**Java 调用**

**从Clojure 调用Java**

Clojure 是基于Java 不仅它是一种方便的，功能丰富的平台。 并且有数千的库，开源项目商业项目是用Java编写。Clojure可以利用这样现有的资源来解决过去困扰大多数新语言的“库问题”。

Clojure 并没有将处理通用事务比如IO操作，网络和数据库连接打包成库。虽然现存的Clojure库的数量正在迅速增长，他仍然是相当少的。幸运的是，任何通用事务处理既可以肯定一个Java库来帮助你。JVM本身带有超过4000个类涵盖网络到图形界面。Clojure的目的就是和Java进行无缝的工作。

**Java Interop 特殊形式**

Clojure 只需要3种特殊形式来和java类经行相互作用。用new这个特殊形式创建一个类的实例。

**(new *classname* & *constructor-arguments*)**

**new** 第一个参数是类名称 (这里不做讨论)。其次是类的构造函数。以下是一些示例。

**user> (new String)**

**""**

**user> (new java.util.Date)**

**#<Date Thu Oct 29 17:04:19 EDT 2009>**

**user> (new java.util.Date 55 10 12)**

**#<Date Sat Nov 12 00:00:00 EST 1955>**

**.**是调用Java的方法或者字段的特殊形式。

**(. *target name* & *arguments*)**

目标参数可以是一个类名或任意表达式。如果目标参数是一个类的名称，则名称应命名一个公共静态方法或一个符号这个类的领域。例如，下面的代码：

**user=> (. Integer valueOf "42")**

**user=> (. Integer MAX\_VALUE)**

**2147483647**

这些相当于Java代码：

**Integer.valueOf(42);**

**Integer.MAX\_VALUE;**

如果目标是一个类名，通常情况下，名称应该是一个名字公共实例方法或字段生成的对象。下面是一些例子：

**user=> (def s "Hello, World!")**

**#'user/s**

**user=> (. s substring 0 5)**

**"Hello"**

这2个表达式相当于Java代码：

**s.substring(0, 5);**

要设置公共的字段，可以用**set!** 这样的特殊形式：

**(set! (. target name) value)**

如上所示，目标是一类对象或者符号命名，名称命名的公共领域是一个符号类或对象，和表达式的值。这相当于Java代码：

***target.name* = *value;***

**new**，**.** 还有 **set!** 这3个特殊形式。他们不遵守同样的规则作为正常Clojure的函数和宏进行计算。特别是，名字参数从来没用进行计算，因为它不能在运行时确定。例如，你不能做到一下：

**;; bad code!**

**(defn call-method [object method-name]**

**(. object method-name))**

这将尝试调用的对象可能不是你所想“方法名”命名的方法。如果您确定需要运行时的方法名称，有两种方法来实现：Java Reflection API 和 Clojure的eval函数。前者是首选，请咨询反射API文档(1)。

**(1)http://java.sun.com/docs/books/tutorial/reflect/**

**Java Interop 首选形式**

虽然**new** 和 **.** 这特殊形式足够与Java相互操作，一些额外的语法可以帮助Java更好地适应Clojure的基于Lisp语法。

首先，Java方法调用可以看起来更像Clojure函数调用把方法的名称，在列表头部有一个特有的前缀。

**(.*method object arguments*)**

**.method** 的形式将处理由Clojure的编译器，就好像它是一个宏，扩展到：

**(. object method arguments)**

通过“就好像它是一个宏，”我的意思是，这个功能是一个纯粹的语法抽象或“语法糖”。这并不神奇地转化第一类函数(2)的Java方法，你不能使用***.method*** 作为函数参数的映射。相反，你必须包裹在Clojure的功能方法。

**user=> (map #(.toUpperCase %) ["one" "two" "three"])**

**("ONE" "TWO" "THREE")**

Clojure的宏**memfn**创建匿名函数之前语法**#()**需要存在。**memfn**需要一个符号，并扩展到一个匿名函数，调用该符号命名方法。匿名函数在前面例子可以写入**(memfn toUpperCase)**，但**#()**形式是首选。

可以通过类名称放置在列表头部构造Java类的新实例，后面跟一句：

**user=> (java.util.Date. 110 3 12)**

**#<Date Mon Apr 12 00:00:00 EDT 2010>**

**user=> (StringBuilder. "Hello")**

**#<StringBuilder Hello>**

你可以调用的语法**(ClassName/method arguments)**的静态方法和类名/字段。例如，下面的代码：

**user=> (Integer/parseInt "101")**

**101**

**user=> Integer/MIN\_VALUE**

**-2147483648**

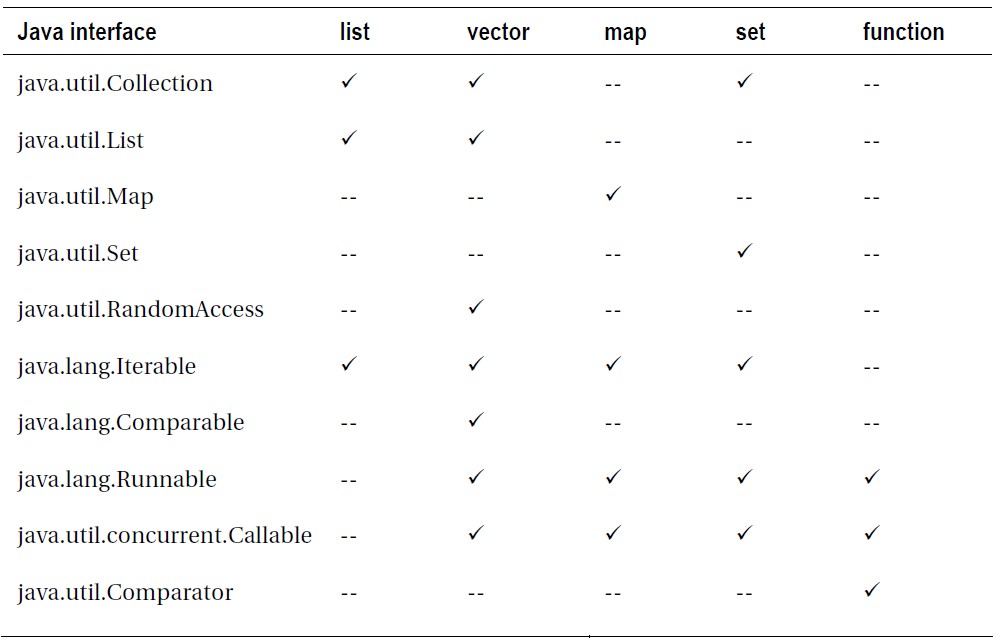
由于这些“语法糖”扩展发生在宏扩展的编译阶段，可能需要复杂的代码生成，以避免他们和使用新的宏。**.** 直接的特殊形式。在所有其他情况下，首选的“语法糖”形式。

**(2)作为第一类函数的功能方法名称已建议为未来的Clojure的版本。**

**Clojure 的类型和Java的接口**

作为一个平台，Java的优势之一就是泛型接口提供产检的数据类型，如List，Set。Clojure的数据结构实现这些接口在适当情况下，所以如果你需要调用一个Java的方法，例如，一个java.util.List，您可以通过它无需任何转换Clojure的数据结构。表10-1显示哪些接口内置在Clojure类型实施。

表10-1 标准的Java实现的接口和Clojure类型



要知道，Clojure的集合类型（List，Vector，Map和Set）仍然是一成不变的，所以他们实现java.util.Collection接口的只读部分。调用一个能产生变化的方法（如List.add或Map.put）一个不可改变的对象将抛出UnsupportedOperationException异常。

什么是Java泛型如List<String> 或者 Map<Integer, Object>？幸运的是，Clojure代码不需要担心，由于他们是在JVM中执行。泛型类型的Java字节码将被忽略，它们存在只能作为Java语言的编译器编译时(3)，Java类型List<String>的提示只是简单的名单。这意味着Clojure的是，你可以调用Java的方法泛型类型的基集合类型的一个实例（例如，List<String>）。只要集合中包含正确的类型（String）的对象，它只是工程。

**(3)相比之下，.NET公共语言运行时类型的泛型，更难以实现像Clojure的动态类型的语言。这是选择Java做Clojure主要平台的原因之一。**

**Java Arrays**

Java数据缺乏Clojure的集合类型的并发安全；它们是可变和非线程安全的。然后，一些Java API的使用数组作为方法的返回值，因此有必要他们一起工作。

此外，一些算法可以更有效地实现原始的集合，尤其是算法处理非常大的集合原始类型。在第14章将讨论提高原始集合的性能。

**创建Arrays**

你可以创建一个Java集合的功能：

**(make-array *type* & *dimensions*)**

类型参数必须是一个类。如果你想用一个Java的基本类型，如int或double，可以使用相应的类的类型字段：

**(make-array Double/TYPE 40) ;; creates a double[40] array**

如果你给的只是一个一维数组，你会得到一个该数组长度的Java数组，如果你给的是个多维的数组，你会得到一个多维数组，这是其他数组的指针在Java中实现。

此外，还有方便创建Java基本类型的函数：**int-array**，**long-array**，**float-array**和**double-array**。每个函数都有几个参数类型：

• **(int-array size)** 创建一个**size**大小的**int[]。**

• **(int-array size initial-value)** 创建一个**size**大小的**int[]**，并且设置每个元素初始值。

• **(int-array collection)** 创建与集合同样大小的数组，并且用集合填充。

• **(int-array size collection)** 创建一个**size**大小的**int[]**，并从集合中填充该，没有使用的元素初始化为0。

Clojure中有一个函数**to-array**可以采取任何Clojure集合的类型，并返回一个**Java Object[]**。如果你有一个二维数组表示的集合，你可以采用**to-array-2d**函数来产生一个二维数组，例如，下面的代码：

**user=> (def matrix [[1 0 0] [0 1 0] [0 0 1]])**

**#'user/matrix**

**user=> (to-array-2d matrix)**

**#<Object[][] [[Ljava.lang.Object;@540984b>**

如果你需要将转换成一个特定类型的数组集合，可以使用**info-array**函数：

**(into-array collection)**

**(into-array type collection)**

调用的时候需要传入一个集合参数，**info-array**函数返回一个相同类型的集合数组。当用两个参数调用的时候，第一个参数指定数组类型。例如，下面的代码：

**user=> (into-array Comparable ["aa" "bb" "cc"])**

**#<Comparable[] [Ljava.lang.Comparable;@a00185>**

**操作Arrays**

用**aget**函数可以从数组中获取一个单个的值（任意维的都行）。

**(aget *array* & *indices*)**

用**aset**函数可以设置任意对象数组中的任意元素：

**(aset array index value)**

**(aset array & indices value)**

这个**setter**函数设置集合的还有：**aset-boolean**，**aset-byte**，**aset-char**，**aset-short**，**aset-int**，**aset-long** 和 **aset-double**。注意，这些函数并不是很有效的操作，实际上他们比**aset**操作数组要慢（见第14章）。只有当你需要处理与Java互相操作的数组相对小的时候使用他们，而不是为了性能。

你可以用**aclone**函数来进行对数组复制，需要获取长度可以用**alenght**函数（虽然**count**也可以）。

**遍历Arrays**

Java 数组中有**map**和**reduce**函数对其操作。但是他们操作的时候会对数组进行转换，你可以直接用数组特定的遍历数组的宏来提高效率。

**(amap a *idx* ret *expr*)**

**amap**宏把**ret**（一个符号）初始化为数组**a**的一个克隆，然后从**idx**标识的数组下标处向后开始，循环执行**expr**。 **expr**执行后的返回值被赋值给**ret**的相同下标位置，最后，**amap**返回**ret**。

**(areduce *a idx ret init expr*)**

**areduce**宏将给**ret**(一个符号)赋值初始化。然后计算与**expr**与**idx**多次绑定的数组**a**的连续索引。由**expr**返回任何值，成为新的**ret**值，最后，**areduce**返回最后一个值的**ret**。

注意，**amap**和**areduce**宏是实现**loop/recur**，所以他们表达式作为参数，而不是**map**和**reduce**函数。

**从Java 调用Clojure**

Clojure代码可以生成真正的Java类，并可以像任何其他Java类调用方法。但是，如果你需要调用从Java中调用Clojure的一个函数，它可能是简单的Clojure的使用Java API，其中包括的静态方法类：clojure.lang.RT，clojure.lang.Compiler 和 clojure.lang.Var。

**Clojure代码的加载和执行**

clojure.lang.RT 是Clojure的“运行时”类。请记住，Clojure没有解释器，不能在单个JVM上有多个“实例”(4)，所以RT大多数方法是静态的。

**class RT {**

**...**

**public static void load(String *name*);**

**public static void loadResourceScript(String *filename*);**

**public static void maybeLoadResourceScript(String *filename*);**

**...**

**}**

**RT.load**方法就像Clojure的加载函数，第7章有描述，参数**name**是一个classpath上的一个文件名，去掉“.clj”或者“.class”扩展名。

**RT.loadResourceScript**方法类似加载。除了文件名，必须包含“.clj”的扩展名。RT.**maybeLoadResourceScript**方法是相同的，但是如果文件不存在是不会抛出异常的。

**class RT {**

**...**

**public static Object readString(String *code*);**

**...**

**}**

**class Compiler {**

**...**

**public static Object eval(Object *obj*);**

**...**

**}**

**RT.ReadString**方法等同与Clojure中的**read-string**函数。它需要一个Clojure源代码的字符串并返回该字符串表示的数据结构。**Compiler.eval**方法就像Clojure中的**eval**函数，执行数据并返回结果。

**使用Clojure的函数和变量**

**class RT {**

**...**

**public static Var var(String ns, String name);**

**public static Var var(String ns, String name, Object value);**

**...**

**}**

**RT.var**方法返回给定命名空间和名称的Clojure的变量，创建命名空间和需要的变量（见第七章）。可选择的第3个参数设置变量初始值或者根绑定。

一旦你有个变量对象，你可以用**Var.get**得到它的值，或者作为一个函数调用它**Var.invoke**：

**class Var {**

**...**

**public Object get();**

**public Object invoke(Object args...);**

**...**

**}**

**创建Java类**

通常情况下，Clojure的Java API不能足够的结合Clojure代码Java代码。许多Java库需要你实现一个特定的接口或者继承一个特定的类。幸运的是，Clojure可以创建真正的Java类，可以像任何其他的Java方法调用，而不需要编写任何“包装”在Java代码中。

**代理的Java类**

如果你需要实现一个Java接口或者继承一个基类为Java互操作目的，你应该首先看看代理宏，每次代理调用，它创建一个代理类，一个匿名类从你指定的基类（和/或者）接口的新实例。

**(*proxy* [*base-class-and-interfaces*...] [*constructor-args*...]**

**(*methodName* [*params*...] *method-body*...)**

**(*methodName* ...))**

proxy函数的第一个参数，是一个由类名和接口名组成的vector。最多可以有一个类，但是可以有多个接口（因为Java不允许多重继承）。如果没有指定父类，代理类将继承java.lang.Object。

第二个参数构造函数传入值，如果构造函数没有参数，vector是空的，但是它必须提供。

其余的参数，以代理列表的形式（**method [args] body**），其中**method**是一个公共（public）或者受保护（protected）的成员类。args是该方法的参数，和body是Clojure代码，你要使用实现的方法。实际上，你定义一个Clojure函数将被代理类的命名方法被调用的时候调用。

Multiple-arity方法（采取不同参数的方法）可能实现像multiple-arity的Clojure函数。

**(method ([arg] body...) ([arg1 arg2] body...))**

这听起来很复杂，但它真不是。让我们看一看实际的例子。Java的SAX类实现基于流的XML处理“push”接口。要使用它们，你必须提供一个实现了org.xml.sax.ContentHandler接口的实例。Clojure的XML库使用这个代理。下面是一个简单的例子，代理的ContentHandler打印出XML文档中的所有文本节点，每行一个。

**(import '(javax.xml.parsers SAXParserFactory)**

**'(org.xml.sax ContentHandler)**

**'(org.xml.sax.ext DefaultHandler2)**

**'(java.io File))**

**(defn proxy-handler []**

**(proxy [DefaultHandler2]**

**[] ;; DefaultHandler2 constructor takes no args**

**(characters [ch start length]**

**(println (String. ch start length)))))**

**(defn extract-text [filename]**

**(let [parser (.newSAXParser (SAXParserFactory/newInstance))]**

**(.parse parser (File. filename) (proxy-handler))))**

这个代理处理函数返回类org.xml.sax.ext.DefaultHandler2代理的一个实例，它提供所有无操作的实现所有org.xml.sax.ContentHandler的方法。代理类将覆盖字符串的方法，它接收一个char数据，并打印出该数组的字符串形式。提取文本函数创建一个使用SAXParserFactory类的SaxParser的新实例，然后调用parse方法与输入文件和代理处理程序，加载此代码后，你可以像这样运行：

**user=> (extract-text "path/to/some/file.xml")**

这将打印在XML文件中的所有文本。将会有很多空行，因为你的实现不会忽略文本元素中的空白。

代理方法可以访问他们被称为特殊的局部变量的对象。例如，要访问名为在当前对象的foo的公共实例的字段值，代理方法可以调用**(.foo this)**。

重要的是，要记住代理不是真正的子类。虽然代理可以覆盖受保护的方法，但是他们无法访问他们的父类的私有或者受保护的领域。他们无法提供自己的构造函数，不能添加未在父类或者接口中定义的新方法。代理实例enerated像

clojure.proxy.org.xml.sax.ext.DefaultHander2类的名称。

代理方法没有直接访问父类对象像Java中的super关键字。不过代理可以用宏**proxy-super**的方式来调用父类的方法。

**(proxy-super method & args)**

**method**是一个符号（未计算）命名的父类的方法，**args**是该方法的参数。代理父类中相应方法将呗调用当前对象（**this**）。

**生成Java类**

虽然代理通常有足够处理Java API的能力。有时会怎么做，而是一个真正的Java类。Clojure中有一个gen-class宏，它采用一系列的键值对作为参数，你可以这样创建Java类。

**(gen-class**

**:name generated-name**

**:extends base-class-name**

**:implements [interfaces ...]**

**:init initialization-function**

**:constructors {[types ...] [super-types ...], ...}**

**:post-init post-initialization-function**

**:methods [[name [types ...] return-type], ...]**

**:main boolean**

**:factory factory-name**

**:state state-field-name**

**:exposes {field {:get name, :set name}, ...}**

**:exposes-methods {method exposed, ...}**

**:prefix string**

**:impl-ns namespace**

**:load-impl-ns boolean)**

没办法，在**gen-class**有一堆的参数。幸运的是，他们除了**:name**其他的都是可选的。你很少需要超过他们几个。在你进入选项前看看**gen-class**如何与Java工作。

当你用**javac**编译Java源文件的时候，你会得到一个包含Java字节码的Java类文件。字节码中定义了类的字段和方法和它们的实现。当你运行**java**，Java虚拟机会加载类文件并执行它包含的字节码。

在Clojure中，相反，在运行时生成的字节码。你可以在Clojure的REPL类型中生成或者加载。.clj文件和Clojure将编译成Java字节码，然后Java虚拟机执行字节码。Clojure是当前所有的代码。而这是一个文件，当你想Java代码调用Clojure代码，因为Clojure函数可执行文件运行时字节码不存在。

可以想想，你写一个小的“包装”的Java类，其方法通过Clojure的API调用Clojure的函数，像这样：

**import clojure.lang.RT;**

**class MyWrapper {**

**public static Object doStuff() {**

**return RT.var("my-namespace", "do-stuff").invoke();**

**}**

**}**

然后你通过Java代码可以调用方法MyWrapper.doStuuff()，这个将调用Clojure函数my-namespace/do-stuff。

从本质上来讲，gen-class做同样的事情，而不许编写任何Java代码。它会产生一个Java的类文件，其中包含调用Clojure函数的“stub”方法。

由于gen-class需要生成一个类文件，将可能被用于其他静态编译的Java类，它不能运行时使用。相反，它必须调用一个单独编译步骤。Clojure的正常运行时编译的代码，将编译Clojure代码运行之前被称之为AOT编译。

**AOT编译**

任何Clojure的命名空间可以AOT编译，通常有什么理由这样做，除非**gen-class**参与。AOT编译Clojure和静态编译比速度。它仍然需要Clojure的运行时库（clojure.jar）。然而，AOT编译的代码将启动稍快一些，因为Clojure的编译器不需要编译所有的源文件时加载程序，这可能是有意的。

要编译一个命名空间，使用**compile**函数：

**(compile name)**

**name**参数是一个引用的符号命名的命名空间，你要编译。Clojure的将载入该命名空间中的源文件，使用文件名的命名空间名称转换为**require**相同的规则（见第七章）;编译为Java字节码，字节码写在目标目录中的类文件。一个命名空间，会产生许多。类文件，对每个功能之一。

AOT编译复杂的部分是正确配置classpath中。编译.class文件中写入的目标目录是存储在 **\*compile-path\***。当你调用**compile**，这个目录和源的.clj文件必须在Java classpath中可用。 **\*compile-path\***是“类”，假定当前工作目录内的目录。你可以改变的Java命令行上设置Java系统属性“clojure.compile.path”。

下面是一个例子。假设你有一个项目包含三个目录：在“sources”的源代码，在“target”的编译代码，并在“lib”库。您的Clojure代码文件中的“ource/com/example/my\_library.clj”，与下面的命名空间声明：

**(ns com.example.my-library)**

要编译这个命名空间，你就可以开始Clojure的像这样的项目的根目录（所有在同一行）：

**java -cp lib/clojure.jar:sources:target**

**-Dclojure.compile.path=target clojure.main**

请注意，classpath中包含三个要素：在Clojure的JAR文件，“sources”的目录，和“target”目录。（你会为您的项目使用的任何其他库添加JAR文件）。此外，系统属性clojure.compile.path设置为“目标”。 “目标”目录必须存在！然后，在Clojure的REPL，你可以运行：

**user=> (compile 'com.example.my-library)**

这将加载的源文件，从“source/com/example/my\_library.clj”，编译它，并写了一堆.class文件目录中的“target/com/example/”。

一旦做到这一点，你可以加载和使用命名空间**com.example.my-library**没有原始文件。所有你需要的类文件和clojure.jar。很明显，你不应该删除源文件，因为重新编译后你可能要改变他们。

为了使更容易整合构建脚本的AOT编译，你可以开始，而不是clojure.main代替clojure.lang.Compile，像以前那样设置classpath和系统属性，通过命名空间作为参数在命令行编译。在Apache Ant构建系统，例如，XML配置将包含类似下面的代码片段：

**<java classname="clojure.lang.Compile"**

**classpath="clojure.jar:target:source">**

**<sysproperty key="clojure.compile.path"**

**value="target"/>**

**<arg value="my.first.namespace"/>**

**<arg value="my.second.namespace"/>**

**</java>**

**gen-class**如何刚好适合？当编译是编译一个文件，调用**gen-class**，它会产生额外的.class配置选项所描述的文件。在其他任何时间，也就是说，当不使用AOT编译，**gen-clas**什么也不做。

**gen-class基本选项**

现在，你准备好**gen-class**的处理选项，你通常只需要前三个选项，**:name**，**:extends**和**:implements**，但我们这里把它们都覆盖。无论参数调用一个类或接口的名称，该名称可能是作为一个符号（将不会评估）或String，必须与Java包名完全合格的。

**:name**参数是要生成的类的名称。记住，这是一个Java风格的包+类名，所以你必须使用下划线或驼峰，而不是连字符。

**:extends**参数是一个字符串或一个符号的一个Java类（接口）的完全限定名称。生成的类将这个类的子类。

**:implements**参数是一个Java接口名称的载体。生成的类将被为实现这些接口，将包括在这些接口中定义的所有方法存根方法。

**定义生成的类的方法**

如前所述，由**gen-class**生成的类将只包含存根方法。这些方法的实现是一个命名空间中的正常Clojure的功能。每个Clojure的功能将其相应的方法相同的名称，添加前缀。**gen-class**与**:prefix**参数前缀默认为“ - ”，并可以改变的职能将被调用对象实例作为其第一个参数。例如，如果您生成的类实现一个Java接口的方法**doStuff(int i)**以和**doMoreStuff(String s)**，您的命名空间应包含下面的函数定义：

**(defn -doStuff [this i] ...)**

**(defn -doMoreStuff [this s] ...)**

**gen-class**默认情况下，使用当前的命名空间来查找方法的定义，这可以与**gen-class**的**:impl-ns**参数改变。

**生成的类中添加状态**

您可能要创建一个类，可以调用Java代码，但保留Clojure的不可改变的状态。**:state**参数的名字将被添加到生成的类的一个公共实例字段（类型为Object）。在你的方法，你可以进入这一字段，就像任何其他Java字段的值。请注意，声明**:state**字段是**final**，所以它可能无法设置以外的对象的构造。通常情况下，**:state**字段的值将Clojure的可变引用类型（Ref, Agent, or Atom）之一。通过这种方式，您可以创建状态的Java对象，采取Clojure的事务语义的优势。

如果你的对象有**:state**你必须提供一种方法来初始化它。 **:init**参数名“初始化函数”调用父类的构造前，作为构造相同的论点。初始化函数必须返回一个向量**[[args...] state]**，其中**状态**是**:state**字段的值和**args**参数将被传递回父类的构造。

为了做额外的计算后，父类的构造，**:post-init**参数名，将父类的构造后立即被调用的函数，与新构造的对象作为它的参数。**:post-init**函数的返回值将被忽略。

**生成的类中添加方法**

默认情况下，生成的类包含所有非私有方法的父类和接口的存根方法。如果你想加入到这一套方法，你可以做:methods选项为**gen-class**。它的参数是一个方法的签名，形成

**[name [arg-types...] return-type]**向量。然后，这些方法实施Clojure的功能与父类的方法一样名字前缀。要创建一个静态方法，添加**:static**真正的元数据。

例如，假设您想要添加到您的类下面的Java代码原型两种方法：

**public int add(int a, int b);**

**public static String getNextID();**

你会使用这样的**gen-class**。

**(gen-class ...**

**:methods [[add [int int] int]**

**#^{:static true} [getNextID [] String]])**

**...**

**(defn -add [this a b] ...)**

**(defn -getNextID [] ...)**

请记住，**:methods**是只添加不存在的方法使用

父类/接口。您不必重新定义现有的Java方法的原型。

**添加构造函数和工厂**

自动生成的类将公共构造与类型相匹配的父类的构造函数的签名。您可以用**gen-class**的**:constructors**选项添加其他构造，**:constructors**的参数是一个map的形式**{[types...] [super-types...], ...}**，map的键添加构造函数参数类型的载体，它必须映射到现有的超类的构造函数，其参数类型的vector确定。例如，如果你生成的类**:extends**选项是一个类**Foo**的构造**Foo(int, int)**，要添加一个单个**String**参数的构造函数，你可以这样做以下**gen-class**形式：

**(gen-class ...**

**:constructors {[String] [int int]} ...)**

你还必须提供一个**:init**函数，接受并返回相应的类型。

一些Java的开发方式，鼓励静态工厂方法，而不是公共构造。您可以添加与**:factory**选项你生成的类的静态工厂方法，**gen-class**它的参数是生成的工厂方法的名称;重载这个方法将接受所有相同的参数类型的构造函数。

**公开基类成员**

因为你的方法实现Clojure的函数，而不是真正的Java方法，他们不具有访问父类的保护领域，也不能调用父类的方法。为了解决这个问题，您可以添加用**gen-class**的**:exposes**和**:exposes-methods**选项。

**:exposes**需要map的形式**{field {:get getter, :set setter}, ...}**。每个键是超类的受保护的实例字段的名称，该值指定公共的getter和setter方法​​，将被添加到生成的类的名称。您不需要提供这些方法的实现，他们是自动生成的。

**:exposes-methods**需要一个map的形式**{super exposed, ...}**，其中**super**是一个超类方法的名称，**exposed**是一个公共方法，将被添加到生成的类的名称。公开的方法调用超级方法。您可以使用此功能时，例如，你一个方法需要实现以同样的方法调用超类版本。

**生成的命令行程序**

Java允许任何运行命令行可执行的类，它有一个方法，声明**public static void main(String[] args)**。您可以在**gen-class**指定**:main**您生成的类添加静态的主要方法。实施这一方法的功能应该被称为**-main**（除非你改变的前缀）。而不是单一的数组参数，它会被称为然而，许多参数是在命令行中。一个简单的方法来处理这个问题是定义的功能，采取可变数目的参数：

**(gen-class ...**

**:main true ...)**

**(defn -main [& args] ...)**

一旦你编译了一个含有**:main**方法的命名空间，你可以在这样的命令行执行：

**java -cp ... your.class.name arguments...**

请记住，你的编译类文件和Clojure的JAR文件必须在classpath中。

**加载实现**

默认情况下，用产生的任何类的**gen-class**将自动加载classpath中的第一次，它是用来执行命名空间，就像你有**require**命名空间。如果您使用的是一些替代的代码加载机制，你不想干涉生成的类，添加**gen-class**的**:load-impl-ns false**选项。

**gen-class的命名空间声明**

**gen-class**可以作为一个命名空间中声明的**ns**宏的一部分。在这种情况下，它写成**(:gen-class options. . .)**。在**ns**，**:name**和**:impl-ns**选项的默认命名空间被声明和**:main**默认值为**true**。一切是相同的。但是请记住，你不必限制自己的每一个生成类的命名空间，你可以生成几个类，不同的**:prefix**选项，并把所有的方法实现在同一个Clojure的命名空间中。

**简单的命令行程序**

如果你需要的是一个可以在命令行运行的程序，你只需要一个**-main**函数和**ns**声明（**:gen-class**），在这个例子中：

**(ns com.example.app**

**(:gen-class))**

**(defn -main [& args]**

**(println "Hello, World!"))**

当AOT编译到一个目录命名的类，这个例子可以运行命令：

**java -cp classes:clojure.jar com.example.app**

**摘要**

Clojure是无意取代Java语言。相反，它是设计来增强Java平台的能力，用不同的编程风格。Clojure的新手可能不喜欢到他们的Clojure代码Java类和方法名称的干涉，并急于包装Java方法调用每一个Clojure的功能。经验丰富的Clojure的程序员理解Java库提供电源和舒适的混合Java方法和Clojure功能。世界太大，一切从头开始实现。 Clojure在一个基于Java的环境得到了Java库的广阔的优势。