果が得られます。1行で書くと、Javaのコンパイラがコンパイル時に、自動的にStringBuilderを使ったコードと同じ処理に置き換えてくれるのです。なかなかJavaコンパイラは賢いですね。ですが、日頃の習慣として文字列の連結には、StringBuilderを使うように心掛けましょう。

図2●+演算子では複数のStringオブジェクトを生成してしまう



リスト1●+演算子を使って文字列を追加したプログラム

```
String message = "こんにちは ";
message += name;
message += "さん、今回は ";
message += count;
message += "回目のアクセスですね! ";
```

+演算子を使って文字列を連結するコード

リスト2●リスト1と同じ処理を記述したプログラム

```
String message1 = "こんにちは ";
String message2 = new String(message1 + name);
String message3 = new String(message2 + "さん、今回は ");
String message4 = new String(message3 + count);
String message5 = new String(message4 + "回目のアクセスですね! ");
```

いちいちオブジェクトを生成してしまう

リスト3●リスト1をStringBuilderで書き直したプログラム

```
StringBuilder message = new StringBuilder("こんにちは ");
message.append(name);
message.append( "さん、今回は ");
message.append(count);
message.append( "回目のアクセスですね! ");

return message.toString();
```

生成するオブジェクトは1つだけで済む

```
String message = "こんにちは "
+ name + "さん、今回は "
+ count+ "回目のアクセスですね! ";
```

staticを1回も使っていない

mainメソッドがstaticメソッドなので、アクセス修飾子のstaticを1度は使っているはずです。ここでの意味は、main以外でstaticを使ったことがないという場合です。プログラムを書いていると、便利なメソッドは共通化して利用したり、再利用したりするものです。このように処理をまとめたクラスを一般的にユーティリティクラスと呼びます。例えば、**リスト4**です。何が問題になるでしょうか。

この問題点は、メソッドをstaticにしていないことです。ユーティリティクラスのメソッドのように様々なオブジェクトから利用するメソッドは、staticを付けてスタティックメソッドとして定義しておくべきです。メソッドをstaticにしなければ、呼び出し側ではユーティリティクラスのインスタンスをわざわざ作成する必要があります。この処理は、面倒ですし、わずかとはいえメモリーも消費します。呼び出し元のコードは、次のようになるでしょう。

リスト4●ユーティリティクラスを定義したプログラム

```
public class FileUtil {
  public List<String> readLines(String fileName) {
    // 省略
  }
}
```

```
List<String> lines =
  new FileUtil().readLines("test.txt");
```

ここでFileUtilクラスのインスタンスをわざわざ作成しているところが無駄です。無駄を省くために、ユーティリティクラスのメソッドは、staticで宣言しましょう。**リスト5**のようにstaticで宣言しておけば、呼び出し側のメソッドではインスタンスを生成する必要がなくなります（**図3**）。すると先ほどの呼び出し元のコードは、

```
List<String> lines =
  FileUtil.readLines("test.txt");
```

のようにインスタンス化する必要がなく、コードをすっきりと記述でき、インスタンス化しないぶんメモリー使用量の削減にもつながるのです。