```
**********************************************************
   44282 Wed Oct 12 04:37:37 2011
new/src/share/vm/ci/ciEnv.cpp
**********************************************************
______unchanged_portion_omitted_
```

```
// ------------------------------------------------------------------
// ciEnv::get_klass_by_name_impl
ciKlass* ciEnv::get_klass_by_name_impl(ciKlass* accessing_klass,
                                       constantPoolHandle cpool,
                                       ciSymbol* name,
                                       bool require_local) {
  ASSERT_IN_VM;
  EXCEPTION_CONTEXT;

  // Now we need to check the SystemDictionary
  Symbol* sym = name->get_symbol();
  if (sym->byte_at(0) == 'L' &&
    sym->byte_at(sym->utf8_length()-1) == ';') {
    // This is a name from a signature.  Strip off the trimmings.
    // Call recursive to keep scope of strippedsym.
    TempNewSymbol strippedsym = SymbolTable::new_symbol(sym->as_utf8()+1,
                    sym->utf8_length()-2,
                    KILL_COMPILE_ON_FATAL_(_unloaded_ciinstance_klass));
    ciSymbol* strippedname = get_symbol(strippedsym);
    return get_klass_by_name_impl(accessing_klass, cpool, strippedname, require_
  }

  // Check for prior unloaded klass.  The SystemDictionary's answers
  // can vary over time but the compiler needs consistency.
  ciKlass* unloaded_klass = check_get_unloaded_klass(accessing_klass, name);
  if (unloaded_klass != NULL) {
    if (require_local)  return NULL;
    return unloaded_klass;
  }

  Handle loader(THREAD, (oop)NULL);
  Handle domain(THREAD, (oop)NULL);
  if (accessing_klass != NULL) {
    loader = Handle(THREAD, accessing_klass->loader());
    domain = Handle(THREAD, accessing_klass->protection_domain());
  }

  // setup up the proper type to return on OOM
  ciKlass* fail_type;
  if (sym->byte_at(0) == '[') {
    fail_type = _unloaded_ciobjarrayklass;
  } else {
    fail_type = _unloaded_ciinstance_klass;
  }
  KlassHandle found_klass;
  {
    ttyUnlocker ttyul;  // release tty lock to avoid ordering problems
    MutexLocker ml(Compile_lock);
    klassOop kls;
    if (!require_local) {
      kls = SystemDictionary::find_constrained_instance_or_array_klass(sym, load
                                                                       KILL_COMP
    } else {
      kls = SystemDictionary::find_instance_or_array_klass(sym, loader, domain,
                                                           KILL_COMPILE_ON_FATAL
    }
    found_klass = KlassHandle(THREAD, kls);
  }

  // If we fail to find an array klass, look again for its element type.
```

```
  // The element type may be available either locally or via constraints.
  // In either case, if we can find the element type in the system dictionary,
  // we must build an array type around it.  The CI requires array klasses
  // to be loaded if their element klasses are loaded, except when memory
  // is exhausted.
  if (sym->byte_at(0) == '[' &&
      (sym->byte_at(1) == '[' || sym->byte_at(1) == 'L')) {
    // We have an unloaded array.
    // Build it on the fly if the element class exists.
    TempNewSymbol elem_sym = SymbolTable::new_symbol(sym->as_utf8()+1,
                                                 sym->utf8_length()-1,
                                                 KILL_COMPILE_ON_FATAL_(fail_typ

    // Get element ciKlass recursively.
    ciKlass* elem_klass =
      get_klass_by_name_impl(accessing_klass,
                             cpool,
                             get_symbol(elem_sym),
                             require_local);
    if (elem_klass != NULL && elem_klass->is_loaded()) {
      // Now make an array for it
      return ciObjArrayKlass::make_impl(elem_klass);
    }
  }

  if (found_klass() == NULL && !cpool.is_null() && cpool->has_preresolution()) {
    // Look inside the constant pool for pre-resolved class entries.
    for (int i = cpool->length() - 1; i >= 1; i--) {
      if (cpool->tag_at(i).is_klass()) {
        klassOop kls = cpool->resolved_klass_at(i);
        if (Klass::cast(kls)->name() == sym) {
          found_klass = KlassHandle(THREAD, kls);
          break;
        }
      }
    }
  }

  if (found_klass() != NULL) {
    // Found it.  Build a CI handle.
    return get_object(found_klass())->as_klass();
  }

  if (require_local)  return NULL;

#endif /* ! codereview */
  // Not yet loaded into the VM, or not governed by loader constraints.
  // Make a CI representative for it.
  return get_unloaded_klass(accessing_klass, name);
}

// ------------------------------------------------------------------
// ciEnv::get_klass_by_name
ciKlass* ciEnv::get_klass_by_name(ciKlass* accessing_klass,
                                  ciSymbol* klass_name,
                                  bool require_local) {
  GUARDED_VM_ENTRY(return get_klass_by_name_impl(accessing_klass,
                                                 constantPoolHandle(),
                                                 klass_name,
                                                 require_local);)
}

// ------------------------------------------------------------------
// ciEnv::get_klass_by_index_impl
//
// Implementation of get_klass_by_index.
```

```
ciKlass* ciEnv::get_klass_by_index_impl(constantPoolHandle cpool,
                                        int index,
                                        bool& is_accessible,
                                        ciInstanceKlass* accessor) {
  EXCEPTION_CONTEXT;
  KlassHandle klass(THREAD, constantPoolOopDesc::klass_at_if_loaded(cpool, index
  KlassHandle klass (THREAD, constantPoolOopDesc::klass_at_if_loaded(cpool, inde
  Symbol* klass_name = NULL;
  if (klass.is_null()) {
    // The klass has not been inserted into the constant pool.
    // Try to look it up by name.
    {
      // We have to lock the cpool to keep the oop from being resolved
      // while we are accessing it.
      ObjectLocker ol(cpool, THREAD);

      constantTag tag = cpool->tag_at(index);
      if (tag.is_klass()) {
        // The klass has been inserted into the constant pool
        // very recently.
        klass = KlassHandle(THREAD, cpool->resolved_klass_at(index));
      } else if (tag.is_symbol()) {
        klass_name = cpool->symbol_at(index);
      } else {
        assert(cpool->tag_at(index).is_unresolved_klass(), "wrong tag");
        klass_name = cpool->unresolved_klass_at(index);
      }
    }
  }

  if (klass.is_null()) {
    // Not found in constant pool.  Use the name to do the lookup.
    ciKlass* k = get_klass_by_name_impl(accessor,
                                        cpool,
                                        get_symbol(klass_name),
                                        false);
    // Calculate accessibility the hard way.
    if (!k->is_loaded()) {
      is_accessible = false;
    } else if (k->loader() != accessor->loader() &&
               get_klass_by_name_impl(accessor, cpool, k->name(), true) == NULL)
      // Loaded only remotely.  Not linked yet.
      is_accessible = false;
    } else {
      // Linked locally, and we must also check public/private, etc.
      is_accessible = check_klass_accessibility(accessor, k->get_klassOop());
    }
    return k;
  }

  // Check for prior unloaded klass.  The SystemDictionary's answers
  // can vary over time but the compiler needs consistency.
  ciSymbol* name = get_symbol(klass()->klass_part()->name());
  ciKlass* unloaded_klass = check_get_unloaded_klass(accessor, name);
  if (unloaded_klass != NULL) {
    is_accessible = false;
    return unloaded_klass;
  }

  // It is known to be accessible, since it was found in the constant pool.
  is_accessible = true;
  return get_object(klass())->as_klass();
}
```

______unchanged_portion_omitted_

```
// ------------------------------------------------------------------
// ciEnv::get_method_by_index_impl
ciMethod* ciEnv::get_method_by_index_impl(constantPoolHandle cpool,
                                          int index, Bytecodes::Code bc,
                                          ciInstanceKlass* accessor) {
  int holder_index = cpool->klass_ref_index_at(index);
  bool holder_is_accessible;
  ciKlass* holder = get_klass_by_index_impl(cpool, holder_index, holder_is_acces
  ciInstanceKlass* declared_holder = get_instance_klass_for_declared_method_hold

  // Get the method's name and signature.
  Symbol* name_sym = cpool->name_ref_at(index);
  Symbol* sig_sym  = cpool->signature_ref_at(index);

  if (cpool->has_preresolution()
      || (holder == ciEnv::MethodHandle_klass() &&
          methodOopDesc::is_method_handle_invoke_name(name_sym))) {
    // Short-circuit lookups for JSR 292-related call sites.
    // That is, do not rely only on name-based lookups, because they may fail
    // if the names are not resolvable in the boot class loader (7056328).
    switch (bc) {
    case Bytecodes::_invokevirtual:
    case Bytecodes::_invokeinterface:
    case Bytecodes::_invokespecial:
    case Bytecodes::_invokestatic:
      {
        methodOop m = constantPoolOopDesc::method_at_if_loaded(cpool, index, bc)
        if (m != NULL) {
          return get_object(m)->as_method();
        }
      }
    }
  }

  if (holder_is_accessible) { // Our declared holder is loaded.
    instanceKlass* lookup = declared_holder->get_instanceKlass();
    methodOop m = lookup_method(accessor->get_instanceKlass(), lookup, name_sym,
    if (m != NULL &&
        (bc == Bytecodes::_invokestatic
         ?  instanceKlass::cast(m->method_holder())->is_not_initialized()
         : !instanceKlass::cast(m->method_holder())->is_loaded())) {
      m = NULL;
    }
    if (m != NULL) {
      // We found the method.
      return get_object(m)->as_method();
    }
  }

  // Either the declared holder was not loaded, or the method could
  // not be found.  Create a dummy ciMethod to represent the failed
  // lookup.
  ciSymbol* name      = get_symbol(name_sym);
  ciSymbol* signature = get_symbol(sig_sym);
  return get_unloaded_method(declared_holder, name, signature, accessor);

  return get_unloaded_method(declared_holder,
                             get_symbol(name_sym),
                             get_symbol(sig_sym));
}


// ------------------------------------------------------------------
// ciEnv::get_fake_invokedynamic_method_impl
ciMethod* ciEnv::get_fake_invokedynamic_method_impl(constantPoolHandle cpool,
                                                    int index, Bytecodes::Code b
                                                    ciInstanceKlass* accessor) {
                                                    int index, Bytecodes::Code b
  // Compare the following logic with InterpreterRuntime::resolve_invokedynamic.
  assert(bc == Bytecodes::_invokedynamic, "must be invokedynamic");

  bool is_resolved = cpool->cache()->main_entry_at(index)->is_resolved(bc);
  if (is_resolved && cpool->cache()->secondary_entry_at(index)->is_f1_null())
    // FIXME: code generation could allow for null (unlinked) call site
    is_resolved = false;

  // Call site might not be resolved yet.  We could create a real invoker method
  // compiler, but it is simpler to stop the code path here with an unlinked met
  if (!is_resolved) {
    ciInstanceKlass* holder    = get_object(SystemDictionary::MethodHandle_klass
    ciSymbol*        name      = ciSymbol::invokeExact_name();
    ciSymbol*        signature = get_symbol(cpool->signature_ref_at(index));
    return get_unloaded_method(holder, name, signature, accessor);
    ciInstanceKlass* mh_klass = get_object(SystemDictionary::MethodHandle_klass(
    ciSymbol*        sig_sym  = get_symbol(cpool->signature_ref_at(index));
    return get_unloaded_method(mh_klass, ciSymbol::invokeExact_name(), sig_sym);
  }

  // Get the invoker methodOop from the constant pool.
  oop f1_value = cpool->cache()->main_entry_at(index)->f1();
  methodOop signature_invoker = (methodOop) f1_value;
  assert(signature_invoker != NULL && signature_invoker->is_method() && signatur
         "correct result from LinkResolver::resolve_invokedynamic");

  return get_object(signature_invoker)->as_method();
}
```
______***unchanged_portion_omitted_***

```
// ------------------------------------------------------------------
// ciEnv::get_method_by_index
ciMethod* ciEnv::get_method_by_index(constantPoolHandle cpool,
                                     int index, Bytecodes::Code bc,
                                     ciInstanceKlass* accessor) {
  if (bc == Bytecodes::_invokedynamic) {
    GUARDED_VM_ENTRY(return get_fake_invokedynamic_method_impl(cpool, index, bc,
    GUARDED_VM_ENTRY(return get_fake_invokedynamic_method_impl(cpool, index, bc)
  } else {
    GUARDED_VM_ENTRY(return get_method_by_index_impl(          cpool, index, bc,
    GUARDED_VM_ENTRY(return get_method_by_index_impl(cpool, index, bc, accessor)
  }
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   16638 Wed Oct 12 04:37:38 2011
new/src/share/vm/ci/ciEnv.hpp
**********************************************************
/*
 * Copyright (c) 1999, 2011, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_CI_CIENV_HPP
#define SHARE_VM_CI_CIENV_HPP

#include "ci/ciClassList.hpp"
#include "ci/ciObjectFactory.hpp"
#include "classfile/systemDictionary.hpp"
#include "code/debugInfoRec.hpp"
#include "code/dependencies.hpp"
#include "code/exceptionHandlerTable.hpp"
#include "compiler/oopMap.hpp"
#include "runtime/thread.hpp"

class CompileTask;

// ciEnv
//
// This class is the top level broker for requests from the compiler
// to the VM.
class ciEnv : StackObj {
  CI_PACKAGE_ACCESS_TO

  friend class CompileBroker;
  friend class Dependencies;  // for get_object, during logging

private:
  Arena*           _arena;       // Alias for _ciEnv_arena except in init_shared
  Arena            _ciEnv_arena;
  int              _system_dictionary_modification_counter;
  ciObjectFactory* _factory;
  OopRecorder*     _oop_recorder;
  DebugInformationRecorder* _debug_info;
  Dependencies*    _dependencies;
  const char*      _failure_reason;
  int              _compilable;
  bool             _break_at_compile;
  int              _num_inlined_bytecodes;
  CompileTask*     _task;           // faster access to CompilerThread::task
  CompileLog*      _log;            // faster access to CompilerThread::log
```

```
  void*            _compiler_data;  // compiler-specific stuff, if any

  char* _name_buffer;
  int   _name_buffer_len;

  // Cache Jvmti state
  bool  _jvmti_can_hotswap_or_post_breakpoint;
  bool  _jvmti_can_access_local_variables;
  bool  _jvmti_can_post_on_exceptions;

  // Cache DTrace flags
  bool  _dtrace_extended_probes;
  bool  _dtrace_monitor_probes;
  bool  _dtrace_method_probes;
  bool  _dtrace_alloc_probes;

  // Distinguished instances of certain ciObjects..
  static ciObject*              _null_object_instance;
  static ciMethodKlass*         _method_klass_instance;
  static ciKlassKlass*          _klass_klass_instance;
  static ciInstanceKlassKlass*  _instance_klass_klass_instance;
  static ciTypeArrayKlassKlass* _type_array_klass_klass_instance;
  static ciObjArrayKlassKlass*  _obj_array_klass_klass_instance;

#define WK_KLASS_DECL(name, ignore_s, ignore_o) static ciInstanceKlass* _##name;
  WK_KLASSES_DO(WK_KLASS_DECL)
#undef WK_KLASS_DECL

  static ciSymbol*        _unloaded_cisymbol;
  static ciInstanceKlass* _unloaded_ciinstance_klass;
  static ciObjArrayKlass* _unloaded_ciobjarrayklass;

  static jobject _ArrayIndexOutOfBoundsException_handle;
  static jobject _ArrayStoreException_handle;
  static jobject _ClassCastException_handle;

  ciInstance* _NullPointerException_instance;
  ciInstance* _ArithmeticException_instance;
  ciInstance* _ArrayIndexOutOfBoundsException_instance;
  ciInstance* _ArrayStoreException_instance;
  ciInstance* _ClassCastException_instance;

  ciInstance* _the_null_string;      // The Java string "null"
  ciInstance* _the_min_jint_string; // The Java string "-2147483648"

  // Look up a klass by name from a particular class loader (the accessor's).
  // If require_local, result must be defined in that class loader, or NULL.
  // If !require_local, a result from remote class loader may be reported,
  // if sufficient class loader constraints exist such that initiating
  // a class loading request from the given loader is bound to return
  // the class defined in the remote loader (or throw an error).
  //
  // Return an unloaded klass if !require_local and no class at all is found.
  //
  // The CI treats a klass as loaded if it is consistently defined in
  // another loader, even if it hasn't yet been loaded in all loaders
  // that could potentially see it via delegation.
  ciKlass* get_klass_by_name(ciKlass* accessing_klass,
                             ciSymbol* klass_name,
                             bool require_local);

  // Constant pool access.
  ciKlass*   get_klass_by_index(constantPoolHandle cpool,
                                int klass_index,
                                bool& is_accessible,
                                ciInstanceKlass* loading_klass);
  ciConstant get_constant_by_index(constantPoolHandle cpool,
                                   int pool_index, int cache_index,
                                   ciInstanceKlass* accessor);
  ciField*   get_field_by_index(ciInstanceKlass* loading_klass,
                                int field_index);
  ciMethod*  get_method_by_index(constantPoolHandle cpool,
                                 int method_index, Bytecodes::Code bc,
                                 ciInstanceKlass* loading_klass);

  // Implementation methods for loading and constant pool access.
  ciKlass* get_klass_by_name_impl(ciKlass* accessing_klass,
                                  constantPoolHandle cpool,
                                  ciSymbol* klass_name,
                                  bool require_local);
  ciKlass*   get_klass_by_index_impl(constantPoolHandle cpool,
                                     int klass_index,
                                     bool& is_accessible,
                                     ciInstanceKlass* loading_klass);
  ciConstant get_constant_by_index_impl(constantPoolHandle cpool,
                                        int pool_index, int cache_index,
                                        ciInstanceKlass* loading_klass);
  ciField*   get_field_by_index_impl(ciInstanceKlass* loading_klass,
                                     int field_index);
  ciMethod*  get_method_by_index_impl(constantPoolHandle cpool,
                                      int method_index, Bytecodes::Code bc,
                                      ciInstanceKlass* loading_klass);
  ciMethod*  get_fake_invokedynamic_method_impl(constantPoolHandle cpool,
                                                int index, Bytecodes::Code bc,
                                                ciInstanceKlass* accessor);
                                                int index, Bytecodes::Code bc);

  // Helper methods
  bool       check_klass_accessibility(ciKlass* accessing_klass,
                                      klassOop resolved_klassOop);
  methodOop  lookup_method(instanceKlass*  accessor,
                           instanceKlass*  holder,
                           Symbol*         name,
                           Symbol*         sig,
                           Bytecodes::Code bc);

  // Get a ciObject from the object factory.  Ensures uniqueness
  // of ciObjects.
  ciObject* get_object(oop o) {
    if (o == NULL) {
      return _null_object_instance;
    } else {
      return _factory->get(o);
    }
  }

  ciSymbol* get_symbol(Symbol* o) {
    if (o == NULL) {
      ShouldNotReachHere();
      return NULL;
    } else {
      return _factory->get_symbol(o);
    }
  }

  ciMethod* get_method_from_handle(jobject method);

  ciInstance* get_or_create_exception(jobject& handle, Symbol* name);

  // Get a ciMethod representing either an unfound method or
  // a method with an unloaded holder.  Ensures uniqueness of
  // the result.
```

```
  ciMethod* get_unloaded_method(ciInstanceKlass* holder,
                                ciSymbol*        name,
                                ciSymbol*        signature,
                                ciInstanceKlass* accessor) {
    return _factory->get_unloaded_method(holder, name, signature, accessor);
                                ciSymbol*        signature) {
    return _factory->get_unloaded_method(holder, name, signature);
  }

  // Get a ciKlass representing an unloaded klass.
  // Ensures uniqueness of the result.
  ciKlass* get_unloaded_klass(ciKlass*  accessing_klass,
                              ciSymbol* name) {
    return _factory->get_unloaded_klass(accessing_klass, name, true);
  }

  // Get a ciKlass representing an unloaded klass mirror.
  // Result is not necessarily unique, but will be unloaded.
  ciInstance* get_unloaded_klass_mirror(ciKlass* type) {
    return _factory->get_unloaded_klass_mirror(type);
  }

  // Get a ciInstance representing an unresolved method handle constant.
  ciInstance* get_unloaded_method_handle_constant(ciKlass*  holder,
                                                  ciSymbol* name,
                                                  ciSymbol* signature,
                                                  int       ref_kind) {
    return _factory->get_unloaded_method_handle_constant(holder, name, signature
  }

  // Get a ciInstance representing an unresolved method type constant.
  ciInstance* get_unloaded_method_type_constant(ciSymbol* signature) {
    return _factory->get_unloaded_method_type_constant(signature);
  }

  // See if we already have an unloaded klass for the given name
  // or return NULL if not.
  ciKlass *check_get_unloaded_klass(ciKlass*  accessing_klass, ciSymbol* name) {
    return _factory->get_unloaded_klass(accessing_klass, name, false);
  }

  // Get a ciReturnAddress corresponding to the given bci.
  // Ensures uniqueness of the result.
  ciReturnAddress* get_return_address(int bci) {
    return _factory->get_return_address(bci);
  }

  // Get a ciMethodData representing the methodData for a method
  // with none.
  ciMethodData* get_empty_methodData() {
    return _factory->get_empty_methodData();
  }

  // General utility : get a buffer of some required length.
  // Used in symbol creation.
  char* name_buffer(int req_len);

  // Is this thread currently in the VM state?
  static bool is_in_vm();

  // Helper routine for determining the validity of a compilation with
  // respect to method dependencies (e.g. concurrent class loading).
  void validate_compile_task_dependencies(ciMethod* target);

public:
  enum {
    MethodCompilable,
    MethodCompilable_not_at_tier,
    MethodCompilable_never
  };

  ciEnv(CompileTask* task, int system_dictionary_modification_counter);
  // Used only during initialization of the ci
  ciEnv(Arena* arena);
  ~ciEnv();

  OopRecorder* oop_recorder() { return _oop_recorder; }
  void set_oop_recorder(OopRecorder* r) { _oop_recorder = r; }

  DebugInformationRecorder* debug_info() { return _debug_info; }
  void set_debug_info(DebugInformationRecorder* i) { _debug_info = i; }

  Dependencies* dependencies() { return _dependencies; }
  void set_dependencies(Dependencies* d) { _dependencies = d; }

  // This is true if the compilation is not going to produce code.
  // (It is reasonable to retry failed compilations.)
  bool failing() { return _failure_reason != NULL; }

  // Reason this compilation is failing, such as "too many basic blocks".
  const char* failure_reason() { return _failure_reason; }

  // Return state of appropriate compilability
  int compilable() { return _compilable; }

  bool break_at_compile() { return _break_at_compile; }
  void set_break_at_compile(bool z) { _break_at_compile = z; }

  // Cache Jvmti state
  void  cache_jvmti_state();
  bool  jvmti_can_hotswap_or_post_breakpoint() const { return _jvmti_can_hotswap
  bool  jvmti_can_access_local_variables()     const { return _jvmti_can_access_
  bool  jvmti_can_post_on_exceptions()         const { return _jvmti_can_post_on

  // Cache DTrace flags
  void  cache_dtrace_flags();
  bool  dtrace_extended_probes() const { return _dtrace_extended_probes; }
  bool  dtrace_monitor_probes()  const { return _dtrace_monitor_probes; }
  bool  dtrace_method_probes()   const { return _dtrace_method_probes; }
  bool  dtrace_alloc_probes()    const { return _dtrace_alloc_probes; }

  // The compiler task which has created this env.
  // May be useful to find out compile_id, comp_level, etc.
  CompileTask* task() { return _task; }
  // Handy forwards to the task:
  int comp_level();   // task()->comp_level()
  uint compile_id();  // task()->compile_id()

  // Register the result of a compilation.
  void register_method(ciMethod*                 target,
                       int                       entry_bci,
                       CodeOffsets*              offsets,
                       int                       orig_pc_offset,
                       CodeBuffer*               code_buffer,
                       int                       frame_words,
                       OopMapSet*                oop_map_set,
                       ExceptionHandlerTable*    handler_table,
                       ImplicitExceptionTable*   inc_table,
                       AbstractCompiler*         compiler,
                       int                       comp_level,
                       bool                      has_unsafe_access);
```

```
  // Access to certain well known ciObjects.
#define WK_KLASS_FUNC(name, ignore_s, ignore_o) \
  ciInstanceKlass* name() { \
    return _##name;\
  }
  WK_KLASSES_DO(WK_KLASS_FUNC)
#undef WK_KLASS_FUNC

  ciInstance* NullPointerException_instance() {
    assert(_NullPointerException_instance != NULL, "initialization problem");
    return _NullPointerException_instance;
  }
  ciInstance* ArithmeticException_instance() {
    assert(_ArithmeticException_instance != NULL, "initialization problem");
    return _ArithmeticException_instance;
  }

  // Lazy constructors:
  ciInstance* ArrayIndexOutOfBoundsException_instance();
  ciInstance* ArrayStoreException_instance();
  ciInstance* ClassCastException_instance();

  ciInstance* the_null_string();
  ciInstance* the_min_jint_string();

  static ciSymbol* unloaded_cisymbol() {
    return _unloaded_cisymbol;
  }
  static ciObjArrayKlass* unloaded_ciobjarrayklass() {
    return _unloaded_ciobjarrayklass;
  }
  static ciInstanceKlass* unloaded_ciinstance_klass() {
    return _unloaded_ciinstance_klass;
  }

  ciKlass*  find_system_klass(ciSymbol* klass_name);
  // Note:  To find a class from its name string, use ciSymbol::make,
  // but consider adding to vmSymbols.hpp instead.

  // Use this to make a holder for non-perm compile time constants.
  // The resulting array is guaranteed to satisfy "can_be_constant".
  ciArray*  make_system_array(GrowableArray<ciObject*>* objects);

  // converts the ciKlass* representing the holder of a method into a
  // ciInstanceKlass*.  This is needed since the holder of a method in
  // the bytecodes could be an array type.  Basically this converts
  // array types into java/lang/Object and other types stay as they are.
  static ciInstanceKlass* get_instance_klass_for_declared_method_holder(ciKlass*

  // Return the machine-level offset of o, which must be an element of a.
  // This may be used to form constant-loading expressions in lieu of simpler en
  int       array_element_offset_in_bytes(ciArray* a, ciObject* o);

  // Access to the compile-lifetime allocation arena.
  Arena*    arena() { return _arena; }

  // What is the current compilation environment?
  static ciEnv* current() { return CompilerThread::current()->env(); }

  // Overload with current thread argument
  static ciEnv* current(CompilerThread *thread) { return thread->env(); }

  // Per-compiler data.  (Used by C2 to publish the Compile* pointer.)
  void* compiler_data() { return _compiler_data; }
  void set_compiler_data(void* x) { _compiler_data = x; }
  // Notice that a method has been inlined in the current compile;
  // used only for statistics.
  void notice_inlined_method(ciMethod* method);

  // Total number of bytecodes in inlined methods in this compile
  int num_inlined_bytecodes() const;

  // Output stream for logging compilation info.
  CompileLog* log() { return _log; }
  void set_log(CompileLog* log) { _log = log; }

  // Check for changes to the system dictionary during compilation
  bool system_dictionary_modification_counter_changed();

  void record_failure(const char* reason);
  void record_method_not_compilable(const char* reason, bool all_tiers = true);
  void record_out_of_memory_failure();
};
```
______***unchanged_portion_omitted_***

```
**********************************************************
   41525 Wed Oct 12 04:37:39 2011
new/src/share/vm/ci/ciMethod.cpp
**********************************************************
______unchanged_portion_omitted_


// ------------------------------------------------------------------
// ciMethod::ciMethod
//
// Unloaded method.
ciMethod::ciMethod(ciInstanceKlass* holder,
                   ciSymbol*        name,
                   ciSymbol*        signature,
                   ciInstanceKlass* accessor) :
  ciObject(ciMethodKlass::make()),
  _name(                   name),
  _holder(                 holder),
  _intrinsic_id(           vmIntrinsics::_none),
  _liveness(               NULL),
  _can_be_statically_bound(false),
  _method_blocks(          NULL),
  _method_data(            NULL)
                   ciSymbol* signature) : ciObject(ciMethodKlass::make()) {
  // These fields are always filled in.
  _name = name;
  _holder = holder;
  _signature = new (CURRENT_ENV->arena()) ciSignature(_holder, constantPoolHandl
  _intrinsic_id = vmIntrinsics::_none;
  _liveness = NULL;
  _can_be_statically_bound = false;
  _method_blocks = NULL;
  _method_data = NULL;
#if defined(COMPILER2) || defined(SHARK)
  ,
  _flow(                   NULL),
  _bcea(                   NULL)
  _flow = NULL;
  _bcea = NULL;
#endif // COMPILER2 || SHARK
{
  // Usually holder and accessor are the same type but in some cases
  // the holder has the wrong class loader (e.g. invokedynamic call
  // sites) so we pass the accessor.
  _signature = new (CURRENT_ENV->arena()) ciSignature(accessor, constantPoolHand
#endif /* ! codereview */
}


// ------------------------------------------------------------------
// ciMethod::load_code
//
// Load the bytecodes and exception handler table for this method.
void ciMethod::load_code() {
  VM_ENTRY_MARK;
  assert(is_loaded(), "only loaded methods have code");

  methodOop me = get_methodOop();
  Arena* arena = CURRENT_THREAD_ENV->arena();

  // Load the bytecodes.
  _code = (address)arena->Amalloc(code_size());
  memcpy(_code, me->code_base(), code_size());

  // Revert any breakpoint bytecodes in ci's copy
  if (me->number_of_breakpoints() > 0) {
```

```
    BreakpointInfo* bp = instanceKlass::cast(me->method_holder())->breakpoints()
    for (; bp != NULL; bp = bp->next()) {
      if (bp->match(me)) {
        code_at_put(bp->bci(), bp->orig_bytecode());
      }
    }
  }

  // And load the exception table.
  typeArrayOop exc_table = me->exception_table();

  // Allocate one extra spot in our list of exceptions.  This
  // last entry will be used to represent the possibility that
  // an exception escapes the method.  See ciExceptionHandlerStream
  // for details.
  _exception_handlers =
    (ciExceptionHandler**)arena->Amalloc(sizeof(ciExceptionHandler*)
                                         * (_handler_count + 1));
  if (_handler_count > 0) {
    for (int i=0; i<_handler_count; i++) {
      int base = i*4;
      _exception_handlers[i] = new (arena) ciExceptionHandler(
                                 holder(),
            /* start    */      exc_table->int_at(base),
            /* limit    */      exc_table->int_at(base+1),
            /* goto pc  */      exc_table->int_at(base+2),
            /* cp index */      exc_table->int_at(base+3));
    }
  }

  // Put an entry at the end of our list to represent the possibility
  // of exceptional exit.
  _exception_handlers[_handler_count] =
    new (arena) ciExceptionHandler(holder(), 0, code_size(), -1, 0);

  if (CIPrintMethodCodes) {
    print_codes();
  }
}


// ------------------------------------------------------------------
// ciMethod::has_linenumber_table
//
// length unknown until decompression
bool    ciMethod::has_linenumber_table() const {
  check_is_loaded();
  VM_ENTRY_MARK;
  return get_methodOop()->has_linenumber_table();
}


// ------------------------------------------------------------------
// ciMethod::compressed_linenumber_table
u_char* ciMethod::compressed_linenumber_table() const {
  check_is_loaded();
  VM_ENTRY_MARK;
  return get_methodOop()->compressed_linenumber_table();
}


// ------------------------------------------------------------------
// ciMethod::line_number_from_bci
int ciMethod::line_number_from_bci(int bci) const {
  check_is_loaded();
  VM_ENTRY_MARK;
  return get_methodOop()->line_number_from_bci(bci);
}


// ------------------------------------------------------------------
// ciMethod::vtable_index
//
// Get the position of this method’s entry in the vtable, if any.
int ciMethod::vtable_index() {
  check_is_loaded();
  assert(holder()->is_linked(), "must be linked");
  VM_ENTRY_MARK;
  return get_methodOop()->vtable_index();
}


#ifdef SHARK
// ------------------------------------------------------------------
// ciMethod::itable_index
//
// Get the position of this method’s entry in the itable, if any.
int ciMethod::itable_index() {
  check_is_loaded();
  assert(holder()->is_linked(), "must be linked");
  VM_ENTRY_MARK;
  return klassItable::compute_itable_index(get_methodOop());
}
#endif // SHARK


// ------------------------------------------------------------------
// ciMethod::native_entry
//
// Get the address of this method’s native code, if any.
address ciMethod::native_entry() {
  check_is_loaded();
  assert(flags().is_native(), "must be native method");
  VM_ENTRY_MARK;
  methodOop method = get_methodOop();
  address entry = method->native_function();
  assert(entry != NULL, "must be valid entry point");
  return entry;
}


// ------------------------------------------------------------------
// ciMethod::interpreter_entry
//
// Get the entry point for running this method in the interpreter.
address ciMethod::interpreter_entry() {
  check_is_loaded();
  VM_ENTRY_MARK;
  methodHandle mh(THREAD, get_methodOop());
  return Interpreter::entry_for_method(mh);
}


// ------------------------------------------------------------------
// ciMethod::uses_balanced_monitors
//
// Does this method use monitors in a strict stack-disciplined manner?
bool ciMethod::has_balanced_monitors() {
  check_is_loaded();
  if (_balanced_monitors) return true;

  // Analyze the method to see if monitors are used properly.
```

```
  VM_ENTRY_MARK;
  methodHandle method(THREAD, get_methodOop());
  assert(method->has_monitor_bytecodes(), "should have checked this");

  // Check to see if a previous compilation computed the
  // monitor-matching analysis.
  if (method->guaranteed_monitor_matching()) {
    _balanced_monitors = true;
    return true;
  }

  {
    EXCEPTION_MARK;
    ResourceMark rm(THREAD);
    GeneratePairingInfo gpi(method);
    gpi.compute_map(CATCH);
    if (!gpi.monitor_safe()) {
      return false;
    }
    method->set_guaranteed_monitor_matching();
    _balanced_monitors = true;
  }
  return true;
}


// ------------------------------------------------------------------
// ciMethod::get_flow_analysis
ciTypeFlow* ciMethod::get_flow_analysis() {
#if defined(COMPILER2) || defined(SHARK)
  if (_flow == NULL) {
    ciEnv* env = CURRENT_ENV;
    _flow = new (env->arena()) ciTypeFlow(env, this);
    _flow->do_flow();
  }
  return _flow;
#else // COMPILER2 || SHARK
  ShouldNotReachHere();
  return NULL;
#endif // COMPILER2 || SHARK
}


// ------------------------------------------------------------------
// ciMethod::get_osr_flow_analysis
ciTypeFlow* ciMethod::get_osr_flow_analysis(int osr_bci) {
#if defined(COMPILER2) || defined(SHARK)
  // OSR entry points are always place after a call bytecode of some sort
  assert(osr_bci >= 0, "must supply valid OSR entry point");
  ciEnv* env = CURRENT_ENV;
  ciTypeFlow* flow = new (env->arena()) ciTypeFlow(env, this, osr_bci);
  flow->do_flow();
  return flow;
#else // COMPILER2 || SHARK
  ShouldNotReachHere();
  return NULL;
#endif // COMPILER2 || SHARK
}

// ------------------------------------------------------------------
// ciMethod::raw_liveness_at_bci
//
// Which local variables are live at a specific bci?
MethodLivenessResult ciMethod::raw_liveness_at_bci(int bci) {
  check_is_loaded();
  if (_liveness == NULL) {
    // Create the liveness analyzer.
    Arena* arena = CURRENT_ENV->arena();
    _liveness = new (arena) MethodLiveness(arena, this);
    _liveness->compute_liveness();
  }
  return _liveness->get_liveness_at(bci);
}

// ------------------------------------------------------------------
// ciMethod::liveness_at_bci
//
// Which local variables are live at a specific bci?  When debugging
// will return true for all locals in some cases to improve debug
// information.
MethodLivenessResult ciMethod::liveness_at_bci(int bci) {
  MethodLivenessResult result = raw_liveness_at_bci(bci);
  if (CURRENT_ENV->jvmti_can_access_local_variables() || DeoptimizeALot || Compi
    // Keep all locals live for the user's edification and amusement.
    result.at_put_range(0, result.size(), true);
  }
  return result;
}

// ciMethod::live_local_oops_at_bci
//
// find all the live oops in the locals array for a particular bci
// Compute what the interpreter believes by using the interpreter
// oopmap generator. This is used as a double check during osr to
// guard against conservative result from MethodLiveness making us
// think a dead oop is live.  MethodLiveness is conservative in the
// sense that it may consider locals to be live which cannot be live,
// like in the case where a local could contain an oop or  a primitive
// along different paths.  In that case the local must be dead when
// those paths merge. Since the interpreter's viewpoint is used when
// gc'ing an interpreter frame we need to use its viewpoint  during
// OSR when loading the locals.

BitMap ciMethod::live_local_oops_at_bci(int bci) {
  VM_ENTRY_MARK;
  InterpreterOopMap mask;
  OopMapCache::compute_one_oop_map(get_methodOop(), bci, &mask);
  int mask_size = max_locals();
  BitMap result(mask_size);
  result.clear();
  int i;
  for (i = 0; i < mask_size ; i++ ) {
    if (mask.is_oop(i)) result.set_bit(i);
  }
  return result;
}


#ifdef COMPILER1
// ------------------------------------------------------------------
// ciMethod::bci_block_start
//
// Marks all bcis where a new basic block starts
const BitMap ciMethod::bci_block_start() {
  check_is_loaded();
  if (_liveness == NULL) {
    // Create the liveness analyzer.
    Arena* arena = CURRENT_ENV->arena();
    _liveness = new (arena) MethodLiveness(arena, this);
    _liveness->compute_liveness();
  }
```

```
  return _liveness->get_bci_block_start();
}
#endif // COMPILER1


// ------------------------------------------------------------------
// ciMethod::call_profile_at_bci
//
// Get the ciCallProfile for the invocation of this method.
// Also reports receiver types for non-call type checks (if TypeProfileCasts).
ciCallProfile ciMethod::call_profile_at_bci(int bci) {
  ResourceMark rm;
  ciCallProfile result;
  if (method_data() != NULL && method_data()->is_mature()) {
    ciProfileData* data = method_data()->bci_to_data(bci);
    if (data != NULL && data->is_CounterData()) {
      // Every profiled call site has a counter.
      int count = data->as_CounterData()->count();

      if (!data->is_ReceiverTypeData()) {
        result._receiver_count[0] = 0;  // that's a definite zero
      } else { // ReceiverTypeData is a subclass of CounterData
        ciReceiverTypeData* call = (ciReceiverTypeData*)data->as_ReceiverTypeDat
        // In addition, virtual call sites have receiver type information
        int receivers_count_total = 0;
        int morphism = 0;
        // Precompute morphism for the possible fixup
        for (uint i = 0; i < call->row_limit(); i++) {
          ciKlass* receiver = call->receiver(i);
          if (receiver == NULL)  continue;
          morphism++;
        }
        int epsilon = 0;
        if (TieredCompilation && ProfileInterpreter) {
          // Interpreter and C1 treat final and special invokes differently.
          // C1 will record a type, whereas the interpreter will just
          // increment the count. Detect this case.
          if (morphism == 1 && count > 0) {
            epsilon = count;
            count = 0;
          }
        }
        for (uint i = 0; i < call->row_limit(); i++) {
          ciKlass* receiver = call->receiver(i);
          if (receiver == NULL)  continue;
          int rcount = call->receiver_count(i) + epsilon;
          if (rcount == 0) rcount = 1; // Should be valid value
          receivers_count_total += rcount;
          // Add the receiver to result data.
          result.add_receiver(receiver, rcount);
          // If we extend profiling to record methods,
          // we will set result._method also.
        }
        // Determine call site's morphism.
        // The call site count is 0 with known morphism (onlt 1 or 2 receivers)
        // or < 0 in the case of a type check failured for checkcast, aastore, i
        // The call site count is > 0 in the case of a polymorphic virtual call.
        if (morphism > 0 && morphism == result._limit) {
           // The morphism <= MorphismLimit.
           if ((morphism <  ciCallProfile::MorphismLimit) ||
               (morphism == ciCallProfile::MorphismLimit && count == 0)) {
#ifdef ASSERT
             if (count > 0) {
               this->print_short_name(tty);
               tty->print_cr(" @ bci:%d", bci);
               this->print_codes();
               assert(false, "this call site should not be polymorphic");
             }
#endif
             result._morphism = morphism;
           }
        }
        // Make the count consistent if this is a call profile. If count is
        // zero or less, presume that this is a typecheck profile and
        // do nothing.  Otherwise, increase count to be the sum of all
        // receiver's counts.
        if (count >= 0) {
          count += receivers_count_total;
        }
      }
      result._count = count;
    }
  }
  return result;
}

// ------------------------------------------------------------------
// Add new receiver and sort data by receiver's profile count.
void ciCallProfile::add_receiver(ciKlass* receiver, int receiver_count) {
  // Add new receiver and sort data by receiver's counts when we have space
  // for it otherwise replace the less called receiver (less called receiver
  // is placed to the last array element which is not used).
  // First array's element contains most called receiver.
  int i = _limit;
  for (; i > 0 && receiver_count > _receiver_count[i-1]; i--) {
    _receiver[i] = _receiver[i-1];
    _receiver_count[i] = _receiver_count[i-1];
  }
  _receiver[i] = receiver;
  _receiver_count[i] = receiver_count;
  if (_limit < MorphismLimit) _limit++;
}

// ------------------------------------------------------------------
// ciMethod::find_monomorphic_target
//
// Given a certain calling environment, find the monomorphic target
// for the call.  Return NULL if the call is not monomorphic in
// its calling environment, or if there are only abstract methods.
// The returned method is never abstract.
// Note: If caller uses a non-null result, it must inform dependencies
// via assert_unique_concrete_method or assert_leaf_type.
ciMethod* ciMethod::find_monomorphic_target(ciInstanceKlass* caller,
                                            ciInstanceKlass* callee_holder,
                                            ciInstanceKlass* actual_recv) {
  check_is_loaded();

  if (actual_recv->is_interface()) {
    // %%% We cannot trust interface types, yet.  See bug 6312651.
    return NULL;
  }

  ciMethod* root_m = resolve_invoke(caller, actual_recv);
  if (root_m == NULL) {
    // Something went wrong looking up the actual receiver method.
    return NULL;
  }
  assert(!root_m->is_abstract(), "resolve_invoke promise");

  // Make certain quick checks even if UseCHA is false.

  // Is it private or final?
```

```
  if (root_m->can_be_statically_bound()) {
    return root_m;
  }

  if (actual_recv->is_leaf_type() && actual_recv == root_m->holder()) {
    // Easy case.  There is no other place to put a method, so don't bother
    // to go through the VM_ENTRY_MARK and all the rest.
    return root_m;
  }

  // Array methods (clone, hashCode, etc.) are always statically bound.
  // If we were to see an array type here, we'd return root_m.
  // However, this method processes only ciInstanceKlasses.  (See 4962591.)
  // The inline_native_clone intrinsic narrows Object to T[] properly,
  // so there is no need to do the same job here.

  if (!UseCHA)  return NULL;

  VM_ENTRY_MARK;

  methodHandle target;
  {
    MutexLocker locker(Compile_lock);
    klassOop context = actual_recv->get_klassOop();
    target = Dependencies::find_unique_concrete_method(context,
                                                       root_m->get_methodOop());
    // %%% Should upgrade this ciMethod API to look for 1 or 2 concrete methods.
  }

#ifndef PRODUCT
  if (TraceDependencies && target() != NULL && target() != root_m->get_methodOop
    tty->print("found a non-root unique target method");
    tty->print_cr("  context = %s", instanceKlass::cast(actual_recv->get_klassOo
    tty->print("  method  = ");
    target->print_short_name(tty);
    tty->cr();
  }
#endif //PRODUCT

  if (target() == NULL) {
    return NULL;
  }
  if (target() == root_m->get_methodOop()) {
    return root_m;
  }
  if (!root_m->is_public() &&
      !root_m->is_protected()) {
    // If we are going to reason about inheritance, it's easiest
    // if the method in question is public, protected, or private.
    // If the answer is not root_m, it is conservatively correct
    // to return NULL, even if the CHA encountered irrelevant
    // methods in other packages.
    // %%% TO DO: Work out logic for package-private methods
    // with the same name but different vtable indexes.
    return NULL;
  }
  return CURRENT_THREAD_ENV->get_object(target())->as_method();
}

// ------------------------------------------------------------------
// ciMethod::resolve_invoke
//
// Given a known receiver klass, find the target for the call.
// Return NULL if the call has no target or the target is abstract.
ciMethod* ciMethod::resolve_invoke(ciKlass* caller, ciKlass* exact_receiver) {
   check_is_loaded();
  VM_ENTRY_MARK;

  KlassHandle caller_klass (THREAD, caller->get_klassOop());
  KlassHandle h_recv       (THREAD, exact_receiver->get_klassOop());
  KlassHandle h_resolved   (THREAD, holder()->get_klassOop());
  Symbol* h_name      = name()->get_symbol();
  Symbol* h_signature = signature()->get_symbol();

  methodHandle m;
  // Only do exact lookup if receiver klass has been linked.  Otherwise,
  // the vtable has not been setup, and the LinkResolver will fail.
  if (h_recv->oop_is_javaArray()
       ||
      instanceKlass::cast(h_recv())->is_linked() && !exact_receiver->is_interfa
    if (holder()->is_interface()) {
      m = LinkResolver::resolve_interface_call_or_null(h_recv, h_resolved, h_na
    } else {
      m = LinkResolver::resolve_virtual_call_or_null(h_recv, h_resolved, h_name
    }
  }

  if (m.is_null()) {
    // Return NULL only if there was a problem with lookup (uninitialized class
    return NULL;
  }

  ciMethod* result = this;
  if (m() != get_methodOop()) {
    result = CURRENT_THREAD_ENV->get_object(m())->as_method();
  }

  // Don't return abstract methods because they aren't
  // optimizable or interesting.
  if (result->is_abstract()) {
    return NULL;
  } else {
    return result;
  }
}

// ------------------------------------------------------------------
// ciMethod::resolve_vtable_index
//
// Given a known receiver klass, find the vtable index for the call.
// Return methodOopDesc::invalid_vtable_index if the vtable_index is unknown.
int ciMethod::resolve_vtable_index(ciKlass* caller, ciKlass* receiver) {
   check_is_loaded();

   int vtable_index = methodOopDesc::invalid_vtable_index;
   // Only do lookup if receiver klass has been linked.  Otherwise,
   // the vtable has not been setup, and the LinkResolver will fail.
   if (!receiver->is_interface()
       && (!receiver->is_instance_klass() ||
           receiver->as_instance_klass()->is_linked())) {
     VM_ENTRY_MARK;

     KlassHandle caller_klass (THREAD, caller->get_klassOop());
     KlassHandle h_recv       (THREAD, receiver->get_klassOop());
     Symbol* h_name = name()->get_symbol();
     Symbol* h_signature = signature()->get_symbol();

     vtable_index = LinkResolver::resolve_virtual_vtable_index(h_recv, h_recv, h
     if (vtable_index == methodOopDesc::nonvirtual_vtable_index) {
       // A statically bound method.  Return "no such index".
       vtable_index = methodOopDesc::invalid_vtable_index;
     }
```

```
   }

   return vtable_index;
}

// ------------------------------------------------------------------
// ciMethod::interpreter_call_site_count
int ciMethod::interpreter_call_site_count(int bci) {
  if (method_data() != NULL) {
    ResourceMark rm;
    ciProfileData* data = method_data()->bci_to_data(bci);
    if (data != NULL && data->is_CounterData()) {
      return scale_count(data->as_CounterData()->count());
    }
  }
  return -1;  // unknown
}

// ------------------------------------------------------------------
// Adjust a CounterData count to be commensurate with
// interpreter_invocation_count.  If the MDO exists for
// only 25% of the time the method exists, then the
// counts in the MDO should be scaled by 4X, so that
// they can be usefully and stably compared against the
// invocation counts in methods.
int ciMethod::scale_count(int count, float prof_factor) {
  if (count > 0 && method_data() != NULL) {
    int counter_life;
    int method_life = interpreter_invocation_count();
    if (TieredCompilation) {
      // In tiered the MDO's life is measured directly, so just use the snapshot
      counter_life = MAX2(method_data()->invocation_count(), method_data()->back
    } else {
      int current_mileage = method_data()->current_mileage();
      int creation_mileage = method_data()->creation_mileage();
      counter_life = current_mileage - creation_mileage;
    }

    // counter_life due to backedge_counter could be > method_life
    if (counter_life > method_life)
      counter_life = method_life;
    if (0 < counter_life && counter_life <= method_life) {
      count = (int)((double)count * prof_factor * method_life / counter_life + 0
      count = (count > 0) ? count : 1;
    }
  }
  return count;
}

// ------------------------------------------------------------------
// invokedynamic support

// ------------------------------------------------------------------
// ciMethod::is_method_handle_invoke
//
// Return true if the method is an instance of one of the two
// signature-polymorphic MethodHandle methods, invokeExact or invokeGeneric.
bool ciMethod::is_method_handle_invoke() const {
  if (!is_loaded()) {
    bool flag = (holder()->name() == ciSymbol::java_lang_invoke_MethodHandle() &
                 methodOopDesc::is_method_handle_invoke_name(name()->sid()));
    return flag;
  }
  VM_ENTRY_MARK;
  return get_methodOop()->is_method_handle_invoke();
}
```

```
// ------------------------------------------------------------------
// ciMethod::is_method_handle_adapter
//
// Return true if the method is a generated MethodHandle adapter.
// These are built by MethodHandleCompiler.
bool ciMethod::is_method_handle_adapter() const {
  if (!is_loaded())  return false;
  VM_ENTRY_MARK;
  return get_methodOop()->is_method_handle_adapter();
}

ciInstance* ciMethod::method_handle_type() {
  check_is_loaded();
  VM_ENTRY_MARK;
  oop mtype = get_methodOop()->method_handle_type();
  return CURRENT_THREAD_ENV->get_object(mtype)->as_instance();
}


// ------------------------------------------------------------------
// ciMethod::ensure_method_data
//
// Generate new methodDataOop objects at compile time.
// Return true if allocation was successful or no MDO is required.
bool ciMethod::ensure_method_data(methodHandle h_m) {
  EXCEPTION_CONTEXT;
  if (is_native() || is_abstract() || h_m()->is_accessor()) return true;
  if (h_m()->method_data() == NULL) {
    methodOopDesc::build_interpreter_method_data(h_m, THREAD);
    if (HAS_PENDING_EXCEPTION) {
      CLEAR_PENDING_EXCEPTION;
    }
  }
  if (h_m()->method_data() != NULL) {
    _method_data = CURRENT_ENV->get_object(h_m()->method_data())->as_method_data
    _method_data->load_data();
    return true;
  } else {
    _method_data = CURRENT_ENV->get_empty_methodData();
    return false;
  }
}

// public, retroactive version
bool ciMethod::ensure_method_data() {
  bool result = true;
  if (_method_data == NULL || _method_data->is_empty()) {
    GUARDED_VM_ENTRY({
      result = ensure_method_data(get_methodOop());
    });
  }
  return result;
}


// ------------------------------------------------------------------
// ciMethod::method_data
//
ciMethodData* ciMethod::method_data() {
  if (_method_data != NULL) {
    return _method_data;
  }
  VM_ENTRY_MARK;
  ciEnv* env = CURRENT_ENV;
  Thread* my_thread = JavaThread::current();
```

```
  methodHandle h_m(my_thread, get_methodOop());

  if (h_m()->method_data() != NULL) {
    _method_data = CURRENT_ENV->get_object(h_m()->method_data())->as_method_data
    _method_data->load_data();
  } else {
    _method_data = CURRENT_ENV->get_empty_methodData();
  }
  return _method_data;

}

// ------------------------------------------------------------------
// ciMethod::method_data_or_null
// Returns a pointer to ciMethodData if MDO exists on the VM side,
// NULL otherwise.
ciMethodData* ciMethod::method_data_or_null() {
  ciMethodData *md = method_data();
  if (md->is_empty()) return NULL;
  return md;
}

// ------------------------------------------------------------------
// ciMethod::will_link
//
// Will this method link in a specific calling context?
bool ciMethod::will_link(ciKlass* accessing_klass,
                         ciKlass* declared_method_holder,
                         Bytecodes::Code bc) {
  if (!is_loaded()) {
    // Method lookup failed.
    return false;
  }

  // The link checks have been front-loaded into the get_method
  // call.  This method (ciMethod::will_link()) will be removed
  // in the future.

  return true;
}

// ------------------------------------------------------------------
// ciMethod::should_exclude
//
// Should this method be excluded from compilation?
bool ciMethod::should_exclude() {
  check_is_loaded();
  VM_ENTRY_MARK;
  methodHandle mh(THREAD, get_methodOop());
  bool ignore;
  return CompilerOracle::should_exclude(mh, ignore);
}

// ------------------------------------------------------------------
// ciMethod::should_inline
//
// Should this method be inlined during compilation?
bool ciMethod::should_inline() {
  check_is_loaded();
  VM_ENTRY_MARK;
  methodHandle mh(THREAD, get_methodOop());
  return CompilerOracle::should_inline(mh);
}

// ------------------------------------------------------------------
// ciMethod::should_not_inline
```

```
//
// Should this method be disallowed from inlining during compilation?
bool ciMethod::should_not_inline() {
  check_is_loaded();
  VM_ENTRY_MARK;
  methodHandle mh(THREAD, get_methodOop());
  return CompilerOracle::should_not_inline(mh);
}

// ------------------------------------------------------------------
// ciMethod::should_print_assembly
//
// Should the compiler print the generated code for this method?
bool ciMethod::should_print_assembly() {
  check_is_loaded();
  VM_ENTRY_MARK;
  methodHandle mh(THREAD, get_methodOop());
  return CompilerOracle::should_print(mh);
}

// ------------------------------------------------------------------
// ciMethod::break_at_execute
//
// Should the compiler insert a breakpoint into the generated code
// method?
bool ciMethod::break_at_execute() {
  check_is_loaded();
  VM_ENTRY_MARK;
  methodHandle mh(THREAD, get_methodOop());
  return CompilerOracle::should_break_at(mh);
}

// ------------------------------------------------------------------
// ciMethod::has_option
//
bool ciMethod::has_option(const char* option) {
  check_is_loaded();
  VM_ENTRY_MARK;
  methodHandle mh(THREAD, get_methodOop());
  return CompilerOracle::has_option_string(mh, option);
}

// ------------------------------------------------------------------
// ciMethod::can_be_compiled
//
// Have previous compilations of this method succeeded?
bool ciMethod::can_be_compiled() {
  check_is_loaded();
  ciEnv* env = CURRENT_ENV;
  if (is_c1_compile(env->comp_level())) {
    return _is_c1_compilable;
  }
  return _is_c2_compilable;
}

// ------------------------------------------------------------------
// ciMethod::set_not_compilable
//
// Tell the VM that this method cannot be compiled at all.
void ciMethod::set_not_compilable() {
  check_is_loaded();
  VM_ENTRY_MARK;
  ciEnv* env = CURRENT_ENV;
  if (is_c1_compile(env->comp_level())) {
    _is_c1_compilable = false;
  } else {
```

```
    _is_c2_compilable = false;
  }
  get_methodOop()->set_not_compilable(env->comp_level());
}

// ------------------------------------------------------------------
// ciMethod::can_be_osr_compiled
//
// Have previous compilations of this method succeeded?
//
// Implementation note: the VM does not currently keep track
// of failed OSR compilations per bci.  The entry_bci parameter
// is currently unused.
bool ciMethod::can_be_osr_compiled(int entry_bci) {
  check_is_loaded();
  VM_ENTRY_MARK;
  ciEnv* env = CURRENT_ENV;
  return !get_methodOop()->is_not_osr_compilable(env->comp_level());
}

// ------------------------------------------------------------------
// ciMethod::has_compiled_code
bool ciMethod::has_compiled_code() {
  VM_ENTRY_MARK;
  return get_methodOop()->code() != NULL;
}

int ciMethod::comp_level() {
  check_is_loaded();
  VM_ENTRY_MARK;
  nmethod* nm = get_methodOop()->code();
  if (nm != NULL) return nm->comp_level();
  return 0;
}

int ciMethod::highest_osr_comp_level() {
  check_is_loaded();
  VM_ENTRY_MARK;
  return get_methodOop()->highest_osr_comp_level();
}

// ------------------------------------------------------------------
// ciMethod::code_size_for_inlining
//
// Code size for inlining decisions.
//
// Don't fully count method handle adapters against inlining budgets:
// the metric we use here is the number of call sites in the adapter
// as they are probably the instructions which generate some code.
int ciMethod::code_size_for_inlining() {
  check_is_loaded();

  // Method handle adapters
  if (is_method_handle_adapter()) {
    // Count call sites
    int call_site_count = 0;
    ciBytecodeStream iter(this);
    while (iter.next() != ciBytecodeStream::EOBC()) {
      if (Bytecodes::is_invoke(iter.cur_bc())) {
        call_site_count++;
      }
    }
    return call_site_count;
  }

  // Normal method
  return code_size();
}

// ------------------------------------------------------------------
// ciMethod::instructions_size
//
// This is a rough metric for "fat" methods, compared before inlining
// with InlineSmallCode.  The CodeBlob::code_size accessor includes
// junk like exception handler, stubs, and constant table, which are
// not highly relevant to an inlined method.  So we use the more
// specific accessor nmethod::insts_size.
int ciMethod::instructions_size(int comp_level) {
  GUARDED_VM_ENTRY(
    nmethod* code = get_methodOop()->code();
    if (code != NULL && (comp_level == CompLevel_any || comp_level == code->comp
      return code->insts_end() - code->verified_entry_point();
    }
    return 0;
  )
}

// ------------------------------------------------------------------
// ciMethod::log_nmethod_identity
void ciMethod::log_nmethod_identity(xmlStream* log) {
  GUARDED_VM_ENTRY(
    nmethod* code = get_methodOop()->code();
    if (code != NULL) {
      code->log_identity(log);
    }
  )
}

// ------------------------------------------------------------------
// ciMethod::is_not_reached
bool ciMethod::is_not_reached(int bci) {
  check_is_loaded();
  VM_ENTRY_MARK;
  return Interpreter::is_not_reached(
               methodHandle(THREAD, get_methodOop()), bci);
}

// ------------------------------------------------------------------
// ciMethod::was_never_executed
bool ciMethod::was_executed_more_than(int times) {
  VM_ENTRY_MARK;
  return get_methodOop()->was_executed_more_than(times);
}

// ------------------------------------------------------------------
// ciMethod::has_unloaded_classes_in_signature
bool ciMethod::has_unloaded_classes_in_signature() {
  VM_ENTRY_MARK;
  {
    EXCEPTION_MARK;
    methodHandle m(THREAD, get_methodOop());
    bool has_unloaded = methodOopDesc::has_unloaded_classes_in_signature(m, (Jav
    if( HAS_PENDING_EXCEPTION ) {
      CLEAR_PENDING_EXCEPTION;
      return true;     // Declare that we may have unloaded classes
    }
    return has_unloaded;
  }
}

// ------------------------------------------------------------------
// ciMethod::is_klass_loaded
```

```
bool ciMethod::is_klass_loaded(int refinfo_index, bool must_be_resolved) const {
  VM_ENTRY_MARK;
  return get_methodOop()->is_klass_loaded(refinfo_index, must_be_resolved);
}

// ------------------------------------------------------------------
// ciMethod::check_call
bool ciMethod::check_call(int refinfo_index, bool is_static) const {
  VM_ENTRY_MARK;
  {
    EXCEPTION_MARK;
    HandleMark hm(THREAD);
    constantPoolHandle pool (THREAD, get_methodOop()->constants());
    methodHandle spec_method;
    KlassHandle  spec_klass;
    LinkResolver::resolve_method(spec_method, spec_klass, pool, refinfo_index, T
    if (HAS_PENDING_EXCEPTION) {
      CLEAR_PENDING_EXCEPTION;
      return false;
    } else {
      return (spec_method->is_static() == is_static);
    }
  }
  return false;
}

// ------------------------------------------------------------------
// ciMethod::print_codes
//
// Print the bytecodes for this method.
void ciMethod::print_codes_on(outputStream* st) {
  check_is_loaded();
  GUARDED_VM_ENTRY(get_methodOop()->print_codes_on(st);)
}


#define FETCH_FLAG_FROM_VM(flag_accessor) { \
  check_is_loaded(); \
  VM_ENTRY_MARK; \
  return get_methodOop()->flag_accessor(); \
}

bool ciMethod::is_empty_method() const {         FETCH_FLAG_FROM_VM(is_empty_met
bool ciMethod::is_vanilla_constructor() const {  FETCH_FLAG_FROM_VM(is_vanilla_c
bool ciMethod::has_loops      () const {         FETCH_FLAG_FROM_VM(has_loops);
bool ciMethod::has_jsrs       () const {         FETCH_FLAG_FROM_VM(has_jsrs);
bool ciMethod::is_accessor    () const {         FETCH_FLAG_FROM_VM(is_accessor)
bool ciMethod::is_initializer () const {         FETCH_FLAG_FROM_VM(is_initializ

BCEscapeAnalyzer  *ciMethod::get_bcea() {
#ifdef COMPILER2
  if (_bcea == NULL) {
    _bcea = new (CURRENT_ENV->arena()) BCEscapeAnalyzer(this, NULL);
  }
  return _bcea;
#else // COMPILER2
  ShouldNotReachHere();
  return NULL;
#endif // COMPILER2
}

ciMethodBlocks  *ciMethod::get_method_blocks() {
  Arena *arena = CURRENT_ENV->arena();
  if (_method_blocks == NULL) {
    _method_blocks = new (arena) ciMethodBlocks(arena, this);
  }
  return _method_blocks;
}

#undef FETCH_FLAG_FROM_VM


// ------------------------------------------------------------------
// ciMethod::print_codes
//
// Print a range of the bytecodes for this method.
void ciMethod::print_codes_on(int from, int to, outputStream* st) {
  check_is_loaded();
  GUARDED_VM_ENTRY(get_methodOop()->print_codes_on(from, to, st);)
}

// ------------------------------------------------------------------
// ciMethod::print_name
//
// Print the name of this method, including signature and some flags.
void ciMethod::print_name(outputStream* st) {
  check_is_loaded();
  GUARDED_VM_ENTRY(get_methodOop()->print_name(st);)
}

// ------------------------------------------------------------------
// ciMethod::print_short_name
//
// Print the name of this method, without signature.
void ciMethod::print_short_name(outputStream* st) {
  check_is_loaded();
  GUARDED_VM_ENTRY(get_methodOop()->print_short_name(st);)
}

// ------------------------------------------------------------------
// ciMethod::print_impl
//
// Implementation of the print method.
void ciMethod::print_impl(outputStream* st) {
  ciObject::print_impl(st);
  st->print(" name=");
  name()->print_symbol_on(st);
  st->print(" holder=");
  holder()->print_name_on(st);
  st->print(" signature=");
  signature()->as_symbol()->print_symbol_on(st);
  if (is_loaded()) {
    st->print(" loaded=true flags=");
    flags().print_member_flags(st);
  } else {
    st->print(" loaded=false");
  }
}
```

```
**********************************************************
   11627 Wed Oct 12 04:37:40 2011
new/src/share/vm/ci/ciMethod.hpp
**********************************************************
/*
 * Copyright (c) 1999, 2011, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_CI_CIMETHOD_HPP
#define SHARE_VM_CI_CIMETHOD_HPP

#include "ci/ciFlags.hpp"
#include "ci/ciInstanceKlass.hpp"
#include "ci/ciObject.hpp"
#include "ci/ciSignature.hpp"
#include "compiler/methodLiveness.hpp"
#include "prims/methodHandles.hpp"
#include "utilities/bitMap.hpp"

class ciMethodBlocks;
class MethodLiveness;
class BitMap;
class Arena;
class BCEscapeAnalyzer;


// ciMethod
//
// This class represents a methodOop in the HotSpot virtual
// machine.
class ciMethod : public ciObject {
  friend class CompileBroker;
  CI_PACKAGE_ACCESS
  friend class ciEnv;
  friend class ciExceptionHandlerStream;
  friend class ciBytecodeStream;
  friend class ciMethodHandle;

 private:
  // General method information.
  ciFlags          _flags;
  ciSymbol*        _name;
  ciInstanceKlass* _holder;
  ciSignature*     _signature;
  ciMethodData*    _method_data;
  ciMethodBlocks*   _method_blocks;
```

```
  // Code attributes.
  int _code_size;
  int _max_stack;
  int _max_locals;
  vmIntrinsics::ID _intrinsic_id;
  int _handler_count;
  int _interpreter_invocation_count;
  int _interpreter_throwout_count;

  bool _uses_monitors;
  bool _balanced_monitors;
  bool _is_c1_compilable;
  bool _is_c2_compilable;
  bool _can_be_statically_bound;

  // Lazy fields, filled in on demand
  address              _code;
  ciExceptionHandler** _exception_handlers;

  // Optional liveness analyzer.
  MethodLiveness* _liveness;
#if defined(COMPILER2) || defined(SHARK)
  ciTypeFlow*         _flow;
  BCEscapeAnalyzer*   _bcea;
#endif

  ciMethod(methodHandle h_m);
  ciMethod(ciInstanceKlass* holder, ciSymbol* name, ciSymbol* signature, ciInsta
  ciMethod(ciInstanceKlass* holder, ciSymbol* name, ciSymbol* signature);

  methodOop get_methodOop() const {
    methodOop m = (methodOop)get_oop();
    assert(m != NULL, "illegal use of unloaded method");
    return m;
  }

  oop loader() const                             { return _holder->loader(); }

  const char* type_string()                      { return "ciMethod"; }

  void print_impl(outputStream* st);

  void load_code();

  void check_is_loaded() const                   { assert(is_loaded(), "not load

  bool ensure_method_data(methodHandle h_m);

  void code_at_put(int bci, Bytecodes::Code code) {
    Bytecodes::check(code);
    assert(0 <= bci && bci < code_size(), "valid bci");
    address bcp = _code + bci;
    *bcp = code;
  }

 public:
  // Basic method information.
  ciFlags flags() const                          { check_is_loaded(); return _fl
  ciSymbol* name() const                         { return _name; }
  ciInstanceKlass* holder() const                { return _holder; }
  ciMethodData* method_data();
  ciMethodData* method_data_or_null();

  // Signature information.
  ciSignature* signature() const                 { return _signature; }
  ciType*      return_type() const              { return _signature->return_typ
  int          arg_size_no_receiver() const     { return _signature->size(); }
  // Can only be used on loaded ciMethods
  int          arg_size() const                 {
    check_is_loaded();
    return _signature->size() + (_flags.is_static() ? 0 : 1);
  }
  // Report the number of elements on stack when invoking this method.
  // This is different than the regular arg_size because invokdynamic
  // has an implicit receiver.
  int invoke_arg_size(Bytecodes::Code code) const {
    int arg_size = _signature->size();
    // Add a receiver argument, maybe:
    if (code != Bytecodes::_invokestatic &&
        code != Bytecodes::_invokedynamic) {
      arg_size++;
    }
    return arg_size;
  }


  // Method code and related information.
  address code()                                 { if (_code == NULL) load_code(
  int code_size() const                          { check_is_loaded(); return _co
  int max_stack() const                          { check_is_loaded(); return _ma
  int max_locals() const                         { check_is_loaded(); return _ma
  vmIntrinsics::ID intrinsic_id() const          { check_is_loaded(); return _in
  bool has_exception_handlers() const            { check_is_loaded(); return _ha
  int exception_table_length() const             { check_is_loaded(); return _ha
  int interpreter_invocation_count() const       { check_is_loaded(); return _in
  int interpreter_throwout_count() const         { check_is_loaded(); return _in

  // Code size for inlining decisions.
  int code_size_for_inlining();

  int comp_level();
  int highest_osr_comp_level();

  Bytecodes::Code java_code_at_bci(int bci) {
    address bcp = code() + bci;
    return Bytecodes::java_code_at(NULL, bcp);
  }
  BCEscapeAnalyzer  *get_bcea();
  ciMethodBlocks    *get_method_blocks();

  bool    has_linenumber_table() const;          // length unknown until decompr
  u_char* compressed_linenumber_table() const;   // not preserved by gc

  int line_number_from_bci(int bci) const;

  // Runtime information.
  int           vtable_index();
#ifdef SHARK
  int           itable_index();
#endif // SHARK
  address       native_entry();
  address       interpreter_entry();

  // Analysis and profiling.
  //
  // Usage note: liveness_at_bci and init_vars should be wrapped in ResourceMark
  bool          uses_monitors() const            { return _uses_monitors; } // t
  bool          has_monitor_bytecodes() const    { return _uses_monitors; }
  bool          has_balanced_monitors();

  // Returns a bitmap indicating which locals are required to be
```

```
  // maintained as live for deopt.  raw_liveness_at_bci is always the
  // direct output of the liveness computation while liveness_at_bci
  // may mark all locals as live to improve support for debugging Java
  // code by maintaining the state of as many locals as possible.
  MethodLivenessResult raw_liveness_at_bci(int bci);
  MethodLivenessResult liveness_at_bci(int bci);

  // Get the interpreters viewpoint on oop liveness.  MethodLiveness is
  // conservative in the sense that it may consider locals to be live which
  // cannot be live, like in the case where a local could contain an oop or
  // a primitive along different paths.  In that case the local must be
  // dead when those paths merge. Since the interpreter's viewpoint is
  // used when gc'ing an interpreter frame we need to use its viewpoint
  // during OSR when loading the locals.

  BitMap  live_local_oops_at_bci(int bci);

#ifdef COMPILER1
  const BitMap  bci_block_start();
#endif

  ciTypeFlow*   get_flow_analysis();
  ciTypeFlow*   get_osr_flow_analysis(int osr_bci);  // alternate entry point
  ciCallProfile call_profile_at_bci(int bci);
  int           interpreter_call_site_count(int bci);

  // Given a certain calling environment, find the monomorphic target
  // for the call.  Return NULL if the call is not monomorphic in
  // its calling environment.
  ciMethod* find_monomorphic_target(ciInstanceKlass* caller,
                                    ciInstanceKlass* callee_holder,
                                    ciInstanceKlass* actual_receiver);

  // Given a known receiver klass, find the target for the call.
  // Return NULL if the call has no target or is abstract.
  ciMethod* resolve_invoke(ciKlass* caller, ciKlass* exact_receiver);

  // Find the proper vtable index to invoke this method.
  int resolve_vtable_index(ciKlass* caller, ciKlass* receiver);

  // Compilation directives
  bool will_link(ciKlass* accessing_klass,
                 ciKlass* declared_method_holder,
                 Bytecodes::Code bc);
  bool should_exclude();
  bool should_inline();
  bool should_not_inline();
  bool should_print_assembly();
  bool break_at_execute();
  bool has_option(const char *option);
  bool can_be_compiled();
  bool can_be_osr_compiled(int entry_bci);
  void set_not_compilable();
  bool has_compiled_code();
  int  instructions_size(int comp_level = CompLevel_any);
  void log_nmethod_identity(xmlStream* log);
  bool is_not_reached(int bci);
  bool was_executed_more_than(int times);
  bool has_unloaded_classes_in_signature();
  bool is_klass_loaded(int refinfo_index, bool must_be_resolved) const;
  bool check_call(int refinfo_index, bool is_static) const;
  bool ensure_method_data();  // make sure it exists in the VM also
  int scale_count(int count, float prof_factor = 1.);  // make MDO count commens

  // JSR 292 support
  bool is_method_handle_invoke()  const;
  bool is_method_handle_adapter() const;
  ciInstance* method_handle_type();

  // What kind of ciObject is this?
  bool is_method()                               { return true; }

  // Java access flags
  bool is_public      () const                   { return flags().is_public(); }
  bool is_private     () const                   { return flags().is_private();
  bool is_protected   () const                   { return flags().is_protected()
  bool is_static      () const                   { return flags().is_static(); }
  bool is_final       () const                   { return flags().is_final(); }
  bool is_synchronized() const                   { return flags().is_synchronize
  bool is_native      () const                   { return flags().is_native(); }
  bool is_interface   () const                   { return flags().is_interface()
  bool is_abstract    () const                   { return flags().is_abstract();
  bool is_strict      () const                   { return flags().is_strict(); }

  // Other flags
  bool is_empty_method() const;
  bool is_vanilla_constructor() const;
  bool is_final_method() const                   { return is_final() || holder()
  bool has_loops      () const;
  bool has_jsrs       () const;
  bool is_accessor    () const;
  bool is_initializer () const;
  bool can_be_statically_bound() const           { return _can_be_statically_bou

  // Print the bytecodes of this method.
  void print_codes_on(outputStream* st);
  void print_codes() {
    print_codes_on(tty);
  }
  void print_codes_on(int from, int to, outputStream* st);

  // Print the name of this method in various incarnations.
  void print_name(outputStream* st = tty);
  void print_short_name(outputStream* st = tty);

  methodOop get_method_handle_target() {
    KlassHandle receiver_limit; int flags = 0;
    methodHandle m = MethodHandles::decode_method(get_oop(), receiver_limit, fla
    return m();
  }
};
```
______***unchanged_portion_omitted_***

```
**********************************************************
   27206 Wed Oct 12 04:37:41 2011
new/src/share/vm/ci/ciObjectFactory.cpp
**********************************************************
______unchanged_portion_omitted_

//------------------------------------------------------------------
// ciObjectFactory::get_unloaded_method
//
// Get the ciMethod representing an unloaded/unfound method.
//
// Implementation note: unloaded methods are currently stored in
// an unordered array, requiring a linear-time lookup for each
// unloaded method.  This may need to change.
ciMethod* ciObjectFactory::get_unloaded_method(ciInstanceKlass* holder,
                                               ciSymbol*        name,
                                               ciSymbol*        signature,
                                               ciInstanceKlass* accessor) {
  ciSignature* that = NULL;
  for (int i = 0; i < _unloaded_methods->length(); i++) {
                                               ciSymbol*        signature) {
  for (int i=0; i<_unloaded_methods->length(); i++) {
    ciMethod* entry = _unloaded_methods->at(i);
    if (entry->holder()->equals(holder) &&
        entry->name()->equals(name) &&
        entry->signature()->as_symbol()->equals(signature)) {
      // Short-circuit slow resolve.
      if (entry->signature()->accessing_klass() == accessor) {
        // We've found a match.
        return entry;
      } else {
        // Lazily create ciSignature
        if (that == NULL)  that = new (arena()) ciSignature(accessor, constantPo
        if (entry->signature()->equals(that)) {
#endif /* ! codereview */
          // We've found a match.
          return entry;
        }
      }
    }
  }
#endif /* ! codereview */

  // This is a new unloaded method.  Create it and stick it in
  // the cache.
  ciMethod* new_method = new (arena()) ciMethod(holder, name, signature, accesso
  ciMethod* new_method = new (arena()) ciMethod(holder, name, signature);

  init_ident_of(new_method);
  _unloaded_methods->append(new_method);

  return new_method;
}
______unchanged_portion_omitted_
```

```
**********************************************************
    5093 Wed Oct 12 04:37:43 2011
new/src/share/vm/ci/ciObjectFactory.hpp
**********************************************************
/*
 * Copyright (c) 1999, 2011, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_CI_CIOBJECTFACTORY_HPP
#define SHARE_VM_CI_CIOBJECTFACTORY_HPP

#include "ci/ciClassList.hpp"
#include "ci/ciObject.hpp"
#include "utilities/growableArray.hpp"

// ciObjectFactory
//
// This class handles requests for the creation of new instances
// of ciObject and its subclasses.  It contains a caching mechanism
// which ensures that for each oop, at most one ciObject is created.
// This invariant allows efficient implementation of ciObject.
class ciObjectFactory : public ResourceObj {
  friend class VMStructs;
  friend class ciEnv;

private:
  static volatile bool _initialized;
  static GrowableArray<ciObject*>* _shared_ci_objects;
  static ciSymbol*                 _shared_ci_symbols[];
  static int                       _shared_ident_limit;

  Arena*                    _arena;
  GrowableArray<ciObject*>* _ci_objects;
  GrowableArray<ciMethod*>* _unloaded_methods;
  GrowableArray<ciKlass*>* _unloaded_klasses;
  GrowableArray<ciInstance*>* _unloaded_instances;
  GrowableArray<ciReturnAddress*>* _return_addresses;
  GrowableArray<ciSymbol*>* _symbols;  // keep list of symbols created
  int                       _next_ident;

public:
  struct NonPermObject : public ResourceObj {
    ciObject*      _object;
    NonPermObject* _next;

    inline NonPermObject(NonPermObject* &bucket, oop key, ciObject* object);
```

```
    ciObject*      object()  { return _object; }
    NonPermObject* &next()  { return _next; }
  };
private:
  enum { NON_PERM_BUCKETS = 61 };
  NonPermObject* _non_perm_bucket[NON_PERM_BUCKETS];
  int _non_perm_count;

  int find(oop key, GrowableArray<ciObject*>* objects);
  bool is_found_at(int index, oop key, GrowableArray<ciObject*>* objects);
  void insert(int index, ciObject* obj, GrowableArray<ciObject*>* objects);
  ciObject* create_new_object(oop o);
  static bool is_equal(NonPermObject* p, oop key) {
    return p->object()->get_oop() == key;
  }

  NonPermObject* &find_non_perm(oop key);
  void insert_non_perm(NonPermObject* &where, oop key, ciObject* obj);

  void init_ident_of(ciObject* obj);
  void init_ident_of(ciSymbol* obj);

  Arena* arena() { return _arena; }

  void print_contents_impl();

  ciInstance* get_unloaded_instance(ciInstanceKlass* klass);

public:
  static bool is_initialized() { return _initialized; }

  static void initialize();
  void init_shared_objects();
  void remove_symbols();

  ciObjectFactory(Arena* arena, int expected_size);

  // Get the ciObject corresponding to some oop.
  ciObject* get(oop key);

  ciSymbol* get_symbol(Symbol* key);

  // Get the ciSymbol corresponding to one of the vmSymbols.
  static ciSymbol* vm_symbol_at(int index);

  // Get the ciMethod representing an unloaded/unfound method.
  ciMethod* get_unloaded_method(ciInstanceKlass* holder,
                                 ciSymbol*        name,
                                 ciSymbol*        signature,
                                 ciInstanceKlass* accessor);
                                 ciSymbol*        signature);

  // Get a ciKlass representing an unloaded klass.
  ciKlass* get_unloaded_klass(ciKlass* accessing_klass,
                              ciSymbol* name,
                              bool create_if_not_found);

  // Get a ciInstance representing an unresolved klass mirror.
  ciInstance* get_unloaded_klass_mirror(ciKlass* type);

  // Get a ciInstance representing an unresolved method handle constant.
  ciInstance* get_unloaded_method_handle_constant(ciKlass*  holder,
                                                  ciSymbol* name,
                                                  ciSymbol* signature,
                                                  int       ref_kind);
  // Get a ciInstance representing an unresolved method type constant.
  ciInstance* get_unloaded_method_type_constant(ciSymbol* signature);


  // Get the ciMethodData representing the methodData for a method
  // with none.
  ciMethodData* get_empty_methodData();

  ciReturnAddress* get_return_address(int bci);

  void print_contents();
  void print();
};
```

______***unchanged_portion_omitted_***

```
**********************************************************
    4405 Wed Oct 12 04:37:44 2011
new/src/share/vm/ci/ciSignature.cpp
**********************************************************
______unchanged_portion_omitted_

// ------------------------------------------------------------------
// ciSignature::return_type
// ciSignature::return_ciType
//
// What is the return type of this signature?
ciType* ciSignature::return_type() const {
  return _types->at(_count);
}

// ------------------------------------------------------------------
// ciSignature::type_at
// ciSignature::ciType_at
//
// What is the type of the index'th element of this
// signature?
ciType* ciSignature::type_at(int index) const {
  assert(index < _count, "out of bounds");
  // The first _klasses element holds the return klass.
  return _types->at(index);
}

// ------------------------------------------------------------------
// ciSignature::equals
//
// Compare this signature to another one.  Signatures with different
// accessing classes but with signature-types resolved to the same
// types are defined to be equal.
bool ciSignature::equals(ciSignature* that) {
  // Compare signature
  if (!this->as_symbol()->equals(that->as_symbol()))  return false;
  // Compare all types of the arguments
  for (int i = 0; i < _count; i++) {
    if (this->type_at(i) != that->type_at(i))         return false;
  }
  // Compare the return type
  if (this->return_type() != that->return_type())     return false;
  return true;
}

// ------------------------------------------------------------------
#endif /* ! codereview */
// ciSignature::print_signature
void ciSignature::print_signature() {
  _symbol->print_symbol();
}

// ------------------------------------------------------------------
// ciSignature::print
void ciSignature::print() {
  tty->print("<ciSignature symbol=");
  print_signature();
 tty->print(" accessing_klass=");
  _accessing_klass->print();
  tty->print(" address=0x%x>", (address)this);
}
```

```
**********************************************************
    2245 Wed Oct 12 04:37:45 2011
new/src/share/vm/ci/ciSignature.hpp
**********************************************************
/*
 * Copyright (c) 1999, 2011, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_CI_CISIGNATURE_HPP
#define SHARE_VM_CI_CISIGNATURE_HPP

#include "ci/ciClassList.hpp"
#include "ci/ciSymbol.hpp"
#include "utilities/globalDefinitions.hpp"
#include "utilities/growableArray.hpp"

// ciSignature
//
// This class represents the signature of a method.
class ciSignature : public ResourceObj {
private:
  ciSymbol* _symbol;
  ciKlass*  _accessing_klass;

  GrowableArray<ciType*>* _types;
  int _size;
  int _count;

  friend class ciMethod;
  friend class ciObjectFactory;
#endif /* ! codereview */

  ciSignature(ciKlass* accessing_klass, constantPoolHandle cpool, ciSymbol* sign

  void get_all_klasses();

  Symbol* get_symbol() const                     { return _symbol->get_symbol();

public:
  ciSymbol* as_symbol() const                    { return _symbol; }
  ciKlass*  accessing_klass() const              { return _accessing_klass; }
#endif /* ! codereview */

  ciType* return_type() const;
  ciType* type_at(int index) const;
```

```
  int       size() const                         { return _size; }
  int       count() const                        { return _count; }

  bool equals(ciSignature* that);

#endif /* ! codereview */
  void print_signature();
  void print();
};

#endif // SHARE_VM_CI_CISIGNATURE_HPP
```

****
   75656 Wed Oct 12 04:37:46 2011
new/src/share/vm/prims/methodHandleWalk.cpp
****
______***unchanged_portion_omitted_***

```
// ---------------------------------------------------------------------------
// MethodHandleCompiler
//

// Values used by the compiler.
jvalue MethodHandleCompiler::zero_jvalue = { 0 };
jvalue MethodHandleCompiler::one_jvalue  = { 1 };

// Fetch any values from CountingMethodHandles and capture them for profiles
bool MethodHandleCompiler::fetch_counts(ArgToken arg1, ArgToken arg2) {
  int count1 = -1, count2 = -1;
  if (arg1.token_type() == tt_constant && arg1.basic_type() == T_OBJECT &&
      java_lang_invoke_CountingMethodHandle::is_instance(arg1.object()())) {
    count1 = java_lang_invoke_CountingMethodHandle::vmcount(arg1.object()());
  }
  if (arg2.token_type() == tt_constant && arg2.basic_type() == T_OBJECT &&
      java_lang_invoke_CountingMethodHandle::is_instance(arg2.object()())) {
    count2 = java_lang_invoke_CountingMethodHandle::vmcount(arg2.object()());
  }
  int total = count1 + count2;
  if (count1 != -1 && count2 != -1 && total != 0) {
    // Normalize the collect counts to the invoke_count
    tty->print("counts %d %d scaled by %d = ", count2, count1, _invoke_count);
    if (count1 != 0) _not_taken_count = (int)(_invoke_count * count1 / (double)t
    if (count2 != 0) _taken_count = (int)(_invoke_count * count2 / (double)total
    tty->print_cr("%d %d", _taken_count, _not_taken_count);
    return true;
  }
  return false;
}
```
______***unchanged_portion_omitted_***