根据你提供的文档内容，我将为你设计一个健壮、可扩展的Java接口，用于发票查验（单张发票实时查询下载接口）。我们将定义合适的枚举、常量，并封装入参和出参。

### 1. 定义枚举和常量

首先，我们定义一些枚举和常量来表示发票类型、结果代码等。

```java

public enum InvoiceType {

VAT\_SPECIAL("01", "增值税专用发票"),

VAT\_TRANSPORT("02", "货物运输业增值税专用发票"),

VEHICLE\_SALES("03", "机动车销售统一发票"),

VAT\_NORMAL("04", "增值税普通发票"),

VAT\_ELECTRONIC\_SPECIAL("08", "增值税电子专用发票"),

VAT\_ELECTRONIC\_NORMAL("10", "增值税电子普通发票"),

ROLL("11", "卷式发票"),

TOLL("14", "通行费发票"),

SECOND\_HAND\_VEHICLE("15", "二手车销售统一发票"),

ELECTRONIC\_VAT\_SPECIAL("81", "电子发票（增值税专用发票）"),

ELECTRONIC\_VAT\_NORMAL("82", "电子发票（普通发票）"),

PAPER\_VAT\_SPECIAL("85", "纸质发票（增值税专用发票）"),

PAPER\_VAT\_NORMAL("86", "纸质发票（普通发票）"),

RAILWAY\_ELECTRONIC("51", "电子发票（铁路电子客票）"),

AIR\_ELECTRONIC("61", "电子发票（航空运输客票电子行程单）");

private final String code;

private final String description;

InvoiceType(String code, String description) {

this.code = code;

this.description = description;

}

public String getCode() {

return code;

}

public String getDescription() {

return description;

}

}

public enum ResultCode {

SUCCESS("00", "成功"),

FAILURE("01", "失败"),

NO\_DATA("02", "查无数据"),

MISMATCH("03", "查验不一致"),

NOT\_OWNED("04", "此发票非本单位开具或取得");

private final String code;

private final String description;

ResultCode(String code, String description) {

this.code = code;

this.description = description;

}

public String getCode() {

return code;

}

public String getDescription() {

return description;

}

}

```

### 2. 定义请求和响应类

接下来，我们定义请求和响应类来封装入参和出参。

```java

public class InvoiceVerificationRequest {

private InvoiceType invoiceType;

private String invoiceCode;

private String invoiceNumber;

private String issueDate;

private String checkCode;

private BigDecimal invoiceAmount;

private String taxpayerId;

// Getters and Setters

}

public class InvoiceVerificationResponse {

private ResultCode resultCode;

private String resultMessage;

private String verificationInfo;

// Getters and Setters

}

```

### 3. 定义服务接口

我们定义一个服务接口来处理发票查验请求。

```java

public interface InvoiceVerificationService {

InvoiceVerificationResponse verifyInvoice(InvoiceVerificationRequest request);

}

```

### 4. 实现服务接口

我们实现这个接口，处理发票查验的逻辑。

```java

public class InvoiceVerificationServiceImpl implements InvoiceVerificationService {

@Override

public InvoiceVerificationResponse verifyInvoice(InvoiceVerificationRequest request) {

// 这里实现具体的发票查验逻辑

// 例如调用第三方API进行查验

// 假设我们得到了查验结果

ResultCode resultCode = ResultCode.SUCCESS;

String resultMessage = "查验成功";

String verificationInfo = "查验结果信息";

InvoiceVerificationResponse response = new InvoiceVerificationResponse();

response.setResultCode(resultCode);

response.setResultMessage(resultMessage);

response.setVerificationInfo(verificationInfo);

return response;

}

}

```

### 5. 使用示例

最后，我们展示如何使用这个服务。

```java

public class InvoiceVerificationExample {

public static void main(String[] args) {

InvoiceVerificationRequest request = new InvoiceVerificationRequest();

request.setInvoiceType(InvoiceType.VAT\_SPECIAL);

request.setInvoiceCode("123456789012");

request.setInvoiceNumber("12345678901234567890");

request.setIssueDate("20230101");

request.setCheckCode("123456");

request.setInvoiceAmount(new BigDecimal("1000.00"));

request.setTaxpayerId("12345678901234567890");

InvoiceVerificationService service = new InvoiceVerificationServiceImpl();

InvoiceVerificationResponse response = service.verifyInvoice(request);

System.out.println("Result Code: " + response.getResultCode().getCode());

System.out.println("Result Message: " + response.getResultMessage());

System.out.println("Verification Info: " + response.getVerificationInfo());

}

}

```

### 6. 总结

通过以上步骤，我们实现了一个健壮、可扩展的发票查验接口。我们定义了合适的枚举和常量来管理发票类型和结果代码，封装了请求和响应类，并实现了一个服务接口来处理发票查验的逻辑。这个设计可以根据业务需求进行扩展和修改。为了完成发票查验接口的开发，我们需要定义一个健壮且可扩展的接口。以下是实现该接口的步骤：

### 1. 定义枚举和常量

首先，我们需要定义一些枚举和常量来表示发票的类型、状态等信息。

```java

public enum InvoiceType {

NORMAL\_INVOICE("01", "普通发票"),

SPECIAL\_INVOICE("02", "专用发票"),

ELECTRONIC\_INVOICE("03", "电子发票"),

SECOND\_HAND\_CAR\_INVOICE("15", "二手车销售统一发票"),

AIR\_TICKET\_INVOICE("61", "航空运输客票电子行程单"),

RAILWAY\_TICKET\_INVOICE("51", "铁路电子客票");

private final String code;

private final String description;

InvoiceType(String code, String description) {

this.code = code;

this.description = description;

}

public String getCode() {

return code;

}

public String getDescription() {

return description;

}

}

public enum InvoiceStatus {

VALID("1", "有效"),

INVALID("2", "无效"),

CANCELLED("3", "作废");

private final String code;

private final String description;

InvoiceStatus(String code, String description) {

this.code = code;

this.description = description;

}

public String getCode() {

return code;

}

public String getDescription() {

return description;

}

}

```

### 2. 定义请求和响应对象

接下来，我们定义请求和响应对象来封装发票查验的入参和出参。

```java

public class InvoiceVerificationRequest {

private String invoiceCode; // 发票代码

private String invoiceNumber; // 发票号码

private String invoiceDate; // 开票日期

private String checkCode; // 校验码

private BigDecimal totalAmount; // 价税合计

// Getters and Setters

}

public class InvoiceVerificationResponse {

private String invoiceCode; // 发票代码

private String invoiceNumber; // 发票号码

private String invoiceDate; // 开票日期

private BigDecimal totalAmount; // 价税合计

private InvoiceStatus status; // 发票状态

private String sellerName; // 销售方名称

private String sellerTaxId; // 销售方纳税人识别号

private String buyerName; // 购买方名称

private String buyerTaxId; // 购买方纳税人识别号

// Getters and Setters

}

```

### 3. 定义服务接口

我们定义一个服务接口来处理发票查验的逻辑。

```java

public interface InvoiceService {

InvoiceVerificationResponse verifyInvoice(InvoiceVerificationRequest request);

}

```

### 4. 实现服务接口

接下来，我们实现这个接口。假设我们有一个外部API来验证发票信息。

```java

@Service

public class InvoiceServiceImpl implements InvoiceService {

@Autowired

private ExternalInvoiceApi externalInvoiceApi;

@Override

public InvoiceVerificationResponse verifyInvoice(InvoiceVerificationRequest request) {

// 调用外部API进行发票查验

ExternalInvoiceResponse externalResponse = externalInvoiceApi.verifyInvoice(

request.getInvoiceCode(),

request.getInvoiceNumber(),

request.getInvoiceDate(),

request.getCheckCode(),

request.getTotalAmount()

);

// 将外部API的响应转换为我们的响应对象

InvoiceVerificationResponse response = new InvoiceVerificationResponse();

response.setInvoiceCode(externalResponse.getInvoiceCode());

response.setInvoiceNumber(externalResponse.getInvoiceNumber());

response.setInvoiceDate(externalResponse.getInvoiceDate());

response.setTotalAmount(externalResponse.getTotalAmount());

response.setStatus(InvoiceStatus.valueOf(externalResponse.getStatus()));

response.setSellerName(externalResponse.getSellerName());

response.setSellerTaxId(externalResponse.getSellerTaxId());

response.setBuyerName(externalResponse.getBuyerName());

response.setBuyerTaxId(externalResponse.getBuyerTaxId());

return response;

}

}

```

### 5. 定义外部API接口

假设我们有一个外部API接口来验证发票信息。

```java

public interface ExternalInvoiceApi {

ExternalInvoiceResponse verifyInvoice(String invoiceCode, String invoiceNumber, String invoiceDate, String checkCode, BigDecimal totalAmount);

}

public class ExternalInvoiceResponse {

private String invoiceCode;

private String invoiceNumber;

private String invoiceDate;

private BigDecimal totalAmount;

private String status;

private String sellerName;

private String sellerTaxId;

private String buyerName;

private String buyerTaxId;

// Getters and Setters

}

```

### 6. 异常处理

为了确保系统的健壮性，我们需要处理可能出现的异常情况。

```java

public class InvoiceVerificationException extends RuntimeException {

public InvoiceVerificationException(String message) {

super(message);

}

public InvoiceVerificationException(String message, Throwable cause) {

super(message, cause);

}

}

```

在服务实现中，我们可以捕获并处理异常：

```java

@Override

public InvoiceVerificationResponse verifyInvoice(InvoiceVerificationRequest request) {

try {

ExternalInvoiceResponse externalResponse = externalInvoiceApi.verifyInvoice(

request.getInvoiceCode(),

request.getInvoiceNumber(),

request.getInvoiceDate(),

request.getCheckCode(),

request.getTotalAmount()

);

// 转换响应

return convertToResponse(externalResponse);

} catch (Exception e) {

throw new InvoiceVerificationException("发票查验失败", e);

}

}

private InvoiceVerificationResponse convertToResponse(ExternalInvoiceResponse externalResponse) {

InvoiceVerificationResponse response = new InvoiceVerificationResponse();

response.setInvoiceCode(externalResponse.getInvoiceCode());

response.setInvoiceNumber(externalResponse.getInvoiceNumber());

response.setInvoiceDate(externalResponse.getInvoiceDate());

response.setTotalAmount(externalResponse.getTotalAmount());

response.setStatus(InvoiceStatus.valueOf(externalResponse.getStatus()));

response.setSellerName(externalResponse.getSellerName());

response.setSellerTaxId(externalResponse.getSellerTaxId());

response.setBuyerName(externalResponse.getBuyerName());

response.setBuyerTaxId(externalResponse.getBuyerTaxId());

return response;

}

```

### 7. 测试

最后，我们可以编写单元测试来验证我们的实现。

```java

@SpringBootTest

public class InvoiceServiceTest {

@Autowired

private InvoiceService invoiceService;

@Test

public void testVerifyInvoice() {

InvoiceVerificationRequest request = new InvoiceVerificationRequest();

request.setInvoiceCode("123456");

request.setInvoiceNumber("789012");

request.setInvoiceDate("2023-10-01");

request.setCheckCode("ABCDEF");

request.setTotalAmount(new BigDecimal("100.00"));

InvoiceVerificationResponse response = invoiceService.verifyInvoice(request);

assertNotNull(response);

assertEquals("123456", response.getInvoiceCode());

assertEquals("789012", response.getInvoiceNumber());

assertEquals(InvoiceStatus.VALID, response.getStatus());

}

}

```

### 总结

通过以上步骤，我们实现了一个健壮且可扩展的发票查验接口。我们定义了枚举和常量来表示发票的类型和状态，封装了请求和响应对象，并实现了服务接口来处理发票查验的逻辑。我们还考虑了异常处理，并编写了单元测试来验证我们的实现。