最後の行問題

この記事は、インテル® デベロッパー・ゾーンに公開されている「The Last Line Effect」の日本語参考訳です。

コピーペーストを使用して引き起こされる多くのエラーを調査した結果、プログラマーが最も多くミスを引き起こしているのはホモジニアスなコードブロックの最後の部分であることが分かりました。プログラミングに関する書籍でこの現象についての記述をこれまで見たことがないため、ここに記しておくことにします。私は、この現象を「最後の行問題」と呼んでいます。

はじめに

私 (Andrey Karpov) は、スタティック・アナライザーを使用してさまざまなアプリケーションのプログラムコードを解析し、発見したエラーや不具合の説明を記録するという、変わった作業を行っています。この作業を行っている理由は、弊社のツール PVS-Studio および CppCat を宣伝するという実用的/金銭的なものです。スキームは非常に単純です。まず、バグを発見します。次に、そのバグを記事にします。その記事が弊社の潜在的な顧客の目に止まります。そして、顧客から仕事の依頼があります。しかし、今回の記事はアナライザーに関するものではありません。

さまざまなプロジェクトの解析を行ったときに、私は見つけたバグと対応するコードを特別なデータベースに保存しています。興味のある方はだれでも、このデータベースを見ることができます。データベースは html ページのコレクションに変換され、弊社 Web サイト (英語) の「Detected errors」セクションにアップロードされています。

このデータベースはほかに類を見ないものです。現在、1,800 のコードとエラーが含まれており、プログラマーによる研究で、これらのエラー間に特定のパターンがあることが明らかにされるのを待っています。将来の研究、マニュアル、記事のベースとして役立つかもしれません。

私は、これまで収集した情報について特別な調査を行ったことはありませんでした。しかし、明らかにあるパターンが出現していることに気付き、そのパターンをより深く調査することにしました。私は、多くの記事で、「最後の行に注意してください」というフレーズを書く必要がありました。これには何か理由があると考えたのです。

最後の行問題

プログラムを記述するとき、プログラマーは一連の似た構造のコードを記述しなければいけないことがよくあります。同じコードを何度も入力することは退屈で非効率です。そのため、プログラマーは、コピーペース

トを使用し、コードの一部をコピーしてほかの部分に貼り付けてから、コードを編集することがよくあります。この方法の悪い点は明らかです。貼り付けた行の一部を変更することを忘れると、エラーが発生するリスクがあることです。しかし、現時点でほかに良い選択肢はありません。

では、私が発見したパターンについてお話しましょう。私は、ほとんどのミスは最後に貼り付けたコードのブロックで起こっていることを突き止めました。

次に簡単な短い例を示します。

```
inline Vector3int32& operator+=(const Vector3int32& other) {
  x += other.x;
  y += other.y;
  z += other.y;
  return *this;
}
```

行 "z += other.y;" に注目してください。プログラマーは、'other.y' を 'other.z' に変更することを忘れています。

この例はこの記事用に作成したサンプルであると思われるかもしれませんが、このコードは実際のアプリケーションで実際に使用されていたものです。この記事を最後まで読むと、この問題が頻繁に発生する一般的な問題であること、そして「最後の行問題」と呼んでいる理由が分かるでしょう。プログラマーは、同様の編集作業の最後によくミスをしているのです。

私は、山頂まで残りわずかな距離で登山者が減るという話を聞いたことがあります。彼らは疲れていたのではありません。山頂にほぼ到達したことで、つい気が緩んだ - 目前の勝利に酔い、注意力が低下して、致命的なミスを犯した - だけなのです。私は、同様のことがプログラマーにも起こっていると考えています。

では、具体的に説明しましょう。

バグ・データベースを調べた結果、コピーペーストを使用して記述されたコードは全部で 84 ありました。そのうち、41 のケースでは、コピーペーストされた (最後を除く) ブロックにミスが含まれていました。次に例を示します。

```
strncmp(argv[argidx], "CAT=", 4) &&
strncmp(argv[argidx], "DECOY=", 6) &&
strncmp(argv[argidx], "THREADS=", 6) &&
strncmp(argv[argidx], "MINPROB=", 8)) {
```

"THREADS=" 文字列の長さは 6 ではなく 8 です。

その他の 43 のケースでは、最後にコピーされたコードブロックでミスが見つかりました。

この 43 という数字は 41 よりもわずかに多いだけに見えます。しかし、多くのホモジニアス・ブロック (1 つ目、2 つ目、5 つ目、10 番目など) で見つかったミスをすべて合わせた数が 41 であることに注意してください。つまり、ミスは最後を除くブロック全体では滑らかに分布していて、最後のみ突出しているのです。

ホモジニアス・ブロックの数は平均で 5 でした。

最初の 4 ブロックにわたって 41 のミスが含まれていたと考えると、ブロックごとのミスの数は約 10 になります。

そして、5 つ目のブロックのミスの数は 43 になります。

分かりやすくなるように、図にしてみましょう。

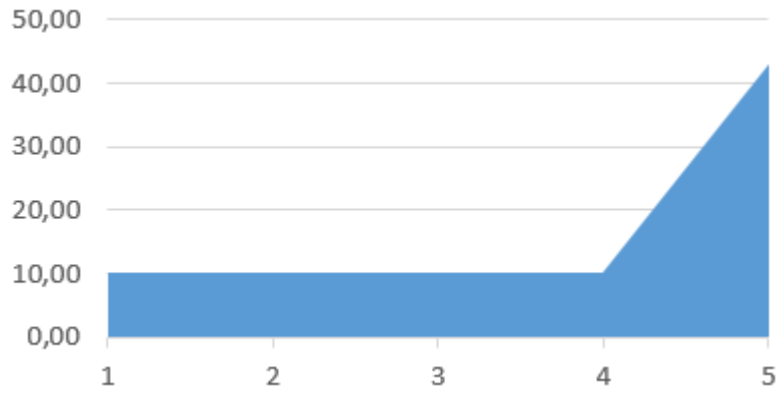

図 1. 5 つのホモジニアス・ブロックにおけるミスの分布。

このことから、次のパターンが得られます。

最後にペーストしたコードのブロックでミスをする確率は別のブロックよりも 4 倍高い。

私はここから壮大な結論を導き出そうとしているのではありません。問題の対策 (コードの最後の部分を記述するときに注意する) に役立つであろう興味深い観察にすぎません。

例

では、この現象が私の想像ではなく、実際の傾向であることを説明することにしましょう。私の言葉を証明するため、いくつかの例を示します。

すべての例ではなく、最も単純なものと最も代表的なものを示しています。

Source Engine SDK

```
inline void Init( float ix=0, float iy=0,
                  float iz=0, float iw = 0 )
{
  SetX( ix );
  SetY( iy );
  SetZ( iz );
  SetZ( iw );
}
```

最後は SetZ() 関数ではなく SetW() 関数を呼び出すべきです。

Chromium*

```
if (access & FILE_WRITE_ATTRIBUTES)
  output.append(ASCIIToUTF16("¥tFILE_WRITE_ATTRIBUTES¥n"));
if (access & FILE_WRITE_DATA)
  output.append(ASCIIToUTF16("¥tFILE_WRITE_DATA¥n"));
if (access & FILE_WRITE_EA)
  output.append(ASCIIToUTF16("¥tFILE_WRITE_EA¥n"));
if (access & FILE_WRITE_EA)
  output.append(ASCIIToUTF16("¥tFILE_WRITE_EA¥n"));
break;
```

最後のブロックと１つ前のブロックが同じです。

ReactOS*

```
if (*ScanString == L'¥"' ||
    *ScanString == L'^' ||
    *ScanString == L'¥"')
```

最初と最後のチェックが同じです。

Multi Theft Auto

```
class CWaterPolySAInterface
{
public:
    WORD m_wVertexIDs[3];
};
CWaterPoly* CWaterManagerSA::CreateQuad (....)
{
  ....
  pInterface->m_wVertexIDs [ 0 ] = pV1->GetID ();
  pInterface->m_wVertexIDs [ 1 ] = pV2->GetID ();
  pInterface->m_wVertexIDs [ 2 ] = pV3->GetID ();
  pInterface->m_wVertexIDs [ 3 ] = pV4->GetID ();
  ....
}
```

最後の行は機械的に貼り付けられたもので冗長です。配列には３つの項目しかありません。

Source Engine SDK

```
intens.x=OrSIMD(AndSIMD(BackgroundColor.x,no_hit_mask),
                AndNotSIMD(no_hit_mask,intens.x));
intens.y=OrSIMD(AndSIMD(BackgroundColor.y,no_hit_mask),
                AndNotSIMD(no_hit_mask,intens.y));
intens.z=OrSIMD(AndSIMD(BackgroundColor.y,no_hit_mask),
                AndNotSIMD(no_hit_mask,intens.z));
```

最後のブロックを "BackgroundColor.y" から "BackgroundColor.z" に置換するのを忘れています。

Trans-Proteomic Pipeline

```
void setPepMaxProb(....)
{
  ....
  double max4 = 0.0;
  double max5 = 0.0;
  double max6 = 0.0;
  double max7 = 0.0;
  ....
  if ( pep3 ) { ... if ( use_joint_probs && prob > max3 ) ... }
  ....
  if ( pep4 ) { ... if ( use_joint_probs && prob > max4 ) ... }
  ....
  if ( pep5 ) { ... if ( use_joint_probs && prob > max5 ) ... }
  ....
  if ( pep6 ) { ... if ( use_joint_probs && prob > max6 ) ... }
  ....
  if ( pep7 ) { ... if ( use_joint_probs && prob > max6 ) ... }
  ....
}
```

最後の条件を "prob > max6" から "prob > max7" に置換するのを忘れています。

SeqAn

```
inline typename Value<Pipe>::Type const & operator*() {
  tmp.i1 = *in.in1;
  tmp.i2 = *in.in2;
  tmp.i3 = *in.in2;
  return tmp;
}
```

最後の代入文は ”*in.in3” にすべきです。

SlimDX

```
for( int i = 0; i < 2; i++ )
{
  sliders[i] = joystate.rglSlider[i];
  asliders[i] = joystate.rglASlider[i];
  vsliders[i] = joystate.rglVSlider[i];
  fsliders[i] = joystate.rglVSlider[i];
}
```

最後の行では rglFSlider 配列を使用すべきです。

Qt

```
if (repetition == QStringLiteral("repeat") ||
    repetition.isEmpty()) {
  pattern->patternRepeatX = true;
  pattern->patternRepeatY = true;
} else if (repetition == QStringLiteral("repeat-x")) {
  pattern->patternRepeatX = true;
} else if (repetition == QStringLiteral("repeat-y")) {
  pattern->patternRepeatY = true;
} else if (repetition == QStringLiteral("no-repeat")) {
  pattern->patternRepeatY = false;
  pattern->patternRepeatY = false;
} else {
  //TODO: exception: SYNTAX_ERR
}
```

最後のブロックに 'patternRepeatX' が含まれていません。正しいコードは次のようになります。

```
pattern->patternRepeatX = false;
pattern->patternRepeatY = false;
```

ReactOS*

```
const int istride = sizeof(tmp[0]) / sizeof(tmp[0][0][0]);
const int jstride = sizeof(tmp[0][0]) / sizeof(tmp[0][0][0]);
const int mistride = sizeof(mag[0]) / sizeof(mag[0][0]);
const int mjstride = sizeof(mag[0][0]) / sizeof(mag[0][0]);
```

'mjstride' 変数は常に 1 に等しくなります。最後の行は次のように記述すべきです。

```
const int mjstride = sizeof(mag[0][0]) / sizeof(mag[0][0][0]);
```

Mozilla* Firefox*

```
if (protocol.EqualsIgnoreCase("http") ||
    protocol.EqualsIgnoreCase("https") ||
    protocol.EqualsIgnoreCase("news") ||
    protocol.EqualsIgnoreCase("ftp") ||          <<<---
    protocol.EqualsIgnoreCase("file") ||
    protocol.EqualsIgnoreCase("javascript") ||
    protocol.EqualsIgnoreCase("ftp")) {          <<<---
```

最後の文字列 "ftp" はすでに比較されています。

Quake-III-Arena*

```
if (fabs(dir[0]) > test->radius ||
    fabs(dir[1]) > test->radius ||
    fabs(dir[1]) > test->radius)
```

dir[2] セルの値がチェックされていません。

Clang

```
return (ContainerBegLine <= ContaineeBegLine &&
        ContainerEndLine >= ContaineeEndLine &&
        (ContainerBegLine != ContaineeBegLine ||
         SM.getExpansionColumnNumber(ContainerRBeg) <=
         SM.getExpansionColumnNumber(ContaineeRBeg)) &&
        (ContainerEndLine != ContaineeEndLine ||
         SM.getExpansionColumnNumber(ContainerREnd) >=
         SM.getExpansionColumnNumber(ContainerREnd)));
```

ブロックの最後の "SM.getExpansionColumnNumber(ContainerREnd)" 式で自身と比較しています。

MongoDB*

```
bool operator==(const MemberCfg& r) const {
  ....
  return _id==r._id && votes == r.votes &&
         h == r.h && priority == r.priority &&
         arbiterOnly == r.arbiterOnly &&
         slaveDelay == r.slaveDelay &&
         hidden == r.hidden &&
         buildIndexes == buildIndexes;
}
```

最後の行の "r." が指定されていません。

Unreal* Engine 4

```
static bool PositionIsInside(....)
{
  return
    Position.X >= Control.Center.X - BoxSize.X * 0.5f &&
    Position.X <= Control.Center.X + BoxSize.X * 0.5f &&
    Position.Y >= Control.Center.Y - BoxSize.Y * 0.5f &&
    Position.Y >= Control.Center.Y - BoxSize.Y * 0.5f;
}
```

最後の行に 2 つミスがあります。1 つ目は、">=" を "<=" に置換するのを忘れています。2 つ目は、マイナス記号をプラス記号に置換するのを忘れています。

Qt

```
qreal x = ctx->callData->args[0].toNumber();
qreal y = ctx->callData->args[1].toNumber();
qreal w = ctx->callData->args[2].toNumber();
qreal h = ctx->callData->args[3].toNumber();
if (!qIsFinite(x) || !qIsFinite(y) ||
    !qIsFinite(w) || !qIsFinite(w))
```

最後の qIsFinite 関数の呼び出しは、'h' 変数を引数として使用すべきです。

OpenSSL*

```
if (!strncmp(vstart, "ASCII", 5))
  arg->format = ASN1_GEN_FORMAT_ASCII;
else if (!strncmp(vstart, "UTF8", 4))
  arg->format = ASN1_GEN_FORMAT_UTF8;
else if (!strncmp(vstart, "HEX", 3))
  arg->format = ASN1_GEN_FORMAT_HEX;
else if (!strncmp(vstart, "BITLIST", 3))
  arg->format = ASN1_GEN_FORMAT_BITLIST;
```

"BITLIST" 文字列の長さは 3 ではなく 7 です。

このあたりで止めておきましょう。すでに十分な例を示せたはずです。

まとめ

この記事では、コピーペーストを使用した場合、最後にペーストしたコードのブロックでミスをする確率が別のブロックよりも 4 倍高くなることを説明しました。

この現象は、技術者のスキルではなく、人間の心理と関係しています。この記事で紹介したように、Clang や Qt のようなプロジェクトの非常にスキルの高い開発者でもこの種のミスをしています。

この観察がプログラマーの注意を促し、弊社のバグ・データベースを調査するきっかけになることを期待しています。その結果、エラーの多くのパターンが明らかにされ、プログラマーに対する新しい推奨事項が提示されるでしょう。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください