目录

简介

这是公司内部一次技术分享活动的讲稿，主要介绍了 Eclipse Vert.x 以及以它为代表的异步编程模型。

0. 开场白

各位同事大家好，今天我分享的题目是 *Vert.x* 与异步无阻塞编程模型初探 。

我刚入职那会儿曾经自学了一段时间 *Nodejs*，它适用于开发常见的 *IO*密集型 的 Web 应用，在处理高并发场景时有优秀的性能。我用 *Nodejs* 帮朋友做过一个网站，在实现一些稍微有些复杂的业务流程时，十分不习惯 *Nodejs* 异步回调的编程方式，写了一些虽然能用但是可读性以及可维护性都非常差的代码，我甚至看不懂网上提供的解决办法，后来学习 *Nodejs* 这件事就不了了之了。

在做消息推送的技术预研时，我们又发现了 *Vert.x* ，一个诞生于2011年最初命名为 *Node.x* 的“框架”（从这个名字大家大概能猜得出，它跟 *Nodejs* 有很多相似之处）。综合考虑性能、业务场景、开发运维成本等各方面因素的考虑，我们最终选择了它。于是，我又入了异步无阻塞编程模型的坑，进行了一些初步的探索，这也是我今天要跟大家分享的主要内容。

我主要从以下三个方面进行分享：

> - 什么是 *Vert.x* ?
> - 为什么要用 *Vert.x* ?
> - 怎么使用 *Vert.x* ?

我觉得枯燥的名词定义和原理阐述并不能引起大家的兴趣，只有当我们真正去编码，去调试运行时，产生了疑惑，才会真正对那些高大上的名词和背后高效运行的原理感兴趣。所以，最后一条会重点讲，大家听完就可以直接用 *Vert.x* 进行开发；前面两条我会简单带过，只在最后留下一些参考资料，如果你对这些感兴趣，可以自行查阅。

1. 什么是 *Vert.x*？

按照官网给的定义，**Eclipse Vert.x** 是一个在 **Java** 虚拟机 上用于构建 响应式 应用的 工具包 。

> Eclipse Vert.x is a tool-kit for building reactive applications on the JVM.

按照 “响应式宣言（*The Reactive Manifesto*）”（一份倡议性的文件） 里的定义，“响应式系统“是具有灵敏性、有回复性、可伸缩性、消息驱动等特点的计算机系统；而这里称之为工具包（而不是“框架”），是在强调 *Vert.x* 是轻量而高度灵活的。下面是它官网上宣称的一些特点，我只进行简单的搬运和翻译：

> 1. 事件驱动，无阻塞。这意味着（为什么?）你的应用可以用少量的内核线程处理大量并发，节省了硬件资源；
> 2. *Vert.x* 支持多语言，包括 *Java*、*JavaScript*、*Groovy*、*Ruby*、*Ceylon*、*Scala* 和 *Kotlin* 。官方并不会给你安利 什么是最好的语言 ，而是希望使用者根据项目需要以及团队技术储备进行选择；
> 3. 通用性强。开发简单的网络应用、复杂的 Web 应用、HTTP/RESTful微服务、大量的事件处理或后台消息服务，*Vert.x* 都是很合适的选择；
> 4. 不自以为是（unopinionated）。*Vert.x* 不是一个严格定义的框架或者容器。官方不会告诉你编写一个应用的正确姿势，只给你一些有用的组件、文档和例子；
> 5. *Vert.x* 很有趣，简约而不简单...blahblah...

总结一下，*Vert.x* 是一个 高效的、轻量的、灵活的、支持多语言的、通用性强的工具包，如图（误）：

2. 为什么要用 *Vert.x* ?

这一段的内容其实还是搬运，不过不是搬运自 *Vert.x* 官网，而是它的一个同类框架 *Akka* 的官网文档，主要有两篇文章：

> 1. Why modern systems need a new programming model
> 2. How the Actor Model Meets the Needs of Modern, Distributed Systems

第一篇文章提出了一些当代计算机系统在开发多线程应用时的问题 ，第二篇文章主要介绍 参与者模型 （Actor Model，也就是 *Vert.x* 和 *Akka* 实现的模型）是 如何解决这些问题 的。我给大家总结下要点，有兴趣的同事可以自己仔细阅读一下。

第一篇首先指出了传统 *Java* 面向对象编程中的通过使用锁来解决多线程资源问题的缺点：

> 1. 锁 在现代 CPU 架构中是开销比较大的，严重限制了高并发；
> 2. 调用线程可能会被 阻塞 ，存在无法工作的情况，对于服务器端开发尤其浪费资源，开更多的线程仍然是一个消耗；
> 3. 锁可能导致 死锁 。

然后介绍了关于共享内存的几点问题：

> 1. 并不存在真正意义的共享内存，为了在 CPU 的多个核心共享数据（比如被声明为 `volatile` 的变量），它们会把大量 CPU 缓存数据传来传去；
> 2. 相比使用共享内存或原子化（atomic）的数据结构，在并发实体中维护自身的状态以及显式地通过消息来传递数据可能是更好的方法。（对 *Golang* 比较了解的同事应该知道，这有点像 *Golang* 所宣称的某种哲学：不要通过共享内存来通信，而应该通过通信来共享内存。）

最后，总结了现代计算机为了实现高并发与高性能，使用调用后台线程处理任务时存在的两个问题：

> 1. 基于调用栈的异常处理不再适用，应当引入显式的异常信号机制；
> 2. 调用者线程应当及时地获得被调线程的处理结果。

以上就是第一篇文章提出的所有问题。

然后，在第二篇文章中，作者提出了通过传递消息来避免 锁 与 阻塞以及“优雅地”处理错误的方法，这里不进行详细的介绍。在介绍如何使用 *Vert.x* 的时候，我会在适当的时候对具体原理进行解释。

关于 “为什么要用 *Vert.x*” 这个问题，除了技术本身效率、功能上的一些原因，还要充分考虑项目需求，开发、运维成本等等要素，请大家结合实际需要，谨慎选择，这里不再赘述。

3. 怎么使用 *Vert.x* ?

下面开始介绍重点——怎么使用 *Vert.x* ， 主要包括六个部分：

> - Java 8 Lambda表达式
> - Hello, Vert.x与异步无阻塞API
> - 黄金法则
> - 事件总线与集群化
> - 回调地狱与Future对象
> - 自定义异步方法

3.1 *Java 8* 的 *Lambda*表达式

*Vert.x*从版本3（当前最新*Vert.x*版本为3.5.0）开始就需要使用*Java 8*,并且大量使用了*Lambda*表达式这个*Java 8*的新特性。所以在真正介绍怎么使用*Vert.x*之前，我们需要先了解一下*Java 8*中的*Lambda*表达式。

*Lambda*表达式在很多语言中都有，其本质就是匿名函数：

JavaScript：

```
function (a, b) {
    return a + b;
};
```

Python：

```
lambda a,b: a + b
```

Lua：

```
function (a, b)
    return a + b
end
```

C++：

```
[](int a, int b) {
    return a + b;
};
```

C#：

```
(a, b) => a + b;
```

*Lambda*表达式 在 *Python*、*JavaScript*、*Lua* 等弱类型语言中可以直接作为一个普通变量，在 *C++* 中可以作为特定类型的函数指针（使用auto作为指针变量的类型即可自动推断类型），在 *C#* 中可以作为一个特定类型的委托变量，而在 *Java 8* 中，*Lambda*表达式 实际上就是一个 匿名内部类。

什么是 匿名内部类 呢？在过去版本的 *Java* 中，如果我们想像传递变量那样传递一个 函数 （或者叫 方法 ），通常是先定义一个 接口 ，像这样：

```
public interface MyFunction {
    int exec(int a, int b);
}
```

然后，我们再实现这个接口，像这样：

```
MyFunction sum = new MyFunction () {
    @Override
    public int exec(int a, int b) {
        return a + b;
    }
```

```
};
```

这个 `sum` 变量（或者叫对象）的 `exec` 就是我们要传递的 函数，我们可以通过传递 `sum` 变量来传递这个 函数，而这个对接口的实现就叫做 匿名内部类。

现在，在*Java 8*中，你可以不用修改接口定义，直接用 *Lambda*表达式 这样写：

```
MyFunction sum = (a, b) -> a + b;
```

也可以这样写：

```
MyFunction sum = (int a, int b) -> a + b;
```

或者这样写：

```
MyFunction sum = (a, b) -> {
    return a + b;
};
```

当然以前的写法仍然是可以的，不过我觉得能一行搞定的事你肯定不会再用四行了。与一般的 匿名内部类 所实现的接口不同的是，配合 *Lambda*表达式 使用的接口（比如这里的 `MyFunction`）需要满足一个条件：

> 除静态方法和默认方法外，该接口只有一个抽象方法。

这个条件是一个接口能用做一个*Lambda*表达式的类型的充分必要条件，而那个抽象方法名是什么并不重要。这样的接口也被称为 函数接口。所以，按照这个条件，*Java*原有的很多只有单个抽象方法的接口都可以写成Lambda表达式了，比如 `Runnable`，以前我们启动一个新线程运行一段代码会这样写：

```
new Thread(new Runnable() {
    @Override
    public void run() {
        // blah...blah...
    }
}).start();
```

而现在，我们可以这样写：

```
new Thread(() -> {
    // blah...blah...
}).start();
```

只要你写的*Lambda*表达式类型正确就可以了，下面是一个错误的例子，无法通过编译，因为 `Runnable` 的 `run` 方法是没有参数的，应当写一对小括号而不是一个变量名 `arg`：

```
new Thread(arg -> {
    // blah...blah...
}).start();
```

另外，*Java 8*还提供了注解 `@FunctionalInterface`，用来加在一个 函数接口 的定义上。它会检查上面那个充分必要条件，如果不满足，比如：

```
@FunctionalInterface
public interface MyFunction {
```

```
    int execute(int a, int b);
    int wtf(int i);
}
```

编译器将会报错：

> MyFunction is not a functional interface.

这个注解仅仅是一种约束和对调用者的提示，虽然函数接口不写这个注解仍然可以用做 *Lambda*表达式 的类型，但是如果这个接口的确是一个函数接口，那么写上是个比较好的习惯。大家可以看一下 *Java* 原有的 `Runable` 、 `Callable` 、 `Comparator` 接口，在 *Java 8* 中都加上了 `@FunctionalInterface` 注解。在 *Vert.x* 中，通常不需要你自己定义函数接口，基本上只需要用到它定义好的 事件处理函数 （以下简称 处理函数）的接口 `Handler<E>` ：

```
@FunctionalInterface
public interface Handler<E> {
    void handle(E event);
}
```

加上 *Java 8* 本身也提供了 `Consumer<T>` 、 `Predicate<T>` 、 `Supplier<T>` 、 `Function<T,R>` 等很多 函数接口，我们基本不用自己定义 函数接口。

最后，再介绍一个与 *Lambda*表达式 相关的 *Java 8* 新特性——η转换（eta-conversion）——一个 函数接口对象还可以写成下面的形式：

```
public class EtaConversion {

    public static void main(String[] args) {

        int a = 4;
        int b = 3;

        // sum 非静态方法需要通过实例获得
        EtaConversion obj = new EtaConversion();
        int c = ofSquare(obj::sum, a, b);

        // diff 静态方法通过类获得即可
        int d = ofSquare(EtaConversion::diff, a, b);

        // 打印计算结果
        System.out.println(a + " 和 " + b + " 的平方和为 " + c + "，平方差为 " + d + "。");
    }

    // 先对两个值分别求平方，再按 func 计算
    public static int ofSquare(MyFunction func, int a, int b) {
        return func.exec(a * a, b * b);
    }

    // 求和
    public int sum(int a, int b) {
        return a + b;
    }

    // 求差（静态方法）
    public static int diff(int a, int b) {
        return a - b;
    }
}
```

以上，就是我要分享的关于 *Java 8* 的新特性 *Lambda*表达式 的所有内容。

3.2 Hello, Vert.x 与异步无阻塞 API

讲完了茴香豆的茴字的几种写法，我们来正式跟 *Vert.x* 打个招呼。下面是参考官网主页上的例子写的一段样例代码：

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.http.HttpHeaders;

public class MyHttpServerVerticle extends AbstractVerticle {

    private static final String[] ARRAY = new String[] {
        "May the Force be with you.",
        "Why so serious?",
        "Farewell, My Concubine."
    };

    private int index;

    @Override
    public void start() {

        index = 0;

        vertx.createHttpServer().requestHandler(req -> {
            req.response()
                .putHeader(HttpHeaders.CONTENT_TYPE, "text/plain")
                .end(ARRAY[index]);
        }).listen(8080, ar -> {
            if (ar.succeeded()) {
                System.out.println("HTTP server started.");
            } else {
                System.out.println("HTTP server error: " + ar.cause().getMessage());
            }
        });

        vertx.setPeriodic(1000, this::rotate);

        System.out.println("MyHttpServerVerticle started.");
    }

    public void rotate(long timerId) {
        index = index + 1;
        index = index % ARRAY.length;
        System.out.println("Current index: " + index);
    }
}
```

> 注：*pom.xml* 文件中要添加依赖：
>
> ```
> <dependency>
>     <groupId>io.vertx</groupId>
>     <artifactId>vertx-core</artifactId>
>     <version>3.5.0</version>
> </dependency>
> ```

这个例子首先创建了一个 *HTTP*服务器，为它设置了 *HTTP*请求 的 处理函数（Handler），对于每个 请求 都直接返回一段文本；然后监听 *8080* 端口，为监听结果设置 处理函数，监听成功则打印 “*HTTP*服务器 启动成功”，监听失败则打印相应的异常信息；另外还设置了一个每1000毫秒触发一次的 定时器 ，每次触发将文本的索引切换

一下，并打印新的索引，这里利用了之前提到的 *η*转换；在做完这一切之后，打印 “`MyHttpServerVerticle` 已启动”。

> 注：这个例子里的三个 处理函数 都是以 *Lambda*表达式 的方式给出的，它们的类型分别是 `Handler<HttpServerRequest>` 、 `Handler<AsyncResult<HttpServer>>` 和 `Handler<Long>` 。

Vert.x 中的基本模块叫做 *Verticle*，最常见的用法是定义一个类继承 `AbstractVerticle` ，然后重载它的 `start` 方法，有些时候还需要重载 `stop` 方法。在一个 *Verticle* 实例被 部署（*deploy*）时，它的 `start` 方法被调用一次，在它被 反部署（*undeploy*，或者叫 撤销）时，它的 `stop` 方法被调用一次。

好的，现在我们 *Verticle* 已经写好了，应该怎么部署呢？首先，我们需要实例化一个 `Vertx` 对象，像这样：

```
Vertx vertx = Vertx.vertx();
```

通常一个程序只需要实例化一个 `Vertx` 对象。然后，像这样部署一个 *Verticle*：

```
vertx.deployVerticle(MyHttpServerVerticle.class.getName(), ar -> {
    if (ar.succeeded()) {
        System.out.println(ar.result());
    } else {
        System.err.println(ar.cause().getMessage());
    }
});
```

如果有需要，你还可以用这个 `Vertx` 实例继续部署更多的 *Verticle*。这里需要说明一下，这里 处理函数（或者说 *Lambda*表达式）的类型是 `Handler<AsyncResult<String>>` ，所以参数类型是 `AsyncResult<String>` 。到目前为止，我们已经两次看见以 `AsyncResult<T>` 为参数的 处理函数 了，它是干什么的呢？

一个 `AsyncResult<T>` 对象代表了一次异步操作的结果（以下简称 异步结果），下面是它的定义：

```
public interface AsyncResult<T> {

    // 获取异步操作的结果（成功时）
    T result();

    // 获取异步操作的异常（失败时）
    Throwable cause();

    // 判断是否成功
    boolean succeeded();

    // 其他方法...
}
```

在异步操作完成时，它会作为参数被传递给我们编写的 处理函数 处理；如果异步操作成功，它会保存类型为 `T` 的结果，我们可以通过 `result` 方法获取这个结果；如果异步操作失败，它会保存导致失败的异常，我们可以通过 `cause` 方法获取这个异常。

在上面部署 *Verticle* 的代码中，如果部署成功，部署生成的 `deploymentID` （一个UUID） 会被打印出来，如果部署失败，异常的描述信息会被打印出来。

如果端口没被占用，我们的网站应该就发布成功了，在浏览器里打开 `http://localhost:8080/` 即可看到返回的文本。

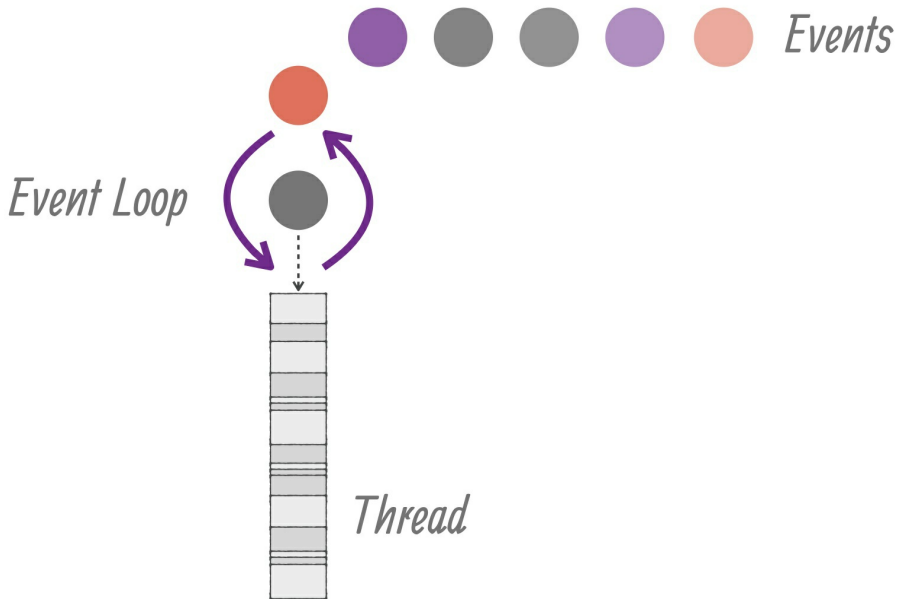
我们之前有提到Vert.x是事件驱动，异步无阻塞的，那么我们回顾一下这个 *HTTP*服务器 样例代码，看看具体体现在哪。

在上面的样例代码中，有三处通过 `System.out.println` 打印出与当前状态有关的信息；通常，运行这段程序，在控制台会按照以下顺序输出：

```
MyHttpServerVerticle started.
HTTP server started.
Current index: 1
Current index: 2
Current index: 0
Current index: 1
Current index: 2
......
```

即 `start` 方法中的信息首先会被打印出来；如果监听 *8080* 端口花费的时间不超过 1s 的话（通常不会出现这种情况），关于端口监听的信息会被打印出来，然后是每秒打印一次的索引信息。之所以有这样的顺序，是因为：

1. 我们目前写的所有代码，所有需要访问网络、读写磁盘等可能耗费较长时间的操作都不是我们在当前线程直接执行的；在样例中，耗时操作包括监听 *8080* 端口、等待 1s 以及等待一个合法的 HTTP 请求；在调用这些 *Vert.x API* 时，方法本身会立即返回，而实际操作会交由操作系统或适当的工作线程执行；这也是使用 *Vert.x API* 不会阻塞当前线程的原因；
2. 在这些耗时操作执行完毕后，被调线程（callee）会通过 事件（*Event*） 通知调用者线程（caller），也就是我们的 *Verticle* 所在的线程；我们写的每一个 事件处理函数 都是在当前 *Verticle* 接收到一个相应 事件 后被调用的；
3. 一个 *Verticle* 中的所有事件处理函数（Handler）都是运行在一个 事件循环（*EventLoop*） 中，即我们在 *Verticle* 中编写业务逻辑代码是完全的运行在单线程中的。



由此引入下面几节的一些概念。

3.3 黄金法则

这里的黄金法则不是说按照这个法则你就能得到好的性能，不按照这个法则你就只能得到正常的性能，而是类似于“铁律”的意思，使用 *Vert.x*，你一定要遵照这一点。这条黄金法则是：

> 不要阻塞事件循环。

开发过客户端程序的同事应该都知道，执行耗时操作的代码应当放到 *Worker*线程 中执行，用户的 *UI*线程 应该永远保持响应。*Vert.x* 中的事件循环就像客户端中的 *UI*线程 一样，不允许执行阻塞或者耗时操作（包括有可能阻塞或耗时的操作），包括但不限于：

> - `Thread.Sleep`
> - 从 *Socket* 读数据
> - 发送一个HTTP请求并等待响应
> - 等待一个锁
> - 执行了一个显著耗时的内存操作

我们前面说过，一个 *Verticle* 里的所有事件处理函数都是运行在一个 事件循环 中，是单线程的，想象一下本来一秒钟可以处理10000个请求（每个 请求 都是一个 事件）的网站服务器程序被一个耗时操作卡了2秒钟是什么感觉，这比用户 *UI*线程 卡2秒严重多了，这可能是一万个用户同时多等待了2秒，按照某些老师的思路，你浪费了大家5个多小时的时间。

不过这个 黄金法则 并不会影响我们实现下面的业务逻辑：

> - 等待一段时间，再运行后续代码
> - 从 *Socket* 读数据，再运行后续代码
> - 发送一个 *HTTP POST* 请求并收到响应，再运行后续代码
> - 等待一个锁，再运行后续代码
> - 执行了一个显著耗时的内存操作，再运行后续代码

——只要你调用 Vert.x 提供的异步无阻塞的 API：

> - `Vertx.setTimer`
> - `NetSocket.handler`
> - `HttpClient.post`
> - `SharedData.getLock`
> - `Vertx.executeBlocking`

它们会完成这些耗时操作并通知你结果，你只需要在 处理函数 中处理这个结果就可以了：

这样设计有什么好处呢？具体有以下两点：

> 1. 通过线程复用提高系统资源利用率，避免高并发启动过多用户线程。如果网络服务器对于每个请求都新开一个用户线程，那么高并发时线程数会显著上升，线程上下文切换将会消耗大量的CPU资源。而 Vert.x 通过使用线程池，复用特定数量的线程，将阻塞的 IO操作 交由操作系统或其他线程异步执行，使用户线程只需要运行无阻塞的 事件处理函数，保证了及时响应以及高性价比的并发支持。
> 2. 减少了资源竞争，天然的线程安全。因为每个 *Verticle* 中的用户代码都是单线程的，所以我们可以在 *Verticle* 中放心的寄存状态变量而不用担心线程安全问题，这使 *Vert.x* 非常适合开发 有状态 的服务。

3.4 事件总线与集群化

到目前为止，我们的介绍都是围绕着单个 *Verticle* ，但是一个项目通常不止一个模块，所以问题来了——

> 如果有多个 *Verticle*，它们之间怎么通信呢？

首先，我想定义一下“多个 *Verticle*”。我们这里说的“多个 *Verticle*”指的是我们实现的多个不同的 *Verticle* 类，区别于同一个 *Verticle* 类部署成多个实例——是的，这一点我前面没有讲，一个 *Verticle* 是单线程的，但是如果某个 *Verticle* 类对并发有更高的要求，想利用多核 CPU 的话，可以像这样部署：

```
vertx.deployVerticle(MyHttpServerVerticle.class.getName(),
    new DeploymentOptions().setInstances(8));
```

这段代码会为 `MyHttpServerVerticle` 部署 8 个实例，分别跑在 8 个线程上；我们不用担心它们监听同一个端口会有冲突， *Vert.x* 会为它们做分发处理。通常这种情况，这个实例数应当设置为 `Runtime.getRuntime().availableProcessors() * 2` （为什么？）。

上面插了一段单个 *Verticle* 利用多核 CPU 横向扩展的介绍，下面开始回答关于 “真 · 多个 *Verticle*” 的通信问题。

Vert.x 模块间通信使用的是 事件总线（*EventBus*） 。在每一个 *Vert.x* 实例中，都存在一个唯一的 事件总线（*EventBus*） 实例，可以通过下面的方法获得：

```
EventBus eb = vertx.eventBus();
```

作为 ***Vert.x*** 的神经系统，事件总线 允许应用的各个部分——无论这些部分是用哪种语言写的、是否在一个 *Vert.x* 实例中——都可以通过它进行通信。在通信的时候，消息会被发送到特定的 地址（*Address*）（一个自定义的字符串），由这个 地址 上注册的 处理函数 处理。一个 地址 上可以注册多个 处理函数。

事件总线 有以下两种通信方式：

1. 发布/订阅模式
2. 点对点模式（支持 请求/响应 模式）

在 发布/订阅模式 中，发布 出去的消息会被发送到注册到目标 地址 上的每个 处理函数 中，类似于 聊天群发。在 点对点模式 中，消息会按照不严格的 轮询算法 发送到一个 处理函数 中，类似于“负载均衡”；在 点对点模式 中，如果需要消息接收方反馈处理结果，还可以在发送方法里添加一个 处理函数，实现类似于 *HTTP* 的 请求/响应模式。

在 事件总线上 可以传输任意类型的对象，通常如果是 基本数据类型、 `String` 以及 *Vert.x* 提供的 `JsonObject` 、 `JsonArray` 、 `Buffer` 等对象，事件总线 可以直接传输；如果是某些自定义的类型，则需要先为这个类型实现一个用于序列化和反序列化的编解码器并注册到 事件总线 上。

事件总线 真正强大之处在于支持集群化。*Vert.x* 提供了集群模式，在执行一个 *Vert.x* 程序或模块时，可以通过添加 `-cluster` 参数开启集群模式。事件总线 可以通过这样一个 *Vert.x* 集群实现消息跨进程、跨节点传输。

基于这样一个集群化的 事件总线，*Vert.x* 提供了类似于 *Dubbo* 那样的远程过程调用模块，可以实现服务发现、管理、发布、订阅等分布式服务相关的功能。同时，官方还提供了一个 *JavaScript* 库，支持在 *Web* 前端页面直接与 *Vert.x* 后台的 事件总线 通信。

3.5 回调地狱与 *Future* 对象

什么技术方案都是有坑的。接下来我给大家介绍一个几乎所有人在用这种模型编程时都会遇到的坑，以及 *Vert.x* 提供的解决方法。

在使用 *Vert.x* 的异步无阻塞 API 时，如果我们要保证一系列操作的执行顺序，通常不能像一般的框架那样简单的依次调用，而是依次把要调用的方法放在前一个方法的事件处理函数中，用 回调函数 用的比较多的同事一定遇到过这种情况：

```
String filePath = "X:\\path\\to\\file";
String backupPath = "X:\\path\\to\\backup\\folder";
Buffer buffer = Buffer.buffer("file content");

vertx.fileSystem().writeFile(filePath, buffer, write -> {
    if (write.succeeded()) {
        vertx.createNetClient().connect(1234, "localhost", connect -> {
            if (connect.succeeded()) {
                connect.result().sendFile(filePath, send -> {
                    connect.result().close(); // 关闭不再使用的Socket
                    if (send.succeeded()) {
                        vertx.fileSystem().copy(filePath, backupPath, copy -> {
                            if (copy.succeeded()) {
                                vertx.fileSystem().delete(filePath, delete -> {
                                    if (delete.succeeded()) {
                                        logger.info("Hello, callback hell.");
                                    } else {
                                        logger.error(delete.cause().getMessage());
                                    }
                                });
                            } else {
                                logger.error(copy.cause().getMessage());
                            }
                        });
                    } else {
                        logger.error(send.cause().getMessage());
                    }
                });
            } else {
                logger.error(connect.cause().getMessage());
            }
        });
    } else {
        logger.error(write.cause().getMessage());
    }
});
```

这段代码先是把一段内容写到一个新文件里，然后建立一个 *TCP* 连接把文件发过去，再把这个文件拷贝到另一个目录作为备份，最后把原文件删掉。回调函数一层层的嵌套，形成了这样的代码结构，这就是 回调地狱。

如果是嵌套个两三层，其实还可以接受，如果业务流程比较长，这样的代码就很难看了。*Vert.x* 提供了四种方法解决这个问题：

- Future
- Vert.x Rx
- Vert.x Async
- Kotlin coroutine

其中，除 `Future` 外的其他三种都需要额外增加依赖，其中 *Vert.x Rx* 是使用观察者模式实现的响应式接口，依赖于 *Reactive X*，而 *Vert.x Async* 和 *Kotlin coroutine* 则是利用了“协程（Fiber/Coroutine）”来实现像调用同步阻塞接口那样使用异步无阻塞接口。我今天只重点介绍使用 Vert.x 核心中提供的 `Future` 来解决回调地狱的方法，大家如果对其它的方法感兴趣可以找官网的文档和例子看一下。

我们直接看 Future 的定义，下面只列出了与我们要介绍的内容有关的部分：

```
public interface Future<T> extends AsyncResult<T>, Handler<AsyncResult<T>> {

    // 用结果 result 将这个未完成的 AsyncResult<T> 设为成功
    void complete(T result);

    // 用异常 cause 将这个未完成的 AsyncResult<T> 设为失败
    void fail(Throwable cause);

    // 设置该 Future 对象被完成时应该调用的处理函数
    Future<T> setHandler(Handler<AsyncResult<T>> handler);

    // 其他方法
}
```

它是一个泛型接口，继承自 异步结果 `AsyncResult<T>` 和 异步结果处理函数 `Handler<AsyncResult<T>>`；我们前面介绍过 异步结果 对象可以保存异步操作的状态（未完成、成功或失败），以及提供结果和异常的获取方法，`Future<T>` 继承自它，表明 `Future<T>` 也有这些特性；同时，不同于只能读取状态和结果的 `AsyncResult<T>`，它还提供了 `complete` 方法和 `fail` 方法，使其可以从外部被完成；

但是，它同时还继承自 异步结果处理函数 `Handler<AsyncResult<T>>` 是怎么回事？一个 `Future<T>` 对象同时还是一个函数？它作为一个函数，要处理的这个 异步结果 `AsyncResult<T>` 又是哪来的呢？

我在 `Future<T>` 接口的实现 `FutureImpl<T>` 类里找到了这个：

```
    @Override
    public void handle(AsyncResult<T> asyncResult) {
        if (asyncResult.succeeded()) {
            complete(asyncResult.result());
        } else {
            fail(asyncResult.cause());
        }
    }
```

没错，这里是它作为一个 处理函数 `Handler<AsyncResult<T>>` 的实现，当它被调用时，就是在处理另一个 异步结果 `AsyncResult<T>`，用另一个 异步结果 的状态和值来完成作为 异步结果 `AsyncResult<T>` 的自己。

我们再看一下刚刚在 回调地狱 里用的那几个 *Vert.x* 提供的 API 的定义：

```
    // 写一段内容到新文件里
    FileSystem writeFile(String path, Buffer data, Handler<AsyncResult<Void>> handler);

    // 建立与某个地址的Socket连接
    NetClient connect(int port, String host, Handler<AsyncResult<NetSocket>> connectHandler);

    // 使用Socket发送一个文件
    NetSocket sendFile(String filename, Handler<AsyncResult<Void>> resultHandler)

    // 复制一个文件到另一个位置
    FileSystem copy(String from, String to, Handler<AsyncResult<Void>> handler);

    // 删除一个文件
    FileSystem delete(String path, Handler<AsyncResult<Void>> handler);
```

我们还可以继续罗列更多的 *Vert.x* API，它们的最后一个参数往往是 `Handler<AsyncResult<T>>` 。我们先按照文档中的方法实例化一个 `Future` 对象，因为写文件的结果是 `Void` ，所以我们实例化的是 `Future<Void>` ：

```
Future<Void> futureWrite = Future.future();
```

然后，我们调用那个写文件的方法，不过不再使用 *Lambda*表达式，而是把 `futureWrite` 放到原来 *Lambda*表达式 的位置：

```
vertx.fileSystem().writeFile(filePath, buffer, futureWrite);
```

如果你还记得 `Future<T>` 继承自 `Handler<AsyncResult<T>>` ，应该不会惊讶于这样的写法。然后，我们给 `futureWrite` 设置一个它在完成时应该调用的 处理函数：

```
futureWrite.setHandler(ar -> {
    if (ar.succeeded()) {
        // success
    } else {
        // error
    }
});
```

这样就实现了读取文件后 `futureWrite` 对象被完成， `futureWrite` 对象的 处理函数 被调用的逻辑。

这样真的有助于解决回调地狱吗？我们可以尝试继续使用这种方法改造上面的回调地狱：

```
Future<Void> futureWrite = Future.future();
Future<NetSocket> futureConnect = Future.future();
Future<Void> futureSend = Future.future();
Future<Void> futureCopy = Future.future();
Future<Void> futureDelete = Future.future();

vertx.fileSystem().writeFile(filePath, buffer, futureWrite);

futureWrite.setHandler(ar -> {
    if (ar.succeeded()) {
            vertx.createNetClient().connect(1234, "localhost", futureConnect);
    } else {
        logger.error(ar.cause().getMessage());
    }
});

futureConnect.setHandler(ar -> {
    if (ar.succeeded()) {
        ar.result().sendFile(filePath, futureSend);
    } else {
        logger.error(ar.cause().getMessage());
    }
});

futureSend.setHandler(ar -> {
    futureConnect.result().close(); // 关闭不再使用的Socket
    if (ar.succeeded()) {
        vertx.fileSystem().copy(filePath, backupPath, futureCopy);
    } else {
        logger.error(ar.cause().getMessage());
    }
});
```

```
futureCopy.setHandler(ar -> {
    if (ar.succeeded()) {
        vertx.fileSystem().delete(filePath, futureDelete);
    } else {
        logger.error(ar.cause().getMessage());
    }
});

futureDelete.setHandler(ar -> {
    if (ar.succeeded()) {
        logger.info("Welcome to the future!!!");
    } else {
        logger.error(ar.cause().getMessage());
    }
});
```

看起来效果还不错，整齐多了，代码也不会随着业务流程长度而无限制缩进了。不过这样还是存在两个问题：

1. 颠倒两个代码块的顺序，该程序仍然是可以运行的，这样一来没有顺序上的约束，很容易产生混乱的代码；
2. 异常处理存在大量重复代码。

好在 `Future` 还提供了一个用于链式调用的方法 `compose`，我们使用 `Future` 的 `compose` 方法再次重构这部分代码：

```
Future<Void> futureWrite = Future.future();
Future<NetSocket> futureConnect = Future.future();
Future<Void> futureSend = Future.future();
Future<Void> futureCopy = Future.future();
Future<Void> futureDelete = Future.future();

vertx.fileSystem().writeFile(filePath, buffer, futureWrite);

futureWrite.compose(v -> {
    vertx.createNetClient().connect(1234, "localhost", futureConnect);
}, futureConnect).compose(socket -> {
    socket.sendFile(filePath, futureSend);
}, futureSend).compose(v -> {
    futureConnect.result().close(); // 关闭不再使用的Socket
    vertx.fileSystem().copy(filePath, backupPath, futureCopy);
}, futureCopy).compose(v -> {
    vertx.fileSystem().delete(filePath, futureDelete);
}, futureDelete).setHandler(ar -> {
    if (ar.succeeded()) {
        logger.info("Hello, future compose!!!");
    } else {
        if (futureConnect.succeeded()) {
            futureConnect.result().close(); // 关闭不再使用的Socket
        }
        logger.error(ar.cause().getMessage());
    }
});
```

除了最后一个 回调函数 ，前面的所有 回调函数 的参数并不是一个 `AsyncResult<T>` 对象，而是我们期望的结果，即一个类型为 `T` 的对象；也就是说每次 `compose` 只处理上一步成功的情况，失败的异常会被层层传递到最后一个 回调函数 处理——这是不是有点像传统的 `try catch` 结构。

好的，这几乎就是使用 `Future` 改造回调地狱的终极解决方案了， `compose` 方法还有另外一个重载实现，有兴趣的同事可以自己尝试写写看。

3.6 自定义异步方法

最后，我介绍一下 如何组合 *Vert.x* 提供异步 *API* 实现自己的异步方法。

我们通常会自定义一些像下面这样方法，它的结果将通过返回值得到：

```
Result getSomething(Argument ...args);
```

在调用时我们会用 `try` 把它包裹起来，以便在无法得到结果的时候处理异常:

```
try {
    Result result = getSomething(args);
    System.out.println("Result is " + result.toString());
} catch (Exception e) {
    System.err.println(e.getMessage());
}
```

现在，我准备使用 Vert.x API 自定义一个根据主键从数据库特定表中取出一行数据的方法，它包括两个异步操作：

1. 从连接池中取出一个连接；
2. 用它查询一条记录并返回；

我们开始尝试使用 Vert.x 提供的 API 自定义一个这样方法，如下：

```
public ResultSet getRecord(SQLClient client, String primaryKey) {

    sqlClient.getConnection(conn -> {
        if (conn.succeeded()) {
            SQLConnection connection = conn.result();
            connection.query("SELECT ID, DATA FROM RECORD WHERE ID='" + pk + "'", rec -> {
                connection.close(); // 一定要将不使用的连接放回连接池
                if (rec.succeeded()) {
                    ResultSet resultSet = rec.result();
                    // 成功，需要 return
                } else {
                    // 失败，需要 throw exception
                }
            });
        } else {
            // 失败，需要 throw exception
        }
    });

    // return ??;
}
```

到这我们已经得到查询结果了，问题是：结果怎么返回呢？如果 `return` 写在 *Lambda*表达式 里，返回的是那个 *Lambda*表达式，并不是我们的 `getRecord`；写在最后的话，按照异步调用的顺序，好像那时候结果还没查到；异常怎么抛出呢？在 *Lambda*表达式 里抛好像是抛到 事件循环（*EventLoop*） 里去了，不是从我们的 `getRecord` 抛出去，外面 `catch` 不到了。　　我想大概很多同事已经知道该怎么实现了，不过这种初到 异步世界 的水土不服卡住过很多人。一种正确的实现方法如下：

```
public void getRecord(SQLClient client, String primaryKey, Handler<AsyncResult<ResultSet>> handler) {
    client.getConnection(conn -> {
```

```
        if (conn.succeeded()) {
            SQLConnection connection = conn.result();
            connection.query("SELECT ID, DATA FROM RECORD WHERE ID='" + primaryKey + "'", rec -> {
                connection.close(); // 一定要将不使用的连接放回连接池

                handler.handle(rec);
            });
        } else {
            handler.handle(Future.failedFuture(conn.cause()));
        }
    });
}
```

这样调用：

```
getRecord(client, primaryKey, ar -> {
    if (ar.succeeded()) {
        ResultSet record = ar.result();
        // TODO: 得到 record，继续处理
    } else {
        System.err.println(ar.cause().getMessage()); // 打印异常信息
    }
});
```

另一种正确的实现方法如下：

```
public Future<ResultSet> getRecord(SQLClient client, String primaryKey) {
    Future<ResultSet> futureRecord = Future.future();

    client.getConnection(conn -> {
        if (conn.succeeded()) {
            SQLConnection connection = conn.result();
            connection.query("SELECT ID, DATA FROM RECORD WHERE ID='" + primaryKey + "'", rec -> {
                connection.close(); // 一定要将不使用的连接放回连接池

                futureRecord.handle(rec);
            });
        } else {
            futureRecord.fail(conn.cause());
        }
    });

    return futureRecord;
}
```

这样调用：

```
getRecord(client, primaryKey).setHandler(ar -> {
    if (ar.succeeded()) {
        ResultSet record = ar.result();
        // TODO: 得到 record，继续处理
    } else {
        // 打印异常信息
        System.err.println(ar.cause().getMessage());
    }
});
```

或者:

```
getRecord(client, primaryKey).compose(resultSet -> {
```

```
    // TODO: 得到 record，继续处理
}, nextFuture).compose(nextResult -> {

...... // 若干次 compose 组成的链式调用

}, finalFuture).setHandler(ar -> {
    if (ar.succeeded()) {
        // 这里得到的是整个链式调用的结果，不是 getRecord 的结果
    } else {
        // 如果 getRecord 失败，会跳到这里打印异常
        System.err.println(ar.cause().getMessage());
    }
});
```

所以，我们使用 *Vert.x* 的异步 API 自定义方法时，应当实现为像:

```
void getRecord(SQLClient client, String primaryKey, Handler<AsyncResult<ResultSet>> handler);
```

或

```
Future<ResultSet> getRecord(SQLClient client, String primaryKey);
```

这样的形式，也正是 函数式编程 中 高阶函数 的两种典型的形式。

> 注：高阶函数 是指接收另外一个函数作为参数，或返回一个函数的函数。

4. 参考资料

原理与哲学：

> Why modern systems need a new programming model
>
> How the Actor Model Meets the Needs of Modern, Distributed Systems
>
> 响应式宣言（The Reactive Manifesto）

Vert.x 官网：

> http://vertx.io/

Vert.x 官方文档：

> http://vertx.io/docs/

Vert.x GitHub：

> https://github.com/eclipse/vert.x

Vert.x Google 论坛：

> https://groups.google.com/forum/m/?fromgroups#!forum/vertx

书籍：

> Java 8 函数式编程（[英]Richard Warburton）

电子书：

> A gentle guide to asynchronous programming with Eclipse Vert.x for Java developers
>
> Building Reactive Microservices in Java

GitHub 上的例子：

> 官方例子 *GitHub* 库： vert-x3/vertx-examples
>
> 与 *Vert.x* 相关的开源项目列表： vert-x3/vertx-awesome
>
> *Vert.x* 蓝图 - *Micro-Shop* 微服务： sczyh30/vertx-blueprint-microservice
>
> *Vert.x* 蓝图 - 待办事项服务： sczyh30/vertx-blueprint-todo-backend
>
> 基于 *Vert.x* 的异步无阻塞 *FastDFS* 客户端实现： gengteng/vertx-fastdfs-client

QQ群：

> *Vert.x* 中国用户组：515203212