JPCERT CC®

Javaセキュアコーディングセミナー東京

第2回
数値データの取扱いと入力値の検証

2012年10月14日(日)

JPCERTコーディネーションセンター
脆弱性解析チーム
久保 正樹，戸田 洋三

- **本資料について**
  - 本セミナーに使用するテキストの著作権はJPCERT/CCに帰属します。
  - 事前の承諾を受けた場合を除いて、本資料に含有される内容（一部か全部かを問わない）を複製・公開・送信・頒布・譲渡・貸与・使用許諾・転載・再利用できません。

- **本セミナーに関するお問い合わせ**
  - JPCERTコーディネーションセンター
  - セキュアコーディング担当
  - E-mail : secure-coding@jpcert.or.jp
  - TEL : 03-3518-4600

本セミナーについて

- 第1回 9月9日（日）
  - オブジェクトの生成と消滅におけるセキュリティ
- 第2回 10月14日（日）
  - 数値データの取扱いと入力値検査
- 第3回 11月11日（日）
  - 入出力(ファイル，ストリーム)と例外時の動作
- 第4回 12月16日
  - メソッドのセキュリティ

- 開発環境持参

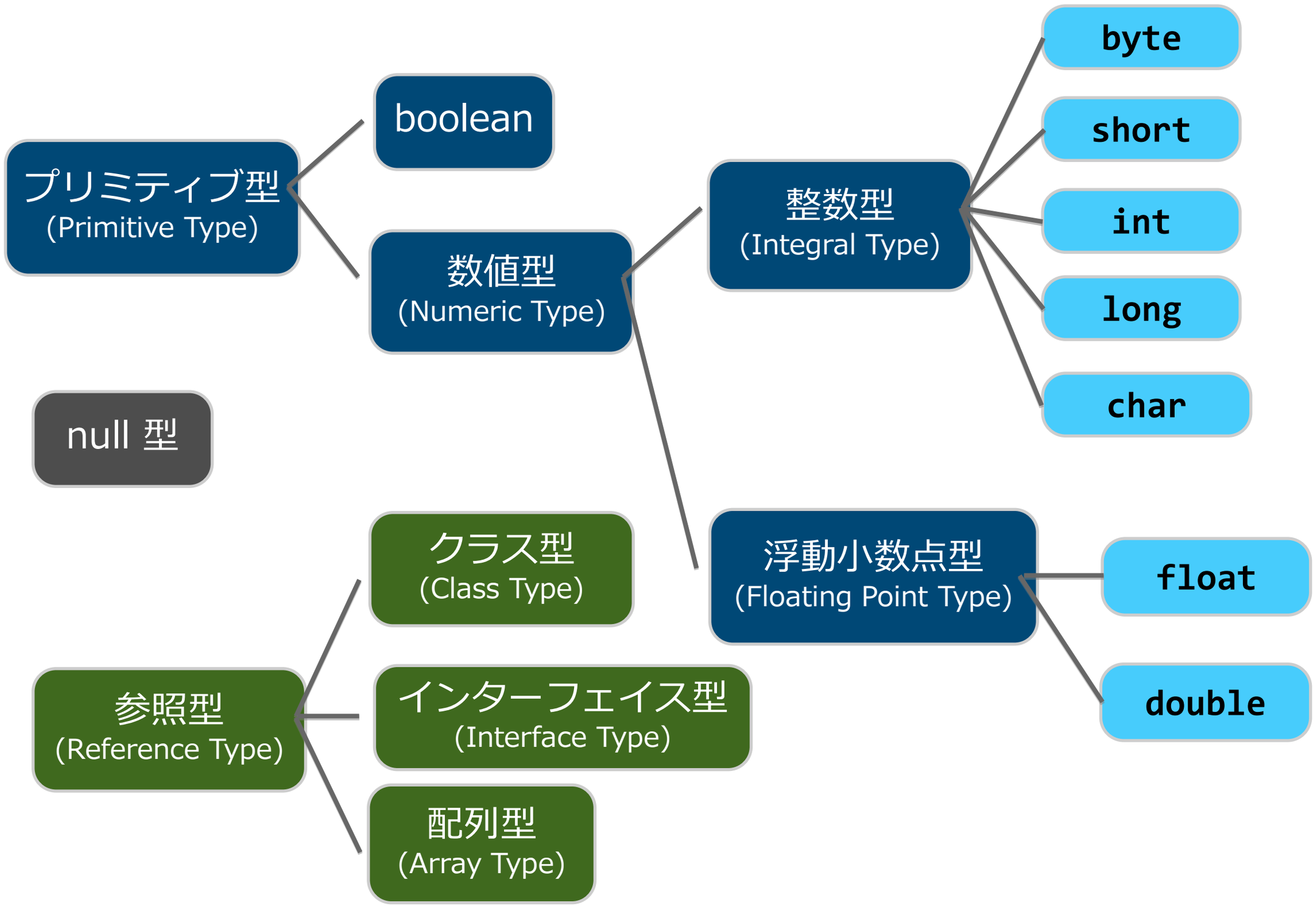
数値データの取扱い

- 「脅威度の高い」
- 脆弱性の
- 85%以上の原因

基本のおさらい

Javaの数値型

Javaの型



整数型

- byte, short, int, long
  - 値は2の補数で表現される

型	ビット幅	最小値	最大値
byte			
short			
char			
int			
long			

2の補数表現

- 負の数は、1の補数に1を加えて表す

- 8ビットの符号付き整数で考えてみる

元の数　　０ ０ １ ０ １ ０ ０ １　　０ ０ １ ０ １ ０ ０ １

(10進の41)

1の補数　　１ １ ０ １ ０ １ １ ０　＋１＝　１ １ ０ １ ０ １ １ １

2の補数
(10進の-41)

浮動小数点型

- float, double
  - IEEE 754 (IEEE 浮動小数点数演算標準)

型	ビット幅	最小値	最大値
float	32	-3.40E+38	3.40E+38
double	64	-1.70E+308	-1.70E+308

型	符号	指数部	仮数部
float	1 bit	8 bit	23 bit
double	1 bit	11 bit	52 bit

- 例： 2.99792458 x 10^8 [m/sec]

デフォルト値

- 宣言されても初期化されていないフィールドは、コンパイラによって適当なデフォルト値を設定される

データ型	デフォルト値
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	‘¥u0000’
String	null
boolean	FALSE

デフォルト値

- ただし例外がある
  - コンパイラは、未初期化のローカル変数にはデフォルト値を設定しない
    - 未初期化のローカル変数へのアクセスはコンパイルエラーになる

整数演算は黙ってオーバーフローする

コード1

パズル3：長除法

```
public class LongDivision {
     public static void main(String[] args) {
          final long MICRO_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
          final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;

          System.out.println(MICRO_PER_DAY / MILLIS_PER_DAY);
     }
}
```

- 一日あたりのミリ秒とマイクロ秒数を計算するコード
- マイクロ秒数は
  - 24時間 * 60分 * 60秒 * 1,000ミリ秒 * 1,000マイクロ秒
- ミリ秒数は、最後の * 1,000がないだけ

乗算演算の結果

long MICRO_PER_DAY = 24 * 60 * 60 * 1000 * 1000	86,400,000,000 (0x141DD76000)
long MILLIS_PER_DAY = 24 * 60 * 60 * 1000	86,400,000
intの最大値	2,147,483,647 (0x7FFFFFFF)
longの最大値	9,223,372,036,854,780,000

- MICRO_PER_DAY も MILLIS_PER_DAY もlong型。かけ算の結果を格納するには十分なサイズ
- MICRO_PER_DAY / MILLIS_PER_DAY は、1000を返すはず

- でも実行結果は、5が表示される。なぜ？

整数オーバーフロー

```
final long MICRO_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
```

- 右辺式は、32bitのintの幅で演算される
  - 0x141DD76000 (数学的かけ算の結果)
  - 0x7FFFFFFF (intの最大値)
  - 0x1DD76000 (実際に得られた値、10進の500,645,080)
    - この値をワイドニング変換し、long型の変数に代入している

- MICRO_PER_DAY / MILLIS_PER_DAY は 500,645,080 ÷ 86,400,000 の演算になり、結果は5になる

整数オーバーフローと演算子

整数オーバーフローが発生しうる演算子			
+	-=	<<	<
-	*=	>>	>
*	/=	&	>=
/	%/	¥	<=
%	<<=	^	==
++	>>=	~	!=
--	&=	!	
=	|=	unary +	
+=	^=	unary -	

なぜint幅でかけ算が行われるのか

```
final long MICRO_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
```

- 数値リテラルの型は、末尾にLかlがある場合、long 型、それ以外の場合は int 型になる
  - なので、右辺式のオペランドはぜんぶ int 型
- int * int の結果は int
- Javaは対象型付け（target typing）な言語ではない
  - 結果を保存する変数の型に、計算の型は影響を受けない

- 整数オーバーフローの発生を防ぐあるいは検知するコードを書くことが重要！

コード1の修正

```
public class LongDivision {
    public static void main(String[] args) {
        final long MICRO_PER_DAY = 24L * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24L * 60 * 60 * 1000;

        System.out.println(MICRO_PER_DAY / MILLIS_PER_DAY);
    }
}
```

- かけ算の最初の値をlong型の数リテラルにする
  - 式中のすべての計算がlongで行われる

整数のラップアラウンドに注意

- Javaの整数は、オーバーフローの発生を通知せず、黙ってラップアラウンドします
  - 整数演算の結果が与えられた型で表現できない場合におこる
    - 計算間違いや予期せぬプログラムの動作につながる

- オーバーフローを検知・防止することが重要です！
  - 事前条件のテスト
  - アップキャスト
  - `BigInteger`
    - 参考：NUM00-J. 整数オーバーフローを検出あるいは防止する（https://www.jpcert.or.jp/java-rules/num00-j.html ）

符号拡張とゼロ拡張

数値の格上げ

- 算術演算子のオペランドに適用される
  - 演算子のオペンラドの型が異なると計算できないので、共通の型にあわせるために行う

> シフト演算子以外の整数演算子のオペランドのいずれかがlong型である場合、演算は64ビットの精度で行われ、演算結果はlong型になる。この際、他方のオペランドがlong型でなければ数値の格上げ変換によってlong型へとワイドニング変換される。さもなければ、演算は32ビットの精度で行われ、演算結果はint型になる。

JLS 4.2.2 整数演算

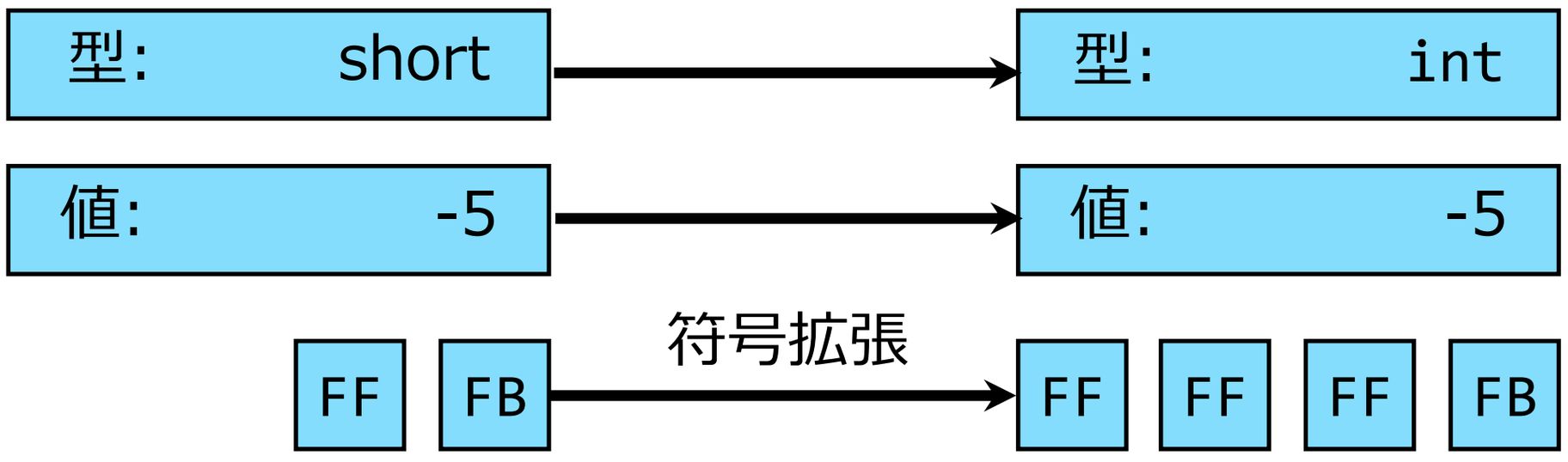
符号拡張とゼロ拡張

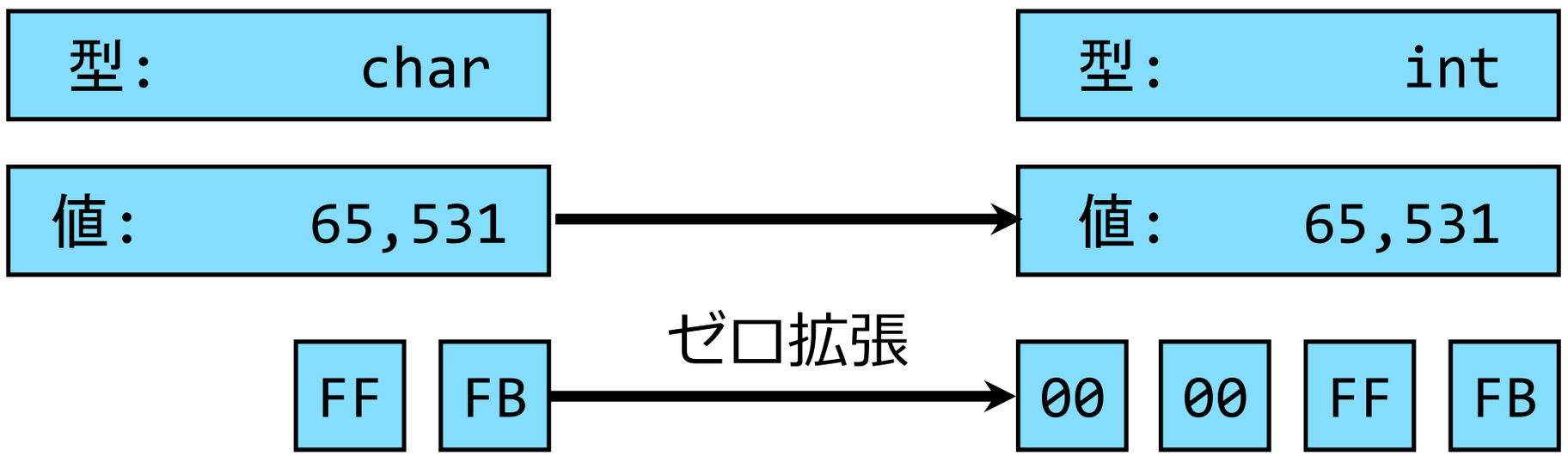
- ある型の値を、より大きな型に変換するときに発生する
- 変換元の型に依存：
  - 符号付きのとき：符号拡張
  - 符号無し(つまりchar型)のとき：ゼロ拡張

- 符号拡張
  - 上位ビットを符号ビットで埋める
  - 例：int i = (short) c;　// c は char型

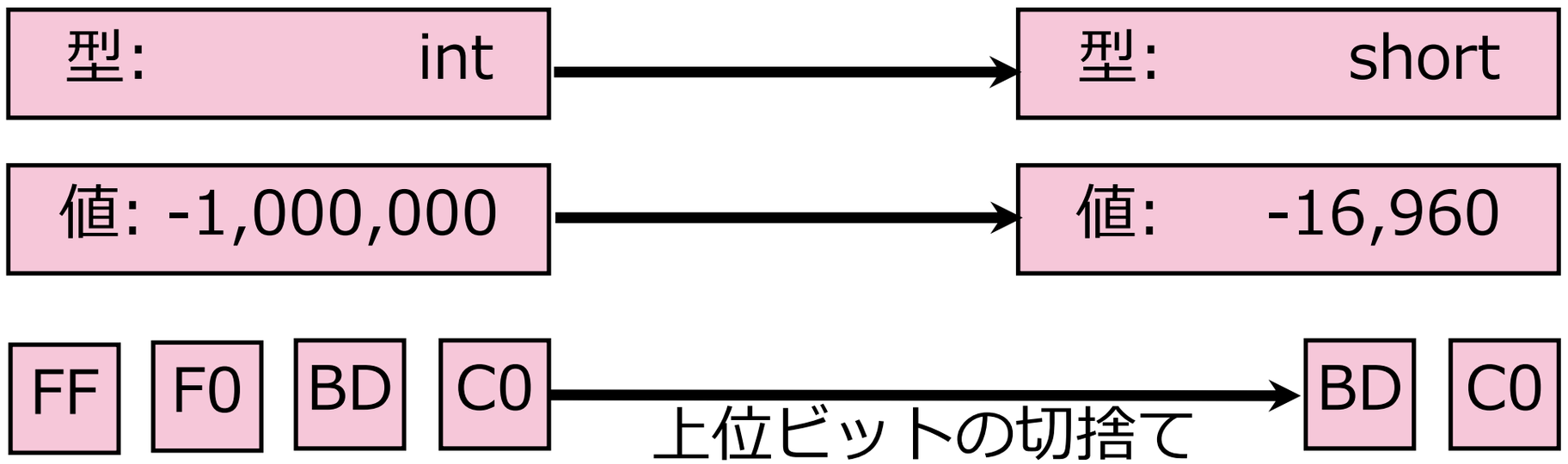
- ゼロ拡張
  - 上位ビットをゼロで埋める
  - 例：long l = c;

ワイドニング変換

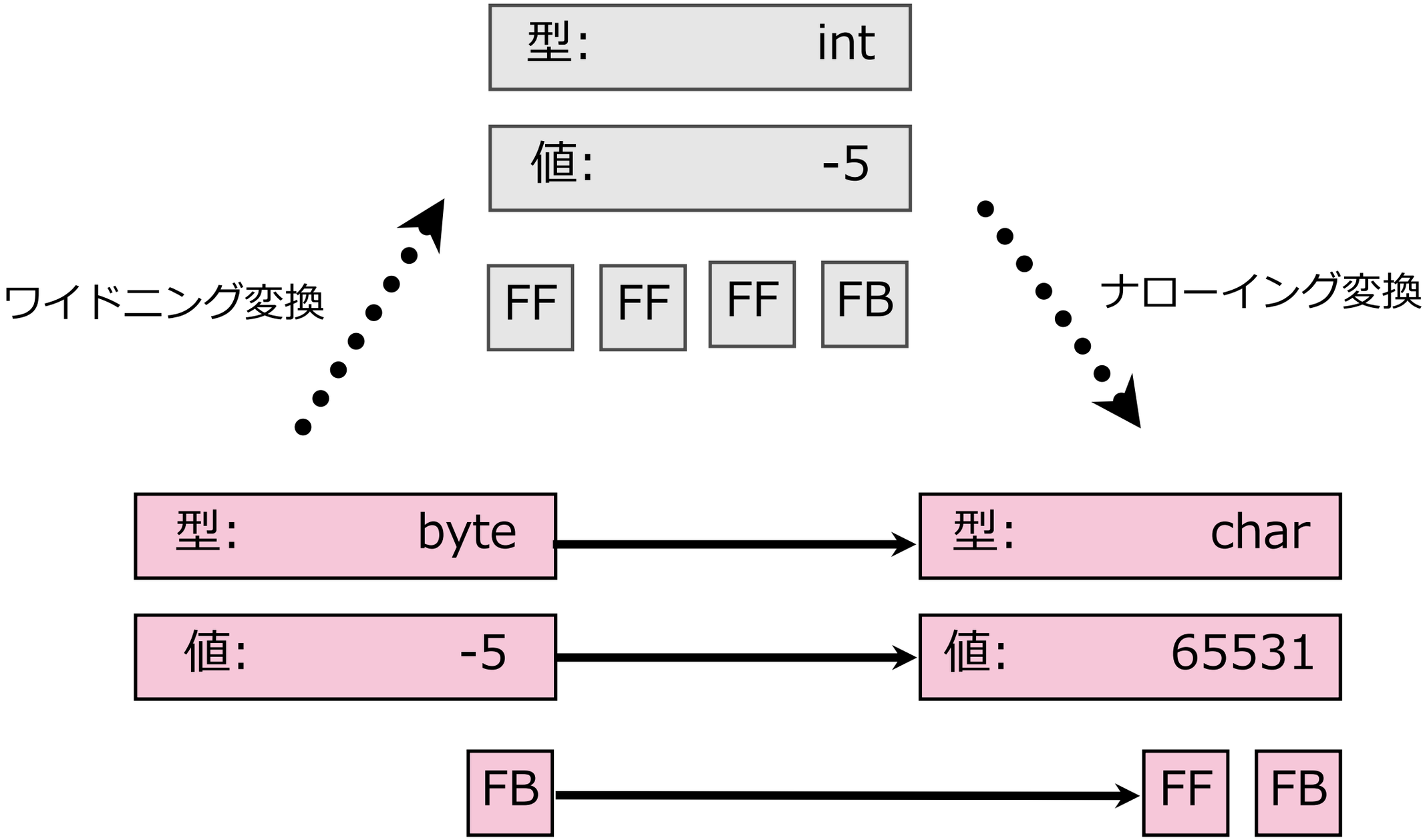




ナローイング変換



ワイドニング&ナローイング変換



コード2

パズル6：マルチキャスト

```
public class Multicast {
    public static void main(String[] args) {
        System.out.println((int) (char) (byte) -1);
    }
}
```

- -1 = 0xFFFFFFFF (int型)
  - (byte)-1 → 0xFF
    - byte型へのナローイング変換、上位24ビットを切り捨て
  - (char)(byte)-1 → 0xFFFF
    - ビットの並びは変わらない(厳密には一旦intに変換し、intがcharに変換されると考える)
  - (int)(char)(byte)-1 → 0x0000FFFF
    - 符号無し整数のワイドニング変換なので、ゼロ拡張する

符号拡張を行いたくないとき

```
byte b = ...;
char c = b & 0xff;  // ビットマスクを使えば、符号拡張は行われない
```

- -1 = 0xFFFFFFFF (int型)
- (byte)-1 → 0xFF
  - byte型へのナローイング変換、上位24ビットを切り捨て
- (char)(byte)-1 → 0xFFFF
  - ビットの並びは変わらない(厳密には一旦intに変換し、intがcharに変換されると考える)
- (int)(char)(byte)-1 → **0x0000FFFF**
  - 符号無し整数のワイドニング変換なので、ゼロ拡張する

暗黙的な整数の型変換

コード3

パズル24：バイトごとに大きな喜び

```
class BigDelight {
   public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE;
           b < Byte.MAX_VALUE; b++) {
           if (b == 0x90)
                   System.out.print("Joy");
       }
 }
```

- 実行しても何も表示されない。なぜ？

コード3

```
for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++) {
   if (b == 0x90)
      System.out.print("Joy");
}
```

- bの値
  - -128 (0x80, 1000 0000)
  - -127 (0x81, 1000 0001)
  - …
  - -113 (0x8f, 1000 1111)
  - -112 (0x90, 1001 0000)　　== 0x90
  - -111 (0x91, 1001 0001)
  - …
  - 126 (0x7e, 1111 1110)

型：int
10進：144

異なる型の値を比較すると、型変換が生じて分かりにくい

コード3の修正

- パターン1: byte 型同士の比較を行う

```
if (b == (byte)0x90)
```

- パターン2: ビットマスクにより符号拡張せずにbyte値をintへ変換させ、int値同士の比較を行う

```
if ((b & 0xff) == 0x90)
```

コード3の修正

- パターン3: 定数をループ外に出し、定数宣言する

```
class BigDelight {
    private static final byte TARGET = (byte)0x90;
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++) {
            if (b == TARGET)
                System.out.print("Joy");
        }
    }
}
```

シフト演算子

コード4

パズル27：不審なi

```
public class Shifty {
   public static void main(String[] args) {
      int i = 0;
      while (-1 << i != 0)
          i++;
      System.out.println(i);
   }
}
```

- 整数リテラル-1は、32ビットすべて1であるint値
  - `0xffffffff`
- 左シフト演算子は、シフトして空いた下位ビットを0で埋める
  - 最下位のiビットを0で埋める
- 32回目のループで、すべてのビットがゼロになり、while文が終了、32が表示されると思いきや……

while (-1 << i != 0) i++ の実行

```
i: 0, -1<<i: 11111111111111111111111111111111(-1)
i: 1, -1<<i: 11111111111111111111111111111110(-2)
i: 2, -1<<i: 11111111111111111111111111111100(-4)
i: 3, -1<<i: 11111111111111111111111111111000(-8)
i: 4, -1<<i: 11111111111111111111111111110000(-16)
i: 5, -1<<i: 11111111111111111111111111100000(-32)
   i: 6, -1<<i: 11111111111111111111111111000000(-64)
i: 7, -1<<i: 11111111111111111111111110000000(-128)
...(snip)...
i:25, -1<<i: 11111110000000000000000000000000(-33554432)
i:26, -1<<i: 11111100000000000000000000000000(-67108864)
i:27, -1<<i: 11111000000000000000000000000000(-134217728)
i:28, -1<<i: 11110000000000000000000000000000(-268435456)
i:29, -1<<i: 11100000000000000000000000000000(-536870912)
i:30, -1<<i: 11000000000000000000000000000000(-1073741824)
i:31, -1<<i: 10000000000000000000000000000000(-2147483648)
i:32, -1<<i: 11111111111111111111111111111111(-1)
i:33, -1<<i: 11111111111111111111111111111110(-2)
i:34, -1<<i: 11111111111111111111111111111100(-4)
...(snip. infinite loop)...
```

-1に戻るのはなぜ？

シフト演算子

- op1(シフトされる値) << op2(シフトする大きさ)
  - 左シフト：<<
  - 符号付き右シフト(算術シフト)： >>
  - 符号無し右シフト(論理シフト)： >>>

- シフト演算子に関するルール (JLS§15.19 シフト演算子)
  - オペランドは、プリミティブ整数型に変換可能な型のみ
  - シフト結果の型は、格上げ後の左オペランドの型
  - 格上げ後の左オペランドが`int`型である場合、右オペランドの**下位5ビット**のみがシフトの大きさとして用いられる
    - 右オペランドの値を `&0x1f` しているのと同じ

左シフト演算子

- (-1 << 32) は、なぜ0ではなく、-1になるのか、言語仕様のルールに従って考えてみる
  - 左オペランド（-1）の型はint
    - ということは、右オペランドの下位5ビット分しかシフトしない
  - 32(10進)は、100000 (2進)
    - その下位5ビットはすべてゼロ
    - つまり0ビットのシフトを行うということ

コード4の修正

```
public class Shifty {
        public static void main(String[] args) {
       int distance = 0;
       for (int val = -1; val != 0; val <<= 1)
           distance++;
       System.out.println(distance);
   }
}
```

- ループ毎に異なるシフト距離でシフトするのではなく、1ビットづつシフトし、シフト結果を保存する操作を繰り返す

- 負のシフト距離(例えば-1)を指定された場合の動作は？
  - C言語では「未定義の動作」
  - Javaでは32の法(longなら64)として計算される

整数の境界条件

コード5

パズル26：「内輪」の話

```
public class InTheLoop {
    public static final int END = Integer.MAX_VALUE;
    public static final int START = END - 100;

    public static void main(String[] args) {
          int count = 0;
          for (int i = START; i <= END; i++)
                count++;
          System.out.println(count);
    }
}
```

- i == ENDが成立とき、i++を行うとiはラップアラウンドして、Integer.MIN_VALUEになってしまう
  - ループカウンタにlong使う

整数型は非対称

最小値を負にすると、最小値

コード6

- **次のループが無限ループとなるような i の宣言は？**

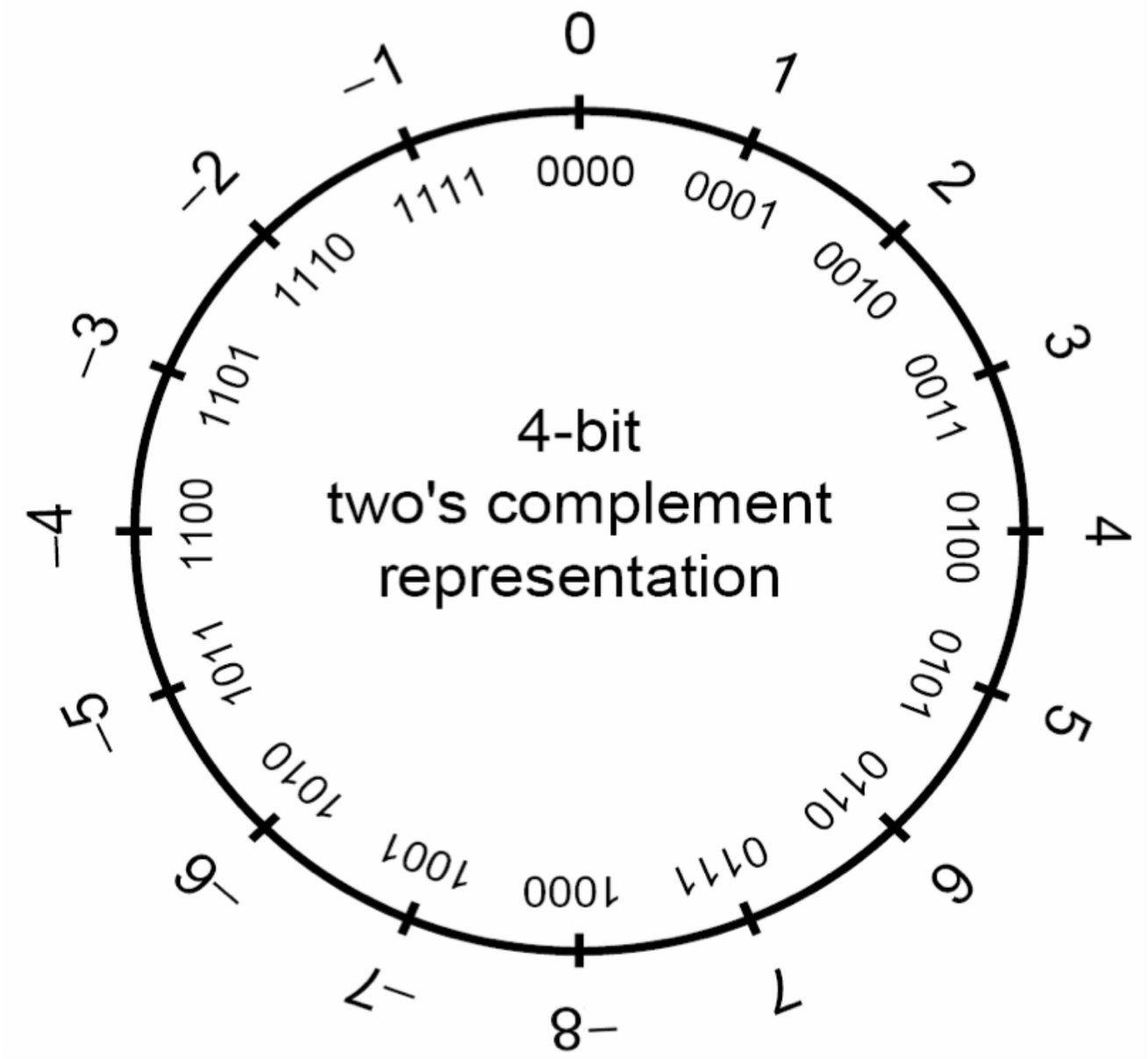
パズル33：ループ職人と狼男

```
while (i != 0 && i == -i) {
}
```

- -i　は単項演算なので、数値じゃないとダメ
- NaNはダメ。いかなる値とも異なる。
- 浮動小数点数もダメ。ビットを反転すると自分自身とは異なる。

- ヒント：2の補数表現

2の補数表現



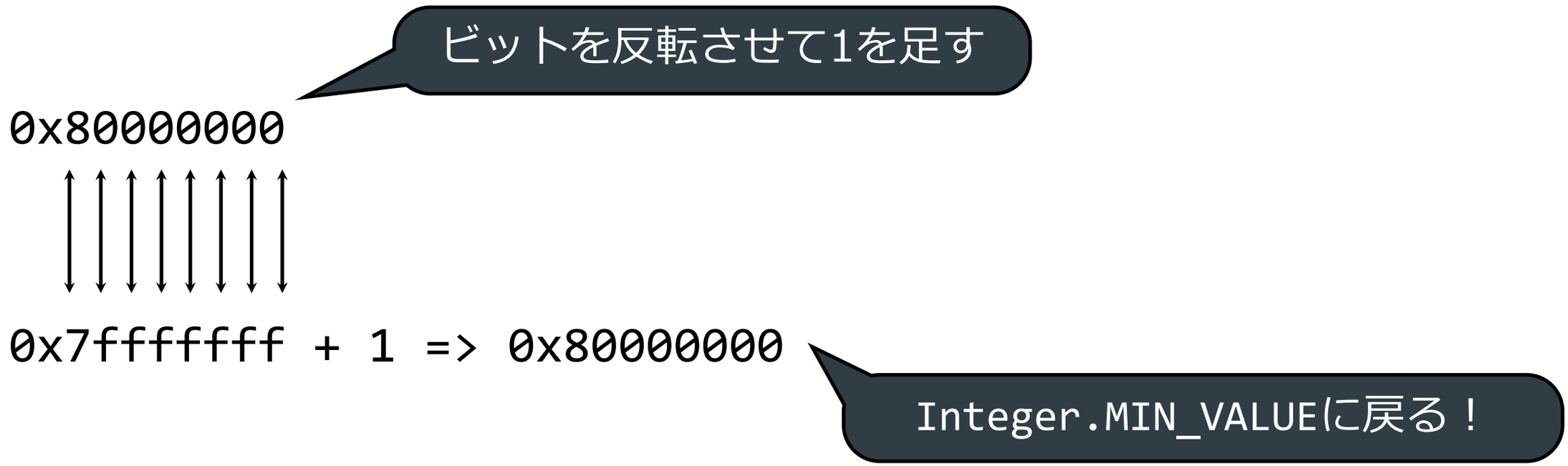
コード6

- int もしくは long の最小値

```
int i = Integer.MIN_VALUE;
// or long i = Long.MIN_VALUE;

while (i != 0 && i == -i) {
}
```

- Integer.MIN_VALUE (-231)



条件演算子の第2、第3オペランドには同じ型を使う

Javaの条件演算子

- 基本形
  - operand1 ? operand2 : operand3
    - operand1の値がtrueなら、operand2の式が選ばれる
    - operand1の値がfalseなら、operand3の式が選ばれる
  - シンタックスは右結合
    - a?b:c?d:e?f:g は a?b:(c?d:(e?f:g))と同じ

- 型変換のルールが複雑（次スライド）

条件式の型を決めるルール

- 条件式の型は以下のように決定される：
- 2番目と3番目のオペランドが同じ方である場合（null型でも可）、それがこの条件式の型となる。
- 2番目と3番目のオペランドの一方がboolean型であり、もう一方がBoolean型である場合、この条件式の型はboolean となる。
- 2番目と3番目のオペランドの一方がnull型であり、もう一方が参照型である場合、この条件式の型はその参照型となる。
- そうでなく、2番目と3番目のオペランドが数値型へ変換可能（§5.1.8）な型である場合、以下の場合に分けられる：
  - 一方のオペランドがbyte型かByte型で、もう一方のオペンランドがshort型かShort型である場合、この条件式の型はshortとなる。
  - 一方のオペランドがbyte, short, charをTとした際のT型で、もう一方のオペランドがT型で表現可能な値となるint型の定数式である場合、この条件式の型はTとなる。
  - 一方のオペランドがByte型であり、もう一方のオペランドがbyte型で表現可能な値となるint型の定数式である場合、この条件式の型はbyteとなる。
  - 一方のオペランドがShort型であり、もう一方のオペランドがshort型で表現可能な値となるint型の定数式である場合、この条件式の型はshortとなる。
  - 一方のオペランドがCharacter型であり、もう一方のオペランドがchar型で表現可能な値となるint型の定数式である場合、この条件式の型はcharとなる。
  - さもなければ、オペランド型に対して二項の数値格上げ変換（§5.6.2）が適用され、この条件式の型は格上げ変換された2番目と3番目のオペランド型となる。二項の数値格上げ変換では、アンボクシング変換（§5.1.8）と数値集合変換（§5.1.13）が適用されることに注意。
- さもなければ、2番目と3番目のオペランドの型をそれぞれS1とS2とする。そして、S1に対するボクシング変換の適用結果をT1、S2に対するボクシング変換の適用結果をT2とする。この条件式の型は、lub(T1, T2)（§15.12.2.7）に対する捕捉変換（§5.1.13）の適用結果となる。

JLS§15.25 条件演算子

条件式の型を決めるルール

ルール	operand2	operand3	結果の型
1	T型	T型	T型
2	boolean	Boolean	boolean
3	Boolean	boolean	boolean
4	null型	参照型	参照型
5	参照型	null型	参照型
6	byte か Byte	short か Short	short
7	short か Short	byte か Byte	short
8	byte,short,char,Byte,Short,Character	int型の定数式	byte,short,char(もしintの値が表現可能なら)
9	int型の定数式	byte,short,char,Byte,Short,Character	同上
10	その他の値	その他の値	格上げされた第2第3オペランドの型
11	T1=ボクシング変換(S1)	T2=ボクシング変換(S2)	lub(T1,T2)の補足変換結果

コード7

```
public class Expr {
  public static void main(String[] args) {
    char alpha = 'A';
    int i = 0;
    /* i の値は変わるかも */
    boolean trueExp = true;
    System.out.print(trueExp ? alpha : 0); // Aを出力
    System.out.print(trueExp ? alpha : i); // 65を出力
  }
}
```

- 問題は第2、第3オペランドの型が何になるか
- 問題は、第3オペランドの型が何になるか
  - 0：char型で表現可能な値の定数式→char型
  - i：二項の数値格上げ変換の結果、int型になる

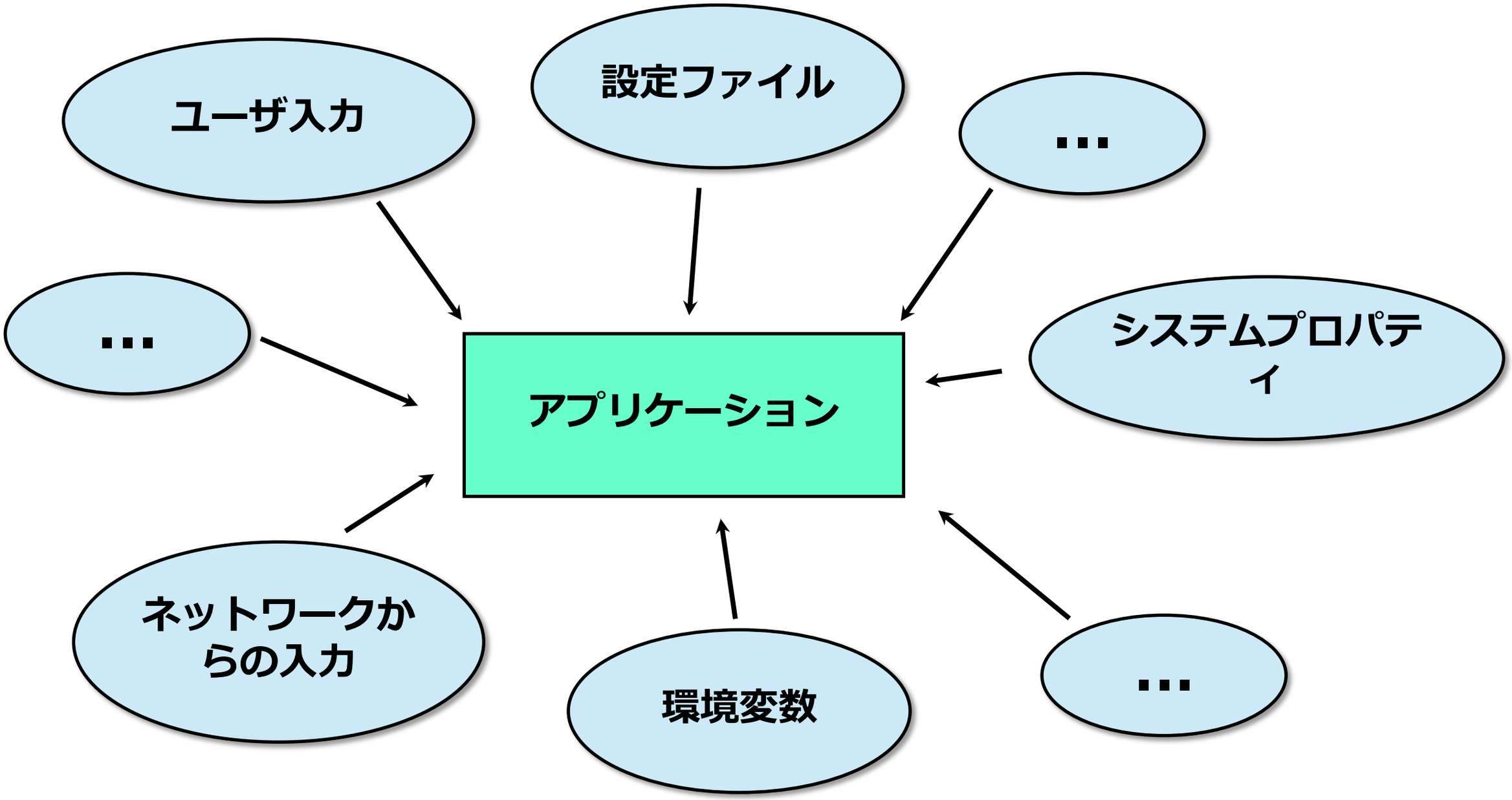
コード7の修正

```
public class Expr {
  public static void main(String[] args) {
    char alpha = 'A';
    int i = 0;
    /* i の値は変わるかも */
    boolean trueExp = true;
    System.out.print(trueExp ? alpha : ((char)0));
    System.out.print(trueExp ? alpha : ((char)i));
  }
}
```

- 第2、第3オペランドを明示的にキャストし、混乱を避ける

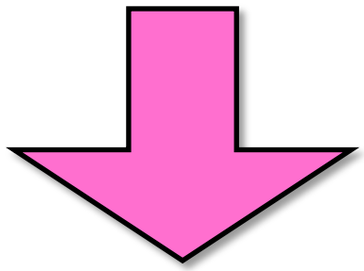
入力値検査とデータの無害化

Java アプリケーションに対する「入力」



予期せぬ入力と想定外の動作と脆弱性

予期せぬ入力によって想定外の動作につながる可能性がある



想定外の動作が様々な脆弱性に結びつく！

コマンドインジェクション, SQL インジェクション, ディレクトリトラバーサル, クロスサイトスクリプティング, リソース枯渇攻撃など

サンプルプログラム: ディレクトリトラバーサル

```
class filecat {
  static private String BASEDIR="/home/yozo/tmp/";

  static public void cat(String s) throws IOException {
      BufferedReader bf =
          new BufferedReader(new FileReader(BASEDIR + s));
      String line;
      while (null != (line = bf.readLine())){
        System.out.println(line);
      }
  }

  public static void main(String[] args) throws IOException {
    String filename = args[0];
    cat(filename);
  }
}
```

対象とするディレクトリ

単純な文字列連結だけ行っている

ファイル名を期待

サンプルプログラム: ディレクトリトラバーサル

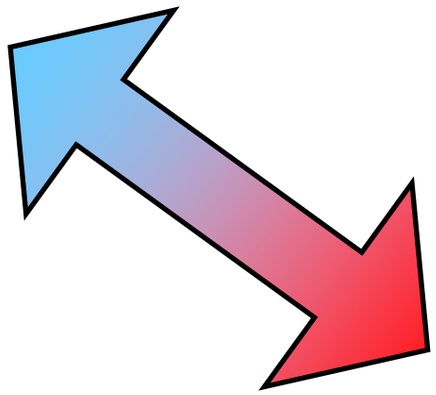
想定通りの動作

```
$ java filecat choi
hoihoi
$ java
filecat ../Documents/WORK/SecureCoding/20121014.Tokyo_part2/filecat.java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
class filecat {
  static private String BASEDIR="/Users/yozo/tmp/";
(...... 以下省略 ......)
```

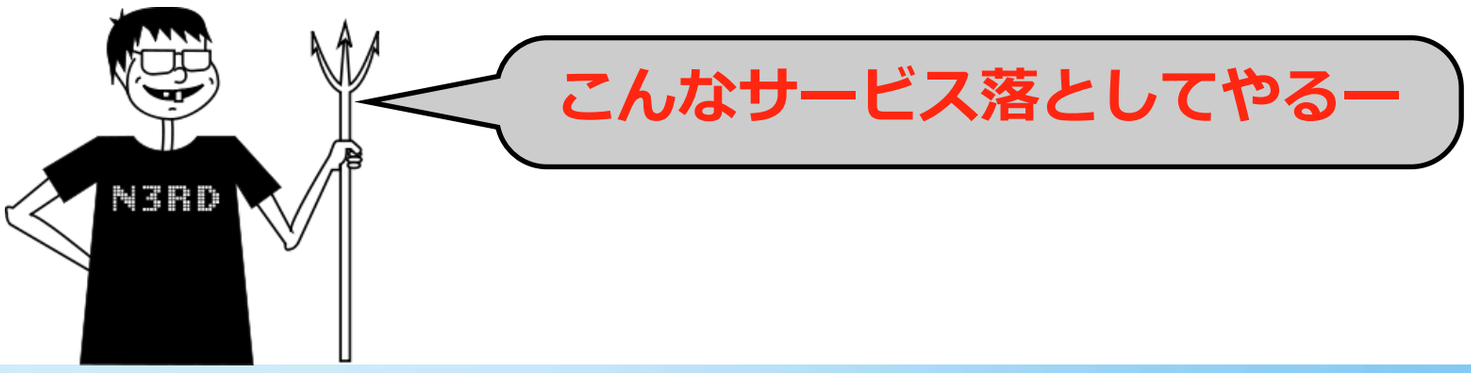
想定外の入力！

プログラマーの視点と攻撃者の視点

プログラマーの視点:
プログラムでデータを制御する



攻撃者の視点:
入力データでプログラムを制御する



重要: 想定外の入力に備える

- **「想定外でした」は禁句!**
  - 整数値の入力に浮動小数点数が来たら…
  - 浮動小数点数の入力に整数値が来たら…
  - 非負の値を想定している入力に負の値が来たら…
  - 文字列の入力にバイナリデータが来たら…
  - 「ファイル名」のところに「パス」が来たら…

入力値検査エラーの実例

Struts2 のクロスサイトスクリプティング脆弱性

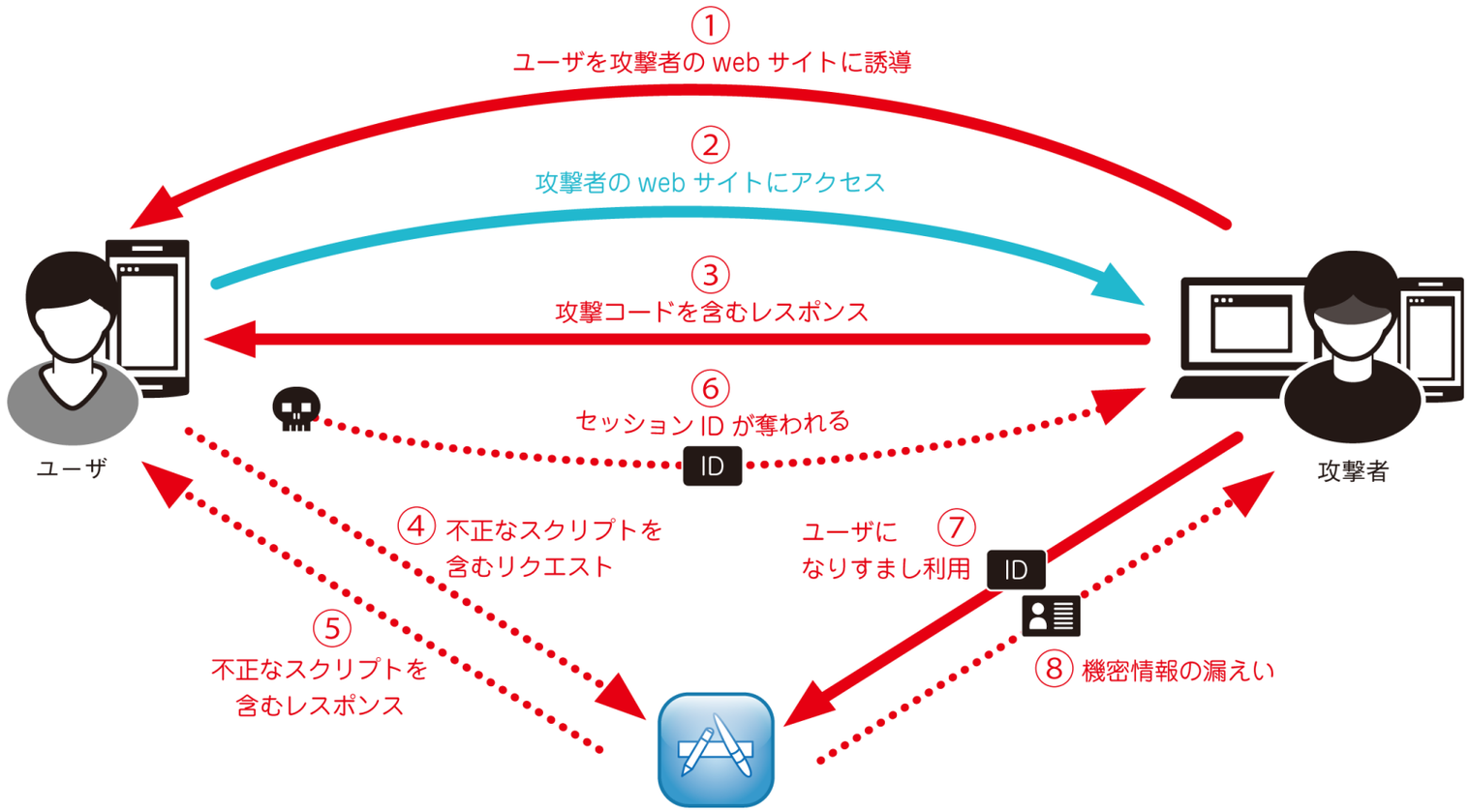
Struts2におけるXSS

- Struts 2 <s:submit> XSS vulnerability（CVE-2011-1772）
  - HTTPリクエストで受け取ったアクション名を、無害化せずにそのままエラーページに出力していた

> **By default, XWork doesn't escape action's names in automatically generated error page, allowing for a successful XSS attack.**

https://cwiki.apache.org/confluence/display/WW/S2-006

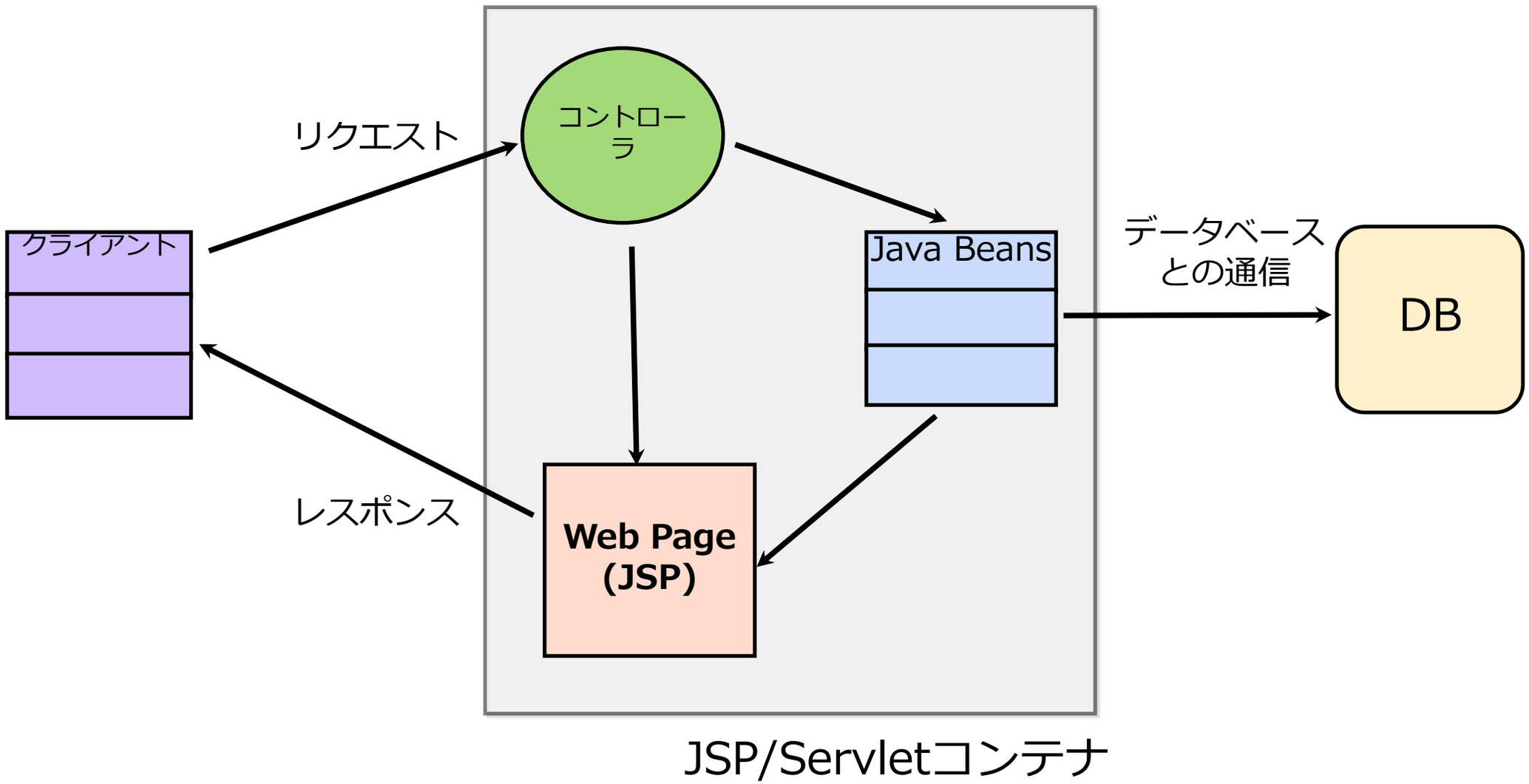
クロスサイトスクリプティングとは



- クロスサイトスクリプティングとは、ユーザの入力値を元にHTMLやURLなどを動的に生成しているWebサイトにおいて、攻撃者によって被害者のブラウザに表示されるコンテンツに任意のHTMLやJavaScriptを埋め込む攻撃

Struts2におけるXSS

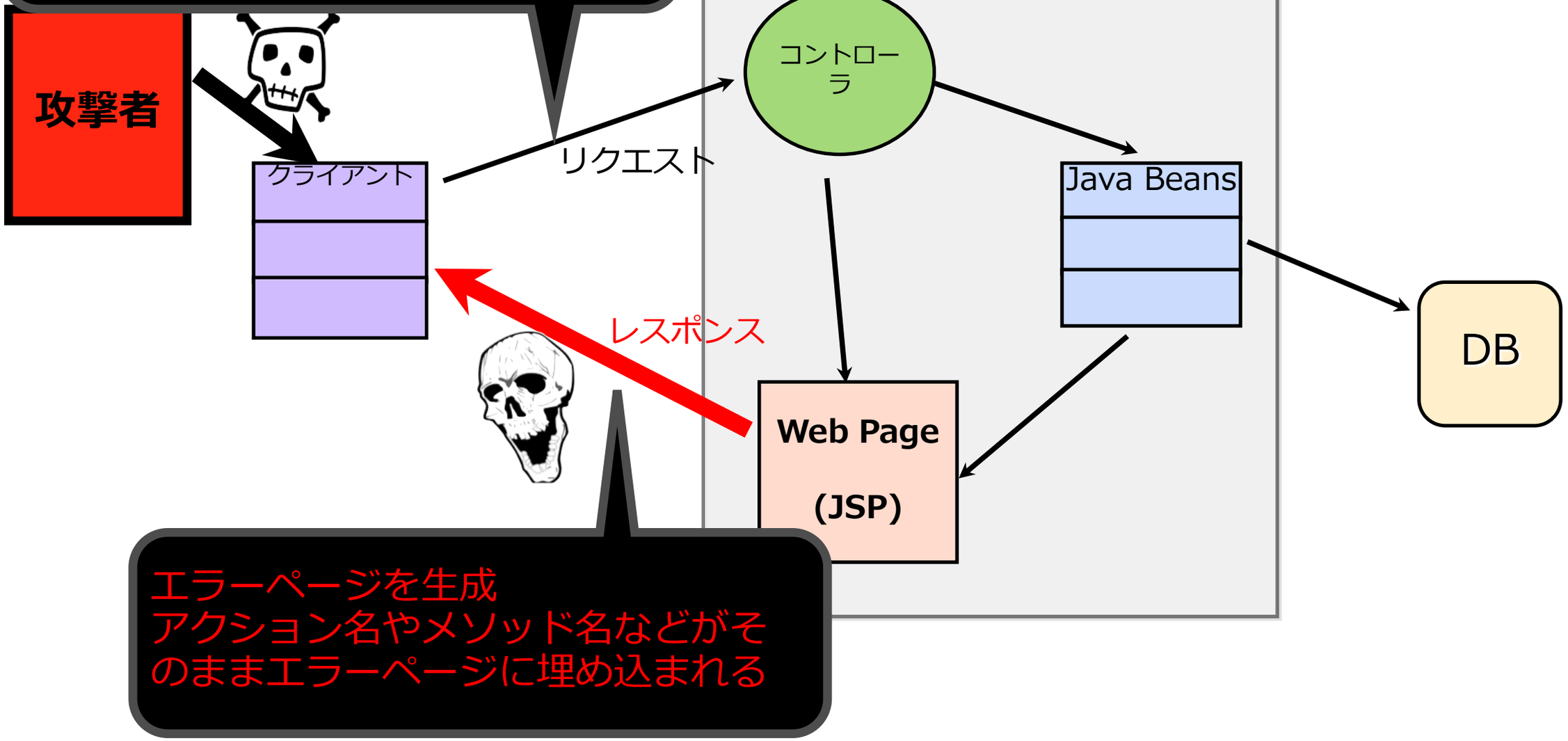
- Struts の仕組み: Model-View-controller



Struts2におけるXSS

細工された入力
http://...home.action?...action!login<script>alert(document.cookie)</script>:cantLogin=some_value

actionName=login<script>alert(document.cookie)</script>
methodName=cantLogin=some_value



エラーページを生成
アクション名やメソッド名などがそのままエラーページに埋め込まれる

Struts2におけるXSS

修正前

```
import java.io.Serializable;
import java.util.Locale;
import org.apache.commons.lang.StringUtils;

/**
 * The Default ActionProxy implementation
 */
...
    Protected DefaultActionProxy(..., String actionName, String methodName, …)
{
        LOG.debug("Creating an DefaultActionProxy for namespace " + namespace
                + " and action name " + actionName);


        this.actionName = actionName;
        this.namespace = namespace;
        this.executeResult = executeResult;
        this.method = methodName;
    }
```

外部から受け取った
ActionNameとmethodName
をそのまま使っている

Struts2におけるXSS

修正後

```
import org.apache.commons.lang.StringEscapeUtils;
import org.apache.commons.lang.StringUtils;
import java.io.Serializable;
import java.util.Locale;

/**
 * The Default ActionProxy implementation
 */
...
    Protected DefaultActionProxy(..., String actionName, String methodName, …)
{
        LOG.debug("Creating an DefaultActionProxy for namespace " + namespace
                + " and action name " + actionName);

        this.actionName = StringEscapeUtils.escapeHtml(actionName);
        this.namespace = namespace;
        this.executeResult = executeResult;
        this.method = StringEscapeUtils.escapeJavaScript(
                        StringEscapeUtils.escapeHtml(methodName) );
    }
```

Apache Commonsライブラリの
escape*()を活用して無害化

Struts2におけるXSS: まとめ

信頼できない入力である actionName と methodName が不適切な値を持っている可能性を考慮すべきだった

攻撃者は細工した入力を Struts2 に処理させ,
(Struts2 開発者にとっては想定外の)データを
エラーページに出力させた.

入力値検査エラーの実例

ZipBomb

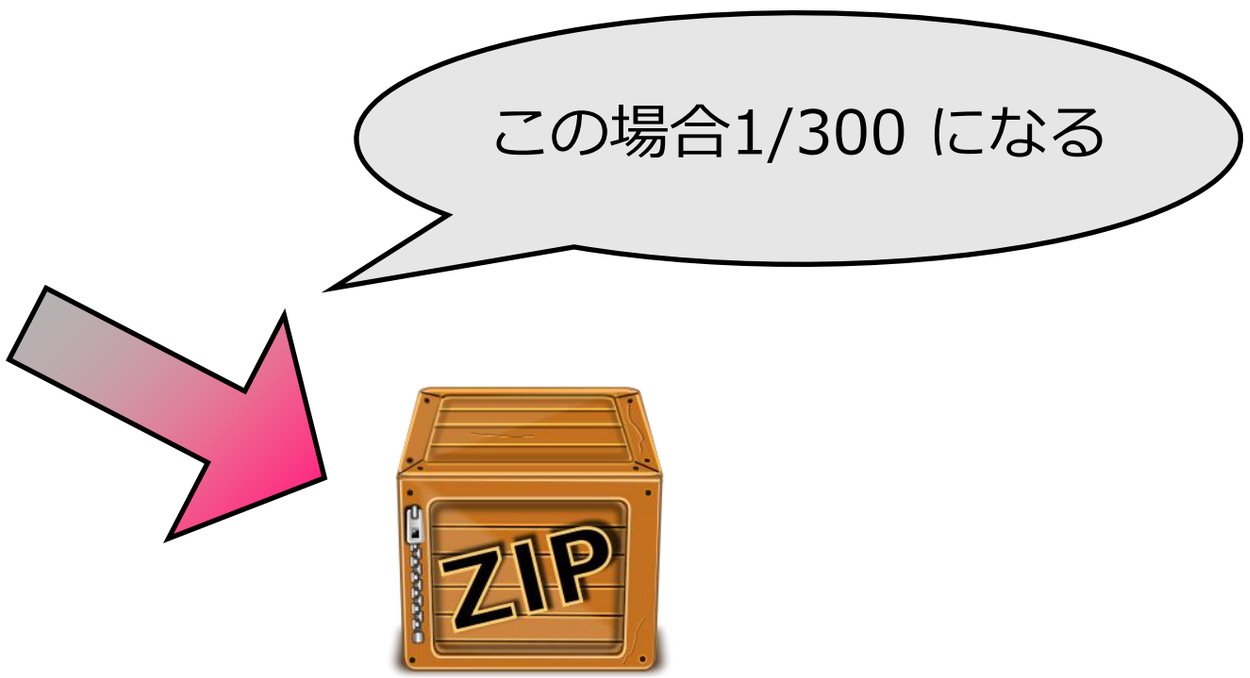
ZipBomb

- **高圧縮ファイル爆弾**
  - http://ja.wikipedia.org/wiki/高圧縮ファイル爆弾
- **Zip Bomb**
  - http://en.wikipedia.org/wiki/Zip_bomb

ZIP とは?

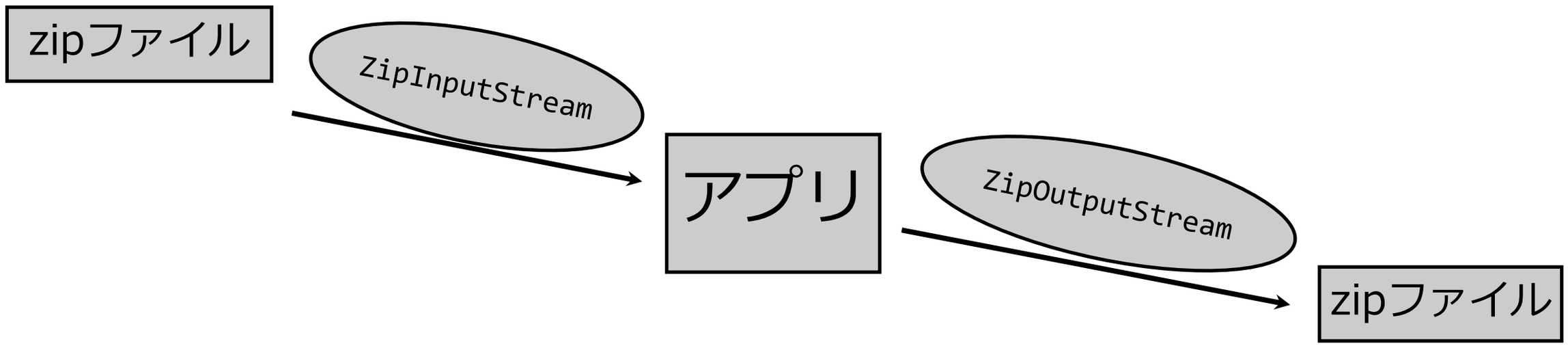
- ZIP: データ圧縮形式のひとつ
- MS Windows のエクスプローラは標準で対応している
- Java 標準 API では `java.util.zip` パッケージが提供されている
- データによっては非常に高い圧縮率で圧縮される
- (例えば, 30MB データが 100KB になる)

AAAAAAAAA....
AAAAAAAAA....
AAAAAAAAA....
AAAAAAAAA....



Java.util.zip パッケージ

- **ZIP や GZIP ファイルを読み書きする機能を提供するクラス**
  - `ZipInputStream` -- ZIP形式データを解凍する
  - `ZipOutputStream` -- データをZIP形式に圧縮する
  - `ZipEntry` -- ZIP形式内部のエントリにアクセス
  - `GZIPInputStream` -- GZIP形式データを解凍する
  - `GZIPOutputStream` -- データをGZIP形式に圧縮する

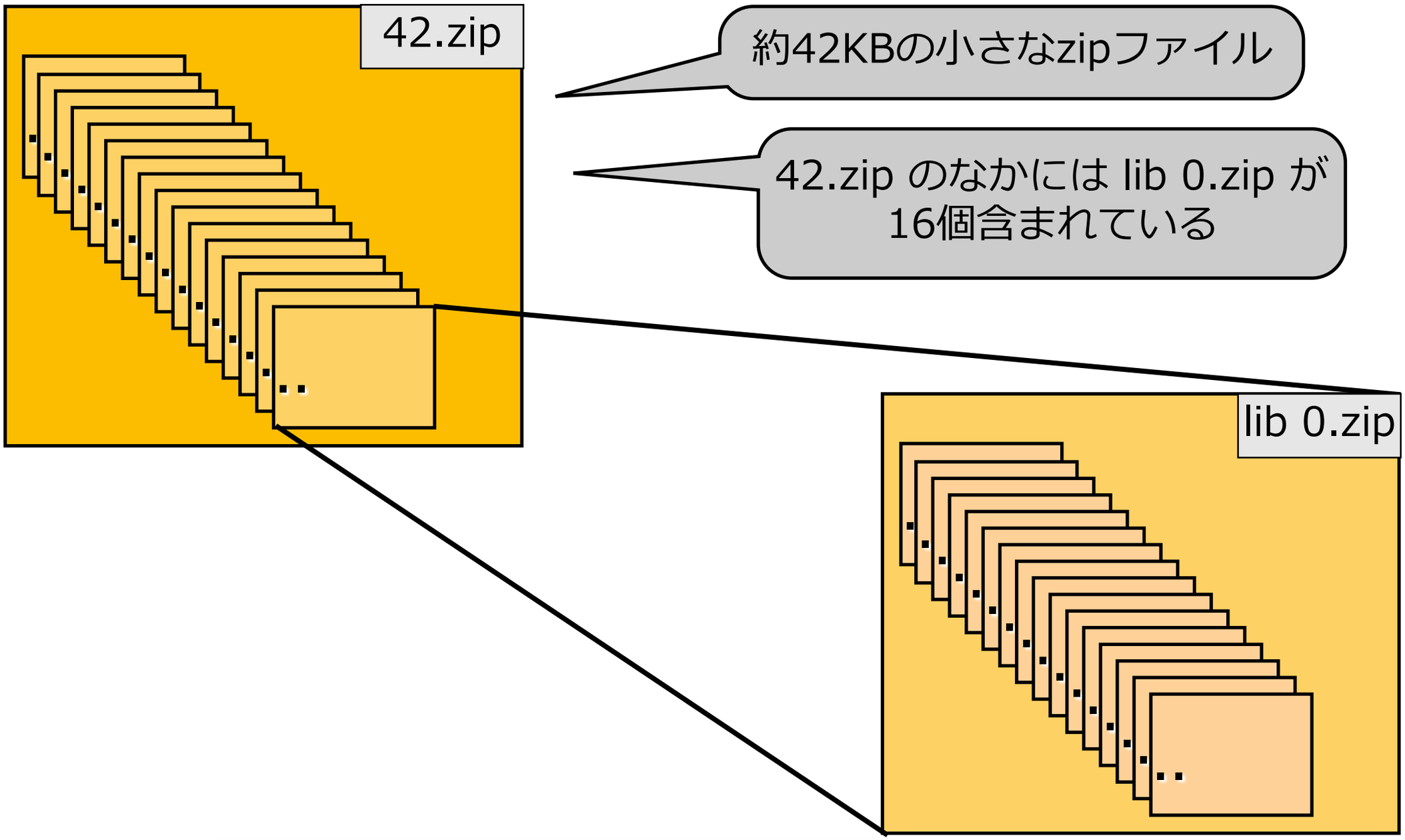


リソース消費攻撃の実例: ZipBomb

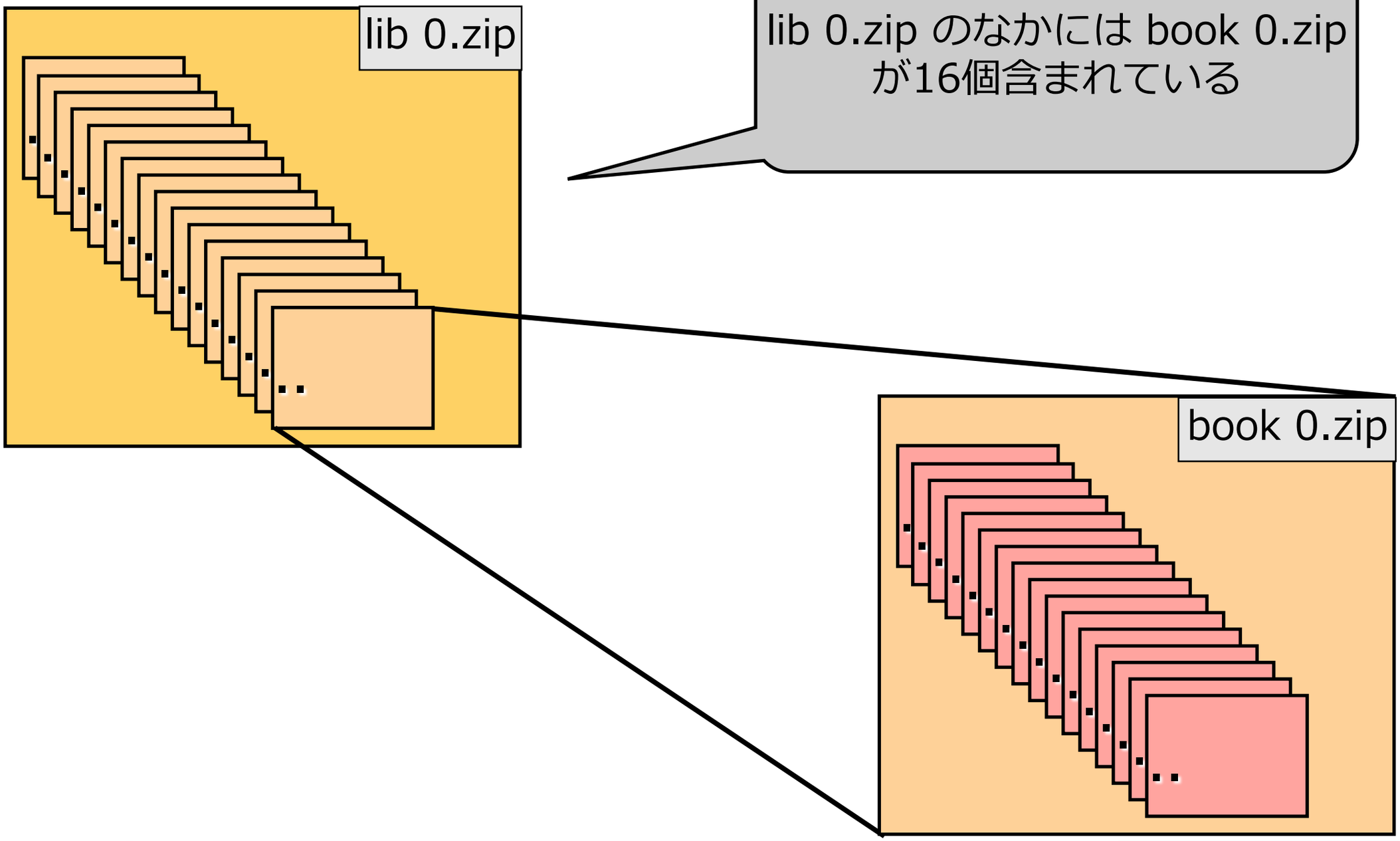
- **ZipBomb**
  - 小さなサイズのzipファイルを展開させる
  - じつは高圧縮率のファイル
  - 展開すると大きなサイズのファイルが出てくる
  - **メモリ**や**ディスク**を圧迫

42.zip(1)

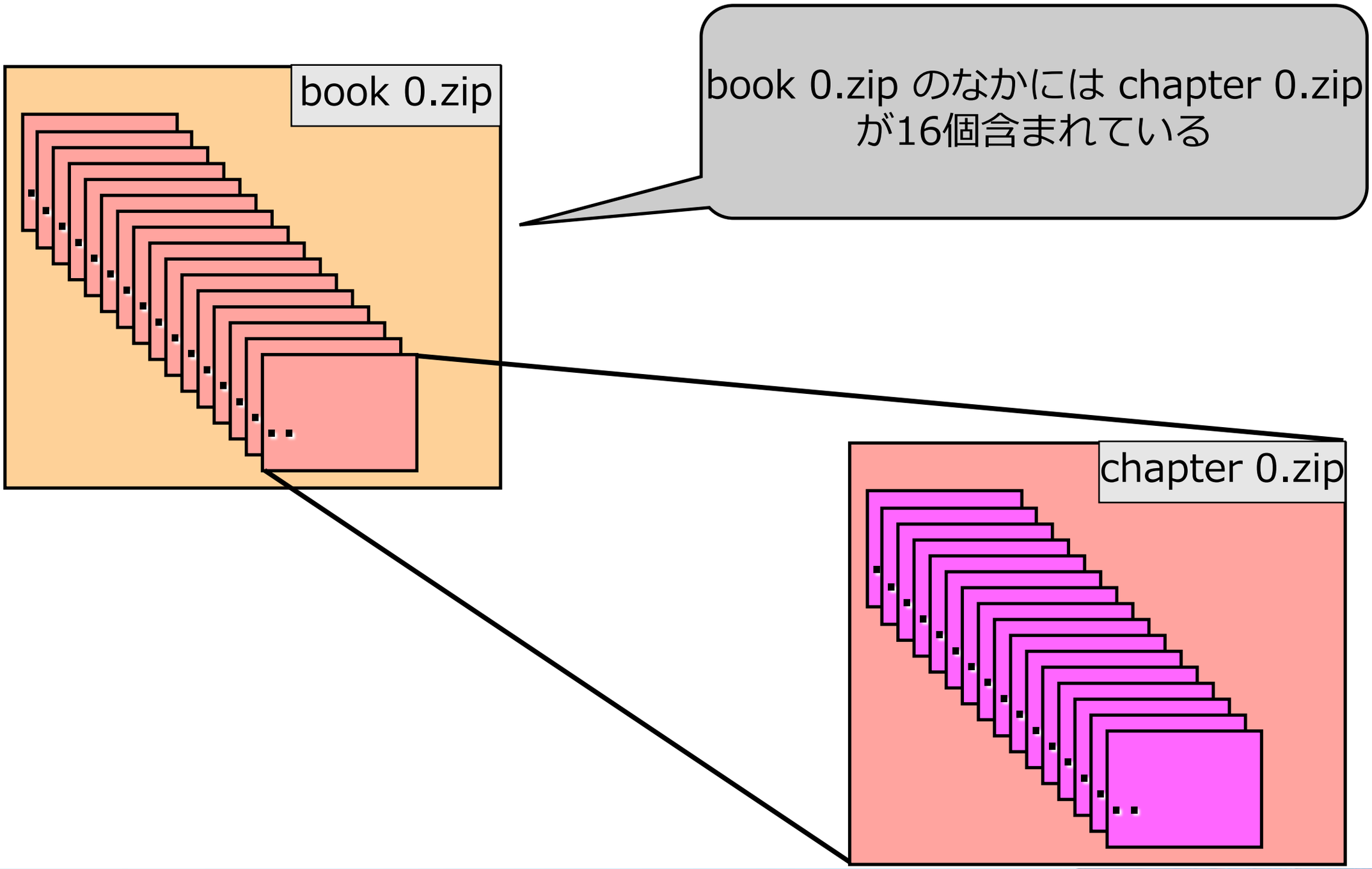
有名な zipbomb の例: 42.zip
(http://www.unforgettable.dk/)



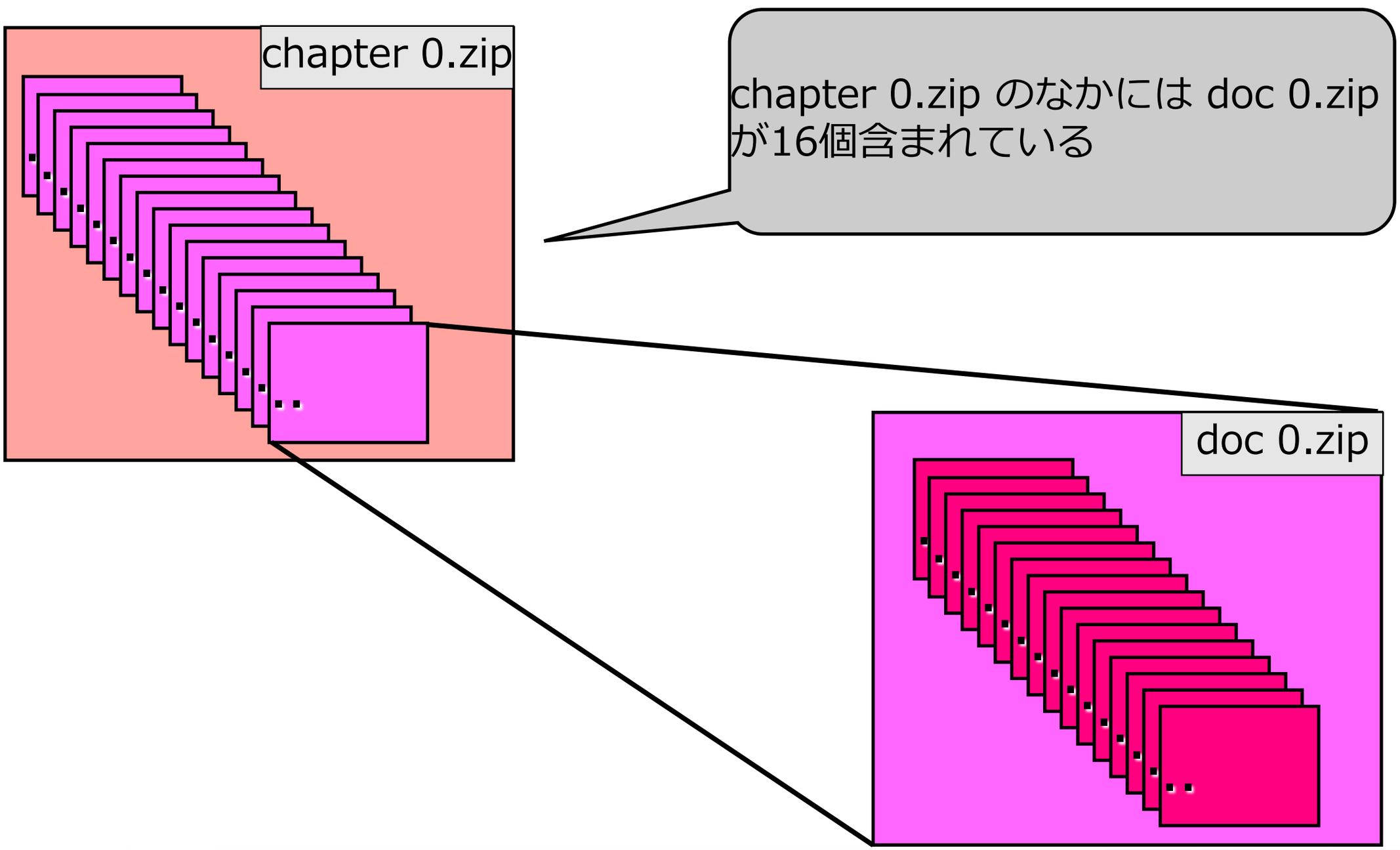
42.zip(2)



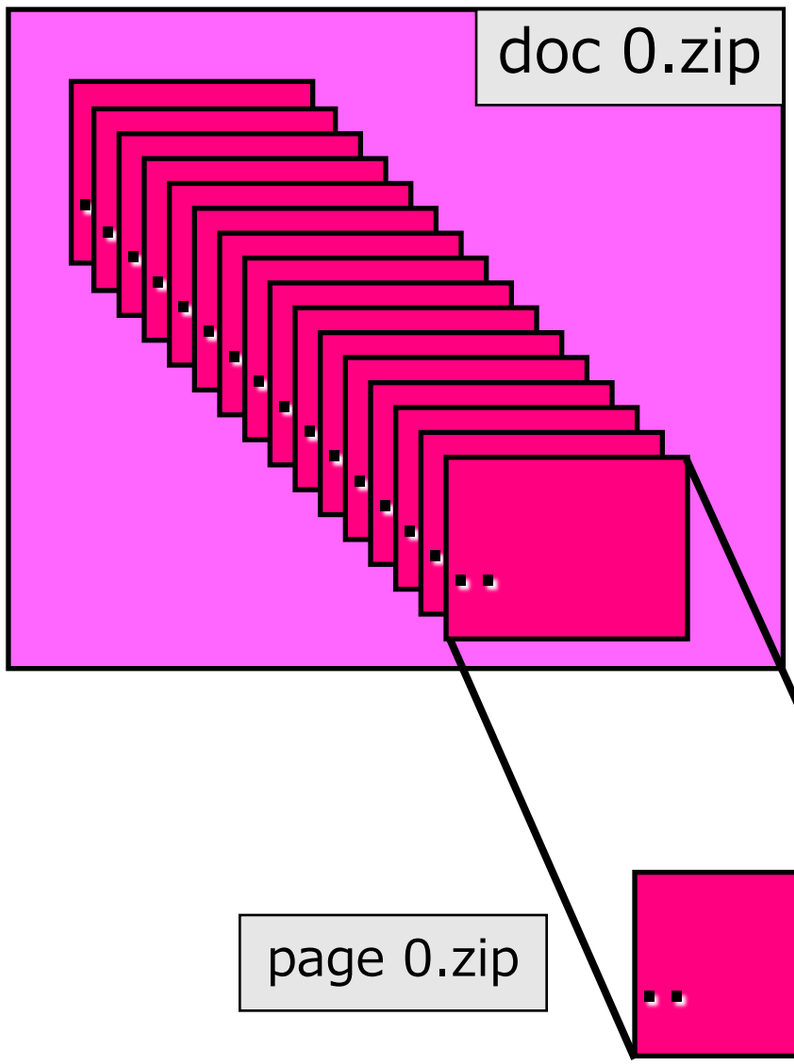
42.zip(3)



42.zip(4)

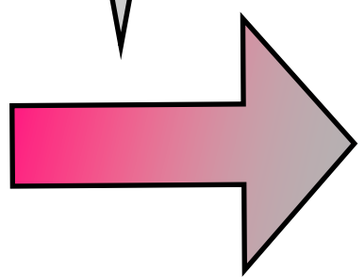


42.zip(5)



doc 0.zip のなかには page 0.zip が16個含まれている

page 0.zip は 0.dll をzip したもの



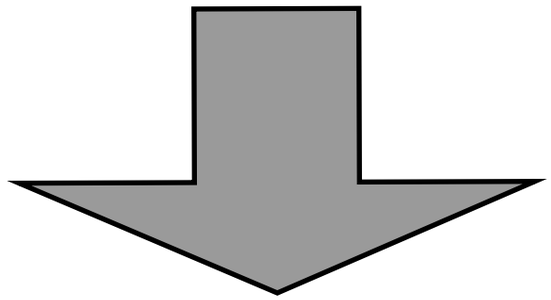
0.dll (約4.3GB)

```
1010101010....
1010101010....
1010101010....
1010101010....
```

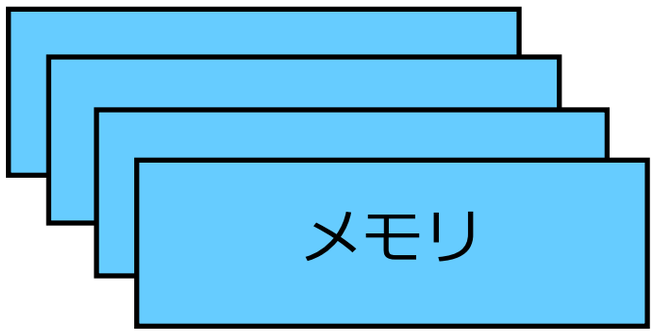
全ての zip ファイルを展開すると
16*16*16*16*16 個の 0.dll が出てくる
⇒ 4.3GB * 16^5 == 4.5PB になる!!

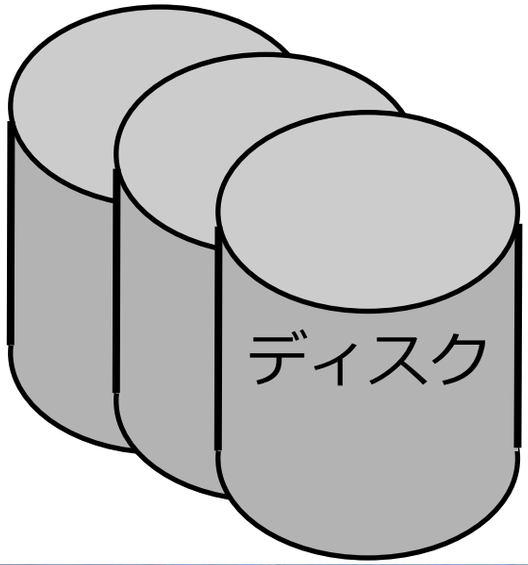
ZipBombの効果

小さなサイズの圧縮データを入力として与え、
大きなサイズのファイルに展開させる



リソース消費攻撃!





ZipBombによる攻撃

- 例: アンチウイルスソフトへの攻撃
  - メールに添付されたZIPファイルを展開して中身をチェック
  - 展開の過程でアンチウイルスソフトが停止, あるいはPC自体がフリーズ

ZipBombの影響を受けるコード例(1)

```
class Unzip {
  static final int BUFFER = 512;

  public static void main(String[] args) throws FileNotFoundException,IOException {
    BufferedOutputStream dest = null;
    ZipInputStream zis =
         new ZipInputStream(new BufferedInputStream(new FileInputStream(args[0])));
    ZipEntry entry;
    while ((entry = zis.getNextEntry()) != null){
        System.out.println("Extracting: " + entry);
        int count;
        byte data[] = new byte[BUFFER];
        FileOutputStream fos = new FileOutputStream(entry.getName());
        dest = new BufferedOutputStream(fos, BUFFER);
        while ((count=zis.read(data,0,BUFFER)) != -1){
            dest.write(data, 0, count);
        }
        dest.flush();
        dest.close();
    }
    zis.close();
  }
}
```

ZIPアーカイブ内の各エントリの展開後のサイズを事前にチェックしていない

ZipBombの影響を受けるコード例(1)

```
class Unzip {
  static final

  public stati                                                    n {
    BufferedOu
    ZipInputSt
         new Z                                                    ));
    ZipEntry e
    while ((er
        System
        int co
        byte d
        FileOu
        dest = new BufferedOutputStream(fos, BUFFER);
        while ((count=zis.read(data,0,BUFFER)) != -1){
            dest.write(data, 0, count);
        }
        dest.flush();
        dest.close();
    }
    zis.close();
  }
}
```

対策:
ZIPアーカイブ内の各エントリの展開後のサイズを**事前に**チェックする

ZIPアーカイブ内の各エントリの展開後のサイズを事前にチェックしていない

ZipBombの影響を受けるコード例(3)

```
class Unzip {
  static final int TOOBIG = 0x6400000; // 100MB
  static final int BUFFER = 512;

  public static void main(String[] args) throws FileNotFoundException,IOException {
    BufferedOutputStream dest = null;
    ZipInputStream zis = new ZipInputStream(new BufferedInputStream(new
FileInputStream(args[0])));
    ZipEntry entry;
    while ((entry = zis.getNextEntry()) != null){
        if (entry.getSize() > TOOBIG){
              throw new IllegalStateException("uncompressed size too huge");
        }
        else if (entry.getSize() == -1){
              throw new IllegalStateException("uncompressed size unknown");
        }
        System.out.println("Extracting: " + entry);
        int count;
        byte data[] = new byte[BUFFER];
        FileOutputStream fos = new FileOutputStream(entry.getName());
        dest = new BufferedOutputStream(fos, BUFFER);
        while ((count=zis.read(data,0,BUFFER)) != -1){ dest.write(data, 0, count); }
        dest.flush();
        dest.close();
    }
    zis.close();
  }
}
```

展開後のサイズの上限を決める

展開後のサイズを事前にチェック

ZipBomb: まとめ

ZIPアーカイブファイルを解凍する場合, 細工したファイルによる**リソース消費攻撃**の危険を考慮する!

展開後のサイズを**事前に**確認する!

More Bombs...

…ところでこれって…
ZIPアーカイブファイルだけじゃないよ!

More Bombs...

- ZIP以外の圧縮ファイル形式
- gzip, bzip2, lha, rar, ……

これらの圧縮形式に対応したアプリは大丈夫かな?

- 圧縮アルゴリズムって画像形式にも使われてるよ
- jpeg, png, gif, ……

細工した画像ファイルでDoS攻撃されちゃうかも!?

- http の転送方式に deflate 圧縮なんてあるよ

悪意あるwebサーバへのアクセスでブラウザがクラッシュとか!?

More Bombs...

Decompression bomb vulnerabilities
AERAsec Network Services and Security GmbH
http://www.aerasec.de/security/advisories/decompression-bomb-vulnerability.html

最後のまとめ

アプリケーションへの全ての入力を適切に検査(と必要に応じて無害化)しましょう

特に信頼境界の外からやってくるデータには入力値検査は必須!

参考資料

- 『Java Puzzlers 罠、落とし穴、コーナーケース』ジョシュア・ブロック他著、ピアソン、2005
- Javaセキュアコーディングスタンダード CERT/Oracle版
  - https://www.jpcert.or.jp/java-rules/
- The CERT Oracle Secure Coding Standard for Java Wiki
  - https://www.securecoding.cert.org/
- OWASP Input Validation Cheat Sheet
  - https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

Java セキュアコーディングスタンダードCERT/Oracle版

- IDS: 入力値検査とデータの無害化
  - IDS00-J. 信頼境界を越えて渡される信頼できないデータは無害化する
  - IDS01-J. 文字列は検査する前に標準化する
  - IDS02-J. パス名は検証する前に正規化する
  - IDS03-J. ユーザ入力を無害化せずにログに保存しない
  - IDS04-J. ZipInputStream にわたすファイルサイズは制限する
  - IDS05-J. ファイル名やファイルパスにはASCII文字セットの一部の文字のみを使用する
  - IDS06-J. ユーザからの入力を使って書式を組み立てない
  - IDS07-J. 信頼できない、無害化されていないデータを Runtime.exec() メソッドに渡さない
  - IDS08-J. 信頼できないデータは regex に渡す前に無害化する
  - IDS09-J. 適切なロケールを指定せずにロケール依存メソッドをロケール依存データに対して使用しない
  - IDS10-J. 一文字を構成するデータを分割しない
  - IDS11-J. 非文字コードポイントは検証を行う前に削除する
  - IDS12-J. 異なる文字コードへの文字列データの変換はデータが欠損しないように行う
  - IDS13-J. ファイル入出力やネットワーク入出力の両端で互換性のある文字エンコーディングを使う

付録

HashDos

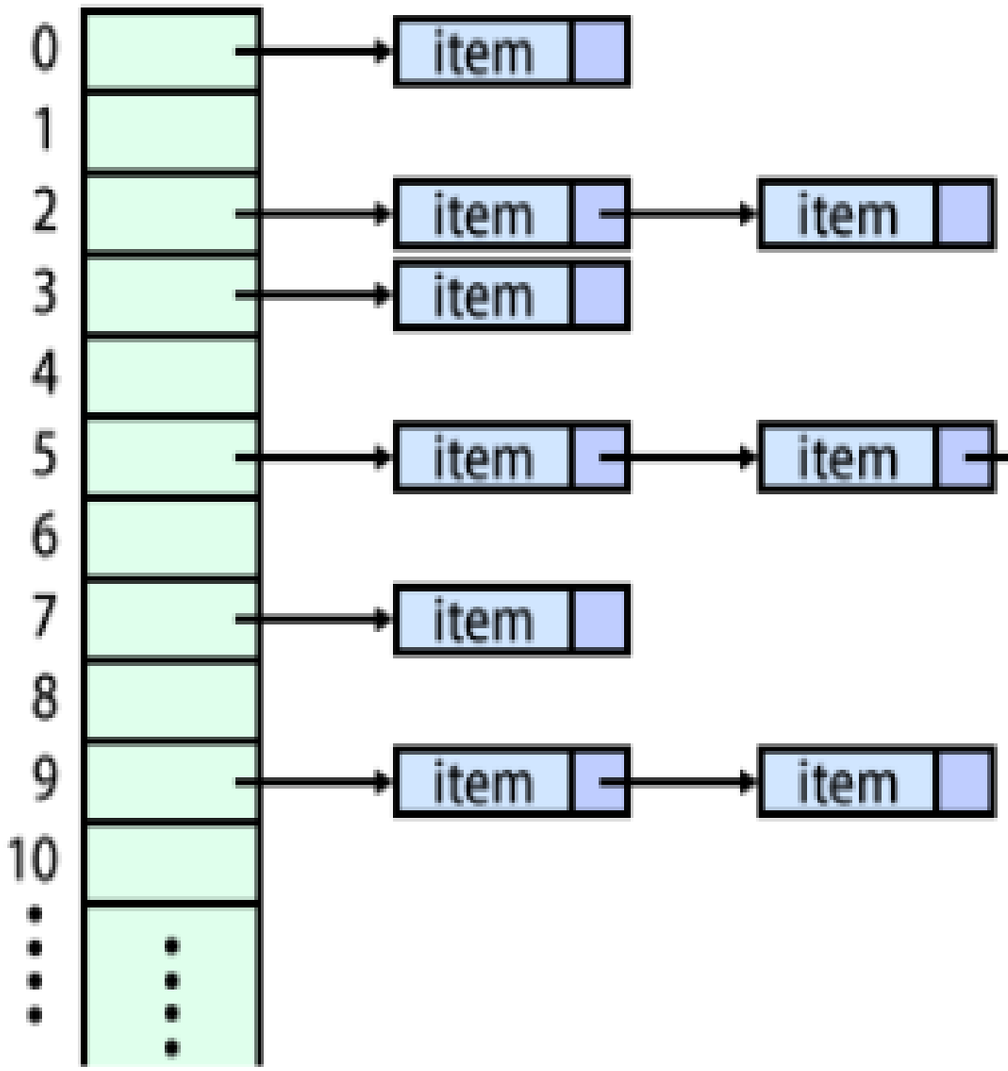
入力データの処理にハッシュテーブルを使っている場合、DoS攻撃を受ける危険がある

ハッシュテーブルとは

- ハッシュテーブルはキーをもとに生成されたハッシュ値を添え字とした配列である。通常、配列の添え字には非負整数しか扱えない。
- そこで、キーを要約する値であるハッシュ値を添え字として値を管理することで、検索や追加を要素数によらず定数時間O(1)で実現する。
- しかしハッシュ関数の選び方(例えば、異なるキーから頻繁に同じハッシュ値が生成される場合)によっては、性能が劣化して最悪の場合O(n)となってしまう。

https://ja.wikipedia.org/wiki/ハッシュテーブル

ハッシュテーブルとは



キーに対応するハッシュ値を添字として使う

ハッシュ値が衝突する場合同じ位置に収められ、追加や検索の計算量は増加する(単純な実装の場合)

Java標準ライブラリのハッシュテーブル

- **Object**クラスのメソッドとして**hashCode()**が実装されている
  - したがって、全てのクラスに**hashCode()**が継承されている

- JavaSEの標準ライブラリとして**Hashtable**クラス、**HashMap**クラスなどが提供されている
- **Hashtable**クラスのサブクラスとして**Java.util.Properties**クラス

- **String**クラスでは**hashcode()**をオーバーライド
  - ハッシュテーブルにおけるキーとして**String**オブジェクトがよく使われる

String.hashCode()の実装

```
// String クラス
public final class String {
  private final char value[]; // 文字列データ
  private final int offset;
  private final int count; // 文字数
  private int hash; // 計算したハッシュ値のキャッシュ
  ...
  public int hashCode() {
    int h = hash;
    if (h == 0 && count > 0) {
      int off = offset;
      char val[] = value;
      int len = count;

      for (int i = 0; i < len; i++) {
        h = 31*h + val[off++];
      }
      hash = h;
    }
    return h;
  }
  ...
}
```

JavaSE7 の src.zip

String.hashCode()の実装

- n文字からなる文字列 s[0] s[1] ... s[n-1]のハッシュ値は以下のように計算される
  - s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

- hash==0ならば、まだ計算したことがないとみなしてハッシュ値を計算
- 計算結果はいつも同じ

ハッシュ値が衝突するような入力を大量に処理させることができれば **DoS**攻撃可能!!

Stringのハッシュ値

- Aa → 2122
- BB → 2122
- AaAa → 2031744
- AaBB → 2031744
- BBAa → 2031744
- BBBB → 2031744
- AaAaAaAa → -540425984
- AaAaAaBB → -540425984
- AaBBAaAa → -540425984
- AaBBAaBB → -540425984
- ....

ハッシュ値が衝突する文字列をいくらでも作ることができる

ハッシュテーブルへの攻撃

- USENIX Security Symposium 2003
  - http://static.usenix.org/events/sec03/tech/crosby.html
- Chaos Communication Congress 2011
  - http://events.ccc.de/congress/2011/Fahrplan/events/4680.en.html

公開日：2012/01/04　最終更新日：2012/02/10

JVNVU#903934
ハッシュ関数を使用しているウェブアプリケーションにサービス運用妨害 (DoS) の脆弱性

概要
ハッシュ関数を使用している複数のウェブアプリケーションには、サービス運用妨害 (DoS) の脆弱性が存在します。

影響を受けるシステム
対象となる製品は複数存在します。詳しくは開発者が提供する情報をご確認ください。

詳細情報
複数のウェブアプリケーションフレームワークで使用されているハッシュ関数の実装には、ハッシュテーブルへのデータ追加処理に問題があり、サービス運用妨害 (DoS) の脆弱性が存在します。

想定される影響
同一のハッシュ値となるリクエストを大量に受信することで、サービス運用妨害 (DoS) 攻撃を受ける可能性があります。

様々なソフトウェアへの影響

- Webアプリケーションフレームワーク
  - .Net Framework, ...
- アプリケーションサーバ
  - Tomcat, …
- 言語
  - Perl, Python, PHP, ...
- ネームサーバ
  - MaraDNS, ...
- ライブラリ
  - libxml2, glib2, ...

HashDoS対策

- ハッシュ関数を攻撃されにくいものにする
  - 乱数化ハッシュ関数
- 入力を制限する
  - POSTリクエストのサイズやパラメータ数を制限
  - (Tomcatや.Net Frameworkなど)

乱数化ハッシュ関数

- Perlが2003年に行なった対策
- Bob Jenkin の one-at-a-time ハッシュと呼ばれるアルゴリズム
- プロセス毎に生成した乱数を、ハッシュ関数の計算に取り入れる
  - プロセス毎にハッシュ関数の性質が変化する
- ハッシュ値が衝突するような入力をあらかじめ用意することが困難になる

以下のコード例ではC言語で書かれた Perl のコードを Java のコードに焼き直しました

乱数化ハッシュ関数

RString クラス(1/2)

```
class RString {

  String string;
  static SecureRandom seed;
  static byte bytes[];

  static {
    seed = new SecureRandom();
    bytes = new byte[4];
    seed.nextBytes(bytes);
  }

  public RString(String s){
    string = s;
  }
```

乱数化ハッシュ関数

RString クラス(2/2)

```
@Override
 public int hashCode() {
   int h = 0;
   for (int i=0; i<4; i++){
     h = h * 256 + bytes[i];
   }
   int len = string.length();
   for (int i=len-1; 0<=i; i--){
     h += string.charAt(i);
     h += (h << 10);
     h ^= (h >> 6);
   }
   h += (h << 3);
   h ^= (h >> 11);
   h = (h + (h << 15));
   return h;
  }
  @Override
  public String toString(){ return string; }
}
```

乱数化ハッシュ関数

RStringTestクラス

```
class RStringTest {
  public static void main(String[] args){
    String[] strlist =
      {“Aa”, “BB”, “AaAa”, “BBAa”};
    for (String s : strlist){
      System.out.println(s + “ -> “
          + “String “ + s.hashCode()
          + “¥tRString “
          + new RString(s).hashCode());
    }
  }
}
```

乱数化ハッシュ関数

RStringTest の実行結果

```
$ java RStringTest
Aa -> String 2112    RString -1651452168
BB -> String 2112    RString 381866510
AaAa -> String 2031744    RString 1223770847
BBAa -> String 2031744    RString 1278012906
$ java RStringTest
Aa -> String 2112    RString 1684681767
BB -> String 2112    RString 2050901943
AaAa -> String 2031744    RString 691142468
BBAa -> String 2031744    RString -1042126578
$ java RStringTest
Aa -> String 2112    RString 1407843984
BB -> String 2112    RString -1833917312
AaAa -> String 2031744    RString 1548747896
BBAa -> String 2031744    RString 1136654132
$
```

ハッシュテーブル

- 信頼できない入力をハッシュテーブルに登録する処理を行っている場合, DoS攻撃に悪用される危険がある

- 入力データのハッシュ値衝突がアプリケーションにとって重要な問題である場合, ハッシュテーブル以外のデータ構造(例えばTreeMap<K,V>)の使用を検討すること