```
val akkaVersion = "2.5.4"
val akkaHttpVersion = "10.0.10"
libraryDependencies += "com.typesafe.akka" %% "akka-http"      % akkaHttpVersion
libraryDependencies += "com.typesafe.akka" %% "akka-actor"    % akkaVersion
libraryDependencies += "com.typesafe.akka" %% "akka-stream" % akkaVersion
```

Common Abstractions (Client- and Server-Side)

HTTP Model

Akka HTTP 为 http 主要的数据结构，例如 request、response、header 等等。
既然提供这些数据模型，可以使用下面的 import 语句引入这些数据模型：

```
import akka.http.scaladsl.model._
```

1、HttpRequest

一个 HttpRequest 包含：
请求方法、URI、请求头、请求体、http 协议

构造 HttpRequest 的方式：

```
import HttpMethods._
```

构造一个简单的 GET 请求：

```
val homeUri = Uri("/abc")
HttpRequest(GET, uri = homeUri)
```

或者

```
HttpRequest(GET, uri = "/index")        // implicit String to Url conversion
```

构造一个简单的包含请求体 POST 请求：

```
val data = ByteString("abc")
HttpRequest(POST, uri = "/receive", entity = data)
```

自定义 HttpRequest 的每一个细节：

```
val userData = ByteString("abc")
val authorization = headers.Authorization(BasicHttpCredentials("user", "pass"))
HttpRequest(
  PUT,
  uri = "/user",
  entity = HttpEntity(`text/plain` withCharset `UTF-8`, userData),
  headers = List(authorization),
  protocol = `HTTP/1.0`)
```

3、HttpResponse

一个 HttpResponse 包含：

状态码、响应头、响应体、http 协议

构造 HttpResponse 的几种方式：

```
// simple OK response without data created using the integer status code
HttpResponse(200)

// 404 response created using the named StatusCode constant
HttpResponse(NotFound)

// 404 response with a body explaining the error
HttpResponse(404, entity = "Unfortunately, the resource couldn't be found.")

// A redirecting response containing an extra header
val locationHeader = headers.Location("http://example.com/other")
HttpResponse(Found, headers = List(locationHeader))
```

4、HttpEntity

一个 HttpEntity 实例中携带着字节（具体的内容 / 数据）以及其相关的 Content-Type 和 Content-Length。

有五种 HttpEntity：HttpEntity.Strict、HttpEntity.Default、HttpEntity.Chunked、HttpEntity.CloseDelimited、HttpEntity.IndefiniteLength

1、HttpEntity.Strict

最简单的模型，可用于当整个正文内容都已被加载到内存的时候。

它把一个 ByteString 包装起来构成一个标准的，非分块的，及带有已知内容长度 Content-Length 的正文。

2、HttpEntity.Default

最通用，非分块的 HTTP/1.1 信息正文模型。 它具有已知的内容长度并其内容数据类型为 Source[ByteString]，该类型只能被实例化一次。

Strict 和 Default 两模型间的区别只是在 API 定义上，在实际传输连接上，两者的正文内容是一样的。

3、HttpEntity.Chunked

为 HTTP/1.1 分块传输内容（即发送时定义 Transfer-Encoding: chunked）设计的模型。内容信息长度未知，而且每一个分块数据体现为 Source[HttpEntity.ChunkStreamPart]。一个 ChunkStreamPart 可以是 Chunk 类型（非空内容）或 LastChunk 类型（包含可能相关的头域）。相应的数据流由零或多个 Chunked 对象组成，并可以被一个 LastChunk 对象（非必要）终止。

4、HttpEntity.CloseDelimited

该模型针对一个非分块的 HTTP 个体，其未知的传输内容长度是当连接结束时才确立的。数据内容的表现方式为 Source[ByteString] 类型。

因为连接的结束必须发生在完成传输该类型正文之后，所以这种模型只能用于在服务器端生成响应内容。

同时，设计 CloseDelimited 类型的主要原因在于它和 HTTP / 1.0 的兼容性，因为旧的协议不支持分块传输。如果开发者在开发一个新的应用程序时没有旧需求的约束，最好不要使用 CloseDelimited 类型。因为使用 Connection：Close 断开 HTTP 连接并不是一个稳

健的做法，特别对于大量使用代理服务器的今天。再加上这个类型使得 HTTP 连接无法复用，会严重影响性能。建议使用 HttpEntity.Chunked 类型！

5、HttpEntity.IndefiniteLength

一个内存长度不确定的流式正文数据模型，对应于 Multipart.BodyPart。

Strict、Default 和 Chunked 是 HttpEntity.Regular 的子类，这三种既可以作为 request 的实体也可以作为 response 的实体。
但是 ClosedDelimited 只能作为 response 的实体。

流式正文类型（除了 Strict 以外的所有）是不能被共享或序列化的。当需要建立一个完备的，可共享的正文内容或消息的时候可使用 HttpEntity.toStrict 或 HttpMessage.toStrict 返回一个 Future，数据流中的字节会被收集成 ByteString 到这个被 Future 包着的 Strict 实例内。

如果我们想单独处理每一种 HttpEntity，可以针对 HttpEntity 使用模式匹配。
但是，大部分的情况下，其实我们并不关心具体是哪种类型的 HttpEntity，因为可以通过 HttpEntity.dataBytes 方法获得 Source[ByteString, Any]类型的结果，从而可以访问 entity 中的具体数据。

什么时候用什么子类型？：

- 如果数据比较小，并且可以直接放到内存中，就使用 Strict；
- 如果数据是从流式数据源中产生(惰性)，并且数据的长度知道，就使用 Default；
- 如果 entity 的长度不知道，就使用 Chunked；
- 当仅做为 Chunked 类型的替代品，以便为某些不支持分块传输模式的旧式客户端提供服务时，使用 CloseDelimited 在响应信息中。否则，请使用 Chunked。
- 当需要在 Multipart.Bodypart 里提供未知长度的正文内容，使用 IndefiniteLength。

限制 HttpEntity 的长度：

所有从网络中读取到的 HttpEntity 都会被进行长度检查，这可以保证 entity 的长度小于或等于 max-content-length 配置的大小，确保服务不会遭到攻击。
但是，在某些 request 允许大尺寸的 entity 时，这样的全局配置，它的扩展性就不是很好了。

为了提高扩展性，HttpEntity 提供了 withSizeLimit 方法，它能让你调节全局的配置(max-content-length)去满足特定的 entity。
在调用 withSizeLimit 方法时，如果 Strict 类型的 entity 的长度在长度限制之内，就返回 entity 自身；否则返回一个 Default 类型的 entity，其包含一个单元素(惰性)。

注意：

akka.http.parsing.max-content-length (applying to server- as well as client-side),
akka.http.server.parsing.max-content-length (server-side only),
akka.http.client.parsing.max-content-length (client-side only)
akka.http.host-connection-pool.client.parsing.max-content-length (only host-connection-pools)

4、Header

1、Header Model

Akka HTTP 包含了丰富多样的 Http Header 模型，它们的解析和渲染工作都交给 Akka HTTP 自动完成，因此我们不需要关系具体的细节。

处理 header 的几种方式：

```
import akka.http.scaladsl.model.headers._

// create a ``Location`` header
val loc = Location("http://example.com/other")

// create an ``Authorization`` header with HTTP Basic authentication data
val auth = Authorization(BasicHttpCredentials("joe", "josepp"))

// custom type
case class User(name: String, pass: String)

// a method that extracts basic HTTP credentials from a request
def credentialsOfRequest(req: HttpRequest): Option[User] =
  for {
    Authorization(BasicHttpCredentials(user, pass)) <- req.header[Authorization]
  } yield User(user, pass)
```

2、Http Headers

当 Akka HTTP 接受到一个 request 时，会尝试解析其中所有的 http header，转换成对应的 http model。

对于那些不知道的 header 或者语法无效的 header，会被解析成 RawHeader；

对于那些解析失败报错的 header，将会被记录日志，但这依赖于 illegal-header-warnings 配置。

3、Custom Header

按下面的方式创建自定义的 header：

```
final class ApiTokenHeader(token: String) extends ModeledCustomHeader[ApiTokenHeader] {
  override def renderInRequests = false
  override def renderInResponses = false
  override val companion = ApiTokenHeader
  override def value: String = token
}
object ApiTokenHeader extends ModeledCustomHeaderCompanion[ApiTokenHeader] {
  override val name = "apiKey"
  override def parse(value: String) = Try(new ApiTokenHeader(value))
}
```

使用场景如下：

```
val ApiTokenHeader(t1) = ApiTokenHeader("token")
t1 should ===("token")

val RawHeader(k2, v2) = ApiTokenHeader("token")
k2 should ===("apiKey")
v2 should ===("token")

// will match, header keys are case insensitive
val ApiTokenHeader(v3) = RawHeader("APIKEY", "token")
v3 should ===("token")
```

5、Registering Custom Media Types

有时候我们需要自定义 media type，并且告诉解析器如何处理这些自定义的 media type。

为了自定义 media type，你需要坐下面这些工作：

```
// similarily in Java: `akka.http.javadsl.settings.[...]`
import akka.http.scaladsl.settings.ParserSettings
import akka.http.scaladsl.settings.ServerSettings

//  自定义 media type:
val utf8 = HttpCharsets.`UTF-8`
val `application/custom`: WithFixedCharset = MediaType.customWithFixedCharset("application", "custom", utf8)

// add custom media type to parser settings:
val parserSettings = ParserSettings(system).withCustomMediaTypes(`application/custom`)
val serverSettings = ServerSettings(system).withParserSettings(parserSettings)

val binding = Http().bindAndHandle(routes, host, port, settings = serverSettings)
```

6、Registering Custom Status Codes

为了自定义 status code，你需要做下面的工作：

```
// similarily in Java: `akka.http.javadsl.settings.[...]`
import akka.http.scaladsl.settings.{ ParserSettings, ServerSettings }

//  自定义 status code:
val LeetCode = StatusCodes.custom(777, "LeetCode", "Some reason", isSuccess = true, allowsEntity = false)

// add custom method to parser settings:
val parserSettings = ParserSettings(system).withCustomStatusCodes(LeetCode)
val serverSettings = ServerSettings(system).withParserSettings(parserSettings)

val clientConSettings = ClientConnectionSettings(system).withParserSettings(parserSettings)
val clientSettings = ConnectionPoolSettings(system).withConnectionSettings(clientConSettings)

//  使用自定义的 status code
val routes = complete(HttpResponse(status = LeetCode))

//  在 server 端使用 serverSettings:
val binding = Http().bindAndHandle(routes, host, port, settings = serverSettings)

//  在客户端使用 clientSettings:
val request = HttpRequest(uri = s"http://$host:$port/")
val response = Http().singleRequest(request, settings = clientSettings)

// futureValue is a ScalaTest helper:
response.futureValue.status should === (LeetCode)
```

URI Model

1、构造 Uri

下面介绍如何构造 URI Model 的实例。在 Uri.from()中你需要传入 scheme、host、path、query 参数。

```
Uri("ftp://ftp.is.co.za/rfc/rfc1808.txt") shouldEqual
  Uri.from(scheme = "ftp", host = "ftp.is.co.za", path = "/rfc/rfc1808.txt")

Uri("http://www.ietf.org/rfc/rfc2396.txt") shouldEqual
  Uri.from(scheme = "http", host = "www.ietf.org", path = "/rfc/rfc2396.txt")

Uri("ldap://[2001:db8::7]/c=GB?objectClass?one") shouldEqual
  Uri.from(scheme = "ldap", host = "[2001:db8::7]", path = "/c=GB", queryString = Some("objectClass?one"))

Uri("mailto:John.Doe@example.com") shouldEqual
  Uri.from(scheme = "mailto", path = "John.Doe@example.com")

Uri("news:comp.infosystems.www.servers.unix") shouldEqual
  Uri.from(scheme = "news", path = "comp.infosystems.www.servers.unix")

Uri("tel:+1-816-555-1212") shouldEqual
  Uri.from(scheme = "tel", path = "+1-816-555-1212")

Uri("telnet://192.0.2.16:80/") shouldEqual
  Uri.from(scheme = "telnet", host = "192.0.2.16", port = 80, path = "/")

Uri("urn:oasis:names:specification:docbook:dtd:xml:4.1.2") shouldEqual
  Uri.from(scheme = "urn", path = "oasis:names:specification:docbook:dtd:xml:4.1.2")
```

下面是 Http 规范定义的 Uri 中的各个部分：

```
  foo://example.com:8042/over/there?name=ferret#nose
  \_/   \______________/\_________/ \_________/ \__/
   |           |            |            |        |
scheme     authority       path        query   fragment
   |   _____________________|__
  / \ /                     \
  urn:example:animal:ferret:nose
```

无效的 URI string

当一个无效的 URI string 传给 Uri()方法时，将会抛出 IllegalUriException。

2、Query string in URI

Uri class's query() method returns the query string of the URI, which is modeled in an instance of the Query class.

示例：

Uri("http://localhost?a=b").query() is equivalent to: Query("a=b")

Marshalling

在 Akka Http 中 Marshalling 表示：将一个类型 T 转换成 lower-level type，例如 MessageEntity (http 请求或相应的实体内容) 或者 HttpRequest、HttpResponse。
Marshalling 的同义词有“Serialization”，“pickling”。

将一个类型 A Marshalling to 类型 B，表现为 Marshalling[A, B]。
Akka-HTTP 针对很多 Marshaller 类型，预定了很多的别名：
type ToEntityMarshaller[T] = Marshaller[T, MessageEntity]
type ToByteStringMarshaller[T] = Marshaller[T, ByteString]
type ToHeadersAndEntityMarshaller[T] = Marshaller[T, (immutable.Seq[HttpHeader], MessageEntity)]
type ToResponseMarshaller[T] = Marshaller[T, HttpResponse]
type ToRequestMarshaller[T] = Marshaller[T, HttpRequest]

Akka-HTTP 针对大部分通用类型预定义了很多 marshaller，例如;
PredefinedToEntityMarshallers

- Array[Byte]
- ByteString
- Array[Char]
- String
- akka.http.scaladsl.model.FormData
- akka.http.scaladsl.model.MessageEntity
- T <: akka.http.scaladsl.model.Multipart

PredefinedToResponseMarshallers

- T, if a ToEntityMarshaller[T] is available
- HttpResponse
- StatusCode
- (StatusCode, T), if a ToEntityMarshaller[T] is available
- (Int, T), if a ToEntityMarshaller[T] is available
- (StatusCode, immutable.Seq[HttpHeader], T), if a ToEntityMarshaller[T] is available
- (Int, immutable.Seq[HttpHeader], T), if a ToEntityMarshaller[T] is available

PredefinedToRequestMarshallers

- HttpRequest
- Uri
- (HttpMethod, Uri, T), if a ToEntityMarshaller[T] is available
- (HttpMethod, Uri, immutable.Seq[HttpHeader], T), if a ToEntityMarshaller[T] is available

GenericMarshallers

- Marshaller[Throwable, T]
- Marshaller[Option[A], B], if a Marshaller[A, B] and an EmptyValue[B] is available
- Marshaller[Either[A1, A2], B], if a Marshaller[A1, B] and a Marshaller[A2, B] is available
- Marshaller[Future[A], B], if a Marshaller[A, B] is available
- Marshaller[Try[A], B], if a Marshaller[A, B] is available

Encoding / Decoding

Http 规范中定义了一个 Content-Encoding 头，用来描述 Http 的实体内容是否被某个算法 encode。通常来说是压缩算法。

目前 Akka Http 支持使用 gzip 或者 deflate 来对 request 和 response 进行压缩和解压缩。

在服务端，默认没有开启压缩/加压缩。如果需要，则必须手动进行编码来支持。

而在客户端，下面的例子展示了如何 decode 带有 Content-Encoding 的 response：

```
implicit val system = ActorSystem()
implicit val materializer = ActorMaterializer()
import system.dispatcher

val http = Http()

val requests: Seq[HttpRequest] = Seq(
  "https://httpbin.org/gzip",      // Content-Encoding: gzip in response
  "https://httpbin.org/deflate",   // Content-Encoding: deflate in response
  "https://httpbin.org/get"         // no Content-Encoding in response
).map(uri ⇒ HttpRequest(uri = uri))

def decodeResponse(response: HttpResponse): HttpResponse = {
  val decoder = response.encoding match {
    case HttpEncodings.gzip ⇒
      Gzip
    case HttpEncodings.deflate ⇒
      Deflate
    case HttpEncodings.identity ⇒
      NoCoding
  }
  decoder.decodeMessage(response)
}

val futureResponses: Future[Seq[HttpResponse]] =
  Future.traverse(requests)(http.singleRequest(_).map(decodeResponse))

futureResponses.futureValue.foreach { resp =>
  system.log.info(s"response is ${resp.toStrict(1.second).futureValue}")
}

system.terminate()
```

Http Timeout

Akka Http 有多种内建的超时机制，来避免你的服务遭受恶意的攻击。

一、通用的超时处理

Idle timeouts

The idle-timeout is a global setting which sets the maximum inactivity time of a given connection.

In other words, if a connection is open but no request/response is being written to it for over idle-timeout time, the connection will be **automatically closed**.

使用下面的配置选项进行配置：

akka.http.server.idle-timeout

akka.http.client.idle-timeout

akka.http.host-connection-pool.idle-timeout

akka.http.host-connection-pool.client.idle-timeout

注意事项：

对于客户端的连接池设置，闲置时段的判断起始点是当连接池中已经没有需要处理的请求在等待写入为准。

二、服务端的超时处理

1、Request timeout

请求超时处理：这是限制一个路由上生成一个 HttpResponse 的所需最大时长的一种机制。

当超过了 Request Timeout，服务端会自动产生一个类似下面的 HttpResponse，并且关闭 http connection 以防止溢出或者防止连接无限期地存在。

```
HttpResponse(StatusCodes.ServiceUnavailable, entity = "The server was not able " +
  "to produce a timely response to your request.\r\nPlease try again in a short while!")
```

使用下面的配置选项进行配置。默认是 20 秒。

akka.http.server.request-timeout

一种推荐的方式是：使用上面的配置进行静态配置，然后利用 TimeoutDirectives 针对某个 route 单独配置。

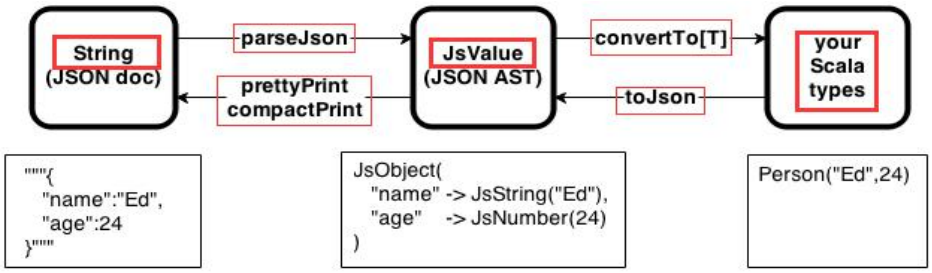
三、客户端的超时处理

1、Connecting timeout

连接超时是指：完成建立相关 TCP 两端连接所需过程的时间长短超过规限。

使用下面的配置选项进行配置

akka.http.client.connecting-timeout



将 JSON 字符串转换成 JSON AST：

```
val source = """{ "some": "JSON source" }"""
val jsonAst = source.parseJson // or JsonParser(source)
```

将 JSON AST 美化打印或者压缩打印：

```
val json = jsonAst.prettyPrint // or .compactPrint
```

一、使用教程

1、对于 Scala 内置的类型，spray-json 定义了 **DefaultJsonProtocol**，它能将 Scala 中**所有的值类型**和**大部分引用类型和集合类型**，通过使用 **toJson** 方法，转换成 JSON AST。

下面列出了 DefaultJsonProtocol 考虑到的类型，只要将 DefaultJsonProtocol 引入到作用域中，就可以使用 toJson 方法将下面的类型转换成 JSON AST。

- Byte, Short, Int, Long, Float, Double, Char, Unit, Boolean
- String, Symbol
- BigInt, BigDecimal
- Option, Either, Tuple1 - Tuple7
- List, Array
- immutable.{Map, Iterable, Seq, IndexedSeq, LinearSeq, Set, Vector}
- collection.{Iterable, Seq, IndexedSeq, LinearSeq, Set}
- JsValue

示例：

```
import spray.json._
object Main extends App {
    import DefaultJsonProtocol._
    val jsonAST = Map(“name” -> “ligx”, “age” -> “23”).toJson
}
```

2、

① 对于自定义的类型，如果自定义的类型是一个 case class，只需要**定义一个 object 扩展 DefaultJsonProtocol**，并通过**调用 jsonFormatX 方法**提供一个隐式的 JsonFormat[T]类型实例，然后**将定义的 object 引入作用域**即可。

jsonFormatX 方法接受 case class 的伴生对象做参数，返回一个符合 case class 构造函数参数个数要求的 JsonFormat。

例如，如果你的 case class 有 13 个类参数，那么就需要使用 jsonFormat13 方法。

示例：

```
case class Color(name: String, red: Int, green: Int, blue: Int)
object MyJsonProtocol extends DefaultJsonProtocol {
    implicit val colorFormat = jsonFormat4(Color)
}
```

```
object Main extends App {
    import MyJsonProtocol._
    val jsonAST = Color("CadetColor", 95, 158, 160).toJson
    val json = jsonAST.convertTo[Color]
}
```

② 有额外的情况：如果 case class 显示地声明了一个伴生对象，那么按照上面的方式做会编译报错，必须显示地调用伴生对象的 apply 方法才能正常执行。

```
case class Color(name: String, red: Int, green: Int, blue: Int)
object Color
object MyJsonProtocol extends DefaultJsonProtocol {
    implicit val colorFormat = jsonFormat4(Color.apply)
}
```

③ 如果自定义的 case class 是一个泛型，这时就需要在扩展 DefaultJsonFormat 的 object 中定义一个隐式转换函数，隐式转换函数的声明中使用上下文界定声明类型参数也是 JsonFormat[T]的实例，并在调用 jsonFormatX 方法时指定参数为显示地调用 case class 的伴生对象的 apply 方法。

```
case class Color[A](name: String, items: List[A])
object MyJsonProtocol extends DefaultJsonProtocol {
    implicit def colorListFormat[A: JsonFormat] = jsonFormat2(Color.apply[A])
}
```

④ 如果自定义的 case class 的类参数中有 Option 类型的参数，当参数值为 None，执行 toJson 方法时默认会忽略这种值；
然而，通过把 NullOptions 特质混入到自定义的 JsonProtocol 单例对象中，那么在执行 toJson 时就会把 None 参数值渲染成 null 值。
但是，反过来对 Json AST 执行 convertTo 时，None 值仍然是 None 值，不会是 null。

```
case class Color(name: Option[String], red: Int, green: Int, blue: Int)
object MyJsonProtocol extends DefaultJsonProtocol with NullOptions {
    implicit val colorFormat = jsonFormat4(Color)
}
object Main extends App {
    import MyJsonProtocol._
    val jsonAST = Color(None, 95, 158, 160).toJson
    println(jsonAST)                          // None 参数值会变成 null
    println(jsonAST.convertTo[color].name)    // 得到的仍然是 None，不会是 null
}
```

3、如果自定义的类型不是 case class，而是一个普通的 class。也可以序列化/反序列化成 JsonAST 或者 class instance。

示例 1：将 Class instance 序列化成 Json Array

```
class Color(val name: String, val red: Int)

object MyJsonProtocol extends DefaultJsonProtocol {
  implicit object ColorJsonFormat extends RootJsonFormat[Color] {  // 定义 RootJsonFormat[T] type class 的隐式实例
    def write(c: Color) =
      JsArray(JsString(c.name), JsNumber(c.red))  // Color instance 序列化得到 Json Array

    // 因为 toJson 方法的结果类型是 JsValue，所以反序列化 jsonAST 时，read 方法的参数类型就是 JsValue
    // 使用模式匹配的方式从 jsonAST 得到 class instance
```

```
    // 因为 JsArray 只接受一个参数，因此所有的 JsString 和 JsNumber 倍 Vector 包起来了
    // 在模式匹配的中变量 red 是 BigDecimal 类型，因此要使用 toInt 方法转换成 int 类型
    def read(value: JsValue) = value match {
      case JsArray(Vector(JsString(name), JsNumber(red))) => new Color(name, red.toInt)
      case _ => deserializationError("Color expected")
    }
  }
}

import MyJsonProtocol._
val json = Color("CadetBlue", 95).toJson        // 得到的 json 是：[“CadetBlue”, 95]
val color = json.convertTo[Color]
```

示例 2：将 Class instance 序列化成 Json Object

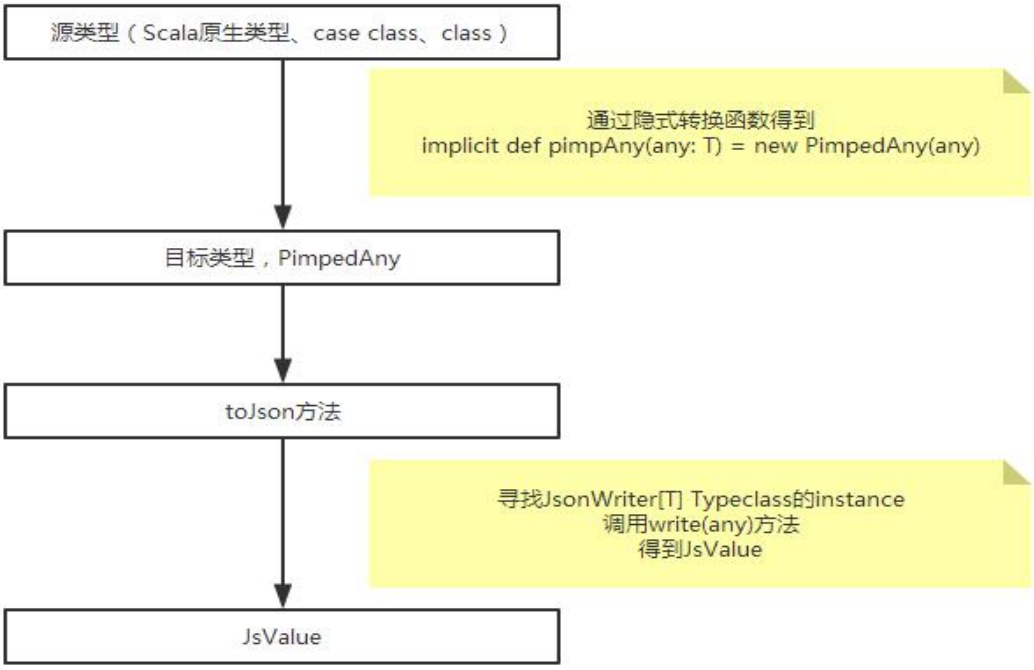
```
object MyJsonProtocol extends DefaultJsonProtocol {
  implicit object ColorJsonFormat extends RootJsonFormat[Color] {
    def write(c: Color) = JsObject(
      "name" -> JsString(c.name),
      "red" -> JsNumber(c.red)
    )
    def read(value: JsValue) = {
      value.asJsObject.getFields("name", "red") match {
        case Seq(JsString(name), JsNumber(red)) => new Color(name, red.toInt)
        case _ => throw new DeserializationException("Color expected")
      }
    }
  }
}

import MyJsonProtocol._
val json = Color("CadetBlue", 95).toJson
// 得到的 json 是：
{
    “name” : “CadetBlue”,
    “red” : 95
}
```

4、spray-json 和 Akka-HTTP 结合

只需要在扩展 DefaultJsonProtocol 时，再混入 SprayJsonSupport 特质就好了。

二、源码分析

toJson 方法内部流程：



convertTo 方法内部流程：

单纯的 import spray.json._，只是把隐式转换函数引入了当前作用域，并没有把需要的 JsonFormat[T] Typeclass instance 引入到当前作用域中，故 toJson 无法发挥作用；

要使 toJson 对原生类型发挥作用，还需要 import DefaultJsonProtocol._

Scala 为大部分的原生类型都提供的对应的 JsonFormat[T] Typeclass 的 instance，存放在 DefaultJsonProtocol 中。

对 spray-json 为 case class 提供的 jsonFormatN 方法进行源码剖析：

因为 toJson 方法内部最终是调用 JsonWriter[T]的 write 方法得到 JsValue；

jsonFormatN(...)方法虽然得到的是 RootJsonFormat[T]类型的一个实例，但是 RootJsonFormat 是 JsonFormat 的子类，最终得到的就是 JsonFormat[T] Typeclass 的 instance，供 toJson 方法使用。

trait RootJsonFormat[T] extends JsonFormat[T] with RootJsonReader[T] with RootJsonWriter[T]

```
case class Color(name: String, red: Int, green: Int, blue: Int)
object MyJsonProtocol extends DefaultJsonProtocol {
  implicit val colorFormat = jsonFormat4(Color)     // 不提供任何参数的部分应用函数在做参数可以省略下划线
}

def jsonFormat4[P1 :JF, P2 :JF, P3 :JF, P4 :JF, T <: Product :ClassManifest](construct: (P1, P2, P3, P4) => T): RootJsonFormat[T] = {
    val Array(p1, p2, p3, p4) = extractFieldNames(classManifest[T])   // 通过使用 ClassManifest、反射，得到 case class 的所有字段的名称
    jsonFormat(construct, p1, p2, p3, p4)
}

def jsonFormat[P1 :JF, P2 :JF, P3 :JF, P4 :JF, T <: Product](construct: (P1, P2, P3, P4) => T, fieldName1: String, fieldName2: String, fieldName3: String, fieldName4: String): RootJsonFormat[T] = new RootJsonFormat[T]{
    def write(p: T) = {
      val fields = new collection.mutable.ListBuffer[(String, JsValue)]
      fields.sizeHint(4 * 5)
      fields ++= productElement2Field[P1](fieldName1, p, 0)     // 使用 Product 特质提供的 productElement(n)方法得到参数的具体值
      fields ++= productElement2Field[P2](fieldName2, p, 1)     // 调用 JsonWriter[T]的 write 方法得到参数值对应的 JsValue
      fields ++= productElement2Field[P3](fieldName3, p, 2)     // 根据字段名称、JsValue，得到 JsField，所有的参数得到 JsField 列表
      fields ++= productElement2Field[P4](fieldName4, p, 3)     // 最终得到一个 JsObject
      JsObject(fields: _*)
    }
    def read(value: JsValue) = {
      val p1V = fromField[P1](value, fieldName1)
      val p2V = fromField[P2](value, fieldName2)
      val p3V = fromField[P3](value, fieldName3)
      val p4V = fromField[P4](value, fieldName4)
      construct(p1V, p2V, p3V, p4V)
    }
}
```

处理 Http Entity 流

一、客户端处理 Http Entity 流

1、消费 Response Entity

在客户端，最通常的场景就是消费服务端传送过来的 http response。
可以使用 dataBytes()方法来消费 http response entity。

下面展示了使用 dataBytes 方法的例子：
把流入的分块数据放到一个一个帧内，然后一行一行地转换类型，最后把管道接到出口（如 File 或其它 Akka Streams 接口的 ）Sink。

```
val response: HttpResponse = ???
response.entity.dataBytes
  .via(Framing.delimiter(ByteString("\n"), maximumFrameLength = 256))
  .map(transformEachLine)
  .runWith(FileIO.toPath(new File("/tmp/example.out").toPath))
def transformEachLine(line: ByteString): ByteString = ???
```

然而有时候，整个正文数据流可能需要被转化成一个 String 实例（就是说把所有的内容都读入到内存中）。Akka HTTP 提供了一个特殊的 toStrict(timeout)函数用于主动地消费正文数据并全部加载到内存中。

```
val response: HttpResponse = ???
// toStrict 强制所以正文数据都读入到内存中。
val strictEntity: Future[HttpEntity.Strict] = response.entity.toStrict(3.seconds)
// 虽然我们使用同样 API 去消费 dataBytes, 但现在他们都已经全部吸收到内存里了:
val transformedData: Future[ExamplePerson] =
  strictEntity flatMap { e =>
    e.dataBytes
      .runFold(ByteString.empty) { case (acc, b) => acc ++ b }
      .map(parse)
  }
```

2、丢弃 Response Entity

discardEntityBytes()方法提供了丢弃 response entity 的能力。它的实现很简单，就是把进来的字节码直接连到 Sink.ignore 上。

下面的两段代码是等价的：

```
val response1: HttpResponse = ???        // 接收一个 HTTP 调用的返回
val discarded: DiscardedEntity = response1.discardEntityBytes()
discarded.future.onComplete { done => println("Entity discarded completely!") }

val response1: HttpResponse = ???       // 接收一个 HTTP 调用的返回
val discardingComplete: Future[Done] = response1.entity.dataBytes.runWith(Sink.ignore)
discardingComplete.onComplete(done => println("Entity discarded completely!"))
```

二、服务端处理 Http Entity 流

1、消费 Request Entity

(1) 最简单的处理 request entity 的方式是将它转换成实际的 domain object。用 entity 方法可以做到。

示例 1：

```
val route =
  path("bid") {
    put {
      entity(as[Bid]) { bid =>
        // incoming entity is fully consumed and converted into a Bid
        complete("The bid was: " + bid)
      }
    }
  }
```

示例 2：

如果请求数据是 json，我们可以使用 spray-json，将其 unmarshall 为对应的 class instance。

```
case class Person(name: String, favoriteNumber: Int)
object PersonJsonSupport extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val PortofolioFormats = jsonFormat2(Person)
}
import PersonJsonSupport._
val route = post {
  entity(as[Person]) { person =>
    complete(s"Person: ${person.name} - favorite number: ${person.favoriteNumber}")
  }
}
```

示例 3：

如果请求数据是 json，但是不想转化成 class instance，而是得到一个 Json Object，也是可以的。

```
val route = post {
  entity(as[JsValue]) { json =>
    complete(s"Person: ${json.asJsObject.fields("name")} - favorite number: ${json.asJsObject.fields("favoriteNumber")}")
  }
}
```

(2) 可以从 request entity 中获得 dataBytes。

```
val route =
  (put & path("lines")) {
    withoutSizeLimit {
      extractDataBytes { bytes =>
          // .....
      }
    }
  }
```

2、抛弃 Request Entity

需要注意“抛弃”表示的是：http connection 不会中断，请求数据仍会流向服务端；但是服务端对数据不感兴趣，即不会消费数据。
通过对收到的 HttpRequest 调用 discardEntityBytes 方法可以显示地抛弃请求数据。

```
val route =
  (put & path("lines")) {
    withoutSizeLimit {
      extractRequest { r: HttpRequest =>
        val finishedWriting = r.discardEntityBytes().future

        // we only want to respond once the incoming data has been handled:
        onComplete(finishedWriting) { done =>
          complete("Drained all data from connection... (" + done + ")")
        }
      }
    }
  }
```

3、取消 Request Entity

取消和抛弃的区别是：取消 Request Entity 会立即中断来自客户端的 http connection，而抛弃不会。
This can be done by attaching the incoming **entity.dataBytes** to a Sink.cancelled()。
示例：

```
val route =
  (put & path("lines")) {
    withoutSizeLimit {
      extractDataBytes { data =>
        // 关闭 connection 方式 1 (粗暴的方式)
        // 我们认定 request 是非法的，粗暴的关闭 connection
        data.runWith(Sink.cancelled)

        // 关闭 connection 方式 2 (优雅的方式)
        // 在 request 请求周期之后，给客户端返回一个“Connection:Close”响应头
        respondWithHeader(Connection("close"))
        complete(StatusCodes.Forbidden -> "Not allowed!")
      }
    }
  }
```

Low-Level Server-Side Api

low-level server 主要是提供 HTTP/1.1 服务端的主要功能：

- 管理 Connection
- parse 和 render 消息体和消息头
- 超时管理（主要是 request 和 connection）
- response 支持

Start and Stop

```
val serverSource: Source[Http.IncomingConnection, Future[Http.ServerBinding]] = Http().bind(interface = "localhost", port = 8080)

val bindingFuture: Future[Http.ServerBinding] =
  serverSource
  .to(Sink.foreach { connection => // foreach materializes the source
    println("Accepted new connection from " + connection.remoteAddress)
    // ... and then actually handle the connection
  })
  .run()
```

通过调用 Http.ServerBinding 实例的 unbind()方法可以释放绑定。

请求-响应

当收到了一个新的 connection，它会被当作 Http.IncomingConnection。Http.IncomingConnection 包含了客户端远程地址。

可以从 connection 取出 HttpRequest，我们可以提供几种 requestHandle 来处理 request，如下面所示

- a Flow[HttpRequest, HttpResponse, _] for handleWith,
- a function HttpRequest => HttpResponse for handleWithSyncHandler,
- a function HttpRequest => Future[HttpResponse] for handleWithAsyncHandler.

示例：

```
val serverSource = Http().bind(interface = "localhost", port = 8080)

val requestHandler: HttpRequest => HttpResponse = {
  case HttpRequest(GET, Uri.Path("/"), _, _, _) =>
    HttpResponse(entity = HttpEntity(
      ContentTypes.`text/html(UTF-8)`,
      "<html><body>Hello world!</body></html>"))

  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>
    HttpResponse(entity = "PONG!")
```

```
  case HttpRequest(GET, Uri.Path("/crash"), _, _, _) =>
    sys.error("BOOM!")
  case r: HttpRequest =>
    r.discardEntityBytes() // important to drain incoming HTTP Entity stream
    HttpResponse(404, entity = "Unknown resource!")
}

val bindingFuture: Future[Http.ServerBinding] =
  serverSource
  .to(Sink.foreach { connection =>
    println("Accepted new connection from " + connection.remoteAddress)
    connection handleWithSyncHandler requestHandler
    // this is equivalent to
    // connection handleWith { Flow[HttpRequest] map requestHandler }
  })
  .run()
```

处理 Http Server failure

发生 failure 的情况主要有以下几种：

- Failure to bind to the specified address/port,
- Failure while accepting new IncommingConnection, for example when the OS has run out of file descriptors or memory,
- Failure while handling a connection, for example if the incoming HttpRequest is malformed.

下面就介绍如何处理不同场景下的 failure，以及这些场景下的 failure 会发生什么。

解决第一种 fialure：

对于第一种 failure 的情况，可能是要绑定的端口已经被占用，或者端口是专有的(只能被 root 用户使用)。如果是这两种情况，failure 会立即发生，可以用下面的方式进行处理。

```
val (host, port) = ("localhost", 80)
val serverSource = Http().bind(host, port)

val bindingFuture: Future[ServerBinding] = serverSource
  .to(handleConnections) // Sink[Http.IncomingConnection, _]
  .run()

bindingFuture.onFailure {       // 因为 bind 操作返回的结果类型是 Future，所以可以添加一个 onFailure 回调监听是否 bind 成功
  case ex: Exception =>
    log.error(ex, "Failed to bind to {}:{}!", host, port)
}
```

解决第二种 failure：

由于某些 failure，导致服务端无法接受新的 http connection。可以用下面的方式解决：

```
val (host, port) = ("localhost", 8080)
val serverSource = Http().bind(host, port)

val failureMonitor: ActorRef = system.actorOf(MyExampleMonitoringActor.props)
```

```
val reactToTopLevelFailures = Flow[IncomingConnection]
  .watchTermination()((_, termination) => termination.onFailure {
    case cause => failureMonitor ! cause    // 将 failure 发送给新创建的 actor，由 actor 来决定是重启服务端还是终止 ActorSystem
  })

serverSource
  .via(reactToTopLevelFailures)
  .to(handleConnections) // Sink[Http.IncomingConnection, _]
  .run()
```

解决第三种 failure：

发生这种 failure 的原因可能是客户端粗暴地中断了底层 TCP 连接。

下面是解决办法：

```
val (host, port) = ("localhost", 8080)
val serverSource = Http().bind(host, port)

val reactToConnectionFailure = Flow[HttpRequest]
  .recover[HttpRequest] {
    case ex =>
      // handle the failure somehow
      throw ex
  }

val httpEcho = Flow[HttpRequest]
  .via(reactToConnectionFailure)     // 解决方法与上一种类似，区别是将 failure handler 作用于 request handler
  .map { request =>
    // simple streaming (!) "echo" response:
    HttpResponse(entity = HttpEntity(ContentTypes.`text/plain(UTF-8)`, request.entity.dataBytes))
  }

serverSource
  .runForeach { con =>
    con.handleWith(httpEcho)
  }
```

第四种 failure：

除了上面介绍的三种 failure，还有第四种 failure：在 route 执行期间抛出的异常会一直向上抛出，直到被 handleExceptions 方法捕获。

通过提供一个自定义的 ExceptionHandler，handleExceptions 方法会将捕获的异常委托给自定义的 ExceptionHandler 处理。

ExceptionHandler 看起来像这样：

```
trait ExceptionHandler extends PartialFunction[Throwable, Route]
```

对于 handleExceptions 方法没能处理的异常，会继续向上抛，直到会被最顶层的 handler 处理。

示例：

```
val myExceptionHandler = ExceptionHandler {
  case _: ArithmeticException =>
```

```
    extractUri { uri =>
      println(s"Request to $uri could not be handled normally")
      complete(HttpResponse(InternalServerError, entity = "Bad numbers, bad result!!!"))
    }
}

object MyApp extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()

  val route: Route =
    handleExceptions(myExceptionHandler) {
      // ... some route structure
    }

  Http().bindAndHandle(route, "localhost", 8080)
}
```

隐式的用法：

```
implicit def myExceptionHandler: ExceptionHandler =
  ExceptionHandler {
    case _: ArithmeticException =>
      extractUri { uri =>
        println(s"Request to $uri could not be handled normally")
        complete(HttpResponse(InternalServerError, entity = "Bad numbers, bad result!!!"))
      }
  }

object MyApp extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()

  val route: Route =                // 不需要提供 handleExceptions 方法
  // ... some route structure

  Http().bindAndHandle(route, "localhost", 8080)
```

一、Route

Route 是 Akka HTTP Routing DSL 的核心概念。
它的类型声明如下：

type Route = RequestContext ⇒ Future[RouteResult]

RequestContext 介绍：

The request context wraps an HttpRequest instance to enrich it with additional information that are typically required by the routing logic, like an ExecutionContext, Materializer, LoggingAdapter and the configured RoutingSettings. It also contains the unmatchedPath, a value that describes how much of the request URI has not yet been matched by a Path Directive.

RequestContext 是不可变的，但是它包含几个方法可以用来将它转换成可变的副本。

RouteResult 介绍：

RouteResult is a simple abstract data type (ADT) that models the possible non-error results of a Route.
它被定义为下面这样：

```
sealed trait RouteResult
object RouteResult {
  final case class Complete(response: HttpResponse) extends RouteResult
  final case class Rejected(rejections: immutable.Seq[Rejection]) extends RouteResult
}
```

通常，我们不需要显示地创建 RouteResult 实例，而是通过 complete、reject、fail 方法来创建。

当一个 route 收到了一个 request，它能做如下其中之一：

- 调用 requestContext.complete(...)方法得到返回值来 complete request；
- 调用 requestContext.reject(....)方法得到返回值来 reject request；
- 调用 requestContext.fail(...)方法得到返回来 fail request，或者仅仅抛出异常。

**1、RouteDirective　　*

(1) complete

complete 方法的各种不同定义形式：

- def complete[T :ToResponseMarshaller](value: T): StandardRoute
- def complete(response: HttpResponse): StandardRoute
- def complete(status: StatusCode): StandardRoute
- def complete[T :Marshaller](status: StatusCode, value: T): StandardRoute
- def complete[T :Marshaller](status: Int, value: T): StandardRoute
- def complete[T :Marshaller](status: StatusCode, headers: Seq[HttpHeader], value: T): StandardRoute
- def complete[T :Marshaller](status: Int, headers: Seq[HttpHeader], value: T): StandardRoute

使用示例：

```
val route =
  path("a") {
    complete(HttpResponse(entity = "foo"))
  } ~
  path("b") {
    complete(StatusCodes.OK)
  } ~
  path("c") {
    complete(StatusCodes.Created -> "bar")
  } ~
  path("d") {
    complete(201 -> "bar")
  } ~
  path("e") {
    complete(StatusCodes.Created, List(`Content-Type`(`text/plain(UTF-8)`)), "bar")
  } ~
  path("f") {
    complete(201, List(`Content-Type`(`text/plain(UTF-8)`)), "bar")
  }
```

(2) reject

reject 方法的不同定义形式：

- def reject: StandardRoute
- def reject(rejections: Rejection*): StandardRoute

使用示例：

```
val route =
  path("a") {
    reject          // don't handle here, continue on
  } ~
    path("a") {
      complete("foo")
    } ~
    path("b") {
      // trigger a ValidationRejection explicitly.    rather than through the `validate` directive
```

```
        reject(ValidationRejection("Restricted!"))
    }
```

(3) failWith

failWith 方法的定义形式：

- def failWith(error: Throwable): StandardRoute

此方法描述：

failWith 显示地触发一个异常，并将触发的异常一直向上抛，直到被 handleException 捕获并被它的 ExceptionHandler 处理。

之所以用 failWith 触发异常，而不是用简单的 throw 语句抛出异常，是因为使用 failWith 能捕获异步处理的 route 发生的异常(在另一个线程或者另一个 actor 中)。

(4) redirect

redirect 方法的定义形式：

- def redirect(uri: Uri, redirectionType: Redirection): StandardRoute

此方法的描述：

调用此方法，用给定的 URI 和 status code，向客户端返回一个**重定向**响应。

使用示例：

```
val route =
  pathPrefix("foo") {
    pathSingleSlash {
      complete("yes")
    } ~
      pathEnd {
        redirect("/foo/", StatusCodes.PermanentRedirect)
      }
  }
```

二、Rejection

如果接受到了一个 request，但是被一个 route reject 了，那么这个 request 仍会流向其它的 route，可能会被其它的 route complete。
若最终 request 没能被 route stracture complete，那么 **handleRejections** 会将 rejection 的集合转换成 HttpResponse。

一个 rejection 描述了为什么一个 route 不能处理 request 的原因。一个 rejection 被建模为 Rejection 类型的实例。

handleRejection directive 将 rejection 集合的转换工作委托给它的参数，一个 **RejectionHandler**，它的定义如下：
trait RejectionHandler extends (immutable.Seq[Rejection] ⇒ Option[Route])
因为 RejectionHandler 返回一个 Option 类型的结果，所以它能选择是否要处理当前的 rejection 集合。
如果返回 None，表示 request 能继续流向其它的 route。

自定义 RejectionHandler
示例：

```
implicit def myRejectionHandler =
  RejectionHandler.newBuilder()
    .handle { case MissingCookieRejection(cookieName) =>
      complete(HttpResponse(BadRequest, entity = "No cookies, no service!!!"))
    }
    .handle { case AuthorizationFailedRejection =>
      complete((Forbidden, "You're out of your depth!"))
    }
    .handle { case ValidationRejection(msg, _) =>
      complete((InternalServerError, "That wasn't valid! " + msg))
    }
    .handleAll[MethodRejection] { methodRejections =>
      val names = methodRejections.map(_.supported.name)
      complete((MethodNotAllowed, s"Can't do that! Supported: ${names mkString " or "}!"))
    }
    .handleNotFound { complete((NotFound, "Not here!")) }
    .result()

object MyApp extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()

  val route: Route =
    // ... some route structure

  Http().bindAndHandle(route, "localhost", 8080)
}
```

三、Exception Handling

在 route 执行期间抛出的异常会一直向上抛出，直到被 handleExceptions 方法捕获。

handleException 将它的工作委托给它的参数 ExceptionHandler 来做。

ExceptionHandler 的定义：

trait ExceptionHandler extends **PartialFunction**[Throwable, Route]

1、自定义 ExceptionHandler

自定义 ExceptionHandler：

```
val myExceptionHandler = ExceptionHandler {
  case _: ArithmeticException =>
    extractUri { uri =>
      println(s"Request to $uri could not be handled normally")
      complete(HttpResponse(InternalServerError, entity = "Bad numbers, bad result!!!"))
    }
}

object MyApp extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()
  val route: Route =
    handleExceptions(myExceptionHandler) {
      // ... some route structure
    }
  Http().bindAndHandle(route, "localhost", 8080)
}
```

一种隐式的用法：

```
implicit def myExceptionHandler: ExceptionHandler =
  ExceptionHandler {
    case _: ArithmeticException =>
      extractUri { uri =>
        println(s"Request to $uri could not be handled normally")
        complete(HttpResponse(InternalServerError, entity = "Bad numbers, bad result!!!"))
      }
  }

object MyApp extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()
  val route: Route =            // 不需要提供 handleExceptions 方法
  // ... some route structure
  Http().bindAndHandle(route, "localhost", 8080)
}
```

2、默认的 ExceptionHandler

- A default ExceptionHandler is used if no custom instance is provided.
- It will handle every NonFatal throwable, write its stack trace and complete the request with InternalServerError (500) status code.
- The message body will contain a string obtained via Throwable#getMessage call on the exception caught.
- In case getMessage returns null (which is true for e.g. NullPointerException instances), the class name and a remark about the message being null are included in the response body.
- Note that IllegalRequestExceptions`stack traces are not logged, since instances of this class normally contain enough information to provide a useful error message.

四、Directive

directives 的结构类似下面这样：

```
name(arguments) { extractions =>
  ... // inner route
}
```

directives 有一个名称、0 或多个参数、一个可选的内部 route。

另外，directive 能抽取多个值，使它们能作为内部 route 的函数参数。

Directive 能做什么：

- 对接受到的 RequestContext，在将它传递给内部 route 之前，对它进行转换（例如修改 request）；
- 根据一些逻辑过滤 RequestContext；
- 从 RequestContext 中抽取出一些值，并将这些值作为内部 route 的函数参数；
- complete the request

1、Directive 组合操作符

1、可以使用~操作符将 get 或 put 等 directive 组合在一起。

例：

```
val route: Route =
  path("order" / IntNumber) { id =>
    get {
      complete {
        "Received GET request for order " + id
      }
    } ~
    put {
      complete {
        "Received PUT request for order " + id
      }
    }
  }
```

2、也可以使用 | 操作符：

例：

```
val route =
  path("order" / IntNumber) { id =>
    (get | put) {
      extractMethod { m =>
        complete(s"Received ${m.name} request for order $id")
      }
    }
  }
```

3、使用 & 操作符：

```
val route =
  (path("order" / IntNumber) & (get | put) & extractMethod) { (id, m) =>
    complete(s"Received ${m.name} request for order $id")
  }
```

2、BaseDirective *

BaseDirective 提供了四种功能类型的 directives，如下面的所示。

(1) 给内部 route 提供值

1、extract

extract 用来构建 **Custom Directive**，从 **RequestContext** 中抽取出数据，并将数据传给内部的 route。

使用示例：

```
val uriLength = extract(_.request.uri.toString.length)
val route =
  uriLength { len =>
    complete(s"The length of the request URI is $len")
  }
```

测试：

```
Get("/abcdef") ~> route ~> check {
  responseAs[String] shouldEqual "The length of the request URI is 25"
}
```

2、extractDataBytes

从 **RequestContext** 中抽取出实体数据作为 **Source[ByteString, Any]**类型。

使用示例：

```
val route =
  extractDataBytes { data ⇒
    val sum = data.runFold(0) { (acc, i) ⇒ acc + i.utf8String.toInt }
    onSuccess(sum) { s ⇒
      complete(HttpResponse(entity = HttpEntity(s.toString)))
    }
  }
```

测试：

```
val dataBytes = Source.fromIterator(() ⇒ Iterator.range(1, 10).map(x ⇒ ByteString(x.toString)))
Post("/abc", HttpEntity(ContentTypes.`text/plain(UTF-8)`, data = dataBytes)) ~> route ~> check {
  responseAs[String] shouldEqual "45"
}
```

3、extractRequestContext

抽取出底层的 **RequestContext**。

```
val route =
  extractRequestContext { ctx =>
    ctx.log.debug("Using access to additional context availablethings, like the logger.")
    val request = ctx.request          // 得到 HttpRequest
    complete(s"Request method is ${request.method.name} and content-type is ${request.entity.contentType}")
  }
```

测试：

```
Post("/", "text") ~> route ~> check {
  responseAs[String] shouldEqual "Request method is POST and content-type is text/plain; charset=UTF-8"
}
```

4、extractRequest

抽取出 **HttpRequest** 实例。

使用示例：

```
val route =
  extractRequest { request =>
    complete(s"Request method is ${request.method.name} and content-type is ${request.entity.contentType}")
  }
```

测试：

```
Post("/", "text") ~> route ~> check {
  responseAs[String] shouldEqual "Request method is POST and content-type is text/plain; charset=UTF-8"
}
```

5、extractRequestEntity

从 **RequestContext** 中抽取 **RequestEntity**。

示例：

```
val route =
  extractRequestEntity { entity =>
    complete(s"Request entity content-type is ${entity.contentType}")
  }
```

测试：

```
val httpEntity = HttpEntity(ContentTypes.`text/plain(UTF-8)`, "req")
Post("/abc", httpEntity) ~> route ~> check {
  responseAs[String] shouldEqual s"Request entity content-type is text/plain; charset=UTF-8"
}
```

6、extractUnmatchedPath

从 **RequestContext** 中抽取出还未被 PathDirective 匹配的剩余 path。

```
val route =
  pathPrefix("abc") {
    extractUnmatchedPath { remaining =>
      complete(s"Unmatched: '$remaining'")
    }
  }
```

测试：

```
Get("/abc/456") ~> route ~> check {
  responseAs[String] shouldEqual "Unmatched: '/456'"
}
```

7、extractUri

从 request 中抽取出 **URI**。

```
val route =
  extractUri { uri =>
    complete(s"Full URI: $uri")
  }
```

测试：

```
Get("/test") ~> route ~> check {
  responseAs[String] shouldEqual "Full URI: http://example.com/test"
}
```

8、provide

方法声明：def provide[T](value: T): Directive1[T]

方法描述：给内部的 route 提供一个常量值。

示例：

```
def providePrefixedString(value: String): Directive1[String] = provide("prefix:" + value)
val route =
  providePrefixedString("test") { value =>
    complete(value)
  }
```

测试：

```
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "prefix:test"
}
```

9、extractClientIP

从 request 中抽取出 **X-Forwarded-For、Remote-Address、X-Real-IP** 这几个 headers 表示的客户端 ip，并作为 HttpIp 类型实例供内部 route 使用。

示例：

```
val route = extractClientIP { ip =>
  complete("Client's ip is " + ip.toOption.map(_.getHostAddress).getOrElse("unknown"))
}
```

测试：

```
Get("/").withHeaders(`Remote-Address`(RemoteAddress(InetAddress.getByName("192.168.3.12")))) ~> route ~> check {
  responseAs[String] shouldEqual "Client's ip is 192.168.3.12"
}
```

10、withoutSizeLimit

方法声明：def withoutSizeLimit: Directive0

方法描述：跳过 reuqest entity 长度的检查。

11、withSizeLimit

方法声明：def withSizeLimit(maxBytes: Long): Directive0

方法描述：限制 request entity 的长度，这个方法的值会覆盖在配置文件配置的属性：akka.http.parsing.max-content-length

示例：

```
val route = withSizeLimit(500) {
  entity(as[String]) { _ ⇒
    complete(HttpResponse())
  }
}
```

测试：

```
def entityOfSize(size: Int) =
  HttpEntity(ContentTypes.`text/plain(UTF-8)`, "0" * size)

Post("/abc", entityOfSize(501)) ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.BadRequest
}
```

(2) 对 Request 做一些转换

1、mapRequest

方法声明：def mapRequest(f: HttpRequest ⇒ HttpRequest): Directive0

方法描述：mapRequest 方法接受一个函数类型的参数，将函数作用于接受到的 HttpRequest。

示例：

```
def transformToPostRequest(req: HttpRequest): HttpRequest = req.copy(method = HttpMethods.POST)
val route =
  mapRequest(transformToPostRequest) {
    extractRequest { req =>
      complete(s"The request method was ${req.method.name}")
    }
  }
```

测试：

```
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "The request method was POST"
}
```

(3) 对 Response 做一些转换

1、mapResponse

方法声明：def mapResponse(f: HttpResponse ⇒ HttpResponse): Directive0

方法描述：对 route 返回的 **HttpResponse** 做些转换再返回给客户端。

示例：

```
def overwriteResultStatus(response: HttpResponse): HttpResponse =
  response.copy(status = StatusCodes.BadGateway)
val route = mapResponse(overwriteResultStatus)(complete("abc"))
```

测试：

```
Get("/abcdef?ghi=12") ~> route ~> check {
  status shouldEqual StatusCodes.BadGateway
}
```

2、mapResponseEntity

方法声明：def mapResponseEntity(f: ResponseEntity ⇒ ResponseEntity): Directive0

方法描述：接受一个函数类型的参数，将函数作用于 route 返回的 HttpResponse 中的 **ResponseEntity**。

示例：

```
def prefixEntity(entity: ResponseEntity): ResponseEntity = entity match {
  case HttpEntity.Strict(contentType, data) =>
    HttpEntity.Strict(contentType, ByteString("test") ++ data)
  case _ => throw new IllegalStateException("Unexpected entity type")
}
val route = mapResponseEntity(prefixEntity) (complete("abc"))
```

测试：

```
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "testabc"
}
```

3、mapResponseHeaders

方法声明：def mapResponseHeaders(f: immutable.Seq[HttpHeader] ⇒ immutable.Seq[HttpHeader]): Directive0

方法描述：对内部 route 返回的 HttpResponse 中的 **headers** 做一些转换。

(4) 对 RouteResult 做一些转换

1、cancelRejection

方法声明：def cancelRejection(rejection: Rejection): Directive0

方法描述：

在内部 route 发生 rejection 时，**默认的 handleRejection** 方法会将它们转换为 error response。

我们也可以提供自定义的 RejectionHandler 对内部 route 发生的 rejection 做处理。

但是，当我们不想处理内部 route 发生的 rejection，应该怎么做呢？可以使用 cancelRejection 方法。

示例：

```
val route =
  cancelRejection(MethodRejection(HttpMethods.POST)) {
    post {
      complete("Result")
    }
  }
```

测试：

```
Get("/") ~> route ~> check {
  rejections shouldEqual Nil
  handled shouldEqual false
}
```

2、cancelRejections

方法声明：

- def cancelRejections(classes: Class[_]*): Directive0
- def cancelRejections(cancelFilter: Rejection ⇒ Boolean): Directive0

方法描述：

cancelRejections 方法与 cancelRejection 类似，区别是它是批量的取消 Rejection。

如果 cancelRejections 方法的参数是一个函数，当函数返回返回 true，表示取消某一种 Rejection。

示例：

```
def isMethodRejection: Rejection => Boolean = {
  case MethodRejection(_) => true
  case _                  => false
}
val route =
  cancelRejections(isMethodRejection) {
    post {
      complete("Result")
    }
  }
```

测试：

```
Get("/") ~> route ~> check {
  rejections shouldEqual Nil
  handled shouldEqual false
}
```

3、MethodDirective

extractMethod：从 request context 中抽取出 HttpMethod。

使用示例：

```
val route =
  get {
    complete("This is a GET request.")
  } ~
    extractMethod { method =>
      complete(s"This ${method.name} request, clearly is not a GET!")
    }
```

测试：

```
Put("/") ~> route ~> check {
  responseAs[String] shouldEqual "This PUT request, clearly is not a GET!"
}
```

method

方法声明：

```
def method(httpMethod: HttpMethod): Directive0 =
  extractMethod.flatMap[Unit] {
    case `httpMethod` ⇒ pass
    case _            ⇒ reject(MethodRejection(httpMethod))
  } & cancelRejections(classOf[MethodRejection])
```

方法描述：

要求 request 的请求方法匹配指定的 HttpMethod。如果请求方法不匹配，request 会被 reject 通过 **MethodRejection**，MethodRejection 会被 RejectionHandler 转义成 **405 Method Not Allowed** 的 response。

使用示例：

```
val route = method(HttpMethods.PUT) { complete("This is a PUT request.") }
```

测试：

```
Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.MethodNotAllowed
  responseAs[String] shouldEqual "HTTP method not allowed, supported methods: PUT"
}
```

当然，还有其它像 get、post、delete、options、put 等方法。

**4、PathDirective　　*

(1) 各种 path directive

path

方法声明：def path[L](pm: PathMatcher[L]): Directive[L]

方法描述：

对给定的 PathMatcher 类型参数值进行完全地匹配。并且会从 URI path 中抽取出 0 或多个值。

如果匹配失败，这个 request 将会以一个空的 rejection 集合被 reject。

使用示例：

```
val route =
  path("foo") {                  // 判断请求路径是否是完整的 /foo
    complete("/foo")
  } ~
    path("foo" / "bar") {     // 判断请求路径是否是完整的 /foo/bar
      complete("/foo/bar")
    } ~
    pathPrefix("ball") {
      pathEnd {                          // 如果请求路径到 ball 就结束了，则进入此代码快
        complete("/ball")
      } ~
        path(IntNumber) { int =>       // 如果请求路径 ball 后面还有路径，则进入此代码快；并抽取得到 ball 后面路径的值
          complete(if (int % 2 == 0) "even ball" else "odd ball")
        }
    }
```

测试：

```
Get("/foo") ~> route ~> check {
  responseAs[String] shouldEqual "/foo"
}
Get("/foo/bar") ~> route ~> check {
  responseAs[String] shouldEqual "/foo/bar"
}
Get("/ball/1337") ~> route ~> check {
  responseAs[String] shouldEqual "odd ball"
}
```

pathPrefix

方法声明：def pathPrefix[L](pm: PathMatcher[L]): Directive[L]

方法描述：匹配并消耗请求路径中未匹配部分的前缀。并且会从 URI path 中抽取出 0 或多个值。

pathPrefixTest

方法声明：def pathPrefixTest[L](pm: PathMatcher[L]): Directive[L]

方法描述：检查未匹配路径的部分是否有方法参数给定的前缀。并且会从 URI path 中抽取出 0 或多个值。

使用示例：

```
val completeWithUnmatchedPath =
  extractUnmatchedPath { p =>
    complete(p.toString)
  }
```

```
val route =
  pathPrefixTest("foo" | "bar") {        // 检查路径的前缀是否为 /foo 或者 /bar
    pathPrefix("foo") { completeWithUnmatchedPath } ~
      pathPrefix("bar") { completeWithUnmatchedPath }
  }
```

测试：

```
Get("/foo/doo") ~> route ~> check {
  responseAs[String] shouldEqual "/doo"
}
Get("/bar/yes") ~> route ~> check {
  responseAs[String] shouldEqual "/yes"
}
```

pathSuffix

方法声明：def pathSuffix[L](pm: PathMatcher[L]): Directive[L]

方法描述：**匹配**并**消耗**请求路径中未匹配部分的后缀。并且会从 URI path 中抽取出 0 或多个值。

注意：

pathSuffix 方法是按相反的方向进行匹配，例如 pathSuffix("baz" / "bar") 实际上匹配的是 /foo/bar/baz。

使用示例：

```
val completeWithUnmatchedPath =
  extractUnmatchedPath { p =>
    complete(p.toString)
  }
val route =
  pathPrefix("start") {
    pathSuffix("end") {
      completeWithUnmatchedPath
    }
  }
```

测试：

```
Get("/start/middle/end") ~> route ~> check {
  responseAs[String] shouldEqual "/middle/"
}
```

pathEnd

方法声明：def pathEnd: Directive0

方法描述：当请求路径已经被 path 或者 pathPrefix 匹配完了，没有剩余的路径要匹配了，则传递 request 到 pathEnd 指向的 route。

pathSingleSlash

方法声明：def pathSingleSlash: Directive0

方法描述：当要匹配的路径只有一个单斜线时，将 request 传递到它所指的 route 中。

pathEndOrSingleSlash

方法声明：def pathEndOrSingleSlash: Directive0

方法描述：

当要匹配的路径是空的或者是个单斜线时，将 request 传递到它所指的 route 中。也就是说**等价于 pathEnd | pathSingleSlash**。

使用示例：

```
val route =
  pathPrefix("foo") {
    pathEndOrSingleSlash {
      complete("/foo")
    } ~
      path("bar") {
        complete("/foo/bar")
      }
  }
```

测试：

```
Get("/foo") ~> route ~> check {
  responseAs[String] shouldEqual "/foo"
}
Get("/foo/") ~> route ~> check {
  responseAs[String] shouldEqual "/foo"
}
Get("/foo/bar") ~> route ~> check {
  responseAs[String] shouldEqual "/foo/bar"
}
```

(2) 各种 PathMatcher

在使用 PathDirective 时，常见的几个组合操作符：

Tilde Operator (~)

The tilde is the most basic combinator. It simply concatenates two matchers into one, i.e if the first one matched (and consumed) the second one is tried. The extractions of both matchers are combined type-safely. For example: "foo" ~ "bar" yields a matcher that is identical to "foobar".

Slash Operator (/)

This operator concatenates two matchers and inserts a Slash matcher in between them. For example: "foo" / "bar" is identical to "foo" ~ Slash ~ "bar".

Pipe Operator (|)

This operator combines two matcher alternatives in that the second one is only tried if the first one did not match. The two sub-matchers must have compatible types. For example: "foo" | "bar" will match either "foo" or "bar".

上面说了好多 PathDirective，凡是方法声明中带参数的，参数的类型一般都为 **PathMatcher**。

下面介绍一些常用的 PathMatcher：

String

You can use a String instance as a PathMatcher0. Strings simply match themselves and extract no value.

Regex

You can use a Regex instance as a PathMatcher1[String], which matches whatever the regex matches and extracts one String value

Map[String, T]

You can use a Map[String, T] instance as a PathMatcher1[T], which matches any of the keys and extracts the respective map value for it.

Slash: PathMatcher0

Matches exactly one path-separating slash (/) character and extracts nothing.

Segment: PathMatcher1[String]

把 path 定义的路径抽取成一个字符串类型的参数。

Remaining: PathMatcher1[String]

Matches and extracts the complete remaining unmatched part of the request's URI path as an (encoded!) String.

IntNumber: PathMatcher1[Int]

把 path 定义的路径抽取成一个 Int 类型的参数。

LongNumber: PathMatcher1[Long]

把 path 定义的路径抽取成一个 Long 类型的参数。

DoubleNumber: PathMatcher1[Double]

把 path 定义的路径抽取成一个 Double 类型的参数.

JavaUUID: PathMatcher1[UUID]

Matches and extracts a java.util.UUID instance.

使用示例：

```
// matches /foo/
path("foo"./)

// matches e.g. /foo/bar123 and extracts "123" as a String
path("foo" / """bar(\d+)""".r)

// identical to path("foo" ~ (PathEnd | Slash))
path("foo" ~ Slash.?)

// matches e.g. /i42 or /hCAFE and extracts an Int
path("i" ~ IntNumber | "h" ~ HexIntNumber)
```

```
// matches /red or /green or /blue and extracts 1, 2 or 3 respectively
path(Map("red" -> 1, "green" -> 2, "blue" -> 3))

// matches anything starting with "/foo" except for /foobar
pathPrefix("foo" ~ !"bar")
```

5、ParameterDirective -- GET　　*

(1) parameters

parameters 方法声明：

- def parameters(param: <ParamDef[T]>): Directive1[T]
- def parameters(params: <ParamDef[T_i]>*): Directive[T_0 :: ... T_i ... :: HNil]
- def parameters(params: <ParamDef[T_0]> :: ... <ParamDef[T_i]> ... :: HNil): Directive[T_0 :: ... T_i ... :: HNil]

方法描述：

parameters 方法过滤出请求路径中 **query parameters**，并且抽取出它们的值。

如果客户端传到服务端的 request，缺少指定的 parameter 或者 parameter value 不符合条件，**则这个 request 会被 reject**。

1、各种 query parameter extraction

query parameters 能被抽取出来当作 String 类型，或者转换成其它的类型。

query parameter extraction 能被修改来标记 query parameter 是 require、optional、repeated、或者用具体的参数值过滤 request。

以下是示例：

"color"

抽取出 parameter name 为 color 的参数值（color **必须存在**），参数值类型为 String

"color".?

抽取出 parameter name 为 color 的**可选**的参数值，参数值的类型为 **Option[String]**

"color" ? "red"

抽取出 parameter name 为 color 的**可选**的参数值，类型为 String，**默认值**为 red

"color" ! "blue"

parameter name 为 color 的参数值**必须**是 blue，并且不抽取任何值

"amount".as[Int]

抽取出 parameter name 为 amount 的参数值，类型转换为 Int

"amount".as(deserializer)

抽取出 parameter name 为 amount 的参数值，类型转换为 Deserializer

"distance".*

抽取出 parameter name 为 distance 的 0 或多个参数值，类型转换为 **Iterable[String]**

"distance".as[Int].*

抽取出 parameter name 为 distance 的 0 或多个参数值，类型转换为 **Iterable[Int]**

"distance".as(deserializer).*

抽取出 parameter name 为 distance 的 0 或多个参数值，类型转换为 **Deserializer**

上面 query parameter extraction 的使用示例：

①Required parameter

```
val route =          // 请求路径必须包含指定 parameter name，否则返回一个 error response
  parameters('color, 'backgroundColor) { (color, backgroundColor) =>
    complete(s"The color is '$color' and the background is '$backgroundColor'")
  }
```

测试：

```
Get("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'red'"
}
Get("/?color=blue") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.NotFound
  responseAs[String] shouldEqual "Request is missing required query parameter 'backgroundColor'"
```

```
}
```
②Optional parameter
```
val route =
  parameters('color, 'backgroundColor.?) { (color, backgroundColor) =>
    val backgroundStr = backgroundColor.getOrElse("<undefined>")
    complete(s"The color is '$color' and the background is '$backgroundStr'")
  }
```
测试：
```
Get("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'red'"
}
Get("/?color=blue") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is '<undefined>'"
}
```

另一个例子：
```
val route =
  parameters('color, 'backgroundColor ? "white") { (color, backgroundColor) =>
    complete(s"The color is '$color' and the background is '$backgroundColor'")
  }
```
测试：
```
Get("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'red'"
}
Get("/?color=blue") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'white'"
}
```

③Deserialized parameter
```
val route =
  parameters('color, 'count.as[Int]) { (color, count) =>
    complete(s"The color is '$color' and you have $count of it.")
  }
```
测试：
```
Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and you have 42 of it."
}
Get("/?color=blue&count=blub") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.BadRequest
  responseAs[String] shouldEqual "The query parameter 'count' was malformed:\n'blub' is not a valid 32-bit signed integer value"
}
```

④Repeated parameter
```
val route =
  parameters('color, 'city.*) { (color, cities) =>
    cities.toList match {
      case Nil              => complete(s"The color is '$color' and there are no cities.")
      case city :: Nil      => complete(s"The color is '$color' and the city is $city.")
      case multiple         => complete(s"The color is '$color' and the cities are ${multiple.mkString(", ")}.")
    }
```

```
  }
```
测试：
```
Get("/?color=blue") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and there are no cities."
}
Get("/?color=blue&city=Chicago") ~> Route.seal(route) ~> check {
  responseAs[String] === "The color is 'blue' and the city is Chicago."
}
Get("/?color=blue&city=Chicago&city=Boston") ~> Route.seal(route) ~> check {
  responseAs[String] === "The color is 'blue' and the cities are Chicago, Boston."
}
```

⑤CSV parameter
```
val route =
  parameter("names".as(CsvSeq[String])) { names =>
    complete(s"The parameters are ${names.mkString(", ")}")
  }
```
测试：
```
Get("/?names=") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are "
}
Get("/?names=Caplin,John") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are Caplin, John"
}
```

2、Case Class Extraction

可以使用 Case Class Extraction，将几个 parametervalue 组装成 case class 的 instance。

在 ParameterDirective 后面调用 as(....)方法，并将目标 case class 的伴生对象传递给 as 方法。底层的 directive 就会将 parameter value 转换成 case class 的 instance。

示例：
```
case class Color(red: Int, green: Int, blue: Int)
val route =
  path("color") {
    parameters('red.as[Int], 'green.as[Int], 'blue.as[Int]).as(Color) { color =>
      // ... route working with the `color` instance
    }
  }
```

在 case class 中添加先决条件 require 方法，为接受到的请求路径中的参数值进行验证。

如果在验证过程中，有参数值不符合条件，那么将会生成 **ValidationRejection**。默认情况下，ValidationRejection 会被 **RejectionHandler** 转换成 **400 Bad Request**。

例：
```
case class Color(name: String, red: Int, green: Int, blue: Int) {
  require(0 <= red && red <= 255, "red color component must be between 0 and 255")
  require(0 <= green && green <= 255, "green color component must be between 0 and 255")
  require(0 <= blue && blue <= 255, "blue color component must be between 0 and 255")
}
```

(2) parameterSeq

方法声明：def parameterSeq: Directive1[immutable.Seq[(String, String)]]

方法描述：Extracts all parameters at once in the original order as (name, value) tuples of type (String, String).

示例：

```
val route =
  parameterSeq { params =>
    def paramString(param: (String, String)): String = s"""${param._1} = '${param._2}'"""
    complete(s"The parameters are ${params.map(paramString).mkString(", ")}")
  }
```

测试：

```
Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are color = 'blue', count = '42'"
}
Get("/?x=1&x=2") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are x = '1', x = '2'"
}
```

(3) parameterMap

方法声明：def parameterMap: Directive1[Map[String, String]]

方法描述：将所有的 query parameters 映射为 Map[String, String]。如果某个 parameter 出现多次，只取最后一个。

使用示例：

```
val route =
  parameterMap { params =>
    def paramString(param: (String, String)): String = s"""${param._1} = '${param._2}'"""
    complete(s"The parameters are ${params.map(paramString).mkString(", ")}")
  }
```

测试：

```
Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are color = 'blue', count = '42'"
}
Get("/?x=1&x=2") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are x = '2'"
}
```

(4) parameterMultiMap

方法声明：def parameterMultiMap: Directive1[Map[String, List[String]]]

方法描述：此方法与 parameterMap 类似，区别是：

如果 query parameter 中某个参数名出现多次，parameterMap 只会取最后一个；

而 parameterMultiMap 会将同一参数名的多个参数值放在 List 中。

6、FormFieldDirective -- POST　　*

前面讲了很多都是与 query parameter 有关。也就是说参数值都是根据 get 请求得来。
如果是 post 请求，请求中的参数值如何抽取。下面将进行介绍。

formFields

方法声明：

- def formFields(field: <FieldDef[T]>): Directive1[T]
- def formFields(fields: <FieldDef[T_i]>*): Directive[T_0 :: ... T_i ... :: HNil]
- def formFields(fields: <FieldDef[T_0]> :: ... <FieldDef[T_i]> ... :: HNil): Directive[T_0 :: ... T_i ... :: HNil]

方法描述：从 post 请求中抽取出指定参数名的参数值。

使用示例：

```
val route =
  formFields('color, 'age.as[Int]) { (color, age) =>
    complete(s"The color is '$color' and the age ten years ago was ${age - 10}")
  }
```

测试：

```
Post("/", FormData("color" -> "blue", "age" -> "68")) ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the age ten years ago was 58"
}
Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.BadRequest
  responseAs[String] shouldEqual "Request is missing required form field 'color'"
}
```

formFieldMap 和 **formFieldMultiMap** 与 parameterMap 和 parameterMultiMap 类似，具体概述不讲了，只看使用示例吧：

```
val route =
  formFieldMap { fields =>
    def formFieldString(formField: (String, String)): String =
      s"""${formField._1} = '${formField._2}'"""
    complete(s"The form fields are ${fields.map(formFieldString).mkString(", ")}")
  }
```

测试：

```
Post("/", FormData("color" -> "blue", "count" -> "42")) ~> route ~> check {
  responseAs[String] shouldEqual "The form fields are color = 'blue', count = '42'"
}
Post("/", FormData("x" -> "1", "x" -> "5")) ~> route ~> check {
  responseAs[String] shouldEqual "The form fields are x = '5'"        // 用 formFieldMap，同名的多个参数值只会取最后一个
}
```

如果想统一地既处理 query parameter 又处理 form parameter，可以使用 anyParams。

7、MarshallingDirective　　*

一、entity

方法声明：def entity[T](um: FromRequestUnmarshaller[T]): Directive1[T]

方法描述：

entity 方法将请求体 **Request Entity** 转换成**给定的类型 T**，并将它传给内部的 route。

entity 方法需要结合 **as 方法**和 akka.http.scaladsl.unmarshalling 一起使用。

下面的示例中，使用 spray-json 去 unmarshalling 一个请求实体数据是 json 的 request：

```
case class Person(name: String, favoriteNumber: Int)
object PersonJsonSupport extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val PortofolioFormats = jsonFormat2(Person)
}

import PersonJsonSupport._
val route = post {
  entity(as[Person]) { person =>
    complete(s"Person: ${person.name} - favorite number: ${person.favoriteNumber}")
  }
}
```

测试：

```
Post("/", HttpEntity(`application/json`, """{ "name": "Jane", "favoriteNumber" : 42 }""")) ~>
  route ~> check {
    responseAs[String] shouldEqual "Person: Jane - favorite number: 42"
  }
```

另一个例子：

如果我们没有自定义一个具体的类型，但是请求实体中仍是 json 格式数据，这时候可以使用 spray-sjon 将 json 格式数据转换成 JsValue。

```
val route = post {
  entity(as[JsValue]) { json =>
    complete(s"Person: ${json.asJsObject.fields("name")} - favorite number: ${json.asJsObject.fields("favoriteNumber")}")
  }
}
```

测试：

```
Post("/", HttpEntity(`application/json`, """{ "name": "Jane", "favoriteNumber" : 42 }""")) ~>
  route ~> check {
    responseAs[String] shouldEqual """Person: "Jane" - favorite number: 42"""
  }
```

8、HeaderDirective *

Header directives 用来从 request 中抽取出 header value。

1、headerValue

方法声明：def headerValue[T](f: HttpHeader ⇒ Option[T]): Directive1[T]

方法描述：将一个函数作用于 request 中所有 header，直到抽取出函数第一个返回的 Some(value)。

如果函数返回 None，request 将会被 reject 通过 NotFound。

示例：

```
def extractHostPort: HttpHeader => Option[Int] = {
  case h: `Host` => Some(h.port)
  case x         => None
}
val route =
  headerValue(extractHostPort) { port =>
    complete(s"The port was $port")
  }
```

测试：

```
Get("/") ~> Host("example.com", 5043) ~> route ~> check {
  responseAs[String] shouldEqual "The port was 5043"
}
Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual NotFound
  responseAs[String] shouldEqual "The requested resource could not be found."
}
```

2、headerValueByName

方法声明：

- def headerValueByName(headerName: Symbol): Directive1[String]
- def headerValueByName(headerName: String): Directive1[String]

从 request 的 headers 中抽取出**指定 name 的 header 的 value**。如果没有找到，request 将会被 reject 通过 **MissingHeaderRejection**。

示例：

```
val route =
  headerValueByName("X-User-Id") { userId =>
    complete(s"The user is $userId")
  }
```

测试：

```
Get("/") ~> RawHeader("X-User-Id", "Joe42") ~> route ~> check {
  responseAs[String] shouldEqual "The user is Joe42"
}
Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual BadRequest
  responseAs[String] shouldEqual "Request is missing required HTTP header 'X-User-Id'"
```

3、optionalHeaderValue

方法声明：

```
def optionalHeaderValue[T](f: HttpHeader ⇒ Option[T]): Directive1[Option[T]]
```

这个方法与 headerValue 类似。区别是 headerValue 直接拿到 Option 中包装的值，而 optionalHeaderValue 需要手动进行模式匹配抽取。

示例：

```
def extractHostPort: HttpHeader => Option[Int] = {
  case h: `Host` => Some(h.port)
  case x         => None
}
val route =
  optionalHeaderValue(extractHostPort) {
    case Some(port) => complete(s"The port was $port")
    case None       => complete(s"The port was not provided explicitly")
  }
```

测试：

```
Get("/") ~> Host("example.com", 5043) ~> route ~> check {
  responseAs[String] shouldEqual "The port was 5043"
}
```

9、RespondWithHeaderDirective

1、respondWithHeaders

方法声明：

- def respondWithHeaders(responseHeaders: HttpHeader*): Directive0
- def respondWithHeaders(responseHeaders: immutable.Seq[HttpHeader]): Directive0

方法描述：用来从 route 中返回的 HttpResponse 添加一些 headers。

使用示例：

```
val route =
  path("foo") {
    respondWithHeaders(RawHeader("Funky-Muppet", "gonzo"), Origin(HttpOrigin("http://akka.io"))) {
      complete("beep")
    }
  }
```

测试：

```
Get("/foo") ~> route ~> check {
  header("Funky-Muppet") shouldEqual Some(RawHeader("Funky-Muppet", "gonzo"))
  header[Origin] shouldEqual Some(Origin(HttpOrigin("http://akka.io")))
  responseAs[String] shouldEqual "beep"
}
```

10、CookieDirective

(1) cookie

方法声明：def cookie(name: String): Directive1[HttpCookiePair]

方法描述：从 request 中抽取出**指定名称的 cookie value**。如果没有找到则 reject request 通过 **MissingCookieRejection**。

使用示例：

```
val route =
  cookie("userName") { nameCookie =>
    complete(s"The logged in user is '${nameCooskie.value}'")
  }
```

测试：

```
Get("/") ~> Cookie("userName" -> "paul") ~> route ~> check {
  responseAs[String] shouldEqual "The logged in user is 'paul'"
}
// missing cookie
Get("/") ~> route ~> check {
  rejection shouldEqual MissingCookieRejection("userName")
```

(2) optionalCookie

方法声明：def optionalCookie(name: String): Directive1[Option[HttpCookiePair]]

方法描述：与 cookie 方法的作用类似。只是它的返回值 Option 类型，需要手动模式匹配抽取出具体的 cookie value。

```
val route =
  optionalCookie("userName") {
    case Some(nameCookie) => complete(s"The logged in user is '${nameCookie.value}'")
    case None             => complete("No user logged in")
  }
```

测试：

```
Get("/") ~> Cookie("userName" -> "paul") ~> route ~> check {
  responseAs[String] shouldEqual "The logged in user is 'paul'"
}
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "No user logged in"
}
```

(3) setCookie

方法声明：def setCookie(first: HttpCookie, more: HttpCookie*): Directive0

方法描述：在 response 添加一个 cookie。

使用示例：

```
val route =
  setCookie(HttpCookie("userName", value = "paul")) {
    complete("The user was logged in")
  }
```

测试：

```
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "The user was logged in"
  header[`Set-Cookie`] shouldEqual Some(`Set-Cookie`(HttpCookie("userName", value = "paul")))
}
```

(4) deleteCookie

方法声明：

- def deleteCookie(first: HttpCookie, more: HttpCookie*): Directive0
- def deleteCookie(name: String, domain: String = "", path: String = ""): Directive0

方法描述：在返回给客户端的 response 中删除指定名称的 cookie。

使用示例：

```
val route =
  deleteCookie("userName") {
    complete("The user was logged out")
  }
```

测试：

```
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "The user was logged out"
  header[`Set-Cookie`] shouldEqual Some(`Set-Cookie`(HttpCookie("userName", value = "deleted", expires = Some(DateTime.MinValue))))
}              // 客户端收到的 cookie 中带有 expires 字段，并且时间为 DateTime.MinValue，表示删除 cookie。
```

11、SchemeDirective

(1) scheme

方法声明：def scheme(name: String): Directive0

方法描述：

匹配给定的 Http 协议，如果不匹配，则 reject request。

示例：

```
val route =
  scheme("http") {
    extract(_.request.uri) { uri =>
      redirect(uri.copy(scheme = "https"), MovedPermanently)
    }
  } ~
  scheme("https") {
    complete(s"Safe and secure!")
  }
```

测试：

```
Get("http://www.example.com/hello") ~> route ~> check {
  status shouldEqual MovedPermanently
  header[Location] shouldEqual Some(Location(Uri("https://www.example.com/hello")))
}
Get("https://www.example.com/hello") ~> route ~> check {
  responseAs[String] shouldEqual "Safe and secure!"
}
```

(2) extractScheme

方法声明：def extractScheme: Directive1[String]

方法描述：从接受到的 request 中抽取出 http 协议。

示例：

```
val route =
  extractScheme { scheme =>
    complete(s"The scheme is '${scheme}'")
  }
```

测试：

```
Get("https://www.example.com/") ~> route ~> check {
  responseAs[String] shouldEqual "The scheme is 'https'"
}
```

12、HostDirective

(1) host

方法声明：

- def host(hostNames: String*): Directive0
- def host(predicate: String ⇒ Boolean): Directive0 = extractHost.require(predicate)
- def host(regex: Regex): Directive1[String]

使用示例：

a、匹配一个 host 列表

```
val route =
  host("api.company.com", "rest.company.com") {
    complete("Ok")
  }
```

测试：

```
Get() ~> Host("rest.company.com") ~> route ~> check {
  status shouldEqual OK
  responseAs[String] shouldEqual "Ok"
}
Get() ~> Host("notallowed.company.com") ~> route ~> check {
  handled shouldBe false
}
```

b、用给定的断言函数匹配 host

```
val shortOnly: String => Boolean = (hostname) => hostname.length < 10

val route =
  host(shortOnly) {
    complete("Ok")
  }
```

测试：

```
Get() ~> Host("short.com") ~> route ~> check {
  status shouldEqual OK
  responseAs[String] shouldEqual "Ok"
}
Get() ~> Host("verylonghostname.com") ~> route ~> check {
  handled shouldBe false
}
```

c、用正则表达式匹配 host

```
val route =
  host("api|rest".r) { prefix =>
    complete(s"Extracted prefix: $prefix")
  } ~
  host("public.(my|your)company.com".r) { captured =>
    complete(s"You came through $captured company")
  }
```

测试：

```
Get() ~> Host("api.company.com") ~> route ~> check {
```

```
    status shouldEqual OK
    responseAs[String] shouldEqual "Extracted prefix: api"
}
Get() ~> Host("public.mycompany.com") ~> route ~> check {
    status shouldEqual OK
    responseAs[String] shouldEqual "You came through my company"
}
```

(2) extractHost

方法声明：def extractHost: Directive1[String]

作用：从接受到的 requet 中抽取出 host，并将它作为一个 string 传给内部的 route。

使用示例：

```
val route =
    extractHost { hn =>
        complete(s"Hostname: $hn")
    }
```

测试：

```
Get() ~> Host("company.com", 9090) ~> route ~> check {
    status shouldEqual OK
    responseAs[String] shouldEqual "Hostname: company.com"
}
```

13、TimeoutDirectives

(1) withRequestTimeout

方法声明：

def withRequestTimeout(timeout: Duration): Directive0

def withRequestTimeout(timeout: Duration, handler: HttpRequest ⇒ HttpResponse): Directive0

def withRequestTimeout(timeout: Duration, handler: Option[HttpRequest ⇒ HttpResponse]): Directive0

方法描述：用来设置服务端的 Request timeout。

使用示例：

```
val route =
  path("timeout") {
    withRequestTimeout(1.seconds) {     // modifies the global akka.http.server.request-timeout for this request
      val response: Future[String] = slowFuture() // very slow
      complete(response)
    }
  }

// 测试
runRoute(route, "timeout").status should ===(StatusCodes.ServiceUnavailable)    // the timeout response
```

也可以将服务端的 Request timeout 和超时的 HttpResponse 设置在一起：

```
val timeoutResponse = HttpResponse(
  StatusCodes.EnhanceYourCalm,
  entity = "Unable to serve response within time limit, please enhance your calm.")

val route =
  path("timeout") {
    // updates timeout and handler at
    withRequestTimeout(1.milli, request => timeoutResponse) {
      val response: Future[String] = slowFuture() // very slow
      complete(response)
    }
  }

// check
runRoute(route, "timeout").status should ===(StatusCodes.EnhanceYourCalm)    // the timeout response
```

(2) withoutRequestTimeout

方法声明：def withoutRequestTimeout: Directive0

方法描述：关闭服务端的 Request timeout 机制。

示例：

```
val route =
  path("timeout") {
    withoutRequestTimeout {
      val response: Future[String] = slowFuture() // very slow
      complete(response)
    }
  }

// no check as there is no time-out, the future would time out failing the test
```

14、CodingDirectives

(1) decodeRequest

作用：

如果收到的 request 是被 **gzip** 或者 **deflate** 压缩过的，则进行**解压缩**。

如果收到的 request 为被 encode，则可以直接被 route 使用。

如果收到的 request 是被其他的方式进行 encode 的，则这个 request 将以 **UnsupportedRequestEncodingRejection** 被 **reject**。

使用示例：

```
val route =
  decodeRequest {
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

// tests:
Post("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'Hello'"
}
Post("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'Hello'"
}
Post("/", "hello uncompressed") ~> `Content-Encoding`(identity) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'hello uncompressed'"
}
```

(2) decodeRequestWith

作用：

如果收到的 request 是被指定 encoder encode，则进行解压缩。

如果收到的 request 的 encode 方式不匹配服务端指定的 encode 方式，则这个 request 将以 **UnsupportedRequestEncodingRejection** 被 **reject**。

使用示例：

```
val route =
  decodeRequestWith(Gzip) {        // 服务端指定了只能解压缩使用了 gzip 的 request
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

// 测试
Post("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'Hello'"
}
Post("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  rejection shouldEqual UnsupportedRequestEncodingRejection(gzip)
```

```
}
Post("/", "hello") ~> `Content-Encoding`(identity) ~> route ~> check {
  rejection shouldEqual UnsupportedRequestEncodingRejection(gzip)
}

val route =
  decodeRequestWith(Gzip, NoCoding) {  // 服务端指定了只能解压缩使用了 gzip 或者非 gzip 非 deflate 的 request
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

// tests:
Post("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'Hello'"
}
Post("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  rejections shouldEqual List(UnsupportedRequestEncodingRejection(gzip), UnsupportedRequestEncodingRejection(identity))
}
Post("/", "hello uncompressed") ~> `Content-Encoding`(identity) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'hello uncompressed'"
}
```

(3) encodeResponse

作用：

如果 request 中带有 Accept-Encoding，并且是 identity、gzip 或者 deflate 中的一种，则使用 request 指定的 encoder 进行 encode response。

如果 request 的请求头中没有 Accept-Encoding，或者 Accept-Encoding 的值是空的，或者 Accept-Encoding 指定的 encoder 不是 identity、gzip 或者 deflate 中的一种，则不会对 response 进行 encode。

使用示例：

```
val route = encodeResponse { complete("content") }

// 测试
Get("/") ~> route ~> check {
  response should haveContentEncoding(identity)
}
Get("/") ~> `Accept-Encoding`(gzip, deflate) ~> route ~> check {
  response should haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding`(deflate) ~> route ~> check {
  response should haveContentEncoding(deflate)
}
Get("/") ~> `Accept-Encoding`(identity) ~> route ~> check {
  response should haveContentEncoding(identity)
}
```

五、支持 WebSocket

数据模型：

因为我们知道，在 WebSocket 中客户端和服务端交换的数据主要是二进制数据和文本数据两种。

所以 Akka Http 提供了两种数据模型：

```
sealed trait Message

sealed trait TextMessage extends Message {
  def textStream: Source[String, _]
}
sealed trait BinaryMessage extends Message {
  def dataStream: Source[ByteString, _]
}
```

六、Route TestKit

Akka Http 为了测试 route logic，专门提供一个测试工具。
要使用这个测试工具，需要先引入 jar 包：**akka-http-testkit**

1、使用

使用这个测试工具最基本的语法是：

```
REQUEST ~> ROUTE ~> check {
  ASSERTIONS
}
```

在 ROUTE 部分，如果只是测试一些普通的 route logic，则直接使用 route 的变量名没问题。但是，当想验证你的 rest 服务是否将 rejection 转换成你期望的 Http response 时，就需要使用 Route.seal 方法。
通过将 route 的变量名传给 Route.seal(...)方法，seal 方法会将作用域中的 ExceptionHandler 或 RejectionHandler 应用于 route 中发生的 exception 或 rejection，将它们转换成期望的 HttpReponse。

Testkit 使用示例：

```
class FullTestKitExampleSpec extends WordSpec with Matchers with ScalatestRouteTest {
  val smallRoute =
    get {
      pathSingleSlash {
        complete {
          "Captain on the bridge!"
        }
      } ~
      path("ping") {
        complete("PONG!")
      }
    }

  // 测试
  "The service" should {
    "return a greeting for GET requests to the root path" in {
      Get() ~> smallRoute ~> check {
        responseAs[String] shouldEqual "Captain on the bridge!"
      }
    }

    "return a 'PONG!' response for GET requests to /ping" in {
      Get("/ping") ~> smallRoute ~> check {
        responseAs[String] shouldEqual "PONG!"
      }
    }

    "leave GET requests to other paths unhandled" in {
      Get("/kermit") ~> smallRoute ~> check {
        handled shouldBe false
```

```
        }
      }

      "return a MethodNotAllowed error for PUT requests to the root path" in {
        Put() ~> Route.seal(smallRoute) ~> check {
          status === StatusCodes.MethodNotAllowed
          responseAs[String] shouldEqual "HTTP method not allowed, supported methods: GET"
        }
      }
    }
}
```

2、Inspector 介绍

上面在对 route logic 进行测试，使用了一些 Inspector，例如：responseAs、handle、status。
其实还有很多的 Inspector，下标列出了全部的 Inspector：

Inspector	Description
charset: HttpCharset	Identical to contentType.charset
chunks: Seq[HttpEntity.ChunkStreamPart]	Returns the entity chunks produced by the route. If the entity is not chunkedreturns Nil.
closingExtension: String	Returns chunk extensions the route produced with its last response chunk. If the response entity is unchunked returns the empty string.
contentType: ContentType	Identical to responseEntity.contentType
definedCharset: Option[HttpCharset]	Identical to contentType.definedCharset
entityAs[T :FromEntityUnmarshaller]: T	Unmarshals the response entity using the in-scopeFromEntityUnmarshaller for the given type. Any errors in the process trigger a test failure.
handled: Boolean	Indicates whether the route produced an HttpResponse for the request. If the route rejected the request handled evaluates to false.
header(name: String):Option[HttpHeader]	Returns the response header with the given name or None if no such header is present in the response.
header[T <: HttpHeader]: Option[T]	Identical to response.header[T]
headers: Seq[HttpHeader]	Identical to response.headers
mediaType: MediaType	Identical to contentType.mediaType
rejection: Rejection	The rejection produced by the route. If the route did not produce exactly one rejection a test failure is triggered.
rejections: Seq[Rejection]	The rejections produced by the route. If the route did not reject the request a test failure is triggered.
response: HttpResponse	The HttpResponse returned by the route. If the route did not return anHttpResponse instance (e.g. because it rejected the request) a test failure is triggered.
responseAs[T:FromResponseUnmarshaller]: T	Unmarshals the response entity using the in-scopeFromResponseUnmarshaller for the given type. Any errors in the process trigger a test failure.
responseEntity: HttpEntity	Returns the response entity.
status: StatusCode	Identical to response.status
trailer: Seq[HttpHeader]	Returns the list of trailer headers the route produced with its last chunk. If the response entity is unchunked returns Nil.

七、启动 Akka Http

1、第一种方式

可以使用 HttpApp 来启动一个 Akka Http 应用。你仅仅需要做：
继承 HttpApp，然后实现 routes()方法即可。
示例：

```
// Server definition
object WebServer extends HttpApp {
  override def routes: Route =
    path("hello") {
      get {
        complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<h1>Say hello to akka-http</h1>"))
      }
    }
}
// Starting the server
WebServer.startServer("localhost", 8080)
```

(1) HttpApp 也提供了很多钩子方法，在成功启动或者失败启动应用时调用。通过重写它们，可以改变它们默认的行为。
例如，通过重写下面的两个钩子，当绑定端口成功或者绑定端口失败时执行自定义的行为：

```
// Server definition
object WebServer extends HttpApp {
  override def routes: Route =
    path("hello") {
      get {
        complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<h1>Say hello to akka-http</h1>"))
      }
    }
  override protected def postHttpBinding(binding: Http.ServerBinding): Unit = {
    super.postHttpBinding(binding)
    val sys = systemReference.get()
    sys.log.info(s"Running on [${sys.name}] actor system")
  }
  override protected def postHttpBindingFailure(cause: Throwable): Unit = {
    println(s"The server could not be started due to $cause")
  }
}
// Starting the server
WebServer.startServer("localhost", 80, ServerSettings(ConfigFactory.load))
```

(2) 如果没有提供自定义的配置，那么 HttpApp 将默认读取 ServerSettings 的配置信息。
可以使用类似下面的方式提供自定义的配置信息：

```
// Server definition
object WebServer extends HttpApp {
  override def routes: Route =
    path("hello") {
      get {
        complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<h1>Say hello to akka-http</h1>"))
```

```
        }
      }
}
// Creating own settings
val settings = ServerSettings(ConfigFactory.load).withVerboseErrorMessages(true)
WebServer.startServer("localhost", 8080, settings)
```

(3) HttpApp 默认使用它自己创建的 ActorSystem。如果我们有自定义好的 ActorSystem，可以将它传递给 startServer 方法。

```
// Server definition
object WebServer extends HttpApp {
  override def routes: Route =
    path("hello") {
      get {
        complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<h1>Say hello to akka-http</h1>"))
      }
    }
}
// Starting the server
val system = ActorSystem("ownActorSystem")
WebServer.startServer("localhost", 8080, system)
system.terminate()
```

(4) 重写中断机制

the default trigger that **shuts down the server** is pressing **ENTER**.

可以通过重写 **waitForShutdownSignal** 方法来重新定义 HttpApp shut down 的信号机制。

示例：

```
// Server definition
object WebServer extends HttpApp {
  override def routes: Route =
    path("hello") {
      get {
        complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<h1>Say hello to akka-http</h1>"))
      }
    }
  override def waitForShutdownSignal(actorSystem: ActorSystem)(implicit executionContext: ExecutionContext): Future[Done] = {
    // 5 秒后 shut down HttpApp
    pattern.after(5 seconds, actorSystem.scheduler)(Future.successful(Done))
  }
}
// Starting the server
WebServer.startServer("localhost", 8080, ServerSettings(ConfigFactory.load))
```

(5) 在 HttpApp shut down 时收到通知

通过重写 HttpApp 的 postServerShutdown 方法来实现。

注意：如果端口绑定失败，导致 HttpApp shut down，这个方法也会被调用。

示例：

```
// Server definition
class WebServer extends HttpApp {
  override def routes: Route =
    path("hello") {
```

```
        get {
          complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<h1>Say hello to akka-http</h1>"))
        }
      }

  private def cleanUpResources(): Unit = ???

  override def postServerShutdown(attempt: Try[Done], system: ActorSystem): Unit = {
    cleanUpResources()
  }
}
// Starting the server
new WebServer().startServer("localhost", 8080, ServerSettings(ConfigFactory.load))
```

2、第二种方式

使用 Http.bindAndHandle 来启动 Akka Http 应用。

在 Akka Http 中处理阻塞操作

因为在使用 Akka Http 时，必须提供 ExecutionContext，即
implicit val executionContext = system.dispatcher

但是如果处理阻塞操作时，和 route infrastructure 处理 HttpRequet 一样，使用相同的 executionContext，就可能会带来问题。
问题：如果 executionContext 中的所有线程都用来处理阻塞的操作，即线程池中所有的线程都被阻塞住，那么就没有空闲线程来处理接受到的 HttpRequest。

解决办法：提供一个专门的 ExecutionContext 来为阻塞操作服务。
在 application.conf 中为阻塞操作配置一个专门的 dispatcher

```
my-blocking-dispatcher {
  type = Dispatcher
  executor = "thread-pool-executor"
  thread-pool-executer {
    fixed-pool-size = 16
  }
  throughput = 100
}
```

一旦，有阻塞操作时，就使用专门的 dispatcher，而不是使用默认的 dispatcher：

```
// GOOD (the blocking is now isolated onto a dedicated dispatcher):
implicit val blockingDispatcher = system.dispatchers.lookup("my-blocking-dispatcher")
val routes: Route = post {
  complete {
    Future {       // uses the good "blocking dispatcher" that we configured,
      // instead of the default dispatcher- the blocking is isolated.
      Thread.sleep(5000)
      System.currentTimeMillis().toString
    }
  }
}
```

根据应用需要，可以选择 3 种不同抽象层次的 API：

Request-Level Client-Side API

Akka HTTP 负责所有连接管理，大多数用户场景推荐使用。

Host-Level Client-Side API

对于每个特定主机/端口端点，Akka HTTP 都会维护一个连接池。Recommended when the user can supply a Source[HttpRequest, NotUsed] with requests to run against a single host over multiple pooled connections.

Connection-Level Client-Side API

用于完全控制何时打开、何时关闭连接，以及请求调度，仅推荐用于特定场景。

Request-Level Client-Site Api

Request-Level API 是最推荐、最方便的使用 Akka-HTTP 的方式。它内部构建在 Host-Level API 之上，可以很容易地从远程服务器上获取 HTTP 响应。

根据你的喜好，你可以选用 flow-based 或者 future-based 两种变体。

注意：

- Request-Level 内部实现了一个 connection pool，这个 connection pool 在一个 actor system 中是共享的。
- 使用 connection pool 的一个后果就是，如果有长时间运行的请求，会阻塞着一个 connection，导致其他请求拿不到 connection。
- 当需要执行长时间的请求时，确保最好不要使用 Request-Level api。可以使用 Connection-Level api，或者分配一个单独的 connection pool 给运行时间较长的请求。

1、Future-based 变体

Http().singleRequest()方法：用于将 HttpRequest 转换成 Future[HttpResponse]类型。

内部实现上，根据请求的 URI，将其分派到（被缓存的）主机连接池上。

注意：

一定要确保消费 response entity stream（类型为 Source[ByteString, Unit]），例如将其导入 Sink（若不需要实体内容，可使用 response.discardEntityBytes()）。

否则 Akka HTTP 会将此视为**反压**信号，并停止读取 TCP 连接。

Host-Level Client-Side Api

注意：

host-level api autonomously manages a configurable pool of connections to one particular target endpoint (i.e. host/port combination).