

钱进培训是哈法地区资深工程师组成的培训机构，通过各位老师的现身说法，帮助各位学员迅速掌握实战知识，为求职打下坚实的基础。电子邮件：[jin.qian.canada@gmail.com](mailto:jin.qian.canada@gmail.com) 钱老师报名、答疑微信号：qianjincanada，或扫描以下二维码添加：



钱老师 

加拿大



扫一扫上面的二维码图案，加我微信

 钱进老师



 钱进老师

# Java高级班（JavaEE方向）

## 讲义5

钱进培训立足Halifax地区，面向在校生提供软件开发技能培训和课程辅导。

本课是钱进培训组织的软件开发系列课程的主打课程，主要针对已经学习过CSCI1100（JAVA1）的同学，目标是通过大约10次课的学习，掌握JavaEE开发必备的技能，对现代软件企业的开发方式、常用类库、方法论等在校很难学到的知识点进行全面讲解，做到心中有数，提前具备求职的基本技术素质。

Exception

Thread

XML

JSON

MAVEN

异常的英文单词是exception，字面翻译就是“意外、例外”的意思，也就是非正常情况。事实上，异常本质上是程序上的错误，包括程序逻辑错误和系统错误。比如使用空的引用、数组下标越界、内存溢出错误等，这些都是意外的情况，背离我们程序本身的意图。错误在我们编写程序的过程中会经常发生，包括编译期间和运行期间的错误，在编译期间出现的错误有编译器帮助我们一起修正，然而运行期间的错误便不是编译器力所能及了，并且运行期间的错误往往是难以预料的。假若程序在运行期间出现了错误，如果置之不理，程序便会终止或直接导致系统崩溃，显然这不是我们希望看到的结果。因此，如何对运行期间出现的错误进行处理和补救呢？Java提供了异常机制来进行处理，通过异常机制来处理程序运行期间出现的错误。通过异常机制，我们可以更好地提升程序的健壮性。

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。例如，代码少了一个分号，那么运行出来结果是提示是错误 `java.lang.Error`；如果用 `System.out.println(11/0)`，那么你是因为你用0做了除数，会抛出 `java.lang.ArithmeticException` 的异常。

异常发生的原因有很多，通常包含以下几大类：

用户输入了非法数据。

要打开的文件不存在。

网络通信时连接中断，或者JVM内存溢出。

这些异常有的是因为用户错误引起，有的是程序错误引起的，还有其它一些是因为物理错误引起的。

要理解Java异常处理是如何工作的，你需要掌握以下三种类型的异常：

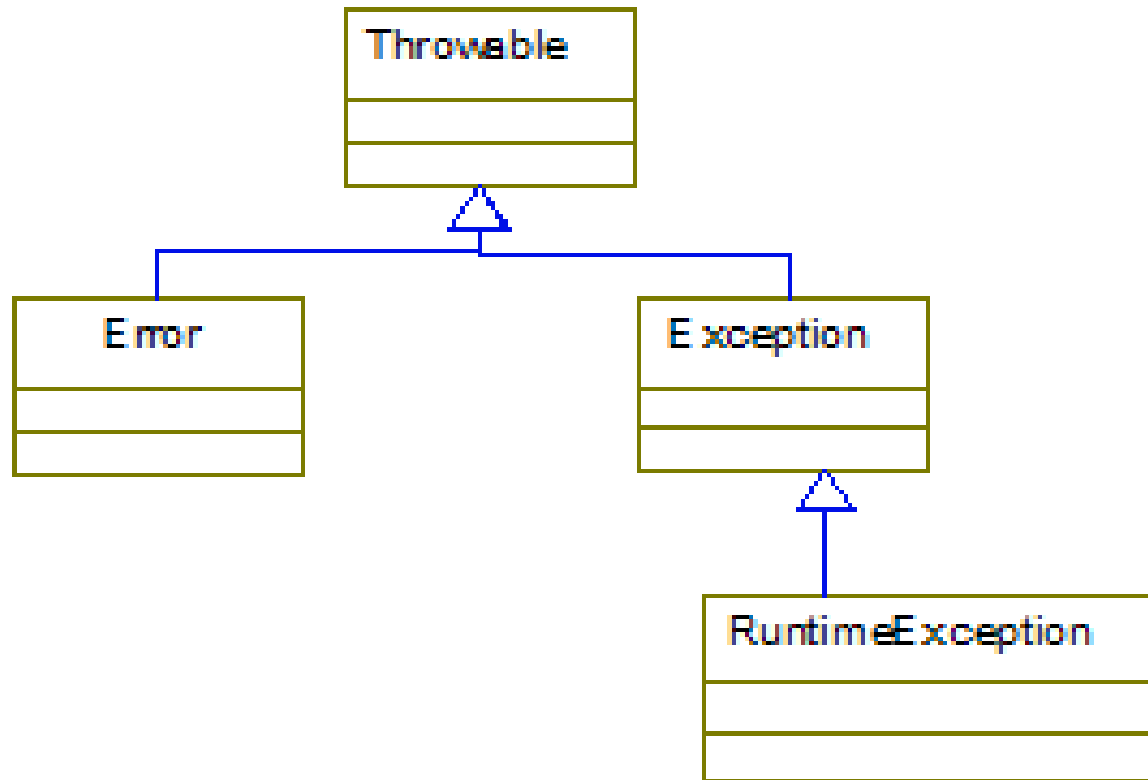
**检查性异常：**最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。

**运行时异常：**运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。

**错误：**错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

- 在讲Java异常实践之前，先理解一下什么是异常。到底什么才算是异常呢？其实异常可以看做在我们编程过程中遇到的一些意外情况，当出现这些意外情况时我们无法继续进程正常的逻辑处理，此时我们就可以抛出一个异常。
- 广义的讲，抛出异常分三种不同的情况：
- **编程错误导致的异常**：在这个类别里，异常的出现是由于代码的错误（譬如 `NullPointerException`、`IllegalArgumentException`、`IndexOutOfBoundsException`）。代码通常对编程错误没有什么对策，所以它一般是非检查异常。
- **客户端的错误导致的异常**：客户端代码试图违背制定的规则，调用API不支持的资源。如果在异常中显示有效信息的话，客户端可以采取其他的补救方法。例如：解析一个格式不正确的XML文档时会抛出异常，异常中含有有效的信息。客户端可以利用这个有效信息来采取恢复的步骤。
- **资源错误导致的异常**：当获取资源错误时引发的异常。例如，系统内存不足，或者网络连接失败。客户端对于资源错误的反应是视情况而定的。客户端可能一段时间之后重试或者仅仅记录失败然后将程序挂起。

在 Java 程序设计语言中，使用一种异常处理的错误捕获机制。当程序运行过程中发生一些异常情况，程序有可能被中断、或导致错误的结果出现。在这种情况下，程序不会返回任何值，而是抛出封装了错误信息的对象。Java 语言提供了专门的异常处理机制去处理这些异常。如图 1 所示为 Java 异常体系结构：



从上面异常继承树可以看出，所以异常都继承自Throwable，这也意味着所有异常都是可以抛出的。

具体来说，广义的异常可以分为Error和Exception两大类。

Error表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java虚拟机运行错误Virtual MachineError，当 JVM 不再有继续执行操作所需的内存资源时，将出现OutOfMemoryError，虚拟机错误还有StackOverflowError、InternalError、UnknownError等。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。经常见到的Error还有LinkageError（结合错误），具体有NoSuchMethodError、IllegalAccessError、NoClassDefFoundError。

所以，对于Error我们编程中基本是用不到的，也就是说我们在编程中可以忽略Error错误。所以我们通常所说的异常只的是Exception，而Exception可分为检查异常和非检查异常。



## 检查异常与非检查异常

通常我们所说的异常指的都是`Exception`的子类，它们具体可以分为两大类，`Exception`的子类和`RuntimeException`的子类，它们分别对应着检查异常和非检查异常。

### Checked exception

检查异常，继承自`Exception`类。对于检查异常，Java强制我们必须进行处理。对于抛出检查异常的API我们有两种处理方式：

对抛出检查异常的API进程try catch

继续把检查异常往上抛

常见的检查异常有：

`SQLException`

`IOException`

`InterruptedException`

## Unchecked exception

非检查异常，也称运行时异常`RuntimeException`，继承自`RuntimeException`，所有非检查都有个特点，就是代码不需要处理它们的异常也能通过编译，所以它们称作unchecked exception。`RuntimeException`本身也是继承自`Exception`。

常见的非检查异常有：

`NullPointerException`

`IllegalArgumentException`

`NumberFormatException`

`IndexOutOfBoundsException`

`IllegalStateException`

## 检查 (Checked) 异常与非检查 (Unchecked) 异常

Java 语言规范将派生于 `Error` 类或 `RuntimeException` 类的所有异常都称为非检查异常。除非检查异常以外的所有异常都称为检查异常。检查异常对方法调用者来说属于必须处理的异常，当一个应用系统定义了大量或者容易产生很多检查异常的方法调用，程序中会有很多的异常处理代码。

如果一个异常是致命的且不可恢复并且对于捕获该异常的方法根本不知如何处理时，或者捕获这类异常无任何益处，笔者认为应该定义这类异常为非检查异常，由顶层专门的异常处理程序处理；像数据库连接错误、网络连接错误或者文件打不开等之类的异常一般均属于非检查异常。这类异常一般与外部环境相关，一旦出现，基本无法有效地处理。

而对于一些具备可以回避异常或预料内可以恢复并存在相应的处理方法的异常，可以定义该类异常为检查异常。像一般由输入不合法数据引起的异常或者与业务相关的一些异常，基本上属于检查异常。当出现这类异常，一般可以经过有效处理或通过重试可以恢复正常状态。

由于检查异常属于必须处理的异常，在存在大量的检查异常的程序中，意味着很多的异常处理代码。另外，检查异常也导致破坏接口方法。如果一个接口上的某个方法已被多处使用，当为这个方法添加一个检查异常时，导致所有调用此方法的代码都需要修改处理该异常。当然，存在合适数量的检查异常，无疑是比较优雅的，有助于避免许多潜在的错误。

到底何时使用检查异常，何时使用非检查异常，并没有一个绝对的标准，需要依具体情况而定。很多情况，在我们的程序中需要将检查异常包装成非检查异常抛给顶层程序统一处理；而有些情况，需要将非检查异常包装成检查异常统一抛出。

在Java中如果需要处理异常，必须先对异常进行捕获，然后再对异常情况进行处理。如何对可能发生异常的代码进行异常捕获和处理呢？使用try和catch关键字即可，如下面一段代码所示：

```
try {  
    File file = new File("d:/a.txt");  
    if(!file.exists())  
        file.createNewFile();  
} catch (IOException e) {  
    // TODO: handle exception  
}
```

被**try**块包围的代码说明这段代码可能会发生异常，一旦发生异常，异常便会被**catch**捕获到，然后需要在**catch**块中进行异常处理。

这是一种处理异常的方式。在**Java**中还提供了另一种异常处理方式即抛出异常，顾名思义，也就是说一旦发生异常，我把这个异常抛出去，让调用者去进行处理，自己不进行具体的处理，此时需要用到**throw**和**throws**关键字。

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            createFile();  
        } catch (Exception e) {  
            // TODO: handle exception  
        }  
    }  
  
    public static void createFile() throws IOException{  
        File file = new File("d:/a.txt");  
        if(!file.exists())  
            file.createNewFile();  
    }  
}
```

这段代码和上面一段代码的区别是，在实际的`createFile`方法中并没有捕获异常，而是用`throws`关键字声明抛出异常，即告知这个方法的调用者此方法可能会抛出`IOException`。那么在`main`方法中调用`createFile`方法的时候，采用`try...catch`块进行了异常捕获处理。

当然还可以采用`throw`关键字手动来抛出异常对象。下面看一个例子：



```
public class Main {  
    public static void main(String[] args) {  
        try {  
            int[] data = new int[]{1,2,3};  
            System.out.println(getDataByIndex(-1,data));  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
  
    public static int getDataByIndex(int index,int[] data) {  
        if(index<0 || index>=data.length){  
            throw new ArrayIndexOutOfBoundsException("数组下标越界");  
            System.out.println("fatal errors!");  
        }  
        return data[index];  
    }  
}
```

也就说在Java中进行异常处理的话，对于可能会发生异常的代码，可以选择三种方法来进行异常处理：

1) 对代码块用**try..catch**进行异常捕获处理；

2) 在 该代码的方法体外用**throws**进行抛出声明，告知此方法的调用者这段代码可能会出现这些异常，你需要谨慎处理。此时有两种情况：

如果声明抛出的异常是非运行时异常，此方法的调用者必须显示地用**try..catch**块进行捕获或者继续向上层抛出异常。

如果声明抛出的异常是运行时异常，此方法的调用者可以选择地进行异常捕获处理。

3) 在代码块用**throw**手动抛出一个异常对象，此时也有两种情况，跟2) 中的类似：

如果抛出的异常对象是非运行时异常，此方法的调用者必须显示地用**try..catch**块进行捕获或者继续向上层抛出异常。

如果抛出的异常对象是运行时异常，此方法的调用者可以选择地进行异常捕获处理。

- .try,catch,finally
- try关键字用来包围可能会出现异常的逻辑代码，它单独无法使用，必须配合catch或者finally使用。Java编译器允许的组合使用形式只有以下三种形式：
- try...catch...; try....finally.....; try....catch...finally...
- 当然catch块可以有多个，注意try块只能有一个,finally块是可选的（但是最多只能有一个finally块）。
- 三个块执行的顺序为try—>catch—>finally。
- 当然如果没有发生异常，则catch块不会执行。但是finally块无论在什么情况下都是会执行的（这点要非常注意，因此部分情况下，都会将释放资源的操作放在finally块中进行）。
- 在有多个catch块的时候，是按照catch块的先后顺序进行匹配的，一旦异常类型被一个catch块匹配，则不会与后面的catch块进行匹配。

- **throws**和**throw**关键字
- 1) **throws**出现在方法的声明中，表示该方法可能会抛出的异常，然后交给上层调用它的方法程序处理，允许**throws**后面跟着多个异常类型；
- 2) 一般会用于程序出现某种逻辑时程序员主动抛出某种特定类型的异常。**throw**只会出现在方法体中，当方法在执行过程中遇到异常情况时，将异常信息封装为异常对象，然后**throw**出去。**throw**关键字的一个非常重要的作用就是 异常类型的转换。
- **throws**表示出现异常的一种可能性，并不一定会发生这些异常；**throw**则是抛出了异常，执行**throw**则一定抛出了某种异常对象。两者都是消极处理异常的方式（这里的消极并不是说这种方式不好），只是抛出或者可能抛出异常，但是不会由方法去处理异常，真正的处理异常由此方法的上层调用处理。

- **Java 内置异常类**
- Java 语言定义了一些异常类在 `java.lang` 标准包中。
- 标准运行时异常类的子类是最常见的异常类。由于 `java.lang` 包是默认加载到所有的 Java 程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。
- Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

`ClassCastException`            当试图将对象强制转换为不是实例的子类时，抛出该异常。

`IndexOutOfBoundsException`        指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。

`NullPointerException`            当应用程序试图在需要对象的地方使用 `null` 时，抛出该异常

Java 定义在 `java.lang` 包中的检查性异常类。

`ClassNotFoundException` 应用程序试图加载类时，找不到相应的类，抛出该异常。

`NoSuchMethodException` 请求的方法不存在

下面的列表是 `Throwable` 类的主要方法:

序号	方法及说明
----	-------

1	<code>public String getMessage()</code>
---	---

返回关于发生的异常的详细信息。这个消息在`Throwable` 类的构造函数中初始化了。

2	<code>public Throwable getCause()</code>
---	--

返回一个`Throwable` 对象代表异常原因。

3	<code>public String toString()</code>
---	---------------------------------------

使用`getMessage()`的结果返回类的串级名字。

4	<code>public void printStackTrace()</code>
---	--

打印`toString()`结果和栈层次到`System.err` , 即错误输出流。

5	<code>public StackTraceElement [] getStackTrace()</code>
---	--

返回一个包含堆栈层次的数组。下标为0的元素代表栈顶, 最后一个元素代表方法调用堆栈的栈底。

6	<code>public Throwable fillInStackTrace()</code>
---	--

用当前的调用栈层次填充`Throwable` 对象栈层次, 添加到栈层次任何先前信息中。

多线程



现代操作系统比如Mac OS X，UNIX，Linux，Windows等，都是支持“多任务”的操作系统。

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒.....这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

对于操作系统来说，一个任务就是一个进程（**Process**），比如打开一个浏览器就是启动一个浏览器进程，打开一个记事本就启动了一个记事本进程，打开两个记事本就启动了两个记事本进程，打开一个**Word**就启动了一个**Word**进程。

有些进程还不止同时干一件事，比如**Word**，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（**Thread**）。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，像**Word**这种复杂的进程可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。当然，真正地同时执行多线程需要多核**CPU**才可能实现。

**CPU+RAM+各种资源**（比如显卡，光驱，键盘，GPS, 等等外设）构成我们的电脑，但是电脑的运行，实际就是**CPU**和相关寄存器以及**RAM**之间的事情。一个最最基础的事实：**CPU**太快，太快，太快了，寄存器仅仅能够追的上他的脚步，**RAM**和别的挂在各总线上的设备完全是望其项背。那当多个任务要执行的时候怎么办呢？轮流着来？或者谁优先级高谁来？不管怎么样的策略，一句话就是在**CPU**看来就是轮流着来。一个必须知道的事实：执行一段程序代码，实现一个功能的过程介绍，当得到**CPU**的时候，相关的资源必须也已经就位，就是显卡啊，GPS啊什么的必须就位，然后**CPU**开始执行。这里除了**CPU**以外所有的就构成了这个程序的执行环境，也就是我们所定义的程序上下文。当这个程序执行完了，或者分配给他的**CPU**执行时间用完了，那它就要被切换出去，等待下一次**CPU**的临幸。在被切换出去的最后一步工作就是保存程序上下文，因为这个是下次他被**CPU**临幸的运行环境，必须保存。串联起来的事实：前面讲过在**CPU**看来所有的任务都是一个一个的轮流执行的，具体的轮流方法就是：先加载程序**A**的上下文，然后开始执行**A**，保存程序**A**的上下文，调入下一个要执行的程序**B**的程序上下文，然后开始执行**B**,保存程序**B**的上下文。。。。

## 进程（Processes）和线程（Threads）

进程和线程是并发编程的两个基本的执行单元。在 **Java** 中，并发编程主要涉及线程。

一个计算机系统通常有许多活动的进程和线程。在给定的时间内，每个处理器只能有一个线程得到真正的运行。对于单核处理器来说，处理时间是通过时间切片来在进程和线程之间进行共享的。

现在多核处理器或多进程的电脑系统越来越流行。这大大增强了系统的进程和线程的并发执行能力。但即便是没有多处理器或多进程的系统，并发仍然是可能的。

## 进程

进程有一个独立的执行环境。进程通常有一个完整的、私人的基本运行时资源;特别是,每个进程都有其自己的内存空间。

进程往往被视为等同于程序或应用程序。然而,用户将看到一个单独的应用程序可能实际上是一组合作的进程。大多数操作系统都支持进程间通信( **Inter Process Communication**, 简称 **IPC**)资源,如管道和套接字。**IPC** 不仅用于同个系统的进程之间的通信,也可以用在不同系统的进程。

大多数 **Java** 虚拟机的实现作为一个进程运行。

## 线程

线程有时被称为轻量级进程。进程和线程都提供一个执行环境,但创建一个新的线程比创建一个新的进程需要更少的资源。

线程中存在于进程中,每个进程都至少一个线程。线程共享进程的资源,包括内存和打开的文件。这使得工作变得高效,但也存在了一个潜在的问题——通信。

多线程执行是 **Java** 平台的一个重要特点。每个应用程序都至少有一个线程,或者几个,如果算上“系统”的线程(负责内存管理和信号处理)那就更多。但从程序员的角度来看,你启动只有一个线程,称为主线程。这个线程有能力创建额外的线程。

Java 给多线程编程提供了内置的支持。一个多线程程序包含两个或多个能并发运行的部分。

在java中要想实现多线程，有两种手段，一种是继承Thread类，另外一种是实现Runnable接口。(其实准确来讲，应该有三种，还有一种是实现Callable接口，并与Future、线程池结合使用)

Java 线程类也是一个 object 类，它的实例都继承自 `java.lang.Thread` 或其子类。可以用如下方式用 java 中创建一个线程：

```
Tread thread = new Thread();
```

执行该线程可以调用该线程的 `start()` 方法：

```
thread.start();
```

在上面的例子中，我们并没有为线程编写运行代码，因此调用该方法后线程就终止了。

编写线程运行时执行的代码有两种方式：一种是创建 `Thread` 子类的一个实例并重写 `run` 方法，第二种是创建类的时候实现 `Runnable` 接口



## java.lang.Thread 类

Thread 类是一个具体的类，即不是抽象类，该类封装了线程的行为。要创建一个线程，程序员必须创建一个从 Thread 类导出的新类。程序员必须覆盖 Thread 的 run() 函数来完成有用的工作。用户并不直接调用此函数；而是必须调用 Thread 的 start() 函数，该函数再调用 run()。

继承Thread类的方法是比较常用的一种，如果说你只是想起一条线程。没有什么其它特殊的要求，那么可以使用Thread.

对于直接继承Thread的类来说，代码大致框架是：

```
class 类名 extends Thread{  
    方法1;  
    方法2;  
    ...  
    public void run(){  
        // other code...  
    }  
    属性1;  
    属性2;  
    ...  
}
```

```
class hello extends Thread {

    public hello() {

    }

    public hello(String name) {
        this.name = name;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(name + "运行  " + i);
        }
    }

    public static void main(String[] args) {
        hello h1=new hello("A");
        hello h2=new hello("B");
        h1.start();
        h2.start();
    }

    private String name;
}
```

因为需要用到CPU的资源，所以每次的运行结果基本是都不一样的

## Runnable 接口

此接口只有一个函数，run()，此函数必须由实现了此接口的类实现。

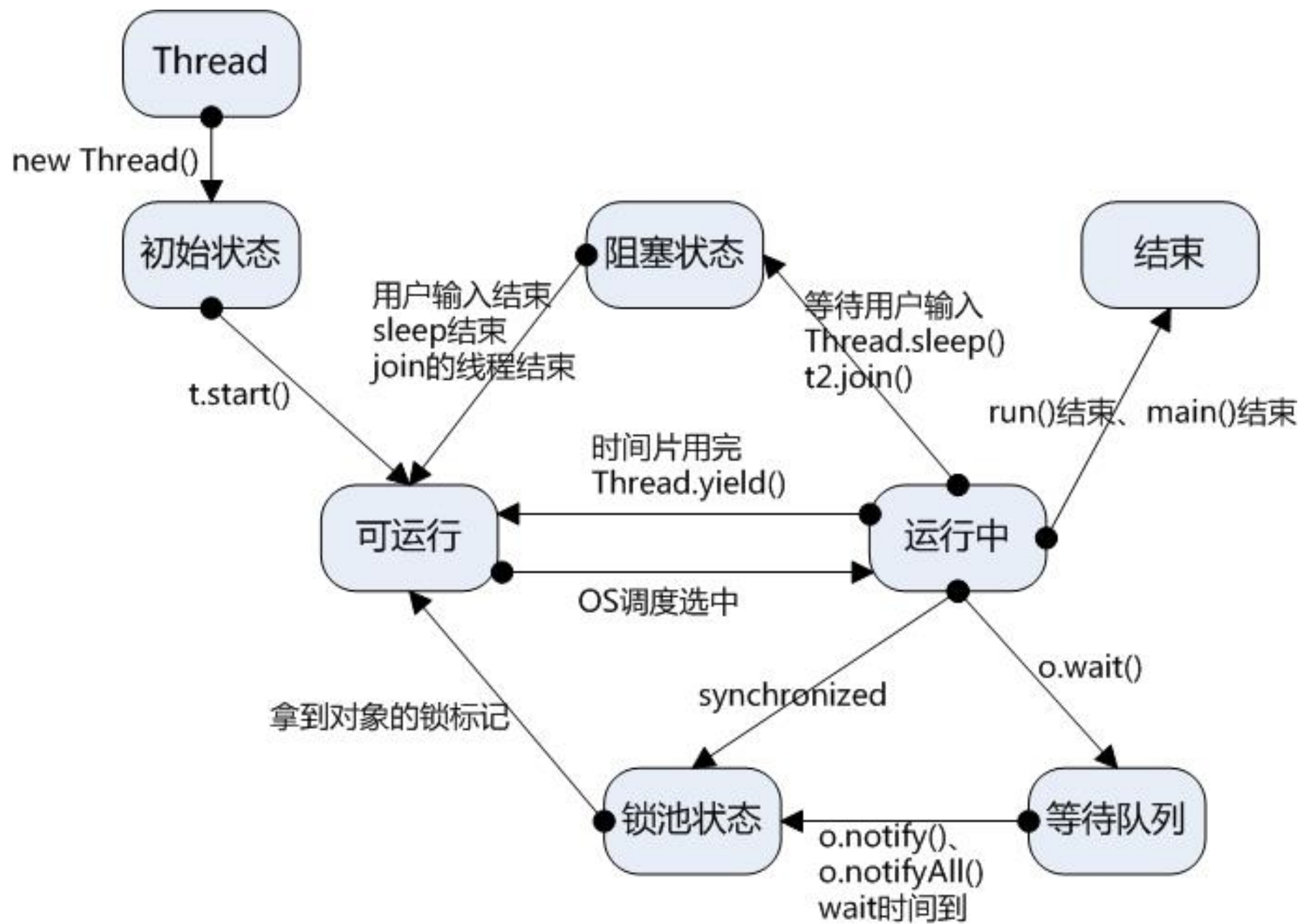
```
class TimePrinter implements Runnable {
    int pauseTime;
    String name;
    public TimePrinter(int x, String n) {
        pauseTime = x;
        name = n;
    }
    public void run() {
        while(true) {
            try {
                System.out.println(name + ":" + new Date(System.currentTimeMillis()));
                Thread.sleep(pauseTime);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
}
```

```
static public void main(String args[]) {  
    Thread t1 = new Thread (new TimePrinter(1000, "Fast Guy"));  
    t1.start();  
    Thread t2 = new Thread (new TimePrinter(3000, "Slow Guy"));  
    t2.start();  
  
    }  
}
```

实现**Runnable**接口，使得该类有了多线程类的特征。**run（）**方法是多线程程序的一个约定。所有的多线程代码都在**run**方法里面。**Thread**类实际上也是实现了**Runnable**接口的类。

在启动的多线程的时候，需要先通过**Thread**类的构造方法**Thread(Runnable target)** 构造出对象，然后调用**Thread**对象的**start()**方法来运行多线程代码。

实际上所有的多线程代码都是通过运行**Thread**的**start()**方法来运行的。因此，不管是扩展**Thread**类还是实现**Runnable**接口来实现多线程，最终还是通过**Thread**的对象的**API**来控制线程的



## 创建 Thread 的子类

创建 Thread 子类的一个实例并重写 run 方法，run 方法会在调用 start()方法之后被执行。例子如下：

```
public class MyThread extends Thread {  
    public void run(){  
        System.out.println("MyThread running");  
    }  
}
```

可以用如下方式创建并运行上述 Thread 子类

```
MyThread myThread = new MyThread();  
myThread.start();
```

一旦线程启动后 start 方法就会立即返回，而不会等待到 run 方法执行完毕才返回。就好像 run 方法是在另外一个 cpu 上执行一样。当 run 方法执行后，将会打印出字符串 MyThread running。

```
public class MyRunnable implements Runnable {  
    public void run(){  
        System.out.println("MyRunnable running");  
    }  
}
```

为了使线程能够执行 `run()` 方法，需要在 `Thread` 类的构造函数中传入 `MyRunnable` 的实例对象。示例如下：

```
Thread thread = new Thread(new MyRunnable());  
thread.start();
```

当线程运行时，它将会调用实现了 `Runnable` 接口的 `run` 方法。上例中将会打印出 “`MyRunnable running`”。



在同一程序中运行多个线程本身不会导致问题，问题在于多个线程访问了相同的资源。如，同一内存区（变量，数组，或对象）、系统（数据库，**web services** 等）或文件。实际上，这些问题只有在一或多个线程向这些资源做了写操作时才有可能发生，只要资源没有发生变化,多个线程读取相同的资源就是安全的。

多线程同时执行下面的代码可能会出错：

```
public class Counter {  
    protected long count = 0;  
    public void add(long value){  
        this.count = this.count + value;  
    }  
}
```

想象下线程 A 和 B 同时执行同一个 Counter 对象的 add()方法，我们无法知道操作系统何时会在两个线程之间切换。JVM 并不是将这段代码视为单条指令来执行的，而是按照下面的顺序：

从内存获取 this.count 的值放到寄存器

将寄存器中的值增加 value

将寄存器中的值写回内存

观察线程 A 和 B 交错执行会发生什么：

this.count = 0;

A: 读取 this.count 到一个寄存器 (0)

B: 读取 this.count 到一个寄存器 (0)

B: 将寄存器的值加 2

B: 回写寄存器值(2)到内存. this.count 现在等于 2

A: 将寄存器的值加 3

A: 回写寄存器值(3)到内存. this.count 现在等于 3

两个线程分别加了 2 和 3 到 `count` 变量上，两个线程执行结束后 `count` 变量的值应该等于 5。然而由于两个线程是交叉执行的，两个线程从内存中读出的初始值都是 0。然后各自加了 2 和 3，并分别写回内存。最终的值并不是期望的 5，而是最后写回内存的那个线程的值，上面例子中最后写回内存的是线程 A，但实际中也可能是线程 B。如果没有采用合适的同步机制，线程间的交叉执行情况就无法预料。

## 竞态条件 & 临界区

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。导致竞态条件发生的代码区称作临界区。上例中 `add()` 方法就是一个临界区,它会产生竞态条件。在临界区中使用适当的同步就可以避免竞态条件。

## Java 同步关键字（synchronized）

Java 中的同步块用 **synchronized** 标记。同步块在 Java 中是同步在某个对象上。所有同步在一个对象上的同步块在同时只能被一个线程进入并执行操作。所有其他等待进入该同步块的线程将被阻塞，直到执行该同步块中的线程退出。

下面是一个同步的实例方法：

```
public synchronized void add(int value){  
    this.count += value;  
}
```

注意在方法声明中同步（**synchronized**）关键字。这告诉 Java 该方法是同步的。

Java 实例方法同步是同步在拥有该方法的对象上。这样，每个实例其方法同步都同步在不同的对象上，即该方法所属的实例。只有一个线程能够在实例方法同步块中运行。如果有多个实例存在，那么一个线程一次可以在一个实例同步块中执行操作。一个实例一个线程。

Xml

XML 指可扩展标记语言（eXtensible Markup Language）。

XML 被设计用来传输和存储数据。

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<note>
```

```
  <to>Tove</to>
```

```
  <from>Jani</from>
```

```
  <heading>Reminder</heading>
```

```
  <body>Don't forget me this weekend!</body>
```

```
</note>
```

上面的这条便签具有自我描述性。它包含了发送者和接受者的信息，同时拥有标题以及消息主体。

但是，这个 XML 文档仍然没有做任何事情。它仅仅是包装在 XML 标签中的纯粹的信息。我们需要编写软件或者程序，才能传送、接收和显示出这个文档。

XML 指可扩展标记语言（EXtensible Markup Language）。

XML 是一种很像HTML的标记语言。

XML 的设计宗旨是传输数据，而不是显示数据。

XML 标签没有被预定义。您需要自行定义标签。

XML 被设计为具有自我描述性。

XML 是 W3C 的推荐标准。

**XML** 仅仅是纯文本

**XML** 没什么特别的。它仅仅是纯文本而已。有能力处理纯文本的软件都可以处理 **XML**。

不过，能够读懂 **XML** 的应用程序可以有针对性地处理 **XML** 的标签。标签的功能性意义依赖于应用程序的特性。

**XML** 简化数据共享

在真实的世界中，计算机系统和数据使用不兼容的格式来存储数据。

**XML** 数据以纯文本格式进行存储，因此提供了一种独立于软件和硬件的数据存储方法。

这让创建不同应用程序可以共享的数据变得更加容易。



## XML 的语法规则

### XML 文档必须有根元素

<root>

  <child>

    <subchild>.....</subchild>

  </child>

</root>

XML 声明文件的可选部分，如果存在需要放在文档的第一行，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
```

所有的 XML 元素都必须有一个关闭标签

在 HTML 中，某些元素不必有一个关闭标签：

`<p>This is a paragraph.`

`<br>`

在 XML 中，省略关闭标签是非法的。所有元素都必须有关闭标签：

`<p>This is a paragraph.</p>`

`<br />`

XML 标签对大小写敏感。标签 <Letter> 与标签 <letter> 是不同的。

必须使用相同的大小写来编写打开标签和关闭标签：

```
<Message>This is incorrect</message>
```

```
<message>This is correct</message>
```

在 HTML 中，常会看到没有正确嵌套的元素：

```
<b><i>This text is bold and italic</b></i>
```

在 XML 中，所有元素都必须彼此正确地嵌套：

```
<b><i>This text is bold and italic</i></b>
```

在上面的实例中，正确嵌套的意思是：由于 `<i>` 元素是在 `<b>` 元素内打开的，那么它必须在 `<b>` 元素内关闭。

与 HTML 类似，XML 元素也可拥有属性（名称/值的对）。

在 XML 中，XML 的属性值必须加引号。

请研究下面的两个 XML 文档。第一个是错误的，第二个是正确的：

```
<note date=12/11/2007>
```

```
<to>Tove</to>
```

```
<to>Tove</to>
```

```
<from>Jani</from>
```

```
</note>
```

```
<note date="12/11/2007">
```

```
<to>Tove</to>
```

```
<from>Jani</from>
```

```
</note>
```

在第一个文档中的错误是，note 元素中的 date 属性没有加引号。

在 XML 中，一些字符拥有特殊的意义。

如果您把字符 "<" 放在 XML 元素中，会发生错误，这是因为解析器会把它当作新元素的开始。

这样会产生 XML 错误：

```
<message>if salary < 1000 then</message>
```

为了避免这个错误，请用实体引用来代替 "<" 字符：

```
<message>if salary &lt; 1000 then</message>
```

在 XML 中，有 5 个预定义的实体引用：

&lt;	<	less than
------	---	-----------

&gt;	>	greater than
------	---	--------------

&amp;	&	ampersand
-------	---	-----------

&apos;	'	apostrophe
--------	---	------------

&quot;	"	quotation mark
--------	---	----------------

在 XML 中编写注释的语法与 HTML 的语法很相似。

```
<!-- This is a comment -->
```



Json

JSON 或者 JavaScript 对象表示法是一种轻量级的基于文本的开放标准，被设计用于可读的数据交换。约定使用 JSON 的程序包括 C , C++ , Java , Python , Perl 等等。

JSON 是 JavaScript Object Notation 的缩写。

这个格式由 Douglas Crockford 提出。

被设计用于可读的数据交换。

它是从 JavaScript 脚本语言中演变而来。

文件名扩展是 .json。

JSON 的网络媒体类型是 application/json。

## SON 特点

JSON 容易阅读和编写。

它是一种轻量级的基于文本的交换格式。

语言无关。

```
{  
  "book": [  
    {  
      "id": "01",  
      "language": "Java",  
      "edition": "third",  
      "author": "Herbert Schildt"  
    },  
    {  
      "id": "07",  
      "language": "C++",  
      "edition": "second",  
      "author": "E.Balagurusamy"  
    }  
  ]  
}
```

## JSON 的基本语法

数据使用名/值对表示。

使用大括号保存对象，每个名称后面跟着一个 ':'（冒号），名/值对使用 ,（逗号）分割。

使用方括号保存数组，数组值使用 ,（逗号）分割。

JSON 格式支持以下数据类型：

类型	描述
----	----

数字型 (Number)	JavaScript 中的双精度浮点型格式
--------------	-----------------------

字符串型 (String)	双引号包裹的 Unicode 字符和反斜杠转义字符
---------------	---------------------------

布尔型 (Boolean)	true 或 false
---------------	--------------

数组 (Array)	有序的值序列
------------	--------

值 (Value)	可以是字符串，数字，true 或 false , null 等等
-----------	----------------------------------

对象 (Object)	无序的键:值对集合
-------------	-----------

空格 (Whitespace)	可用于任意符号对之间
-----------------	------------

null	空
------	---

Maven

在进行软件开发的过程中，无论什么项目，采用何种技术，使用何种编程语言，我们都要重复相同的开发步骤：编码，测试，打包，发布，文档。实际上这些步骤是完全重复性的工作。那为什么让软件开发人员去重复这些工作？开发人员的主要任务应该是关注商业逻辑并去实现它，而不是把时间浪费在学习如何在不同的环境中去打包，发布，。。。

**Maven**正是为了将开发人员从这些任务中解脱出来而诞生的。**Apache Maven** 是一种用作软件项目管理和理解工具。它基于项目对象模型（**POM**）的概念，可以管理一个项目的构建、报告以及从项目核心信息中生成文档。

**Maven**能够：

1) 理解并管理整个软件开发周期，重用标准的构建过程，比如：编译，测试，打包等。同时**Maven**还可以通过相应的元数据，重用构建逻辑到一个项目。

2) **Maven**负责整个项目的构建过程。开发人员只需要描述项目基本信息在一个配置文件中：`pom.xml`。也就是说，**Maven**的使用者只需要回答“**What**”而不是“**How**”。



## Maven设计原则

1) **Convention Over Configuration** (约定优于配置)。在现实生活中，有很多常识性的东西，地球人都知道。比如说：如何过马路(红灯停绿灯行)，如何开门，关门等。对于这些事情，人们已经有了默认的约定。

在软件开发过程中，道理也是类似的，如果我们事先约定好所有项目的目录结构，标准开发过程（编译，测试，。。。），所有人都遵循这个约定。软件项目的管理就会变得简单很多。在现在流行的很多框架中，都使用了这个概念，比如EJB3和 Ruby on Rails。在Maven中默认的目录结构如下：

- NumberOperations
  - src
    - main
      - java
        - net
          - lanfeng
            - tutorials
    - test
      - java
        - net
          - lanfeng
            - tutorials
    - target
      - classes
        - net
          - lanfeng
            - tutorials
        - maven-archiver
        - surefire-reports
      - test-classes

可以看出以下几个标准的Maven目录：

**src：**源代码目录。所有的源代码都被放在了这个目录下。在这个目录下又包括了：

1) **main：**所有的源代码放在这里。对于Java项目，还有一个下级子目录：**java**。对于Flex项目则是**flex**，。。。。

2) **test：**所有的单元测试类放在这里。

**target：**所有编译过的类文件以及生成的打包文件(.jar, .war, ...)放在这里。

2) **Reuse Build Logic (重用构建逻辑)：**Maven把构建逻辑封装到插件中来达到重用的目的。这样在Maven就有用于编译的插件，单元测试的插件，打包的插件，。。。Maven可以被理解成管理这些插件的框架。

3) **Declarative Execution (声明式执行)：**Maven中所有的插件都是通过**POM**中声明来定义的。Maven会理解所有在**POM**中的声明，并执行相应的插件。

src/main/java - 存放项目.java文件;

src/main/resources - 存放项目资源文件;

src/test/java - 存放测试类.java文件;

src/test/resources - 存放测试资源文件;

target - 项目输出目录;

pom.xml - Maven核心文件 ( Project Object Model ) ;

下载Maven : <http://maven.apache.org/>

2) 解压缩下载的zip文件到本地目录下, 比如: D:\Maven

3) 添加D:\Maven\bin到环境变量PATH中

4) 在命令行下运行:

`mvn -version` 或者 `mvn -v`

一些Maven命令

编译: `mvn compile`

单元测试: `mvn test`

构建并打包: `mvn package`

清理: `mvn clean`

安装 `mvn clean install`

maven 项目会有一个 pom.xml 文件，在这个文件里面，只要你添加相应配置，他就会自动帮你下载相应 jar 包，不用你铺天盖地的到处搜索你需要的 jar 包了。

```
<dependency>
```

```
  <groupId>junit</groupId> 项目名
```

```
  <artifactId>junit</artifactId> 项目模块
```

```
  <version>3.8.1</version> 项目版本
```

```
  <scope>test</scope>
```

```
</dependency>
```

maven都会通过，项目名-项目模块-项目版本来maven在互联网上的代码库中下载相应jar包。

所有的POM文件要求有project节点和三个必须字段：groupId, artifactId, version。

在仓库中项目的标识为groupId:artifactId:version。

POM.xml的根节点是project并且其下有三个主要的子节点：

节点	描述
----	----

groupId	项目组织的Id。通常在一个项目或者一个组织之中，这个Id是唯一的。例如，某个Id为com.company.bank的银行组织包含所有银行相关的项目。
---------	--

artifactId	项目的Id，通常是项目的名字。例如，consumer-banking。artifactId与groupId一起定义了仓库中项目构件的路径。
------------	---

version	项目的版本。它与groupId一起，在项目构件仓库中用作区分不同的版本，例如：
---------	---

com.company.bank:consumer-banking:1.0

com.company.bank:consumer-banking:1.1.

Eclipse 提供了一个极佳的插件 m2eclipse，可以无缝地把 Maven 和 Eclipse 集成在一起。

m2eclipse 的一些特性列出如下：

你可以从 Eclipse 中运行 Maven 目标操作。

你可以使用 Eclipse 自身的控制台查看 Maven 命令的输出。

你可以使用 IDE 更新 Maven 依赖。

你可以从 Eclipse 中启动 Maven 构建。

为基于 Maven pom.xml 文件的 Eclipse 构建路径做依赖管理。

解决来自 Eclipse 工作空间的 Maven 依赖，而无需安装依赖到本地 Maven 仓库中（需要依赖的项目在同一个工作空间中）。

自动从 Maven 远程仓库中下载所需依赖及源码。


提供了向导，可供创建 Maven 新项目和 pom.xml 文件以及为已存在的项目开启 Maven 支持。

提供了对 Maven 远程仓库中依赖的快速搜索。



钱进培训是哈法地区资深工程师组成的培训机构，通过各位老师的现身说法，帮助各位学员迅速掌握实战知识，为求职打下坚实的基础。电子邮件：[jin.qian.canada@gmail.com](mailto:jinqian.canada@gmail.com) 钱老师报名、答疑微信号：[qianjincanada](https://www.wechat.com/p/qianjincanada)，或扫描以下二维码添加：



钱老师   
加拿大



扫一扫上面的二维码图案，加我微信

 钱进老师



 钱进老师

