Hotspot Command-line Flags Clean Up - Design Proposal for JDK-8243208

Latest Webrev: http://cr.openjdk.java.net/~iklam/jdk16/8243208-cleanup-jvmflag-impl.v00/

Goals

1. Reduce the complexity of macros related to command-line flags (JDK-8243208)
2. Avoid including large globals.hpp in every file. Just include the <mod>_globals.hpp file of the module you need (separate RFE: JDK-8243205)
3. Improvement footprint/performance

Non Goals

1. The flag declaration is done with large X-macros. Although some people find this objectionably, this proposal does not change that.

Requirements for Command-line Flags

The flags implementation in HotSpot has evolved over 20 years without much documentation.

By reading the JDK 15 code, I can gather the following requirements – which lead to the complex implementation.

(These requirements cannot be implemented elegantly/efficiently using older C++ compilers, leading to the current messy code)

- **[REQ1]** Flags have 3 **types**
  - PRODUCT - readable/writable in all builds
  - DEVELOP - readable in all builds, writable only in debug builds
  - NOTPRODUCT - readable/writable only in debug builds
    - Using a flag of this type in a product build will result in C compiler error.
- **[REQ2]** Flags can have optional **attributes**
  - MANAGEABLE, DIAGNOSTIC, EXPERIMENTAL
- **[REQ3]** Each flag can be in at most one of the following **groups**.
  - C1, C2, JVMCI, ARCH, LP64
  - (many flags aren't in any of these groups)
- **[REQ4]** Each flag must have a **default value**
  - Platform-defined default values are specified by XXX_PD macros
- **[REQ5]** Flag declarations should be **concise** (default flag attributes shouldn't need to be specified)
- **[REQ6]** Metadata for flags must be **compile-time generated**
  - Good: stored in .bss section
  - Bad: initialized with global constructors

Problems with command-line flags in JDK 15

First, although types, attributes and groups are orthogonal, JDK 15 allows you to only specify a limited number of combinations. For example, all experimental flags must be of the PRODUCT type:

```
develop(bool, CleanChunkPoolAsync, true, \
       "Clean the chunk pool asynchronously") \
experimental(bool, AlwaysSafeConstructors, false, \
       "Force safe construction, as if all fields are final.") \
```

In a way, the **[REQ5] conciseness** requirement is the root cause of the messy implementation. If we add an extra "kind" parameter to the macros, we can convert the above to the following, which would allow the second flag to be both EXPERIMENTAL and DEVELOP

```
develop(bool, CleanChunkPoolAsync, true, /*kind=*/ 0\
       "Clean the chunk pool asynchronously") \
develop(bool, AlwaysSafeConstructors, false, /*kind=*/ FLAG_KIND_EXPERIMENTAL \
       "Force safe construction, as if all fields are final.") \
```

Similarly, groups could be implemented as something like the following (instead of the messy macros in here).

```
/*kind=*/ FLAG_KIND_EXPERIMENTAL|FLAG_GROUP_C1
```

Another way to keep the code concise is to use constructor overloading in the flags definition, something like (simplified)

```
#ifndef CONSTEXPR
#define CONSTEXPR
#endif
#define  FLAG_KIND_EXPERIMENTAL 1
struct Flag {
    const char* name; int kind;
    CONSTEXPR Flag(const char* n): name(n), kind(0) {}
    CONSTEXPR Flag(const char* n, int k): name(n), kind(k) {}
};
Flag flags[] = {
    Flag("CleanChunkPoolAsync"),
    Flag("AlwaysSafeConstructors", FLAG_KIND_EXPERIMENTAL),
};
```

However, this runs into problem with **[REQ6].** Even the effect of the above code can be completely decided at compilation time, without constexpr, GCC stubbornly insists to initialize the array using global constructors

```
$ /home/iklam/devkit/latest/bin/gcc -O3 -o - -S test.cpp
....
_GLOBAL__sub_I_flags:
.LFB7:
        .cfi_startproc
        movq        $.LC0, flags(%rip)
        movl        $0, flags+8(%rip)
        movq        $.LC1, flags+16(%rip)
        movl        $1, flags+24(%rip)
        ret
```

Another option is to use clever macros. However, there's no vararg macro that I can think of that can satisfy all of the above macros

Proposal - Use constexpr!

C++ constexpr makes things much easier:

```
$ /home/iklam/devkit/latest/bin/gcc -O3 -o - -S -DCONSTEXPR=constexpr test.cpp
....
flags:
        .quad        .LC0  // name
        .long        0     // kind
        .quad        .LC1  // name
        .long        1     // kind
```

New way of declaring flags - with optional argument for **attributes**. Also, the flag declaration macros now all take only 7 arguments: (compared to old version in globals.hpp)

```
#define RUNTIME_FLAGS(develop, \
                      develop_pd, \
                      product, \
                      product_pd, \
                      /* REMOVED diagnostic,   */\
                      /* REMOVED diagnostic_pd, */ \
                      /* REMOVED experimental,  */\
                      notproduct, \
                      /* REMOVED manageable,   */ \
                      /* REMOVED product_rw,   */\
                      /* REMOVED lp64_product, */\
                      range, \
                      constraint) \
  .....
  develop(bool, CleanChunkPoolAsync, true,  /* no attr */                  \
          "Clean the chunk pool asynchronously")                           \
                                                                           \
  product(uint, HandshakeTimeout, 0, /* attr= */DIAGNOSTIC,                \
          "If nonzero set a timeout in milliseconds for handshakes")       \
                                                                           \
  product(bool, AlwaysSafeConstructors, false, /* attr= */ EXPERIMENTAL,    \
          "Force safe construction, as if all fields are final.")         ....
```

The flags metadata is defined using overloaded constructors (around lin 679 of jvmFlag.cpp)

```
constexpr JVMFlag::JVMFlag(int flag_enum, const char* type, const char* name,
                           void* addr, int attrs, const char* doc) :
  _type(type), _name(name), _addr(addr) NOT_PRODUCT(COMMA _doc(doc)) {}

constexpr JVMFlag::JVMFlag(int flag_enum,  const char* type, const char* name,
                           void* addr, const char* doc) :
  JVMFlag(flag_enum, type, name, addr, /*attrs*/0, doc) {}

....
#define DEVELOP_FLAG_INIT(   type, name, value, ...) JVMFlag(FLAG_MEMBER_ENUM(name), ...snip..., __VA_ARGS__),
#define DEVELOP_FLAG_INIT_PD(type, name,        ...) JVMFlag(FLAG_MEMBER_ENUM(name), ...snip..., __VA_ARGS__),
...snip....

static JVMFlag flagTable[NUM_JVMFlagsEnum + 1] = {
  ALL_FLAGS(DEVELOP_FLAG_INIT,
            DEVELOP_FLAG_INIT_PD,
            PRODUCT_FLAG_INIT,
            PRODUCT_FLAG_INIT_PD,
            NOTPROD_FLAG_INIT,
            IGNORE_RANGE,
            IGNORE_CONSTRAINT)
  ....
// NOTE: ALL_FLAGS() calls RUNTIME_FLAGS()
```

Handling of Groups:

There's a very small number of groups. They don't seem very useful so I don't know if we will add many new groups in the future. So for the time being, I implemented groups by ordering the flags and counting the size of each group:

```
#define ENUM_F(type, name, ...)  enum_##name,
#define IGNORE_F(...)

//                                                 dev     dev-pd  pro     pro-pd  notpro  range     constraint
enum FlagCounter_LP64  { LP64_RUNTIME_FLAGS(        ENUM_F, ENUM_F, ENUM_F, ENUM_F, ENUM_F, IGNORE_F,
IGNORE_F)  num_flags_LP64   };
enum FlagCounter_JVMCI { JVMCI_ONLY(JVMCI_FLAGS(    ENUM_F, ENUM_F, ENUM_F, ENUM_F, ENUM_F, IGNORE_F,
IGNORE_F)) num_flags_JVMCI  };
enum FlagCounter_C1    { COMPILER1_PRESENT(C1_FLAGS(ENUM_F, ENUM_F, ENUM_F, ENUM_F, ENUM_F, IGNORE_F,
IGNORE_F)) num_flags_C1     };
enum FlagCounter_C2    { COMPILER2_PRESENT(C2_FLAGS(ENUM_F, ENUM_F, ENUM_F, ENUM_F, ENUM_F, IGNORE_F,
IGNORE_F)) num_flags_C2     };
enum FlagCounter_ARCH  { ARCH_FLAGS(                ENUM_F,         ENUM_F,         ENUM_F, IGNORE_F,
IGNORE_F)  num_flags_ARCH   };

const int first_flag_enum_LP64   = 0;
const int first_flag_enum_JVMCI  = first_flag_enum_LP64  + num_flags_LP64;
const int first_flag_enum_C1     = first_flag_enum_JVMCI + num_flags_JVMCI;
const int first_flag_enum_C2     = first_flag_enum_C1    + num_flags_C1;
const int first_flag_enum_ARCH   = first_flag_enum_C2    + num_flags_C2;
const int first_flag_enum_other  = first_flag_enum_ARCH  + num_flags_ARCH;

static constexpr inline int flag_group(int flag_enum) {
  if (flag_enum < first_flag_enum_JVMCI) return JVMFlag::KIND_LP64_PRODUCT;
  if (flag_enum < first_flag_enum_C1)    return JVMFlag::KIND_JVMCI;
  if (flag_enum < first_flag_enum_C2)    return JVMFlag::KIND_C1;
  if (flag_enum < first_flag_enum_ARCH)  return JVMFlag::KIND_C2;
  if (flag_enum < first_flag_enum_other) return JVMFlag::KIND_ARCH;

  return 0;
}
```

Handling of types

Same as before, just fewer cases (old version here)

```
// Interface macros
#define DECLARE_PRODUCT_FLAG(type, name, value, ...)      extern "C" type name;
#define DECLARE_PD_PRODUCT_FLAG(type, name, ...)          extern "C" type name;
#ifdef PRODUCT
#define DECLARE_DEVELOPER_FLAG(type, name, value, ...)    const type name = value;
#define DECLARE_PD_DEVELOPER_FLAG(type, name, ...)        const type name = pd_##name;
#define DECLARE_NOTPRODUCT_FLAG(type, name, value, ...)   const type name = value;
#else
#define DECLARE_DEVELOPER_FLAG(type, name, value, ...)    extern "C" type name;
#define DECLARE_PD_DEVELOPER_FLAG(type, name, ...)        extern "C" type name;
#define DECLARE_NOTPRODUCT_FLAG(type, name, value, ...)   extern "C" type name;
#endif // PRODUCT

ALL_FLAGS(DECLARE_DEVELOPER_FLAG,
          DECLARE_PD_DEVELOPER_FLAG,
          DECLARE_PRODUCT_FLAG,
          DECLARE_PD_PRODUCT_FLAG,
          DECLARE_NOTPRODUCT_FLAG,
          IGNORE_RANGE,
          IGNORE_CONSTRAINT)
```

Range and Constraint Checking

The old code has 2 problems

- Builds list of range/constraint checking objects at VM start-up (lots of code and slow start)
- Range/constraint checking needs linear search (further slows down start-up)

The new design (see here for webrev):

- Builds the checker objects (JVMFlagLimit) at build-time using constexpr.
- JVMFlagLimit objects are indexed by each flag's enum (or NULL if no limit exists), so it's O(1) time.

Implementation of JVMFlagLimit

The range/constraint information for a flag of type **T** is described by a **JVMTypedFlagLimit<T>**:

```
class JVMFlagLimit {
  enum {
    HAS_RANGE = 1,
    HAS_CONSTRAINT = 2
  };

  short _constraint_func;
  char  _phase;
  char  _kind; ...};

template <typename T>
class JVMTypedFlagLimit : public JVMFlagLimit {
  const T _min;
  const T _max; ...};
```

Each flag is given a unique enum that starts from **0** to **NUM_JVMFlagsEnum-1**. We use this enum to find the **JVMTypedFlagLimit<T>** of this flag from an array:

```
static constexpr const JVMFlagLimit* const flagLimitTable[1 + NUM_JVMFlagsEnum] = { .... }
const JVMFlagLimit* const* JVMFlagLimit::flagLimits = &flagLimitTable[1]; // excludes dummy

/* E.g., to get the limit of this flag:
  product(intx, ContendedPaddingWidth, 128,                                  \
          "How many bytes to pad the fields/classes marked @Contended with")\
          range(0, 8192)                                                     \
          constraint(ContendedPaddingWidthConstraintFunc,AfterErgo)          \
*/
const JVMTypedFlagLimit<intx>* limit = JVMFlagLimit::flagLimits[Flag_ContendedPaddingWidth_Enum];

// We will see these fields:
// limit->_constraint_func   ==> constraint_enum_ContendedPaddingWidthConstraintFunc  (more on this below)
// limit->_phase             ==> AfterErgo
// limit->_kind              ==> HAS_RANGE | HAS_CONSTRAINT
// limit->_min               ==> 0
// limit->_max               ==> 8192
```

Most flags have neither range nor constraint. For those flags, we want its **flagLimits[Flag_*flagname*_Enum]** to be **NULL**.

To do this, we first define a **JVMTypedFlagLimit<T>** variable for each flag (including the ones that don't have range/constraint). It's done by this macro:

```
//          macro body starts here ------------------+
//                                                   |
//                                                   v
#define FLAG_LIMIT_DEFINE(      type, name, ...)       ); constexpr JVMTypedFlagLimit<type> limit_##name(0
#define FLAG_LIMIT_DEFINE_DUMMY(type, name, ...)       ); constexpr DummyLimit nolimit_##name(0
#define FLAG_LIMIT_PTR(         type, name, ...)       ), LimitGetter<type>::get_limit(&limit_##name, 0
#define FLAG_LIMIT_PTR_NONE(    type, name, ...)       ), LimitGetter<type>::no_limit(0
#define APPLY_FLAG_RANGE(...)                          , __VA_ARGS__
#define APPLY_FLAG_CONSTRAINT(func, phase)             , next_two_args_are_constraint, (short)CONSTRAINT_ENUM
(func), int(JVMFlagConstraint::phase)

constexpr JVMTypedFlagLimit<int> limit_dummy
(
#ifdef PRODUCT
 ALL_FLAGS(FLAG_LIMIT_DEFINE_DUMMY,
           FLAG_LIMIT_DEFINE_DUMMY,
           FLAG_LIMIT_DEFINE,
           FLAG_LIMIT_DEFINE,
           FLAG_LIMIT_DEFINE_DUMMY,
           APPLY_FLAG_RANGE,
           APPLY_FLAG_CONSTRAINT)
#else
 ALL_FLAGS(FLAG_LIMIT_DEFINE,
           FLAG_LIMIT_DEFINE,
           FLAG_LIMIT_DEFINE,
           FLAG_LIMIT_DEFINE,
           FLAG_LIMIT_DEFINE,
           APPLY_FLAG_RANGE,
           APPLY_FLAG_CONSTRAINT)
#endif
);
```

To understand how the macros work, it's best to compile **jvmFlagLimit.c** with **gcc -save-temps**. and look at the generated **jvmFlagLimit.ii** with the macros expanded. Here's an example:

```
// code excerpt prettified manually
constexpr JVMTypedFlagLimit<int>  limit_dummy();
constexpr JVMTypedFlagLimit<bool> limit_UseCompressedOops(0);
....
constexpr JVMTypedFlagLimit<intx> limit_ObjectAlignmentInBytes(0, 8, 256,
             next_two_args_are_constraint,
            (short)constraint_enum_ObjectAlignmentInBytesConstraintFunc, int(JVMFlagConstraint::AtParse));
....
constexpr JVMTypedFlagLimit<intx> limit_JVMCIThreads(0, 1, max_jint);
```

We use overloaded constructors to fill out the necessarily fields of the **JVMTypedFlagLimit<T>** variables. Note that the **min**/**max** parameters, as well as the **constraint_func**/**phase** parameters, can both be integer values. For disambiguation, we pass in a dummy **next_two_args_are_constraint** for the **constraint_func**/**phase**.

We also need to always pass in an initial dummy 0 parameter so that the macros can safely add a comma before passing the **min/max** or **constraint_func/phase**.

These dummy parameters are evaluated at compile time so they can be safely optimized away.

The next step is to fill out the **flagLimitTable[]** array:

```
static constexpr const JVMFlagLimit* const flagLimitTable[1 + NUM_JVMFlagsEnum] = {
  // Because FLAG_LIMIT_PTR must start with an "),", we have to place a dummy element here.
  LimitGetter<int>::get_limit(NULL, 0

#ifdef PRODUCT
  ALL_FLAGS(FLAG_LIMIT_PTR_NONE,
            FLAG_LIMIT_PTR_NONE,
            FLAG_LIMIT_PTR,
            FLAG_LIMIT_PTR,
            FLAG_LIMIT_PTR_NONE,
            APPLY_FLAG_RANGE,
            APPLY_FLAG_CONSTRAINT)
#else
  ALL_FLAGS(FLAG_LIMIT_PTR,
            FLAG_LIMIT_PTR,
            FLAG_LIMIT_PTR,
            FLAG_LIMIT_PTR,
            FLAG_LIMIT_PTR,
            APPLY_FLAG_RANGE,
            APPLY_FLAG_CONSTRAINT)
#endif
  )
};
```

For the flags shown in the example above, the following code is generated by the macros:

```
static constexpr const JVMFlagLimit* const flagLimitTable[1 + NUM_JVMFlagsEnum] = {
  LimitGetter<int>::get_limit(NULL, 0),
  LimitGetter<bool>::get_limit(&limit_UseCompressedOops, 0),
  ....
  LimitGetter<intx>::get_limit(&limit_ObjectAlignmentInBytes, 0,
             8, 256,
             next_two_args_are_constraint,
            (short)constraint_enum_ObjectAlignmentInBytesConstraintFunc, int(JVMFlagConstraint::AtParse)),
  ....
  LimitGetter<intx>::get_limit(&limit_JVMCIThreads, 0, 1, max_jint),
```

- If a flag has neither range nor constraint, we will call the **LimitGetter<T>::get_limit()** function with two parameters, which returns NULL.
- If a flag has range and/or constraint, we will call a **LimitGetter<T>::get_limit()** function with more than two parameters. These functions would return the same **JVMTypedFlagLimit<T>** as passed in.

As a result, we will end up with this in the final output of the C++ compiler:

```
static constexpr const JVMFlagLimit* const flagLimitTable[1 + NUM_JVMFlagsEnum] = {
  NULL, // dummy
  NULL, // UseCompressedOops has no range/constraint
  ....
  &limit_ObjectAlignmentInBytes
  ....
  &limit_JVMCIThreads
```

What happens to unreferenced flag limits

All the flag limits are defined with the **constexpr** keyword, which has internal linkage by default. If a flag has no range/constraint, its flag limit (e.g., **limit_UseCompressedOops** in the example above) will be unused, and will be eliminated by the C++ compiler from the object file. So we don't waste any space.

Why use enums for constraint_func

This is a small optimization: There are 120 flags that use a constraint function, but there are only 65 total constraint functions. By using a short index, we can:

- Reduce the size of the JVMFlagLimit object (the 2 bytes fits in unused space)
- Reduce the number of pointers relocated when libjvm.so is dynamically loaded (from 120 to 65).

The savings are not a big deal, but since we can do it, why not?

Is constexpr really working?

A good way to check is to build the .o with something like "gcc -save-temps" and look at the .s file. Here's an example of jvmFlagLimit.s. You can see that the content of the flagLimitTable is also completely determined at build time (it's in the "ro" section):

```
        .section        .data.rel.ro.local,"aw"
        .align 32
        .type        _ZL14flagLimitTable, @object
        .size        _ZL14flagLimitTable, 9728
_ZL14flagLimitTable:
        .quad        0
        .quad        0
        .quad        0
        .quad        _ZL28limit_ObjectAlignmentInBytes
        .quad        0
        .quad        0
        .quad        0
        .quad        0
        .quad        0
        .quad        0
        .quad        0
        .quad        _ZL18limit_JVMCIThreads
        .quad        _ZL22limit_JVMCIHostThreads
    ....
```