Welcome to this talk about Clojure. The talk is not about the language from the end-user perspective but about the Clojure implementation itself. The implementation we are looking at is of course the reference RCI (Rich's Clojure Implementation). There is a lot of complexity involved in the implementation of such a successful language and this talk is my attempt to understand some of most interesting corner of the Clojure sources.

An amazing growth

Mar 2006: First commit

Oct 2006: 30k LOC 7 months old

Oct 2007: Clojure announced!

Oct 2008: Clojure invited at Lisp50

May 2009: 1.0 + book!

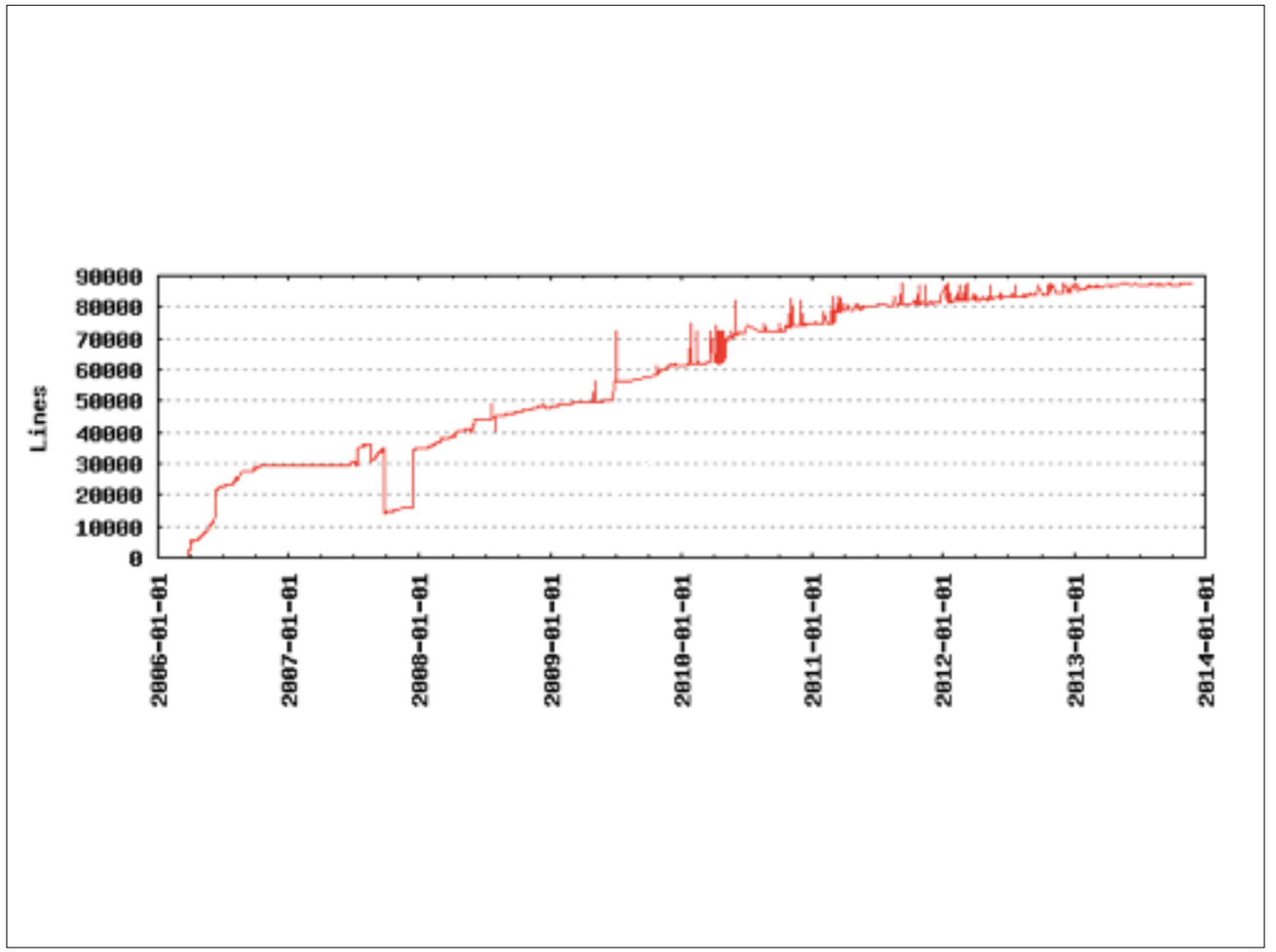
This is a quick overview of the important initial milestones of Clojure. One thing overall: the growth ratio is impressive: 30k LOC after only 7 months of work.

If we ever decide to have a Clojure retreat day, that must definitely be October the 10th.

Info:

http://clojure.org/space/stats/overview

http://clojure-log.n01se.net



Sorry for the brightness bomb.

Initial Code Milestones

Apr ’06: Lisp2Java sources

May ’06: boot.clj appears

May ’06: STM first cut

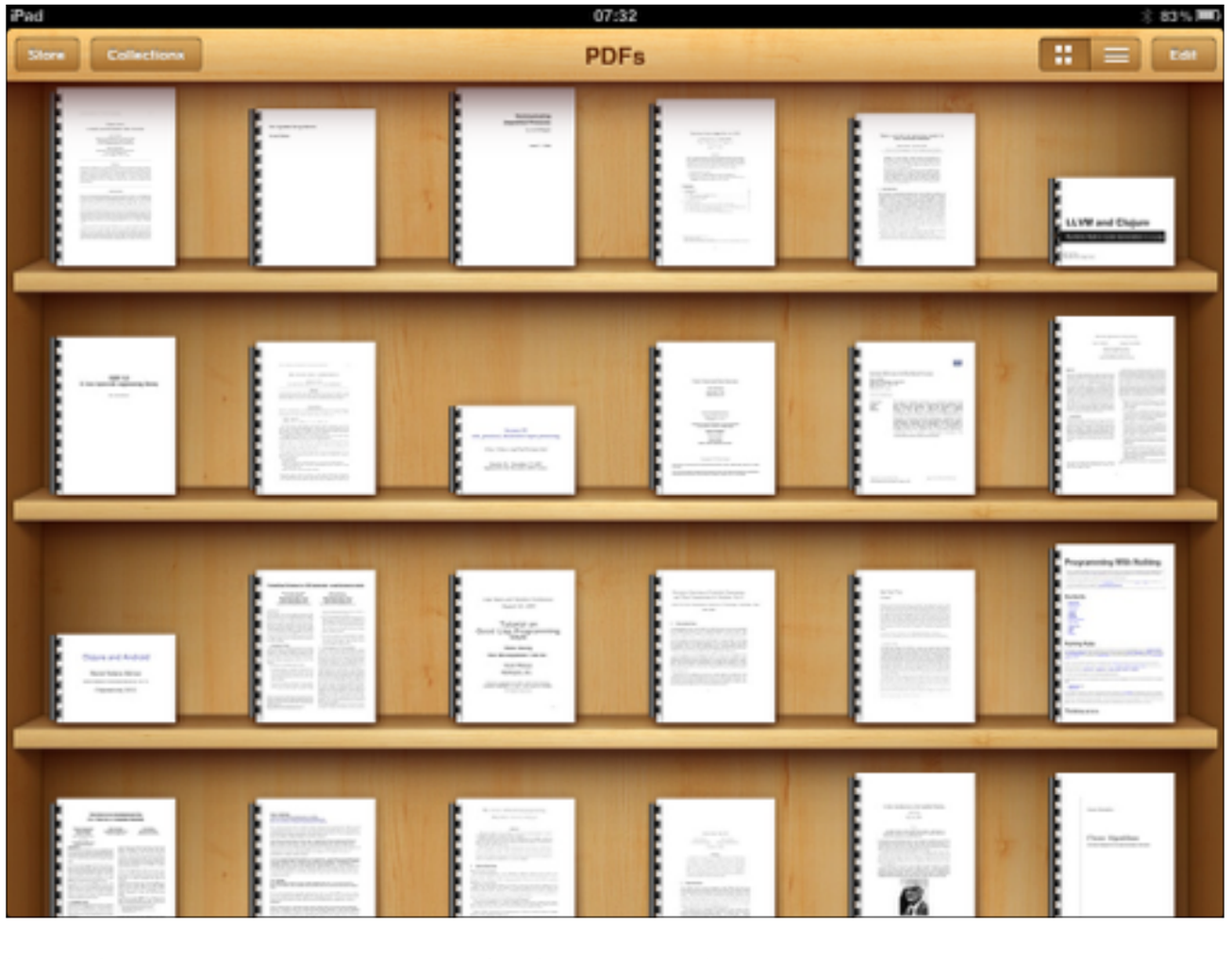
Jun ’06: First persistent data structure

Sep ’06: Java2Java sources

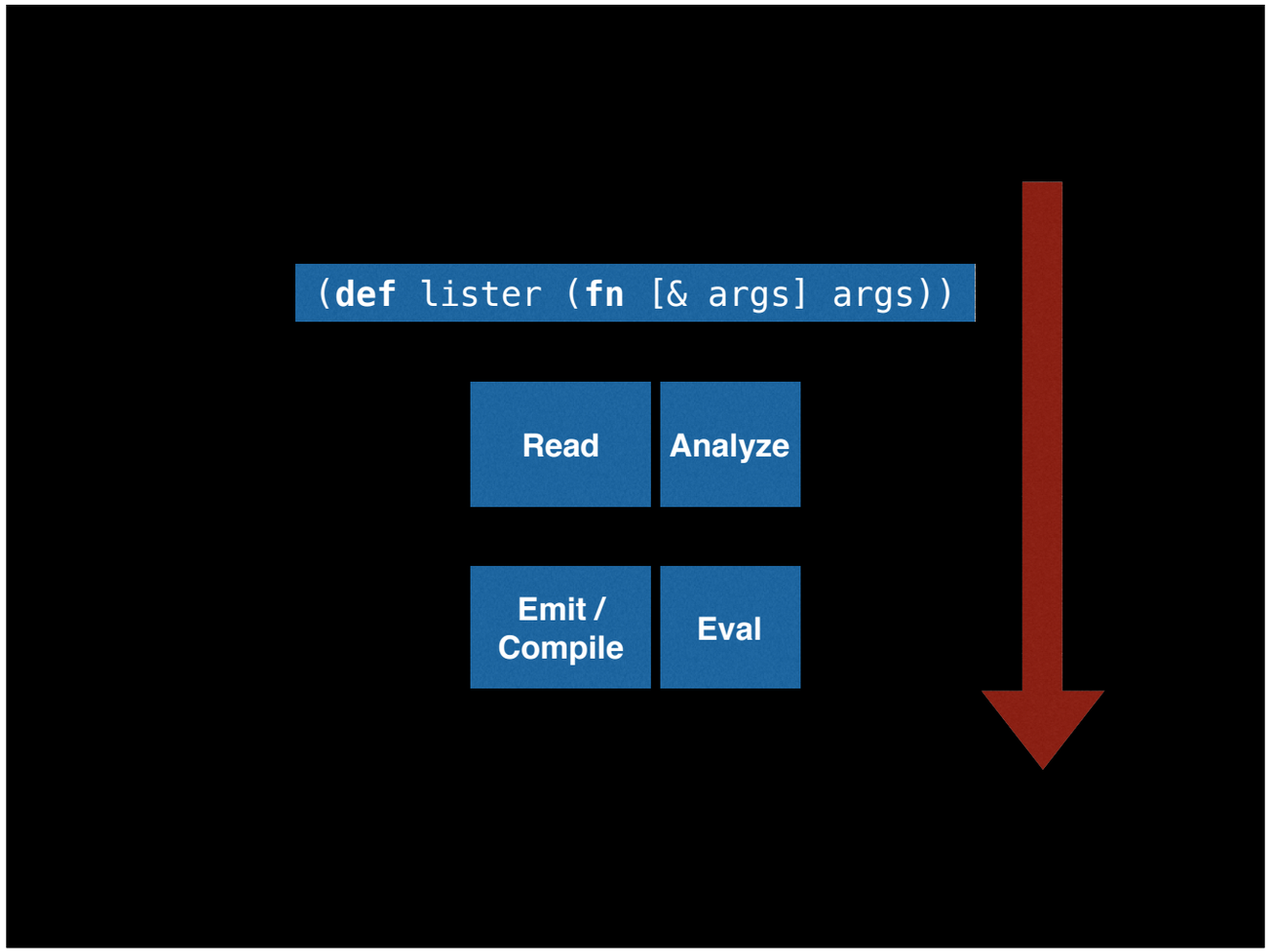
Aug ’07: Java2Bytecode started

Right after: almost all the rest, Refs, LockingTx

Clojure starts its existence as a Lisp DSL. The first Lisp implementation requires LispWorks to run and is generating compilable Java.
lib.lisp is the first incarnation of core.clj and contains the first implementation of the Clojure standard library.
All important language aspects we are used today are present since the very beginning. Work on persistence data structures, STM, Java interop proceeds almost in parallel.
The introduction of the LispReader Java class is the first translation of parts of the Lisp code into Java. The next big part of it being the Compiler.
The Compiler first cut is mainly a translation of the Lisp code and is still generating Java sources.
Curiosity: Clojure at some point got an ANTLR grammar experiment, if you ever interested in one (although maybe not up to date) you need to dig the commits.
The introduction of the BytecodeCompiler is architecturally the last big design change. With the BytecodeCompiler Clojure can finally have a REPL and be truly dynamic.
Of course there is much more since 2007 to nowadays and things got terribly more complicated, but the same compiler architecture is still there with Clojure happily generating tons of byte code fun.



iBook on my iPad, 4 months into studying the Clojure sources. Go refresh your CS degree! Now! Data structures and algorithms are pervasive. Deep JVM concurrency knowledge. Lisp internals, concepts and heritage. Big OO hierarchies and deep FP



30,000 ft view. Clojure can be thought very broadly as a big Lisp DSL to drive byte code generation. We can quickly see this in action by dumping the class that is generated for a function. In order to achieve that result, Clojure needs to parse the original form, understand what instructions are contained in it and emit the necessary bytecode. In the next section we are going to explore the main objects involved during processing of Clojure s-expressions. S-expr means symbolic expression, a form of notation to express recursive lists of “symbols” introduced with Lisp. From the point of view of parsing textual s-expressions Clojure is very similar to Lisp.

Reader

```
(def lister (fn [& args] args))
```

```
(
  def
  lister
  (
    fn
    [& args]
    args))
```

```
PersistentList(
  Symbol(“def"),
  Symbol(“lister"),
  PersistentList(
    Symbol("fn"),
    PersistentVector("&", Symbol(“args”)),
    Symbol("args")))
```

By the way, ever wondered why there is a LispReader in Clojure that reads Clojure forms? Isn’t this a ClojureReader? Some of the early Clojure history is showing here. Let’s take a simple example. We feed Clojure (it can be at the REPL or loading a script with that line in it) with a definition for a function of a variable number of arguments that returns those arguments as a list. The list function indeed. The LispReader decomposes text (which is always a list in Clojure) into instances of internal classes, like PersistentList of Symbols. The resulting outer PersistentList is usually called a “form”. The nice thing about recursive list is that they can be handled recursively, from parsing to analysing and then to the emission phase.

- Transforms Clojure text into symbolic lists
- Cons, PersistentVector, IteratorSeq and so on
- Uses a push back reader to read/unread
- Perform part of the syntax validation

Analyzer

```
(def lister (fn [& args] args))
```

```
PersistentList(
  Sym(“def"),
  Sym(“lister”),
  PersistentList(
    Sym("fn"),
    PersistentVector(
      "&",
      Sym(“args”)),
    Sym("args")))
```

```
DefExpr(
  Var(“lister"),
  FnExpr(
    Sym(“lister"),
    PersistentList(
      FnMethod(
        LocalBinding(Sym(“args”)),
        BodyExpr(
          PersistentVector(
            LocalBindingExpr(“args”)))))))
```

- Transform LispReader forms into Expr
- Compiler.analyze() & friends
- Expr inner classes implementing IParser

There is not really an Analyzer class in Clojure. The compiler contains 4 analyzer functions. The rest of the analysis is done by implementors of the IParser interface, which are all static inner classes of Expr interface implementors. Expr classes are for example: DefExpr, FnExpr, MethodExpr, BodyExpr, LocalBindingExpr and so on.

Let’s continue on the previous expansion. Based on the (first PersistentList) Symbol (def, fn, and so on) a specific Parser class is selected from a corresponding Expr class. The parser instance knows how to build expression objects out of the (rest PersistentList) or stop on a leaf without invoking “analyze” any further. The result is a recursive hierarchy of Expr classes that know how to emit/eval them selves.

Emission

Mainly bytecode generation for Expr

Prerequisite for evaluation

emit() method in Expr interface

Notable exception: called over FnExpr analysis

Emitting has to do with generation of bytecode. The emit() method takes a context, the expression to transform and an ASM::GeneratorAdapter instance. For example DefExpr.emit() will coordinate bytecode generation for its meta info and it's initialisation form. Bytecode generation needs to happen before Evaluation, because evaluation will return the objects ready to be used.

Evaluation

Transform Exprs into their “usable form”

For example: a new Object, a Var, a Namespace

FnExpr is just getCompiledClass().newInstance()

Evaluating means returning the last step of the s-expression transformation process ready to be used in Java code, hence Clojure.

For example, DefExpr.eval() returns the created Var after invoking eval() on its init instance (for an fn that would be a new instance of an object extending AFn).

Compilation

Usually coordination for emit

Compiler.compile namespace -> file

ObjExpr.compile -> coordinate FnExpr emission

Compilation methods tend to coordinate emission action, especially in the case of AOT compilation. Compiler.java contains a compile method that iterates over a list of forms (likely from a clojure script in the classpath) and then analyse, emit and eval each form in them.
Another use of the compile method happens during FnExpr.parser().parse() which is analysing the form. After analysis, FnExpr already contains the generated class.

Emit

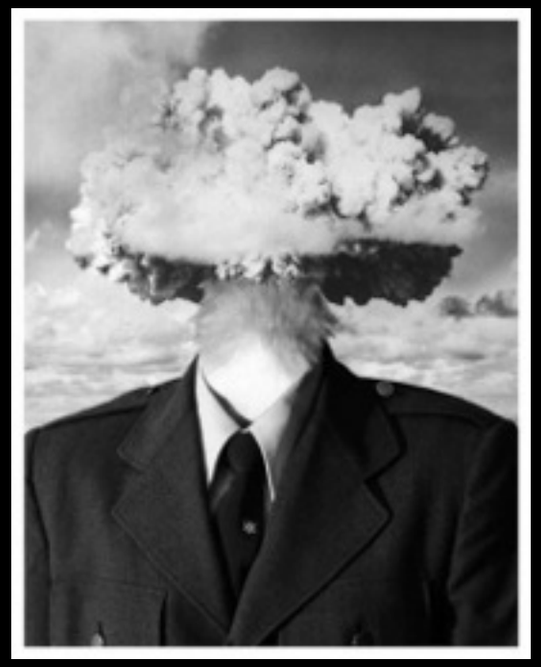
```
(def lister (fn [& args] args))
```

```
DefExpr(
  Var(“lister"),
  FnExpr(
    Sym(“lister"),
    PersistentList(
      FnMethod(
        LocalBinding(
        Sym(“args”)),
        BodyExpr(
          PersistentVector(
          LocalBindingExpr(
“args”)))))))
```

```
import clojure.lang.RestFn;

public class user$lister extends RestFn {

   public Object doInvoke(Object args) {
       return args;
    }

    public int getRequiredArity() {
        return 0;
    }

    public user$lister() {
    }
}
```

The final step for our lister example consists of invoking eval() on the corresponding DefExpr which essentially is var.bindRoot(init.eval()) where “init” is the init form for this def expression, in our case an FnExpr. init.eval() is compiledClass.newInstance(). At the REPL we can finally invoke the function pointed by the created var.

I know how you feel…



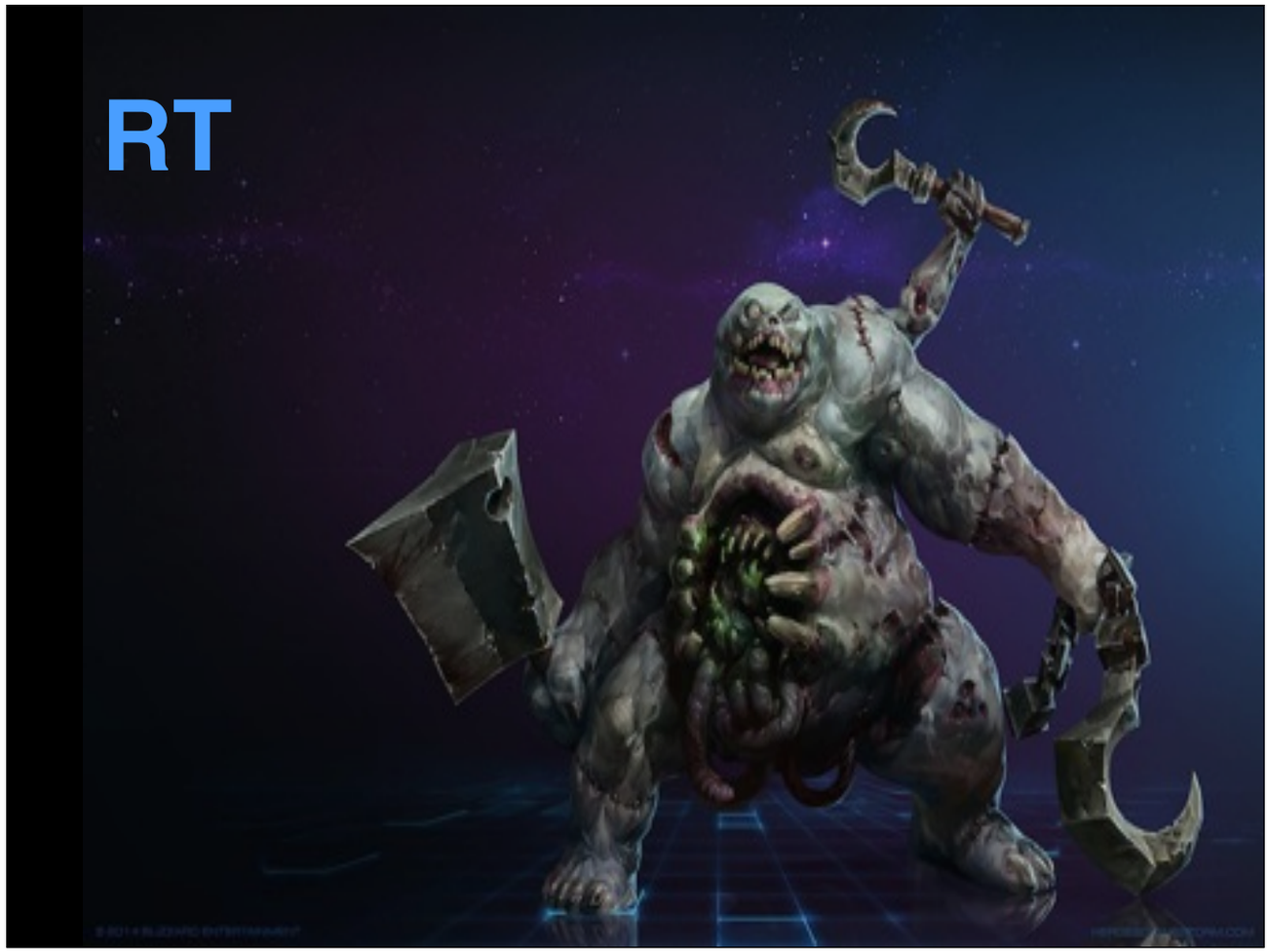
Stress, confusion, hallucinations!

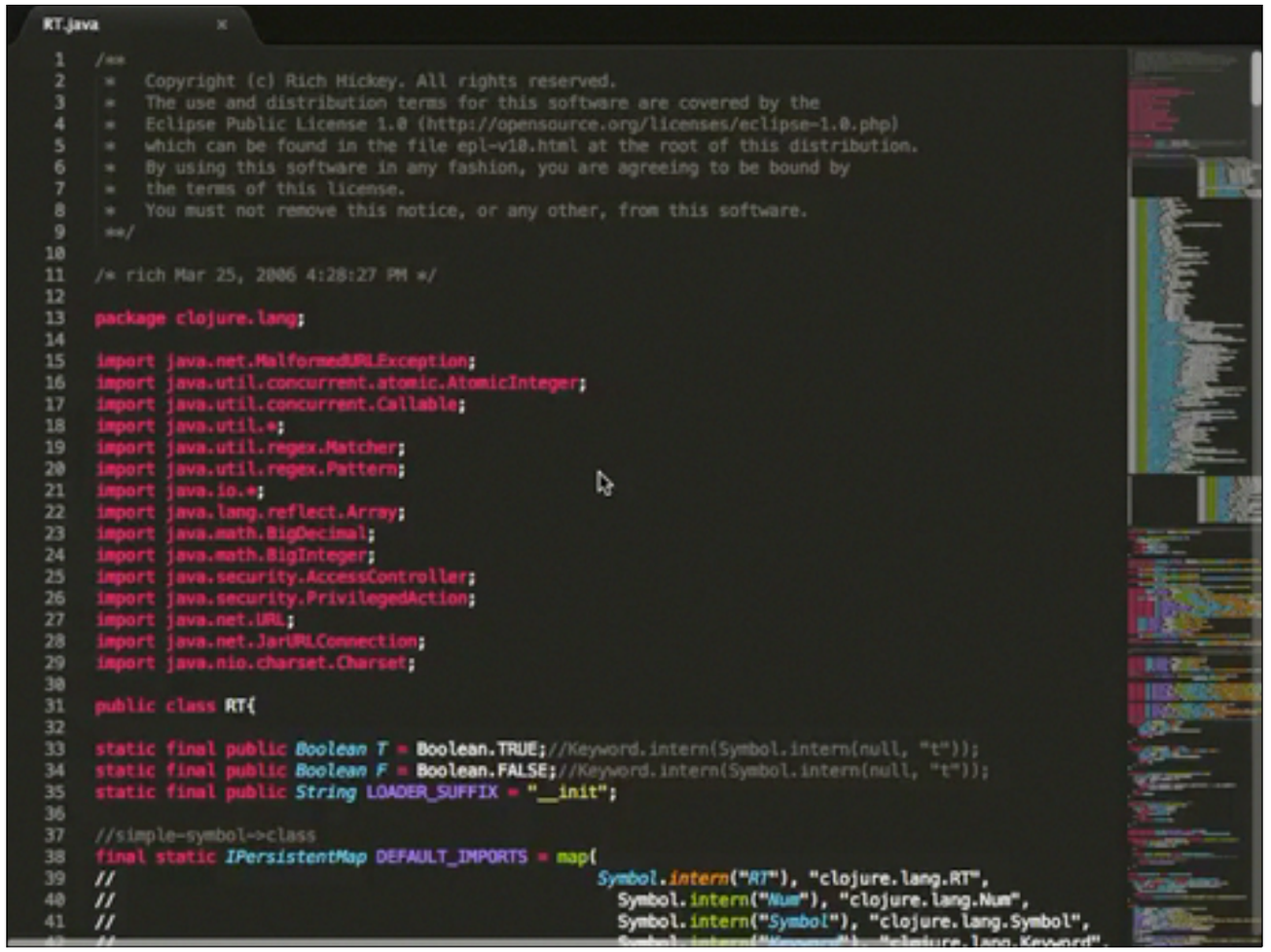
I know how you feel… Let me introduce you to the beasts in more details.

Where are the monsters?

```
package clojure;

import ...

public class main{

final static private Symbol CLOJURE_MAIN = Symbol.intern("clojure.main");
final static private Var REQUIRE = RT.var("clojure.core", "require");
final static private Var LEGACY_REPL = RT.var("clojure.main", "legacy-repl");
final static private Var LEGACY_SCRIPT = RT.var("clojure.main", "legacy-script");
final static private Var MAIN = RT.var("clojure.main", "main");

public static void legacy_repl(String[] args) {...}

public static void legacy_script(String[] args) {...}

public static void main(String[] args) {
    REQUIRE.invoke(CLOJURE_MAIN);
    MAIN.applyTo(RT.seq(args));
}
}
```

Our journey begins from the Java main class, the one you invoke to start Clojure with java -jar. This innocent main is hiding the parsing of the entire standard library. As soon as you touch RT its static initialisation kicks in. That in turns trigger other important static inits in Compiler, Var, Namespace.


RT



RT stands for RunTime, although RT.java today is more an Util class. It's main responsibilities:

- Main Clojure bootstrap
- Basic definitions
- List manipulation

RT <clinit>

```
299 static{
300     Keyword arglistskw = Keyword.intern(null, "arglists");
301     Symbol namesym = Symbol.intern("name");
302     OUT.setTag(Symbol.intern("java.io.Writer"));
303     CURRENT_NS.setTag(Symbol.intern("clojure.lang.Namespace"));
304     AGENT.setMeta(map(DOC_KEY, "The agent currently running an action on this thread, else nil"));
305     AGENT.setTag(Symbol.intern("clojure.lang.Agent"));
306     MATH_CONTEXT.setTag(Symbol.intern("java.math.MathContext"));
307     Var nv = Var.intern(CLOJURE_NS, NAMESPACE, bootNamespace);
308     nv.setMacro();
309     Var v;
310     v = Var.intern(CLOJURE_NS, IN_NAMESPACE, inNamespace);
311     v.setMeta(map(DOC_KEY, "Sets *ns* to the namespace named by the symbol, creating it if needed.",
312                   arglistskw, list(vector(namesym))));
313     v = Var.intern(CLOJURE_NS, LOAD_FILE,
314                    (AFn) invoke(arg1) -> {
316                        try
317                            {
318                            return Compiler.loadFile((String) arg1);
319                            }
320                        catch(IOException e)
321                            {
322                            throw Util.sneakyThrow(e);
323                            }
324                    });
326     v.setMeta(map(DOC_KEY, "Sequentially read and evaluate the set of forms contained in the file.",
327                   arglistskw, list(vector(namesym))));
328     try {
329         doInit();
330     }
331     catch(Exception e) {
332         throw Util.sneakyThrow(e);
333     }
334 }
```

There is quite a lot before this static initialisation block, but this is the most important part. Here you can see we are doing the Var.intern of CLOJURE_NS, something we'll be soon using.

But let's move baby steps. The important part here is that doInit() helper. Let's have a deeper look.

RT.doInit()

```
446  static void doInit() throws ClassNotFoundException, IOException{
447      load("clojure/core");
448
449      Var.pushThreadBindings(
450              RT.mapUniqueKeys(CURRENT_NS, CURRENT_NS.deref(),
451                      WARN_ON_REFLECTION, WARN_ON_REFLECTION.deref()
452                      ,RT.UNCHECKED_MATH, RT.UNCHECKED_MATH.deref()));
453      try {
454          Symbol USER = Symbol.intern("user");
455          Symbol CLOJURE = Symbol.intern("clojure.core");
456
457          Var in_ns = var("clojure.core", "in-ns");
458          Var refer = var("clojure.core", "refer");
459          in_ns.invoke(USER);
460          refer.invoke(CLOJURE);
461          maybeLoadResourceScript("user.clj");
462      }
463      finally {
464          Var.popThreadBindings();
465      }
466  }
```

That's right, we are going to load the entire standard library right there. Also notice the Var.pushThreadBindings(), an idiom that is found often in the Clojure Java sources and clojure standard library as well.

RT.load()

```
414  static public void load(String scriptbase, boolean failIfNotFound) throws IOException, ClassNotFo
415      String classfile = scriptbase + LOADER_SUFFIX + ".class";
416      String cljfile = scriptbase + ".clj";
417      URL classURL = getResource(baseLoader(),classfile);
418      URL cljURL = getResource(baseLoader(), cljfile);
419      boolean loaded = false;
420
421      if((classURL != null &&
422          (cljURL == null
423           || lastModified(classURL, classfile) > lastModified(cljURL, cljfile)))
424         || classURL == null) {
425          try {
426              Var.pushThreadBindings(
427                      RT.mapUniqueKeys(CURRENT_NS, CURRENT_NS.deref(),
428                              WARN_ON_REFLECTION, WARN_ON_REFLECTION.deref()
429                                      ,RT.UNCHECKED_MATH, RT.UNCHECKED_MATH.deref()));
430              loaded = (loadClassForName(scriptbase.replace('/', '.') + LOADER_SUFFIX) != null);
431          }
432          finally {
433              Var.popThreadBindings();
434          }
435      }
436      if(!loaded && cljURL != null) {
437          if(booleanCast(Compiler.COMPILE_FILES.deref()))
438              compile(cljfile);
439          else
440              loadResourceScript(RT.class, cljfile);
441      }
442      else if(!loaded && failIfNotFound)
443          throw new FileNotFoundException(String.format("Could not locate %s or %s on classpath: ",
444  }
```

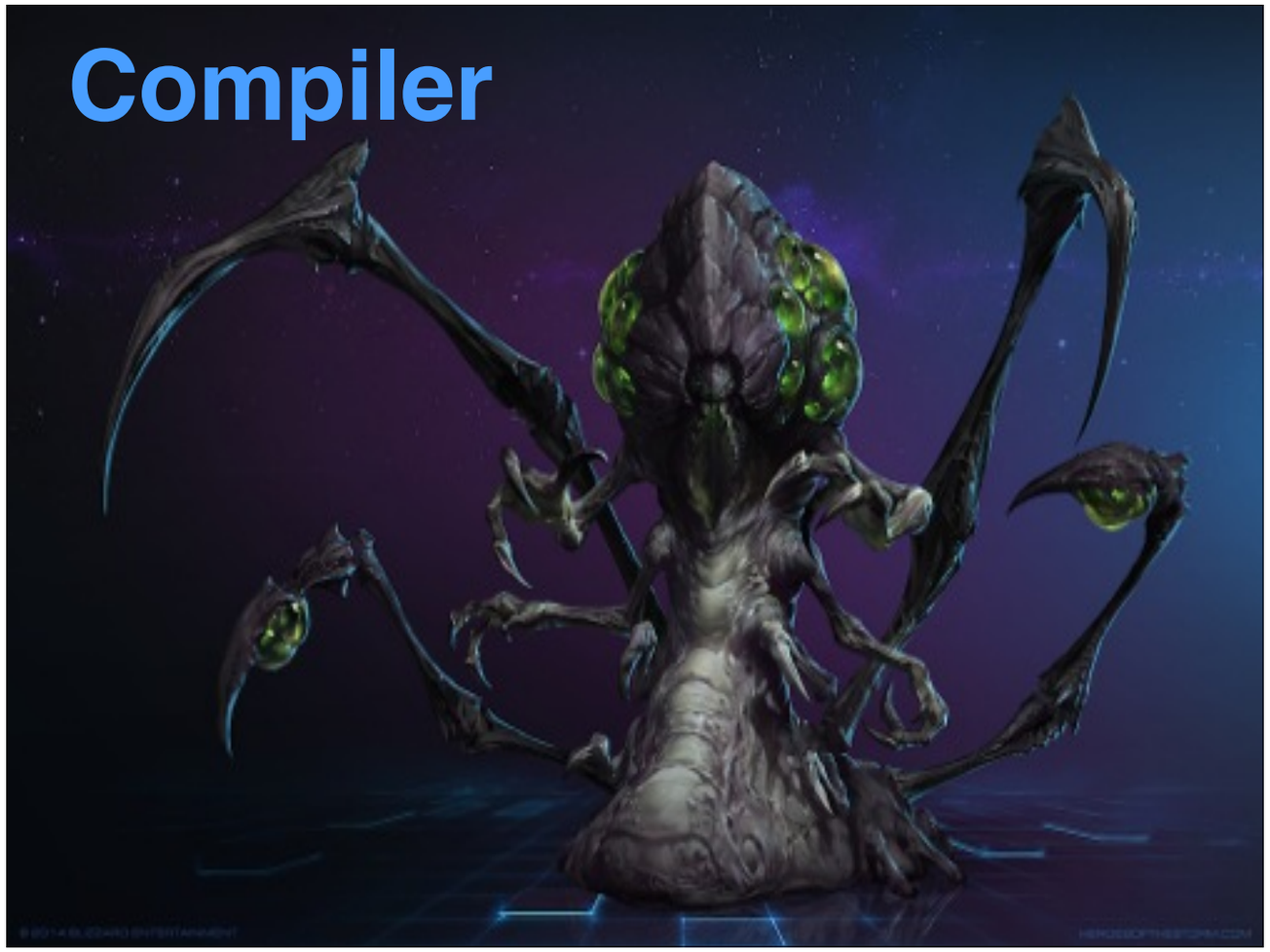
We are not taking the compilation branch here, because we didn’t enter from the std lib compile function that is setting that *compile-files* dynamic var to true. Other interesting bits here are searching for an already compiled class for this script base and if existing we are going to skip an additional compilation.

RT.load()

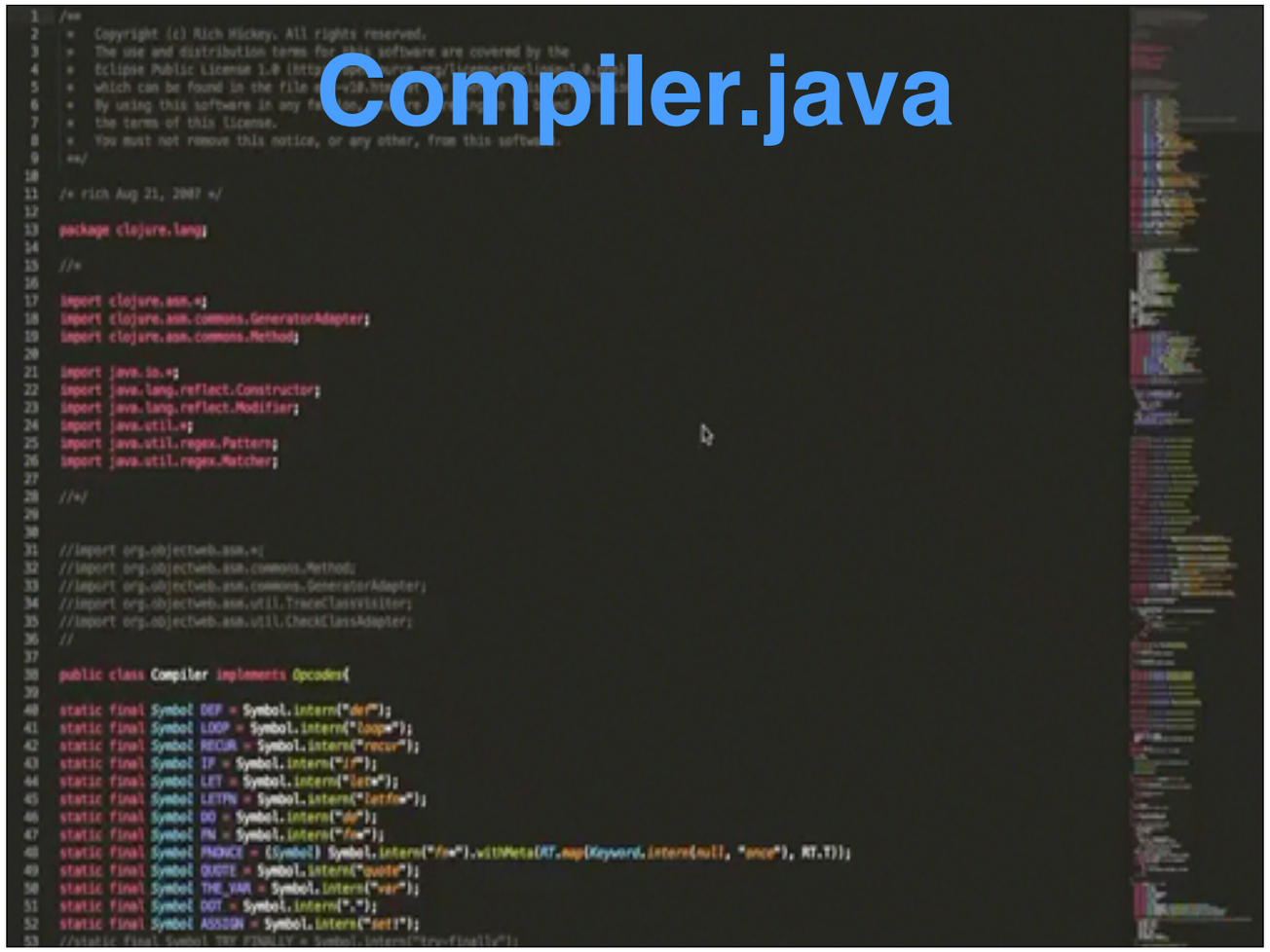
```
364  public static void loadResourceScript(Class c, String name, boolean fail
365      int slash = name.lastIndexOf('/');
366      String file = slash >= 0 ? name.substring(slash + 1) : name;
367      InputStream ins = resourceAsStream(baseLoader(), name);
368      if(ins != null) {
369          try {
370              Compiler.load(new InputStreamReader(ins, UTF8), name, file);
371          }
372          finally {
373              ins.close();
374          }
375      }
376      else if(failIfNotFound) {
377          throw new FileNotFoundException("Could not locate Clojure resour
378      }
379  }
```

Leaving the RT realm for the moment to enter the compiler.

Compiler



Compiler is a fat class in terms of LOC. The compiler as it is designed today is a “third pass”: first it was conceived as a Lisp emitting Java sources, then translated to Java from Lisp, third it was rewritten to support bytecode emission.



Peculiar style in Clojure sources is the use of inner classes, preferred usually instead of packages. One reason why this can be useful can be found on the design page for JScheme, a 1998 Scheme to Java compiler written by Peter Norvig who clearly stated that in order to make Java coding more “lispy” he hated the idea to always prepend the name of the class to the method invocation. With inner classes, methods also from other classes inside the same compilation unit can be called by name only. Remember there was no import static at that time, same for Rich in 2006.

Compiler.load()

```
7128  public static Object load(Reader rdr, String sourcePath, String sourceName) {
7129      Object EOF = new Object();
7130      Object ret = null;
7131      LineNumberingPushbackReader pushbackReader =
7132              (rdr instanceof LineNumberingPushbackReader) ? (LineNumberingPushbackReader) rdr :
7133              new LineNumberingPushbackReader(rdr);
7134      Var.pushThreadBindings(
7135              RT.mapUniqueKeys(LOADER, RT.makeClassLoader(),
7136                      SOURCE_PATH, sourcePath,
7137                      SOURCE, sourceName,
7138                      METHOD, null,
7139                      LOCAL_ENV, null,
7140                       LOOP_LOCALS, null,
7141                       NEXT_LOCAL_NUM, 0,
7142                       RT.READEVAL, RT.T,
7143                      RT.CURRENT_NS, RT.CURRENT_NS.deref(),
7144                      LINE_BEFORE, pushbackReader.getLineNumber(),
7145                      COLUMN_BEFORE, pushbackReader.getColumnNumber(),
7146                      LINE_AFTER, pushbackReader.getLineNumber(),
7147                      COLUMN_AFTER, pushbackReader.getColumnNumber()
7148                      ,RT.UNCHECKED_MATH, RT.UNCHECKED_MATH.deref()
7149                       ,RT.WARN_ON_REFLECTION, RT.WARN_ON_REFLECTION.deref()
7150                      ,RT.DATA_READERS, RT.DATA_READERS.deref()
7151                           ));
7152
7153      try
7154          {
7155          for(Object r = LispReader.read(pushbackReader, false, EOF, false); r != EOF;
7156              r = LispReader.read(pushbackReader, false, EOF, false))
7157              {
7158              LINE_AFTER.set(pushbackReader.getLineNumber());
7159              COLUMN_AFTER.set(pushbackReader.getColumnNumber());
7160              ret = eval(r,false);
7161              LINE_BEFORE.set(pushbackReader.getLineNumber());
```

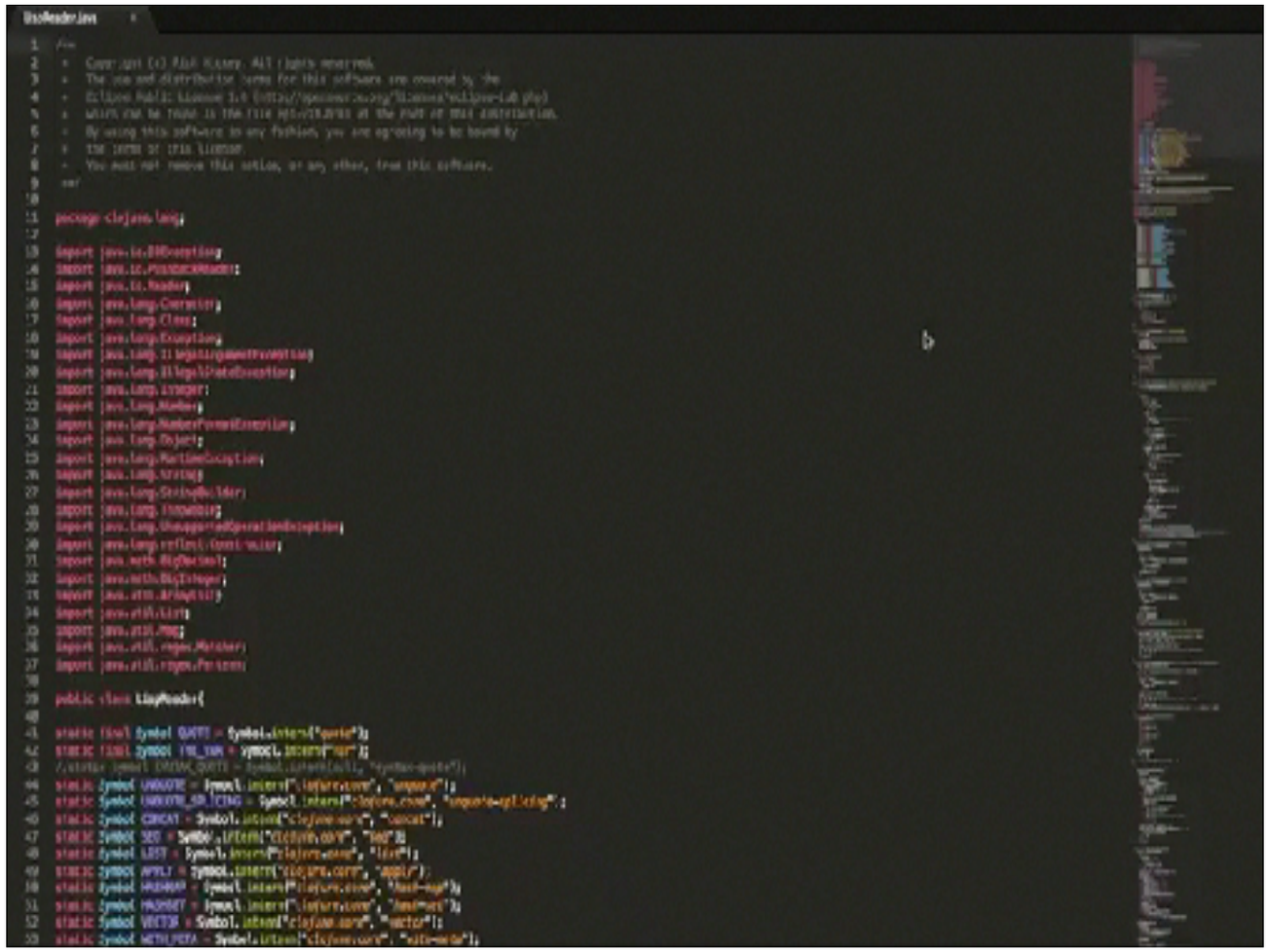
Load main goal is to loop over the forms found in an input stream, parse them with the reader and pass them down to evaluation, one by one.

Pushing and popping from the thread local storage is not idiomatic Java, where the thread-local is reserved for special usage (like session or transaction IDs). It is pervasive in the Clojure codebase to see values pushed on the thread local and popped right after. Nested invocations will make use of that state.

First we'll introduce another interesting beast, the LispReader.

LispReader

Well, you might ask why there is a LispReader instead of a Clojure reader, aren't we reading Clojure after all? I think it all goes back to when Clojure was implemented as a clojure.lisp script of about 1k lines. When the Lisp was ported into Java the name sort of stayed around (svn://svn.code.sf.net/p/clojure/code/trunk@164).



Like the Compiler.java, the LispReader.java contains invokable AFn inner classes, a strategy for each of the thing that can be read. The LispReader is also responsible for a first syntactic check and throws error for malformed maps, vectors, lists and so on.

LispReader.read()

```
151  static public Object read(PushbackReader r, boolean eofIsError, Object eofValue, boolean isRecursive)
152  {
153      if(RT.READEVAL.deref() == UNKNOWN)
154          throw Util.runtimeException("Reading disallowed - *read-eval* bound to :unknown");
155
156      try
157          {
158          for(; ;)
159              {
160              int ch = read1(r);
161
162              while(isWhitespace(ch))
163                  ch = read1(r);
164
165              if(ch == -1)
166                  {
167                  if(eofIsError)
168                      throw Util.runtimeException("EOF while reading");
169                  return eofValue;
170                  }
171
172              if(Character.isDigit(ch))
173                  {
174                  Object n = readNumber(r, (char) ch);
175                  if(RT.suppressRead())
176                      return null;
177                  return n;
178                  }
179
180              IFn macroFn = getMacro(ch);
181              if(macroFn != null)
182                  {
183                  Object ret = macroFn.invoke(r, (char) ch);
```

A special table of reader macros is looked up for the char currently pointed at, if it was not matching other kind of supported literals.

LispReader <clinit>

```
81  static
82      {
83      macros['"'] = new StringReader();
84      macros[';'] = new CommentReader();
85      macros['\''] = new WrappingReader(QUOTE);
86      macros['@'] = new WrappingReader(DEREF);//new DerefReader();
87      macros['^'] = new MetaReader();
88      macros['`'] = new SyntaxQuoteReader();
89      macros['~'] = new UnquoteReader();
90      macros['('] = new ListReader();
91      macros[')'] = new UnmatchedDelimiterReader();
92      macros['['] = new VectorReader();
93      macros[']'] = new UnmatchedDelimiterReader();
94      macros['{'] = new MapReader();
95      macros['}'] = new UnmatchedDelimiterReader();
96  //  macros['|'] = new ArgVectorReader();
97      macros['\\'] = new CharacterReader();
98      macros['%'] = new ArgReader();
99      macros['#'] = new DispatchReader();
100
101
102     dispatchMacros['^'] = new MetaReader();
103     dispatchMacros['\''] = new VarReader();
104     dispatchMacros['"'] = new RegexReader();
105     dispatchMacros['('] = new FnReader();
106     dispatchMacros['{'] = new SetReader();
107     dispatchMacros['='] = new EvalReader();
108     dispatchMacros['!'] = new CommentReader();
109     dispatchMacros['<'] = new UnreadableReader();
110     dispatchMacros['_'] = new DiscardReader();
111     }
```

Using an array of indexes to Reader strategies is a cheap form of dynamic lookup. In order to be extensible without recompilation of the sources, the commonly adopted idiomatic way is a Class.forName of a reader strategy found on the classpath.

LispReader$FnReader

```
624  public static class FnReader extends AFn{
625      public Object invoke(Object reader, Object lparen) {
626          PushbackReader r = (PushbackReader) reader;
627          if(ARG_ENV.deref() != null)
628              throw new IllegalStateException("Nested #()s are not allowed");
629          try
630              {
631              Var.pushThreadBindings(
632                      RT.map(ARG_ENV, PersistentTreeMap.EMPTY));
633              unread(r, '(');
634              Object form = read(r, true, null, true);
635
636              PersistentVector args = PersistentVector.EMPTY;
637              PersistentTreeMap argsyms = (PersistentTreeMap) ARG_ENV.deref();
638              ISeq rargs = argsyms.rseq();
639              if(rargs != null)
640                  {
641                  int higharg = (Integer) ((Map.Entry) rargs.first()).getKey();
642                  if(higharg > 0)
643                      {
644                      for(int i = 1; i <= higharg; ++i)
645                          {
646                          Object sym = argsyms.valAt(i);
647                          if(sym == null)
648                              sym = garg(i);
649                          args = args.cons(sym);
650                          }
651                      }
652                  Object restsym = argsyms.valAt(-1);
653                  if(restsym != null)
654                      {
655                      args = args.cons(Compiler._AMP_);
656                      args = args.cons(restsym);
657                      }
658                  }
```

Some interesting things in a typical reader strategy are highlighted here. LispReader is passed an instance of a push-back reader, a reader with a cursor that can be moved back and forward. This enables look-ahead of symbols or unread to pass the form to a different reader when necessary. Another important aspect is that beginning from the reader and up to the analyzer, Clojure is eating its own dog food. Persistent data structures are pervasive and Java basic collections are rarely used.

Back to Compiler.load()

```
7128  public static Object load(Reader rdr, String sourcePath, String sourceName) {
7129      Object EOF = new Object();
7130      Object ret = null;
7131      LineNumberingPushbackReader pushbackReader =
7132              (rdr instanceof LineNumberingPushbackReader) ? (LineNumberingPushbackReader) rdr :
7133              new LineNumberingPushbackReader(rdr);
7134      Var.pushThreadBindings(
7135              RT.mapUniqueKeys(LOADER, RT.makeClassLoader(),
7136                      SOURCE_PATH, sourcePath,
7137                      SOURCE, sourceName,
7138                      METHOD, null,
7139                      LOCAL_ENV, null,
7140                       LOOP_LOCALS, null,
7141                       NEXT_LOCAL_NUM, 0,
7142                       RT.READEVAL, RT.T,
7143                      RT.CURRENT_NS, RT.CURRENT_NS.deref(),
7144                      LINE_BEFORE, pushbackReader.getLineNumber(),
7145                      COLUMN_BEFORE, pushbackReader.getColumnNumber(),
7146                      LINE_AFTER, pushbackReader.getLineNumber(),
7147                      COLUMN_AFTER, pushbackReader.getColumnNumber()
7148                      ,RT.UNCHECKED_MATH, RT.UNCHECKED_MATH.deref()
7149                       ,RT.WARN_ON_REFLECTION, RT.WARN_ON_REFLECTION.deref()
7150                      ,RT.DATA_READERS, RT.DATA_READERS.deref()
7151                           ));
7152
7153      try
7154          {
7155          for(Object r = LispReader.read(pushbackReader, false, EOF, false); r != EOF;
7156              r = LispReader.read(pushbackReader, false, EOF, false))
7157              {
7158              LINE_AFTER.set(pushbackReader.getLineNumber());
7159              COLUMN_AFTER.set(pushbackReader.getColumnNumber());
7160              ret = eval(r,false);
7161              LINE_BEFORE.set(pushbackReader.getLineNumber());
```

Each read from the LispReader is returning a brand new form as a list of symbols and potentially embedded lists. Despite its controversial name, Compiler.eval() is indeed managing the Analyzer pass.

Compiler.eval()

```
6715        try
6716            {
6717            form = macroexpand(form);
6718            if(form instanceof ISeq && Util.equals(RT.first(form), DO))
6719                {
6720                ISeq s = RT.next(form);
6721                for(; RT.next(s) != null; s = RT.next(s))
6722                    eval(RT.first(s), false);
6723                return eval(RT.first(s), false);
6724                }
6725            else if((form instanceof IType) ||
6726                    (form instanceof IPersistentCollection
6727                    && !(RT.first(form) instanceof Symbol
6728                        && ((Symbol) RT.first(form)).name.startsWith("def"))))
6729                {
6730                ObjExpr fexpr = (ObjExpr) analyze(C.EXPRESSION, RT.list(FN, PersistentVector.EMPTY, form)
6731                                                  , "eval" + RT.nextID());
6732                IFn fn = (IFn) fexpr.eval();
6733                return fn.invoke();
6734                }
6735            else
6736                {
6737                Expr expr = analyze(C.EVAL, form);
6738                return expr.eval();
6739                }
6740            }
6741        finally
6742            {
6743            Var.popThreadBindings();
```

We'll recur on Compiler.eval() until we get back something that is not an sequence. As soon as we reach a leaf, we go down analysing it.

analyze()

```
{
if(form instanceof LazySeq)
    {
    form = RT.seq(form);
    if(form == null)
        form = PersistentList.EMPTY;
    }
if(form == null)
    return NIL_EXPR;
else if(form == Boolean.TRUE)
    return TRUE_EXPR;
else if(form == Boolean.FALSE)
        return FALSE_EXPR;
Class fclass = form.getClass();
if(fclass == Symbol.class)
    return analyzeSymbol((Symbol) form);
else if(fclass == Keyword.class)
    return registerKeyword((Keyword) form);
else if(form instanceof Number)
    return NumberExpr.parse((Number) form);
else if(fclass == String.class)
        return new StringExpr(((String) form).intern());

else if(form instanceof IPersistentCollection && ((IPersistentCollection) form).count() == 0)
        {
        Expr ret = new EmptyExpr(form);
        if(RT.meta(form) != null)
            ret = new MetaExpr(ret, MapExpr
                    .parse(context == C.EVAL ? context : C.EXPRESSION, ((IObj) form).meta()));
        return ret;
        }
else if(form instanceof ISeq)
        return analyzeSeq(context, (ISeq) form, name);
else if(form instanceof IPersistentVector)
        return VectorExpr.parse(context, (IPersistentVector) form);
else if(form instanceof IRecord)
        return new ConstantExpr(form);
else if(form instanceof IType)
        return new ConstantExpr(form);
else if(form instanceof IPersistentMap)
        return MapExpr.parse(context, (IPersistentMap) form);
else if(form instanceof IPersistentSet)
        return SetExpr.parse(context, (IPersistentSet) form);


return new ConstantExpr(form);
```

There are many analysing strategies. One of the most common is analyzeSeq, maybe worth exploring a little further.

```
6649    private static Expr analyzeSeq(C context, ISeq form, String name) {
6650        Object line = lineDeref();
6651        Object column = columnDeref();
6652        if(RT.meta(form) != null && RT.meta(form).containsKey(RT.LINE_KEY))
6653            line = RT.meta(form).valAt(RT.LINE_KEY);
6654        if(RT.meta(form) != null && RT.meta(form).containsKey(RT.COLUMN_KEY))
6655            column = RT.meta(form).valAt(RT.COLUMN_KEY);
6656        Var.pushThreadBindings(
6657                RT.map(LINE, line, COLUMN, column));
6658        try
6659            {
6660            Object me = macroexpand1(form);
6661            if(me != form)
6662                return analyze(context, me, name);
6663
6664            Object op = RT.first(form);
6665            if(op == null)
6666                throw new IllegalArgumentException("Can't call nil");
6667            IFn inline = isInline(op, RT.count(RT.next(form)));
6668            if(inline != null)
6669                return analyze(context, preserveTag(form, inline.applyTo(RT.n
6670            IParser p;
6671            if(op.equals(FN))
6672                return FnExpr.parse(context, form, name);
6673            else if((p = (IParser) specials.valAt(op)) != null)
6674                return p.parse(context, form);
6675            else
6676                return InvokeExpr.parse(context, form);
6677            }
6678        catch(Throwable e)
6679            {
6680            if(!(e instanceof CompilerException))
6681                throw new CompilerException((String) SOURCE_PATH.deref(), lin
6682            else
6683                throw (CompilerException) e;
6684            }
6685        finally
6686            {
6687            Var.popThreadBindings();
6688            }
6689    }
```

analyzeSeq()

Sequence analysis performs another round of macro expansion (possibly because it can be invoked from other part of the code) and then figure out what to do based on the first element of the form. If that is fn (quite a common case) we need to parse an FnExpr.

FnExpr parse

```
3902            try
3903                {
3904                fn.compile(fn.isVariadic() ? "clojure/lang/RestFn" : "clojure/lang/AFunction",
3905                           (prims.size() == 0)?
3906                           null
3907                           :prims.toArray(new String[prims.size()]),
3908                           fn.onceOnly);
3909                }
3910            catch(IOException e)
3911                {
3912                throw Util.sneakyThrow(e);
3913                }
3914            fn.getCompiledClass();
3915
3916            if(fn.supportsMeta())
3917                {
3918                //System.err.println(name + " supports meta");
3919                return new MetaExpr(fn, MapExpr
3920                        .parse(context == C.EVAL ? context : C.EXPRESSION, fmeta));
3921                }
3922            else
3923                return fn;
3924        }
```

FnExpr parsing is quite complicated stuff. Something worth noticing is that bytecode generation is triggered here. The call to getCompiledClass() is used for the side effect of having the freshly new generated class loaded in the class loader and ready to use.

```
4093
4094    void compile(String superName, String[] interfaceNames, boolean oneTimeUse) throws IOException{
4095        //...
4099        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
4100 //     ClassWriter cw = new ClassWriter(0);
4101        ClassVisitor cv = cw;
4102 //...
4104        cv.visit(V1_5, ACC_PUBLIC + ACC_SUPER + ACC_FINAL, internalName, null,superName,interfaceNames);
4105 //...
4107        String source = (String) SOURCE.deref();
4108        int lineBefore = (Integer) LINE_BEFORE.deref();
4109        int lineAfter = (Integer) LINE_AFTER.deref() + 1;
4110        int columnBefore = (Integer) COLUMN_BEFORE.deref();
4111        int columnAfter = (Integer) COLUMN_AFTER.deref() + 1;
4112
4113        if(source != null && SOURCE_PATH.deref() != null)
4114            {
4115            //cv.visitSource(source, null);
4116            String smap = "SMAP\n" +
4117                          ((source.lastIndexOf('.') > 0) ?
4118                           source.substring(0, source.lastIndexOf('.'))
4119                            :source)
4120                                  //                      : simpleName)
4121                          + ".java\n" +
4122                          "Clojure\n" +
4123                          "*S Clojure\n" +
4124                          "*F\n" +
4125                          "+ 1 " + source + "\n" +
4126                          (String) SOURCE_PATH.deref() + "\n" +
4127                          "*L\n" +
4128                          String.format("%d#1,%d:%d\n", lineBefore, lineAfter - lineBefore, lineBefore) +
4129                          "*E";
4130            cv.visitSource(source, smap);
4131            }
4132        addAnnotation(cv, classMeta);
4133        //static fields for constants
4134        for(int i = 0; i < constants.count(); i++)
4135            {
4136            cv.visitField(ACC_PUBLIC + ACC_FINAL
4137                          + ACC_STATIC, constantName(i), constantType(i).getDescriptor(),
4138                          null, null);
4139            }
```

ASM implements a visitor-based generator. Visiting order is important and each visited “context” must be closed when done (the structure of a class can be visualised like visiting a tree where indentation is the depth of the tree). Compilation proceeds top to bottom, emitting static context first (declaration, initialisation), constructors and then methods.

Bytecode generation needs to iterate over the different arities of the Clojure function and their closed overs, generating static variables and instance variables to hold references to other functions.

FnExpr.emitMethods()

```
3766    protected void emitMethods(ClassVisitor cv){
3767        //override of invoke/doInvoke for each method
3768        for(ISeq s = RT.seq(methods); s != null; s = s.next())
3769            {
3770            ObjMethod method = (ObjMethod) s.first();
3771            method.emit(this, cv);
3772            }
3773
3774        if(isVariadic())
3775            {
3776            GeneratorAdapter gen = new GeneratorAdapter(ACC_PUBLIC,
3777                                                        Method.getMethod("int getRequiredArity()"
3778                                                        null,
3779                                                        null,
3780                                                        cv);
3781            gen.visitCode();
3782            gen.push(variadicMethod.reqParms.count());
3783            gen.returnValue();
3784            gen.endMethod();
3785            }
3786    }
```

There is a lot more happening on ObjExpression.compile() before each arity method can be emitted. Static initialisation, constructors and initialisation of instance state.

emitMethods() manages emission of bytcode for each arity of the Clojure function. It just delegates to each ObjMethod instance and emits also the getRequiredArity() helper to return how many parameters are requested.

ObjMethod.emit()
FnMethod.doEmit()

```
5589    public void emit(ObjExpr fn, ClassVisitor cv){
5590        Method m = new Method(getMethodName(), getReturnType(), getArgTypes());
5591
5592        GeneratorAdapter gen = new GeneratorAdapter(ACC_PUBLIC,
5593                                                    m,
5594                                                    null,
5595                                                    //todo don't hardwire this
5596                                                    EXCEPTION_TYPES,
5597                                                    cv);
5598        gen.visitCode();
5599
5600        Label loopLabel = gen.mark();
5601        gen.visitLineNumber(line, loopLabel);
5602        try
5603            {
5604            Var.pushThreadBindings(RT.map(LOOP_LABEL, loopLabel, METHOD, this));
5605
5606            body.emit(C.RETURN, fn, gen);
5607            Label end = gen.mark();
5608            gen.visitLocalVariable("this", "Ljava/lang/Object;", null, loopLabel, end, 0);
5609            for(ISeq lbs = argLocals.seq(); lbs != null; lbs = lbs.next())
5610                {
5611                LocalBinding lb = (LocalBinding) lbs.first();
5612                gen.visitLocalVariable(lb.name, "Ljava/lang/Object;", null, loopLabel, end,
5613                }
5614            }
5615        finally
5616            {
5617            Var.popThreadBindings();
5618            }
```

Each ObjMethod contains a body instance that is requested to emit in turn. BodyExpr is basically a wrapper for an implicit do form which contains any number of instructions. Around the instructions ObjMethod takes care of assigning local variables.

BodyExpr.emit()

BodyExpr.emit() iterates over the collected expressions in turn (the body of a method being an implicit “do form” if none specified).

The list of potential implementors any of the Exprs classes (54).

The bottom! LocalBindingExpr.emit()

```
5757    public boolean canEmitPrimitive(){
5758        return b.getPrimitiveType() != null;
5759    }
5760
5761    public void emitUnboxed(C context, ObjExpr objx, GeneratorAdapter gen){
5762        objx.emitUnboxedLocal(gen, b);
5763    }
5764
5765    public void emit(C context, ObjExpr objx, GeneratorAdapter gen){
5766        if(context != C.STATEMENT)
5767            objx.emitLocal(gen, b, shouldClear);
5768    }
5769
5770    public Object evalAssign(Expr val){ throw new UnsupportedOperationException("Can't
5773
5774    public void emitAssign(C context, ObjExpr objx, GeneratorAdapter gen, Expr val){
5775        objx.emitAssignLocal(gen, b,val);
5776        if(context != C.STATEMENT)
5777            objx.emitLocal(gen, b, false);
5778    }
```

Finally we reached a leaf! the place where args (a local close over that was collected from the LispReader as a list of arguments) is taken and emitted. It is not actually returned here, it needs another specific ASM instruction to close the method.

Quick demo

We're just going to show everything together, evaluating the "lister" function and dumping the generated bytecode to the file system, so we can decompile it and have a look inside.

DynamicClassLoader

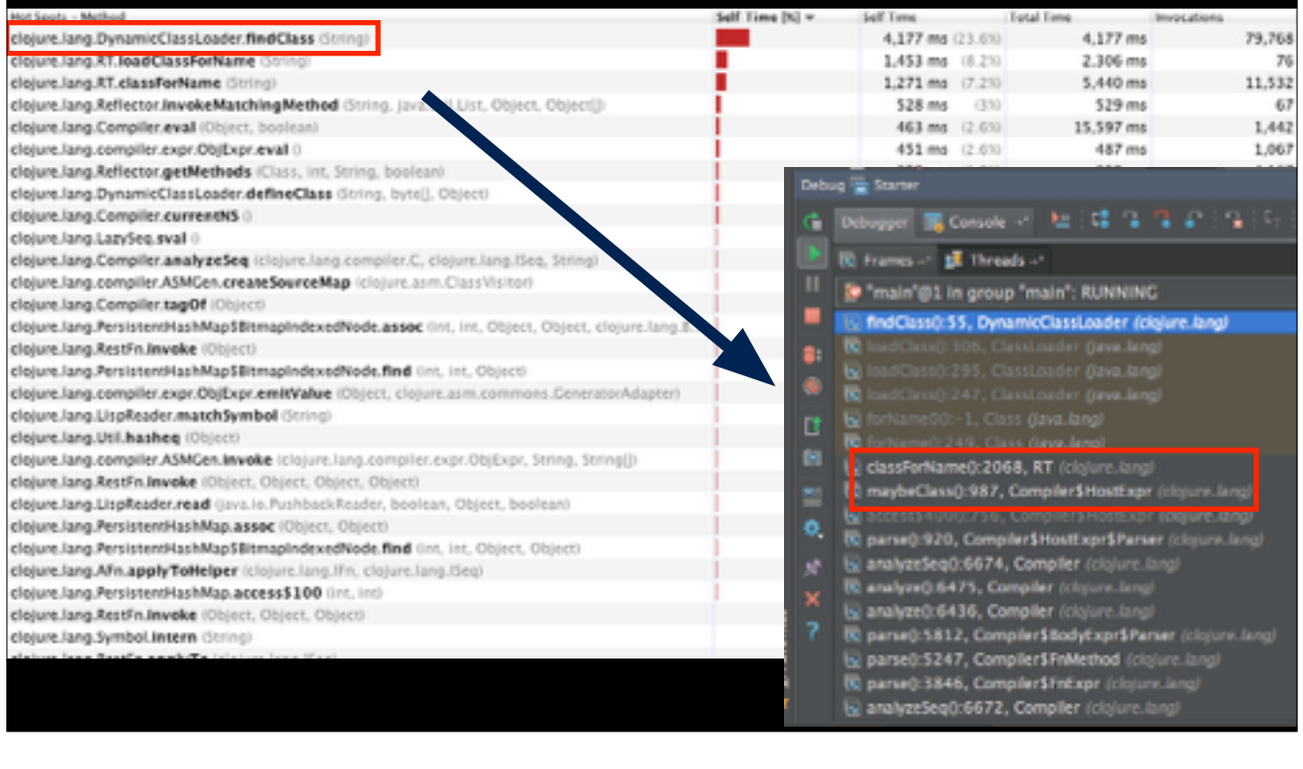
Stressing the compiler

Hot Spots - Method	Self Time [%]	Self Time	Total Time	Invocations
clojure.lang.Compiler$HostExpr.**maybeClass** (Object, boolean)		13,308 ms (43.1%)	13,349 ms	39,189
clojure.lang.Compiler.**eval** (Object, boolean)		2,648 ms (8.6%)	30,799 ms	5,418
clojure.lang.Compiler$ObjExpr.**eval** ()		1,980 ms (6.4%)	1,981 ms	4,200
clojure.lang.Compiler.**macroexpand1** (Object)		1,470 ms (4.8%)	6,874 ms	108,806
clojure.lang.Compiler$ObjExpr.**getCompiledClass** ()		1,313 ms (4.3%)	1,313 ms	10,979
clojure.lang.Compiler.**load** (java.io.Reader, String, String)		923 ms (3%)	30,845 ms	214
clojure.lang.Compiler.**currentNS** ()		795 ms (2.6%)	795 ms	296,779
clojure.lang.Compiler$ObjExpr.**compile** (String, String[], boolean)		771 ms (2.5%)	3,620 ms	6,779
clojure.lang.Compiler.**analyzeSeq** (clojure.lang.Compiler.C, clojure.lang.ISeq, String)		736 ms (2.4%)	23,952 ms	98,397
clojure.lang.Compiler.**tagOf** (Object)		547 ms (1.8%)	547 ms	236,593
clojure.lang.Compiler.**analyze** (clojure.lang.Compiler.C, Object, String)		347 ms (1.1%)	23,959 ms	210,337
clojure.lang.Compiler$FnExpr.**parse** (clojure.lang.Compiler.C, clojure.lang.ISeq, String)		310 ms (1%)	23,425 ms	6,616
clojure.lang.Compiler$ObjExpr.**emitValue** (Object, clojure.asm.commons.GeneratorAdapter)		278 ms (0.9%)	307 ms	55,315
clojure.lang.Compiler.**lookupVar** (clojure.lang.Symbol, boolean, boolean)		222 ms (0.7%)	879 ms	133,443
clojure.lang.Compiler$InstanceMethodExpr.**eval** ()		221 ms (0.7%)	221 ms	2
clojure.lang.Compiler$InvokeExpr.**eval** ()		193 ms (0.6%)	373 ms	151
clojure.lang.Compiler$HostExpr$Parser.**parse** (clojure.lang.Compiler.C, Object)		173 ms (0.6%)	7,966 ms	10,206
clojure.lang.Compiler$StaticMethodExpr.**<init>** (String, int, int, clojure.lang.Symbol, Class, String,...		158 ms (0.5%)	605 ms	5,981
clojure.lang.Compiler.**resolveIn** (clojure.lang.Namespace, clojure.lang.Symbol, boolean)		157 ms (0.5%)	214 ms	28,454
clojure.lang.Compiler$StaticMethodExpr.**eval** ()		155 ms (0.5%)	155 ms	442
clojure.lang.Compiler.**analyzeSymbol** (clojure.lang.Symbol)		140 ms (0.5%)	1,860 ms	63,312
clojure.lang.Compiler$InvokeExpr.**parse** (clojure.lang.Compiler.C, clojure.lang.ISeq)		136 ms (0.4%)	11,203 ms	27,132
clojure.lang.Compiler$InstanceMethodExpr.**<init>** (String, int, int, clojure.lang.Symbol, clojure.lan...		128 ms (0.4%)	206 ms	3,732
clojure.lang.Compiler$ObjExpr.**emitConstants** (clojure.asm.commons.GeneratorAdapter)		123 ms (0.4%)	477 ms	5,614
clojure.lang.Compiler.**registerLocal** (clojure.lang.Symbol, clojure.lang.Symbol, clojure.lang.Compi...		123 ms (0.4%)	644 ms	20,636
clojure.lang.Compiler$FnMethod.**parse** (clojure.lang.Compiler.ObjExpr, clojure.lang.ISeq, boolean)		120 ms (0.4%)	20,855 ms	6,923
clojure.lang.Compiler.**namespaceFor** (clojure.lang.Namespace, clojure.lang.Symbol)		105 ms (0.3%)	105 ms	116,151
clojure.lang.Compiler$ObjExpr.**constantType** (int)		104 ms (0.3%)	104 ms	142,887

Stressing the compiler isn't that difficult. Loading core.clj is probably all that it takes. Just in case, feed it a project with many dependencies and watch the compiler at work while AOT compiling the entire thing.

Some low hanging fruits here (as of 1.6): why that HostExpr.maybeClass() is taking way more than anything else? Little caching might be helpful.

clojure.lang.* at work



Same profiling, but the entire clojure.lang Java package. The DynamicClassLoader.findClass() is quite stressed during compilation. It turns out it is again HostExpr.maybeClass() invoking a RT.classForName, then Class.forName that triggers the findClass on the custom classloader.

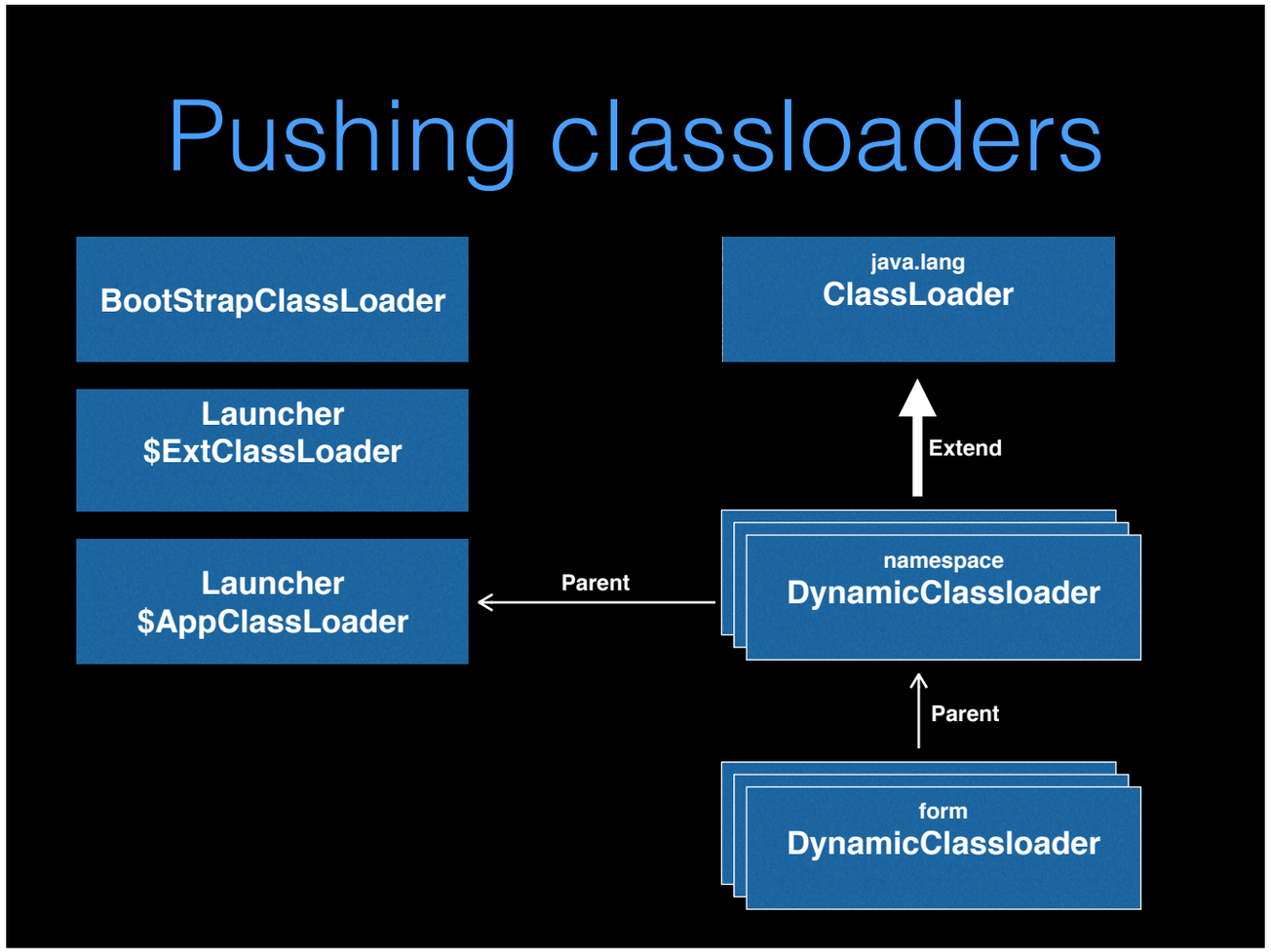
HostExpr is just one example of code that needs to lookup classes, nothing wrong with that. It just happens that is showing that searching and defining classes is a custom feature handled by the DynamicClassLoader.

HostExpr.maybeClass()

```
966     private static Class maybeClass(Object form, boolean stringOk) {
967         if(form instanceof Class)
968             return (Class) form;
969         Class c = null;
970         if(form instanceof Symbol)
971             {
972             Symbol sym = (Symbol) form;
973             if(sym.ns == null) //if ns-qualified can't be classname
974                 {
975                 if(Util.equals(sym,COMPILE_STUB_SYM.get()))
976                     return (Class) COMPILE_STUB_CLASS.get();
977                 if(sym.name.indexOf('.') > 0 || sym.name.charAt(0) == '[')
978                     c = RT.classForName(sym.name);
979                 else
980                     {
981                     Object o = currentNS().getMapping(sym);
982                     if(o instanceof Class)
983                         c = (Class) o;
984                     else
985                         {
986                         try{
987                         c = RT.classForName(sym.name);
988                         }
989                         catch(Exception e){
990                             //...
992                         }
993                         }
994                     }
995                 }
996             }
997         else if(stringOk && form instanceof String)
998             c = RT.classForName((String) form);
999         return c;
1000    }
```

Java interop is used extensively during Clojure bootstrapping. HostExpr parser is delegated to deal with the parsing of the Class or instance target for a method invocation. Maybe class is used to determine if the Symbol is indeed a class and transform it into a real one.

RT.classForName is invoking Class.forName() inside. This in turn mutates into a findClass() call for all the class loaders in the hierarchy. Consider the hundreds of them instantiated, the Compiler is spending quite an amount of time searching for them.



So, what's going on here? A little bit of Clojure class loaders organisation.

Every time a form is evaluated, at the REPL or during AOT compilation, a new DynamicClassLoader instance is created to hold the generated classes related to the bytecode compilation of that form. This form-classloader is child of another DynamicClassLoader instance that was created for the namespace the form is defined into. The namespace-classloader is then in turn child of the standard application classloader.

Once a namespace is required as the natural chain of dependencies during compilation it will be subject to the process of class generation for each function contained in them, **doesn't matter if they are used or not**.

This is what happens if I stick a couple of print lines to show every time a class loader is created and all the time a class loader instance is searched for a class implementation.

Luckily for us, this is only a bootstrap problem. None of that happens for our AOT compiled apps.

Supporting Dynamicity

```
public Class defineClass(String name, byte[] bytes, Object srcForm){
    Util.clearCache(rq, classCache);
    Class c = defineClass(name, bytes, 0, bytes.length);
    classCache.put(name, new SoftReference(c,rq));
    return c;
}

protected Class<?> findClass(String name) throws ClassNotFoundException{
    Reference<Class> cr = classCache.get(name);
    if(cr != null)
        {
        Class c = cr.get();
        if(c != null)
            return c;
        else
            classCache.remove(name, cr);
        }
    return super.findClass(name);
}
```

Clojure classloading is there to support the dynamicity of Clojure, the fact that functions or entire namespaces can be thrown away and redefined at anytime. The class loader that was generated for a namespace and all its form-classloaders is stored in a Var and when a reloaded is required all it takes to remove their only reference is just an “alter-var-root” away.

When the var root holding a reference to the class loader is gone, the class loader can be garbage collected and along with that all the references to classes that are unused.

Since some of the function generated during the namespace compilation might never be used, it makes sense to give the garbage collector a chance to collect them. That’s why they are wrapped in SoftReferences. If PermGen is close to be exhausted, the JVM can claim soft references and free up some space.

Resources

All snippets are from this SHA of clojure fastload branch

The Clojure mailing list *especially older posts*

The Clojure IRC channel logs

Interesting interview with Rich about Clojure history

An old but still useful class diagram

Very nice blog coverage about Lisp50

Presenter notes on these slides!

I so miss the time when Rich was discussing about everything Clojure on the #clojure IRC channel!

Bonus: Clojure in Lisp

Bonus: Lisp2Java

```
/**
 *   Copyright (c) Rich Hickey. All rights reserved.
 *   The use and distribution terms for this software are covered by the
 *   Eclipse Public License 1.0 (http://opensource.org/licenses/eclipse-1.0.php)
 *   which can be found in the file epl-v10.html at the root of this distribution.
 *   By using this software in any fashion, you are agreeing to be bound by
 *   the terms of this license.
 *   You must not remove this notice, or any other, from this software.
 **/

/* rich Aug 21, 2007 */

package clojure.lang;

//*

import clojure.asm.*;
import clojure.asm.commons.GeneratorAdapter;
import clojure.asm.commons.Method;

import java.io.*;
import java.lang.reflect.Constructor;
import java.lang.reflect.Modifier;
import java.util.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

//*/


//import org.objectweb.asm.*;
//import org.objectweb.asm.commons.Method;
//import org.objectweb.asm.commons.GeneratorAdapter;
//import org.objectweb.asm.util.TraceClassVisitor;
//import org.objectweb.asm.util.CheckClassAdapter;
//

public class Compiler implements Opcodes{

static final Symbol DEF = Symbol.intern("def");
static final Symbol LOOP = Symbol.intern("loop*");
static final Symbol RECUR = Symbol.intern("recur");
static final Symbol IF = Symbol.intern("if");
static final Symbol LET = Symbol.intern("let*");
static final Symbol LETFN = Symbol.intern("letfn*");
static final Symbol DO = Symbol.intern("do");
```

Bonus: Lisp2C#

```
/* Generated by Clojure */

namespace clojure.lib {

using System;

using org.clojure.runtime;

public class Clojure{
/* Generated by Clojure from the following Lisp:

(defn* apply ((fn & args+) (.applyTo fn __tld (spread* args+))))

*/

 public class apply : RestFn1{
override public Object doInvoke(ThreadLocalData __tld, Object fn, Cons argsPLUS__)
{
if(__tld == null) __tld = ThreadLocalData.get();
return ((IFn)ACC__clojure__applyTo).invoke(__tld, fn, clojure____tld.getValue(__tld), clojure__spreadSTA
}
}
/* Generated by Clojure from the following Lisp:

(defn* complement ((fn) (fn (& args) (not (apply fn args)))))

*/

 public class complement : AFn{
override public Object invoke(ThreadLocalData __tld, Object fn)
{
return (new FN__2163(fn));
}
 public class FN__2163 : RestFn0{
Object fn;
public FN__2163 (Object fn){
this.fn = fn;
}
override public Object doInvoke(ThreadLocalData __tld, Cons args)
{
return ((clojure__apply.fn.invoke(__tld, fn, args))==null?RT.T:null);
}
}
}
/* Generated by Clojure from the following Lisp:

(defn* constantly ((x) (fn (& args) x)))
```

Special thanks

The entire “Clojure Table” at the Mail Online

Phil Potter and the other EuroClojure speakers

http://mr--jack.deviantart.com incredible monsters

and finally Rich for the great learning opportunity

By the way, **WE ARE HIRING!**