介绍

<https://github.com/duncanhunter/enterprise-angular-mono-repo-patterns-book>

这本书是如何组织的

在第1部分中，我们首先介绍monorepo中的开发。我们将介绍一些关于Nx的基础知识，并了解如何开始使用Nx。

在第2部分中，我们将深入探讨库:如何组织它们、命名它们、组合它们以及其他有助于代码重用和模块化的技术。

在第3部分中，我们将研究如何使用内建在Nx中的工具在整个monorepo中加强质量和一致性。

在第4部分中，我们将介绍如何帮助进行更智能的构建，以及如何在本地和CI管道中使用Nx工具。

在第5部分中，我们将介绍一些常见的开发挑战，这些挑战是在一个monorepo中与许多具有不同发布计划的团队一起工作时遇到的。  
在附录A中，我们将看到如何使用其他工具(Angular Console)与Nx交互。  
在附录B中，我们介绍了可以在Nx中使用的所有命令，并参考了书中的特定章节，您可以在其中找到关于该命令的更多信息。  
在附录C中，我们将介绍一些常见的how-to，并说明常见问题的决策树。

格式化  
代码块的格式是这样的。

示例参考应用程序  
作为贯穿全书的参考，让我们考虑虚构的Nrwl航空公司。在这个组织中有三个团队:

Booking：:团队致力于允许用户预订飞往目的地的航班。  
Check-in：:该团队致力于让用户对他们已经预订的航班使用在线值机。  
Seatmap：该团队致力于让用户以图形方式选择航班上的座位

有四(4)个单独部署的应用程序:Check-in(桌面和移动)和Booking(桌面和移动)。根据终端用户访问的URL (booking.nrwl-airlines.com或check-in.nrwl-airlines.com)和随请求发送的浏览器元数据(为他们发送桌面或移动体验)，终端用户将获得其中一个应用程序。

它的依赖图的代码是这样的:

第1部分:开始

为什么monorepo吗?

许多大型组织和组织内的业务单位都选择在单个存储库中开发它们的所有代码(包括大量前端应用程序和后端应用程序)。这种方法有几个目标:

*增加可见性*

组织或业务单元中的团队通常不知道其他团队正在进行的更改，而这些更改在集成期间会产生很大的影响。如果在代码签入时就发现了集成问题，那么可以节省大量时间。我们将在本书的第5部分讨论一些应对变化的策略。

此外，API契约在单一订单中很容易访问，并且可以被前端和后端直接使用。类型可以从契约中生成，并由前端和后端使用，以确保在编译时捕获错误，而不是在更昂贵的集成时捕获错误。

*重用代码直接*

模块化和共享代码的传统方法是创建一个包，将其部署到私有的npm存储库中，并通过将其添加到项目依赖项中来依赖它。当对这些进行更改时，会有大量的开销，因为需要打包和部署依赖代码。还有一个版本控制问题，因为我们必须引用正确的版本号。

开发人员可能或者使用npm链接或模拟当地发展依赖关系管理的方法,但这也是繁琐的设置和使用:可能有很大的依赖性,我们需要设置以这种方式,我们必须记得我们做的删除链接

在monorepo中工作允许您直接引用依赖项(在Nx中使用工作区相对路径)。开发人员也可以直接使用这些代码，我们将在该书的第5部分讨论如何将更改集成到共享代码中。

*确保依赖项的一致版本*

monorepo中的版本控制变得更加容易——组织或业务单位可以选择在所有项目中拥有单一版本的依赖关系，或者使用“latest-minus-X”策略来确保所有项目都是最新的。这降低了在其代码中依赖于不赞成使用的依赖项和脆弱版本的可能性

Nx Basics

*What is a workspace?*

工作区是使用Nx创建的文件夹。该文件夹包含一个单独的git存储库，包含应用程序和库的文件夹;以及一些脚手架来帮助构建、衬线和测试。

*什么是应用程序?*

应用程序生成二进制文件。它包含了打包许多库以创建部署的工件所需的最小数量的代码。

应用程序定义了如何构建交付给用户的构件。如果我们有两个独立的目标(比如桌面和移动设备)，我们可能有两个独立的应用程序。

在我们的参考示例中，四个(4)应用程序组织如下:

apps/

  booking/

  booking-desktop/ <--- application

  booking-mobile/ <--- application

  check-in/

  check-in-desktop/ <--- application

  check-in-mobile/ <--- application

每个测试程序都生成了相应的e2e测试应用程序文件夹。  
应用程序只是用来将其他库组织成一个可部署的构件——在应用程序中模块文件之外并没有很多代码，可能还有一些基本的路由。应用程序的所有代码都被组织成库

*What is a lib?*

库是一组打包在一起的文件，供应用程序使用。库类似于节点模块或nuget包。它们可以按原样发布到NPM或与已部署的应用程序绑定

使用库的目的是将代码划分为更小的单元，以便更容易维护和促进代码重用。使用Angular CLI 6。库可以在/ apps文件夹中的应用程序中使用，甚至可以捆绑并部署到NPM中作为一个独立的包。

有些库只用于特定的应用程序(如booking)，应该进入相应的目录(如libs/booking)。我们称它们为“特定于应用程序”。尽管这些库可以在多个地方使用，但创建它们的目标不是代码重用，而是将应用程序分解为定义良好的模块，以简化应用程序的维护。

典型的Nx工作区只包含四种类型的库:feature, data-access, ui, and util。您可以在本书的第2部分中详细了解这些类型的库。

与Nx交互

Nx配置为以下配置文件的工作空间:

Nx.json特定于Nx的json  
package.json 由npm提供的json  
angular.json 由Angular CLI提供的json

Nx还提供与工作区、应用程序和库一起工作的命令。这些可以输入到终端中，也可以在Angular控制台中调用，Angular控制台是一个图形工具，可以与  
Angular CLI和Nx。Angular控制台在附录A中讨论

在这本书中，我们在每一节中使用npm提供终端命令。我们在适当的地方提供了Angular控制台的说明。本书的附录B中也列出了所有命令。

*yarn vs. npm*

在这本书中，我们提供了使用npm的示例代码。所有这些命令也可以使用yarn。两者之间有一些区别:

1. npm命令可以接受命令本身和底层命令的参数。例如，npm运行task1——watch——param1=val1将参数watch传递给npm本身，而param1传递给task1的底层实现。——本身指示所有参数被转发到基础任务之后的中断。另一方面，*yarn*不会这样做，而是将所有参数传递给底层任务;因此它不需要——分隔符
2. npm命令用npm run <命令>运行，*yarn*命令用*yarn*<命令>运行
3. npm install安装包，您需要指定—save或—save-dev来保存依赖项;而纱线则安装带有纱线添加的包装，并将纱线添加到包装中。json在默认情况下

在书中，你可以转换一个npm脚本与*yarn*使用以下方法:

安装和设置工作区

*Creating a workspace*

您可以通过以下方式安装Nx:

npm install -g @angular/cli @nrwl/schematics

@nrwl/schematics库包含一个二进制脚本，该脚本提供命令create-nxworkspace。该命令可以用来创建一个新的Nx工作区:

create-nx-workspace myworkspacename

这将使用沙箱环境创建一个新的Nx工作区，并在Nx schematics集合下运行Angular CLI的ng new命令。

您还可以通过运行以下程序将Nx功能添加到现有的CLI项目:

ng add @nrwl/schematics

最后，您可以使用Angular Console创建一个Nx工作区。

图1所示。在Angular Console中创建一个新的工作区  
不管您如何创建工作空间，您最终得到的都是一个包含以下文件/文件夹的Nx工作空间。

apps/

  libs/

  tools/

  angular.json

  nx.json

  tslint.json

  tsconfig.json

它类似于标准的Angular CLI项目，只是做了一些改变:  
•它有一个apps目录，用于存放所有的应用程序  
•它有一个libs目录，所有自定义库代码都放在那里  
下面是对它们的简短描述;我们在进行的过程中对每一个都进行了扩展

tsconfig.json：这是工作空间tsconfig。Nx在这里为每个库添加了路径别名，以允许与工作空间相关的导入。

import { myLib } from '@myProject/shared/my-lib';

新的Nx工作区不设置初始应用程序，但是添加一个很简单。

Creating an app

向Nx工作区添加新应用程序是通过使用Angular CLI generate命令完成的。Nx有一个示意图名为app，可以用来添加一个新的app到我们的工作空间:

ng generate app myapp

ng generate application myapp # same thing

这个命令执行几个动作:

1 - 它创建一个新的应用程序，将其放在apps目录中，并配置angular.json和nx.json文件支持新的应用程序。

2 - 它配置根NgModule来导入NxModule代码，这样我们就可以利用像DataPersistence这样的东西。

3 - 它还为这个应用程序提供了一个包含e2e测试代码的e2e兄弟文件夹。

运行ng generate app——帮助查看可用选项列表。

大多数选项与默认CLI应用程序支持的选项相同，但以下选项是新的或不同的:directory, routing, and tags.

ng generate app myapp --directory=myteam creates a new application in apps/myteam/myapp.

• ng generate app myapp --routing 配置根NgModule来连接路由，并在AppComponent模板中添加<router-outlet>来帮助我们开始。

• ng generate app myapp --tags=scope:shared,type:app 配置根NgModule来连接路由，并在AppComponent模板中添加<router-outlet>来帮助我们开始。

一旦我们创建了应用程序，我们就可以开始创建包含组成应用程序的所有组件和逻辑的库。

Creating a lib

向Nx工作区添加新的库是通过使用Angular CLI generate命令完成的，就像添加一个新的应用程序一样

ng generate lib mylib

  ng generate library mylib # same thing

这将创建一个新的lib，将其放到libs目录中，并配置angular.json和nx.json文件来支持新库。

有关这些选项的进一步描述，请参阅“使用库的说明”一节

Getting help

当使用终端时，所有Nx命令都提供——help标志来显示可用选项和每个选项的描述。Angular控制台可以可视化地显示帮助文本，还可以提供按是否需要分组的选项列表。

示例输出如下:

运行ng generate lib——帮助查看可用选项列表:

使用Angular控制台也可以得到帮助，它为每个命令提供了所有选项以及描述。它还将必需的选项从可选的选项中分离出来，并基于您的工作空间自动填充值(允许您从列表中选择，而不是手动输入)

图2。获得Angular控制台的帮助。注意，选项被分为“重要”和“可选”，并且这些值可以在下拉列表中进行选择。

总结  
这一节为书的其余部分奠定了基础。它涵盖了一些关键概念，为以下内容奠定了基础:

采用monorepo思维方式的一些原因包括:增加所有团队在代码库中的可见性，直接重用代码而不是通过包发布和消费代码管理，并确保所有项目依赖关系的一致版本。  
•介绍了Nx的基本术语和基本概念(工作区、应用程序和库)  
•命令行和如何与Nx交互的描述  
•使用Nx进行设置，创建工作区、应用程序和库  
•如何使用Nx命令获得帮助

第2部分:用库组织代码

简而言之  
应用程序会随着规模和复杂性的增长而增长，在大型组织中，有许多方法可以重用代码以保持一致性并降低开发工作量。可以使用库来实现模块化开发，库只是执行特定任务的相关文件的集合。这些组合在一起就构成了一个应用程序

库需要分类器来描述其内容和预期目的。这些分类器有助于组织库并提供一种定位它们的方法。

Scope

范围与逻辑分组、业务用例或域相关。示例应用程序中的范围示例有seatmap、booking、shared和check-in。它们包含管理应用程序逻辑子域的库。

我们建议使用文件夹结构来表示范围。  
下面的文件夹结构是一个用于描述seatmap特性的范围层次结构示例:

shared/

  seatmap/

feature/

在这里，“shared”和“seatmap”是对文件夹进行分组的，而feature是一个深嵌两层的库。这清楚地表明该特性属于seatmap的一个域，该域是共享项的子域。

这个库中使用的标记将是scope:shared，因为这是顶级范围

Type

类型与库的内容相关，并指示其用途和用法。类型的例子有ui, data-access, and feature.

。请参阅下面较长的讨论

我们建议使用前缀和标记来表示类型。我们建议将类型的数量限制为以下几节中描述的四种类型。  
此特性的文件夹名称将是features -shell，以便它将前缀用于其库类型。  
如上例中所示，seatmap特性库的标记现在是scope:shared,type:feature。

Platform

还可以使用其他分类器来区分类似的库(例如服务器、移动设备和桌面)。Platform就是这样一个分类器。

我们建议使用标记来表示平台

seatmap特性的最后一个标签是scope:shared,type:feature,platform:desktop。

每个库应该按作用域位于文件夹树中，具有格式作用域:scope、type: type、platform: platform中的标记，并具有按其类型设置的前缀

*不要按文件类型组织!  
我们强烈反对文件类型的组织，例如指令/，服务/等。这样做的原因是，当我们想要进行更改时，我们经常需要一次性更改几个相关文件。当按文件类型组织时，我们需要遍历文件夹树多次以找到所需的文件。*

*相反，我们建议代码库按域组织，并包括所有相关文件，例如航空公司，其中包括状态，ui组件等在一个分组文件夹内。*

**Types of libraries**

工作空间中有许多不同类型的库。为了保持一定的秩序，我们建议只使用以下四(4)种类型的库:

Feature libraries:开发人员应该将特性库看作是为特定于该特性的业务用例实现智能UI(带有注入的服务)的库。

• UI libraries: UI库只包含表示组件。

• Data-access libraries: 数据访问库包含用于与后端系统交互的服务和实用程序。

• Utility libraries: 实用程序库包含许多库和应用程序使用的公共实用程序和服务。它还包括与状态管理相关的所有代码。

Feature libraries

特性库包含特定应用程序部分的路由器配置。  
此类库中的大多数UI组件都是与NgRx store交互的智能组件。这种类型的库还包含了大部分的应用程序逻辑、验证等。特性库几乎都是特定于应用程序的，并且通常是延迟加载的

*开发人员应该将特性库视为实现智能UI的库(使用注入的服务)用于特定于该特性的业务用例。*

命名约定:feature或feature-\*(例如feature-shell)

libs/

  booking/

  feature-shell/

src/

index.ts

lib/

booking-feature-shell.module.ts

Example - Feature Library Module:

  @NgModule({

  imports: [ CommonModule,

  RouterModule.forChild([

{path: '', component: DashboardComponent,pathMatch: 'full'}  ])

  ],

  declarations: [DashboardComponent]

  })

export class BookingFeatureShellModule {}

UI libraries

UI组件库只包含表示组件。

特定应用程序组件库包含特定于特定应用程序的组件(或者几个紧密相连的应用程序)。

共享组件库包含用于多个应用程序或跨许多业务功能的组件。

命名约定:ui或ui-\*(例如，components-buttons)

当我们构建应用程序时，有两种类型的组件:

Smart Components （智能组件）  
 管理或委托业务逻辑和使用DI注入服务。  
 有子实例的presentational组件

Presentational Components (aka dumb)显示组件

没有或很少业务逻辑  
◦只依靠投入和产出来与外界沟通。  
◦有智能组件父母  
◦宣布用户交互使用输出  
◦是高度可重复使用，是最容易的测试  
◦呈现数据的演示格式(例如，显示用户票)，和  
◦可以完全通用/领域不可知(例如:data-table))或拥有域上下文(例如，log-book-table)。

考虑下面的OrderCreate智能组件:

@Component({

  selector: 'order-creator',

  styleUrls: ['./order-create.component.css'],

  template: `

<div fxLayout fxLayoutAlign="center center">

<order-card [order]="order" [users]="users$ | async" >

</order-card>

</div>  `

  })

  export class OrderCreatorComponent {

  users$: Observable<User[]> = this.service.users$;

  order: Order = makeNewOrder();

  constructor(public service: OrdersFacade, public users: UsersFacade) {}

  }

这是它的子表示组件OrderCard:

@Component({

  selector: 'order-card',

  styleUrls: ['./order-card.component.css'],

  changeDetection: ChangeDetectionStrategy.OnPush,

  template: `

<mat-card class="order-card" \*ngIf="order">

...

</mat-card>

  `

  })

  export class OrderCardComponent {

  @Input() users: User[];

  @Input() order: Order;

  @Output() save = new EventEmitter<Order>();

  @Output() cancel = new EventEmitter<void>();

  @Output() reassign = new EventEmitter<string>();

  @Output() complete = new EventEmitter<Order>();

  title(order): string {

  const status = order.completed ? 'Finished' : 'Pending';

  return !order.id ? 'Create New Order' : `${status} Order `;

  }

  }

示例-组件(表示)库模块:

  import { LoadingComponent } from './loading/loading.component';

  import { ConfirmButtonComponent } from './confirm-button/confirm-button.component';

  @NgModule({

  imports: [CommonModule],

  declarations: [LoadingComponent, ConfirmButtonComponent],

  exports: [LoadingComponent, ConfirmButtonComponent]

  })

  export class ComponentButtonsModule {}

这个模块的路径是libs/common/ui-buttons/src/commonui-buttons.module.ts

Data-access libraries

数据访问库包含REST或webSocket服务，它们作为服务器层api的客户端委托层。所有与状态管理相关的文件也驻留在数据访问文件夹中。

名称约定:data-access or data-access-\*.

@NgModule({

  imports: [

  CommonModule,

  StoreModule.forFeature(

'customer', customersReducer, {initialState: customersInitialState  }),

  EffectsModule.forFeature([CustomersEffects])

  ],

  providers: [CustomersEffects]

  })

  export class CustomersDataAccessModule {}

重用数据访问库很容易，所以应该重点创建更多的共享数据访问库。

当使用NgRx facade时，开发者应该考虑三层api:

HTTP REST服务的API:这些是简单的、由视图或外观调用的公共方法。这些方法隐藏了使用内部HTTPClient服务的复杂性。  
•NgRx外观API:仅在视图组件级别用于将查询可观察对象发布为属性的API:可观察对象到NgRx存储选择器——在内部向NgRx存储分派动作的公共方法。  
•NgRx Effects 的API:定义为哪个NgRx动作“运行”哪个Effects 。这些Effects 通常运行异步进程

Utility libraries

实用程序包含许多库使用的公共实用utilities/services 。

命名规格: util (if nested), or util-\* (e.g., util-testing)

export { navigateToPage } from './src/navigate-to-page';

  export { HttpMockBackend } from './src/http-mock-backend';

Grouping folders（分组的文件夹）

在我们的引用结构中，libs/booking, libs/check-in, libs/shared, and

libs/shared/seatmap文件夹都是分组文件夹。它们不包含任何东西，除了其他库或分组文件夹

apps/

  booking/

  check-in/

 libs/

  booking/ <---- grouping folder

  feature-shell/ <---- library

  check-in/

  feature-shell/

  shared/ <---- grouping folder

  data-access/ <---- library

  seatmap/ <---- grouping folder

data-access/ <---- library

feature-seatmap/ <---- library

Shared libraries

在我们的参考示例中，有一些共享库

libs/

  shared/

  data-access/ <---- shared library

  seatmap/

data-access/ <---- shared library

feature-seatmap/ <---- shared library

与领域无关的:跨大多数应用程序使用的库。它们进入描述逻辑共享域的分组文件夹或进入共享文件夹。(例如,shared/data-access)

特定于领域的:只在少数应用程序中使用的库。这些进入分组文件夹。(例如,booking)

有时候，要确定一个库是否应该共享并不容易。参见参考部分中的决策树获得一些指导。

**使用库须知**

命令行选项

当使用Nx原理图创建库时，开发人员有几个power选项。这些选项中的大多数与默认CLI库支持的选项相同，但以下选项是新的或不同的。

——lazy标志对于创建延迟加载的库尤其重要。

在创建延迟加载的库时，需要向tsconfig.app添加一个条目。父模块app的json文件，所以TypeScript也知道如何构建它:

{

  "include": [

"\*\*/\*.ts"

/\* add all lazy-loaded libraries here: "../../../libs/my-lib/index.ts" \*/

, "../../../libs/mymodule/src/index.ts"

  ]

  }

在大多数情况下，Nx在默认情况下这样做。有时，您需要手动添加该条目

The barrel file

当我们使用Nx和@nrwl/schematics创建库时，需要执行几个步骤。  
首先创建文件夹(如果指定——directory，则采用嵌套结构)。接下来，在这个结构中为库生成基本文件:

jest.config.js

src/

  index.ts

  lib/

shared-data-access.module.spec.ts

  shared-data-access.module.ts

  shared-data-access.service.spec.ts

  shared-data-access.service.ts

  test-setup.ts

tsconfig.lib.json

tsconfig.spec.json

tslint.json

注意:jest被选为这个库的测试框架。

注意：index.ts 文件在src/文件夹中。这是lib的barrel文件。它包含了与库交互的公共API，我们应该确保我们要公开的所有常量、枚举、类、函数等都导出到这个桶文件中

如果它嵌套在src/文件夹中，那么当我们调用库如下时，桶是如何暴露的:import {SharedDataAccessModule} from ['@nrwl-airlines/shared/dataaccess';?导入需要index.ts文件应该在lib的根文件夹中，而不是在src/中。](mailto:'@nrwl-airlines/shared/dataaccess';?导入需要index.ts文件应该在lib的根文件夹中，而不是在src/中。)

Nx在根中配置它tsconfig.json，它将包含以下条目:

"paths": {

  ...

  "@nrwl-airlines/shared/data-access": [

  "libs/shared/data-access/src/index.ts"

  ]

  ...

}

我们创建的每个应用程序和库都在这个文件中获得一个条目，以帮助使用@ workspace-relative path语法进行映射。如果这种方式有助于开发，您可以用这种方式创建自己的别名。

Import Paths

库中包含的组件和类应该仅通过相对路径导入。使用工作区相对路径引用它们会导致linting错误。

从当前库外部导入的组件和服务必须使用npmScoped导入(例如。@myOrg/customers/)

而不是相对路径。

TsLint规则已配置为显示此类违规的错误。

Module names

库的主模块必须在文件名中包含完整的路径(相对于库)。例如，libs/booking/feature-destination的主模块的文件名为booking-feature-destination.module.ts.这就是CLI创建模块的方式，我们建议保留这种模式。

Documenting libraries

README应该确定库的用途，并概述库的公共API。该文件还可包括其他细节，如:  
•代码的所有者  
•库使用的可视化(依赖图)  
•依赖约束的可视化(哪些应用程序或库被授权使用此库

**总结**在本节中，我们深入了解了库。我们讨论了以下内容:  
•为什么按文件类型分组不是一个可伸缩的策略  
•使用范围、类型、平台和其他分类器来组织库  
•四种类型的库:ui、特性、数据访问和util  
•使用分组文件夹为范围引入层次结构  
•共享的和特定领域的范围

谷歌强制所有项目只有一个版本的Angular。这确保了没有遗留项目使得重构变得困难，并且还允许开发人员在项目之间移动。  
Nx包含一些工具来帮助维护代码库之间的一致性，并确保代码质量

1 一个package.json确保所有应用程序和库使用相同的依赖版本。  
2 构建像prettier这样的工具有助于确保代码格式的一致性(一个直接的好处是消除了PRs中的空白和格式差异)。

3 它允许对库之间的边界进行可配置的执行，例如，util库不能依赖于某个特性库(甚至是一个应用程序)。  
4 Nx允许您生成工作区图表，以一致地处理代码生成。

5 Nx包括一个DataPersistence库来帮助以一致和可靠的方式管理与后端的交互

代码格式化  
在谷歌中，一般的原则是任何可以自动执行的操作都应该自动执行。其中之一就是代码格式化。这就是为什么Nx内置了对Prettier的支持。  
您完全不需要配置任何东西。只需调用format:write来格式化受影响的文件，以及format:检入CI以确保所有格式一致

Run npm run format:write — --help 查看可用选项.

npm run format:write — --base=[SHA1] --base=[SHA2]. Nx计算两个SHAs之间的变化，并格式化变化的文件。 npm run format:write — --base=origin/

master --base=HEAD formats what is affected by a PR.

check命令接受相同的选项，但不是格式化文件，而是在任何文件没有正确格式化时抛出。这对CI/CD很有用。

分析和可视化依赖图

为了能够支持monorepo风格的开发，这些工具必须知道您工作空间中的不同项目是如何相互依赖的。Nx使用高级代码分析来构建这个依赖关系图

你可以通过运行来想象它 npm run dep-graph

通过使用affected: depo -graph命令，您还可以可视化您的更改所影响的内容。

在本书的第4部分中，我们将了解如何通知受影响的命令使用哪些文件更改来确定所有受这些代码更改影响的应用程序和库。depgraph通过突出显示受影响的应用程序库进行响应，并可视化地描述关键路径。

默认情况下，depgraph和affected: depgraph命令打开浏览器以显示图形，但是您也可以通过运行将图形输出到文件中

对库的内置约束

以下不变量应该成立

1 库不能依赖于应用程序

2 特定于应用程序的库不能依赖于另一个应用程序的库(例如，“safe/”只能依赖于“safe/”或共享库)

3 共享库不能依赖于特定于应用程序的库(例如，“common-ui/”不能依赖于  
“safe/”)

4 ui库不能依赖于feature library, or a data-access library.

5 utils库不能依赖于功能库、数据访问库或组件库

6 数据访问库不能依赖于功能库或组件库

7 项目不能具有循环依赖关系

8 延迟加载另一个项目的项目不能直接导入它。

Nx提供了一些适用于所有工作空间的预定义规则:

1 Libs不能导入应用程序

2 通过loadChildren加载库的项目也不能使用ESM导入导入库

3 循环依赖是不允许的

4 不能使用相对导入导入lib。

将您自己的约束强加于库依赖

为了帮助管理复杂性，Nx使用代码分析来确保项目只能依赖于彼此定义良好的公共API。它还允许您声明性地对项目如何相互依赖施加约束。

这些约束是静态执行的:

如果您试图违反这些规则，ide和编辑器会显示一个错误  
 CI失败

要定义并满足约束，您需要在创建新库时提供标签:

ng g lib feature-destination --directory=booking --tags=booking,feature

注意，您还可以在事后修改nx.json中的标记，就像这样

"booking-feature-destination": {

  "tags": ["scope:booking", "type:feature"]

  },

  "booking-shell": {

  "tags": []

  },

一旦标记与每个库相关联，就可以定义tsLint规则来配置约束:

"nx-enforce-module-boundaries": [

  true,

  {

  "allow": [],

  "depConstraints": [ {

"sourceTag": "scope:shared", ①

"onlyDependOnLibsWithTags": ["scope:shared"]

},

{

"sourceTag": "scope:booking", ②

"onlyDependOnLibsWithTags": ["scope:booking", "scope:shared"]

},

{

"sourceTag": "type:util", ③

"onlyDependOnLibsWithTags": ["type:util"] } ]

  }

  ]

A lib tagged scope: 共享只能通过标签从其他库导入 scope:shared. ·

A lib tagged scope:booking 只能从标记为scope:booking或scope:shared的lib中导入。·

A lib tagged type:util只能从标记为util的另一个库导入

使用上面的示例配置，当我们尝试从repo的管理部分导入私有客户端代码时，应该会看到一个错误

有了依赖约束，另一个团队将不会在您的内部库上创建依赖项。您可以定义哪些项目包含组件、NgRx代码和特性，因此，例如，您可以根据NgRx禁止包含表示性UI组件的项目。您可以定义哪些项目是实验性的，哪些是稳定的，因此稳定的应用程序不能依赖于实验性的项目等等。

隐式依赖关系

Nx使用其内置的智能来创建应用程序和库的依赖图，并用于找出需要重建和重新测试的内容。但是，Nx不能分析某些文件。这就是为什么Nx支持隐式依赖关系。它们在nx.json中定义。

{

  "npmScope": "mycompany",

  "implicitDependencies": {

  "package.json": "\*",

  "angular.json": "\*",

  "tsconfig.json": "\*",

  "tslint.json": "\*",

  "nx.json": "\*"

  },

  "projects": {}

  }

The ”package.json”: “\*” 行告诉Nx对package.json的更改会影响每个项目。由于没有列出根级自述文件，因此更改它不会影响任何内容。

我们可以更具体地列出受特定文件影响的项目。

{

  "npmScope": "mycompany",

  "implicitDependencies": {

  "package.json": "\*",

  "angular.json": "\*",

  "tsconfig.json": "\*",

  "tslint.json": "\*",

  "nx.json": "\*",

  "tools/scripts/app1-rename-bundles.js": ["app1"]

  },

  "projects": {}

  }

除了能够列出文件和项目之间的隐式依赖外，我们还可以添加项目之间的隐式依赖。例如，Nx工作区可以包含Angular应用程序的后端代码。后端代码隐式地依赖于Angular应用程序，但它没有在代码中表示出来，因此Nx无法推导出它。我们可以明确定义为:

{

  "npmScope": "mycompany",

  "implicitDependencies": {

  "package.json": "\*",

  "angular.json": "\*",

  "tsconfig.json": "\*",

  "tslint.json": "\*",

  "nx.json": "\*",

  "tools/scripts/app1-rename-bundles.js": ["app1"]

  },

  "projects": {

  "app": {},

  "backend": {

"implicitDependencies": ["app"]

  }

  }

  }

异常  
与任何事情一样，也有例外，这也会添加到tslin .json中。

"nx-enforce-module-boundaries": [

  true,

  {

  "allow": ["@myworkspace/mylib"],

  "depConstraints": [

  ]

  }

  ]

Workspace Schematics

一些最佳实践已经在Angular社区中得到了完善。例如，大多数都不是微不足道的  
Angular项目使用NgRx，所以Nx自带了一组内置的代码生成器和运行时库，以确保应用程序的状态管理是一致的。只要运行ng g ngrx accounts

--module=one/two/app.module.ts 把一切都准备好 (阅读更多 [here](https://nrwl.io/nx/guidesetting-up-ngrx))

然而，有些是特定于您的组织的。Nx也可以在这些方面提供帮助。例如，我们希望促进一种将ngrx相关代码封装到数据访问库中的模式。这是如何容易做到它是与Nx:

首先生成一个新的工作区示意图

(图)

然后，提供实现

(图)

最后，调用它来生成新的数据访问库。

我们通常建议将图表放在/tools文件夹中。

处理具有数据持久化的后端调用

管理状态是一个困难的问题。我们需要协调多个后端、web worker和UI组件，所有这些都可以并发地更新状态。

与URL相比，我们应该在内存中存储什么?那么本地UI状态呢?如何同步服务器上的持久状态、URL和状态?在设计应用程序的状态管理时，必须回答所有这些问题。Nx数据持久性是一组帮助函数，它使开发人员能够使用有意的同步策略管理状态并处理错误状态。请参考在Angular中使用NgRx 4来管理状态应用程序(Using NgRx 4 to Manage State in Angular Applications )，以获得数据持久性正在解决的状态问题的更详细示例。

为什么使用DataPersistence库?

该库被设计为抽象一些常见的逻辑Effects

1 在异步操作期间从存储中获取状态以检索一个或多个值

2 悲观或乐观地更新UI

3 需要以可靠的方式处理具有相同Effects的多个调用

4 在使用switchMap操作时，错误处理常常被忘记，导致在遇到错误时效果完成(因此不能响应触发它的进一步操作)

乐观的更新

为了获得更好的用户体验，optimisticUpdate方法会在更新服务器端数据之前立即更新客户机应用程序上的状态。虽然它解决了按顺序获取数据、删除竞争条件和处理错误的问题，但更新服务器不会失败。在失败的情况下，当使用optimisticUpdate时，客户端上的本地状态已经更新。开发人员必须提供一个撤销操作来恢复以前的状态，以保持它与服务器状态一致。错误处理必须在回调中完成，或者通过撤销操作

import { DataPersistence } from '@nrwl/nx';

...

class TodoEffects {

  @Effect()

updateTodo = this.dataPersistence.optimisticUpdate('UPDATE\_TODO', {

  // 提供存储的操作和当前状态

  run: (a: UpdateTodo, state: TodosState) => {

  return this.backend(state.user, a.payload);

  },

  undoAction: (a: UpdateTodo, e: any) => {

  // 分派撤消操作以撤消客户端状态中的更改

  return ({type: 'UNDO\_UPDATE\_TODO',payload: a  });

  }

  });

  constructor(

  private dataPersistence: DataPersistence<TodosState>,

  private backend: Backend

  ) {}

}

悲观的更新

为了实现更可靠的数据同步，悲观更新方法在更新UI之前首先更新服务器数据。当更改反映在服务器状态时，就会通过发送操作对客户端状态进行更改。悲观更新方法强制执行读取和错误处理的顺序

import { DataPersistence } from '@nrwl/nx';

...

@Injectable()

class TodoEffects {

  @Effect() updateTodo = this.dataPersistence.pessimisticUpdate('UPDATE\_TODO', {

  // 提供存储的操作和当前状态

  run: (a: UpdateTodo, state: TodosState) => {

  // 首先更新后端，然后分派操作

  // 更新客户端

  return this.backend(state.user, a.payload).map(updated => ({

type: 'TODO\_UPDATED',

payload: updated

  }));

  },

  onError: (a: UpdateTodo, e: any) => {

  //我们不需要撤销客户端上的更改.

  // 我们可以分派一个错误，或者简单地记录错误并返回' null '

  return null;

  }

  });

  constructor(

  private dataPersistence: DataPersistence<TodosState>,

  private backend: Backend

  ) {}

}

数据抓取

DataPersistence的获取方法在获取数据时提供了一致性。如果为同一操作调度了多个请求，则只运行最后一个请求。

import { DataPersistence } from '@nrwl/nx';

...

@Injectable()

class TodoEffects {

  @Effect() loadTodos = this.dataPersistence.fetch('GET\_TODOS', {

  // 提供存储的操作和当前状态

  run: (a: GetTodos, state: TodosState) => {

  return this.backend(state.user, a.payload).map(r => ({

type: 'TODOS',

payload: r

  }));

  },

  onError: (a: GetTodos, e: any) => {

  // 分派撤消操作以撤消客户端状态中的更改

  return null;

  }

});

  constructor(

  private dataPersistence: DataPersistence<TodosState>,

  private backend: Backend

  ) {}

}

这是正确的，但是我们可以通过使用访问器函数来提供数据和id，并为不同的ToDo的获取操作添加并发性，从而提高性能。

@Injectable()

class TodoEffects {

  @Effect() loadTodo = this.dataPersistence.fetch('GET\_TODO', {

  id: (a: GetTodo, state: TodosState) => {   return a.payload.id;  },

  // 提供存储的操作和当前状态

  run: (a: GetTodo, state: TodosState) => {

  return this.backend(state.user, a.payload).map(r => ({

type: 'TODO',payload: r  })); },

  onError: (a: GetTodo, e: any) => {

  // 分派撤消操作以撤消客户端状态中的更改

  return null;

  }

  });

  constructor(

  private dataPersistence: DataPersistence<TodosState>,

  private backend: Backend

  ) {}

}

在此设置中，Todo请求与Todo 2请求并发运行。对Todo的连续调用排在第一个调用之后。

在路由器导航上获取数据

由于用户总是可以直接与URL交互，所以我们应该将路由器视为真相的来源和动作的发起者。换句话说，路由器应该调用reducer，而不是反过来

当我们的状态依赖于导航时，我们不能假设在触发新url时发生了路由更改，而是当我们实际知道用户能够导航到该url时才发生。DataPersistence导航方法检查激活的路由器状态是否包含传入的组件类型，如果包含，则运行run回调。它提供与组件关联的激活快照和当前状态。它只运行最后一个请求

import { DataPersistence } from '@nrwl/nx';

...

@Injectable()

class TodoEffects {

  @Effect()

loadTodo = this.dataPersistence.navigation(TodoComponent, {

  run: (a: ActivatedRouteSnapshot, state: TodosState) => {

  return this.backend.fetchTodo(a.params['id']).map(todo => ({

type: 'TODO\_LOADED', payload: todo   }));

  },

  onError: (a: ActivatedRouteSnapshot, e: any) => {

  // 我们可以在这里记录和错误并返回null

  // 我们也可以导航回去

  return null;

  }

  });

  constructor(

  private dataPersistence: DataPersistence<TodosState>,

  private backend: Backend

  ) {}

}

仅重建和重新测试受影响的应用程序和库  
当我们考虑到包含大量库和许多应用程序的大型存储库时，我们意识到，每当合并新的代码时，都必须对所有库进行测试和构建，这将是非常困难的。它没有规模。手动跟踪依赖关系也不可靠:它是不可靠的。如果有一种可靠的方法让我们只测试和构建受影响的库和应用程序，那就太好了。

Nx使用代码分析来确定需要重新构建和重新测试的内容。它通过受影响的命令:affected:build, affected:test和affected:e2e来提供这一点。这些命令可以与下列选项一起运行，以确定仅那些库和应用程序(又名。“项目”)，这些项目会受到代码更改的影响。

针对特定项目的选项

1 Compare changes between 2 git commits: You can run --base=SHA1 --head=SHA2, 其中SHA1是您想要比较的，而SHA2包含更改。这将生成一个文件列表，我们可以使用该列表来确定哪些项目受到这些更改的影响，并且只处理那些更改。

2 Uncommited changes: Use the flag --uncommited 使用当前分支上所有未提交的更改来获取要处理的受影响项目的列表(自上次提交以来更改的文件:可以使用git状态查看它们)

3 Untracked: Use the flag --untracked 只使用未跟踪文件的列表(自上次提交以来创建的文件)来确定要处理的项目(在开发期间很有用)

4 Explicit files: Use --files 提供一个以逗号分隔的文件列表(在开发期间很有用)

5 All projects: Use --all 对所有项目强制执行该命令，而不仅仅是那些受代码更改影响的项目

6 Only last failed: Use --only-failed 只隔离那些以前失败的项目 (default: false)

附加选项

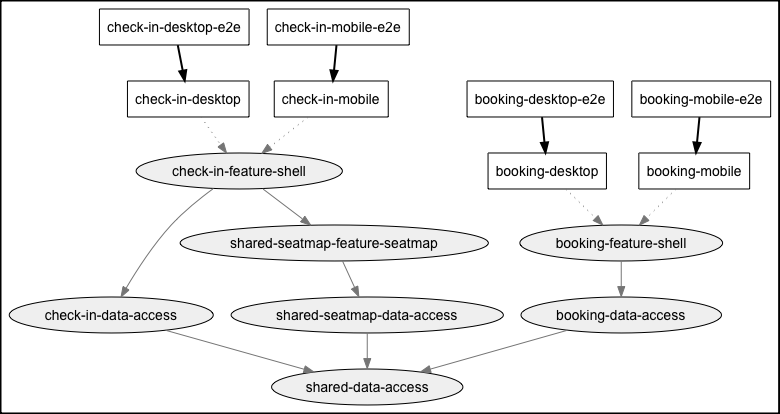
• --parallel: 并行命令 (default: false)

• --maxParallel: 设置并行进程的最大值(default: 3)

• --exclude: 排除正在处理的某些项目 (comma-delimited list)

所有其他选项都传递给底层CLI命令。

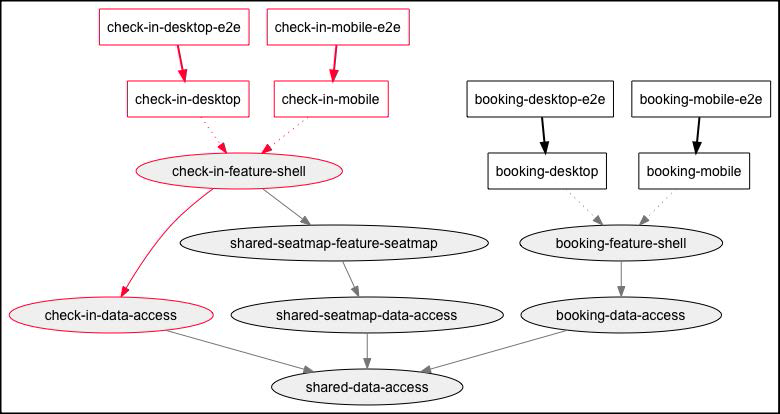
一个介绍  
让我们以我们的样本回购(nrwl-airlines)为例。其依赖关系图如下:



场景1:特定于应用程序的库中的更改

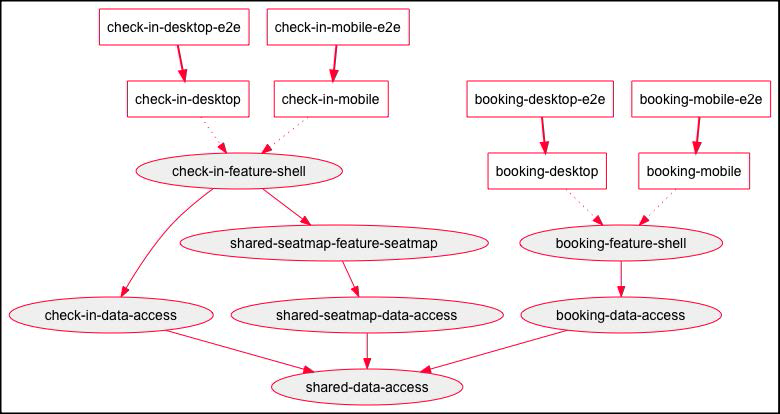
如果我们更改check-in-data-access库中的文件，我们现在可以看到它只影响check-infeature-shell, check-in-desktop, and check-in-mobile.。

我们不需要触碰booking或seatmap



场景2:共享库中的更改

如果我们现在更改共享数据访问库中的一个文件，我们会看到它会影响工作区中的所有项目!影响是巨大的，所以我们知道我们必须格外小心，接触其他项目的所有者，以确保每个人都知道这些变化。



使用“affected”命令的优点

场景1和场景2的差异非常明显:仅重新构建或重新测试受影响的项目会对构建和测试所花费的时间产生巨大影响。我们也不需要将更改部署到没有更改的项目中。

并行运行命令  
在执行这些命令时，Nx对项目进行拓扑排序，并并行运行它能运行的内容。但我们也可以显式传递，像这样:

npm run affected:build -- --base=master --parallel

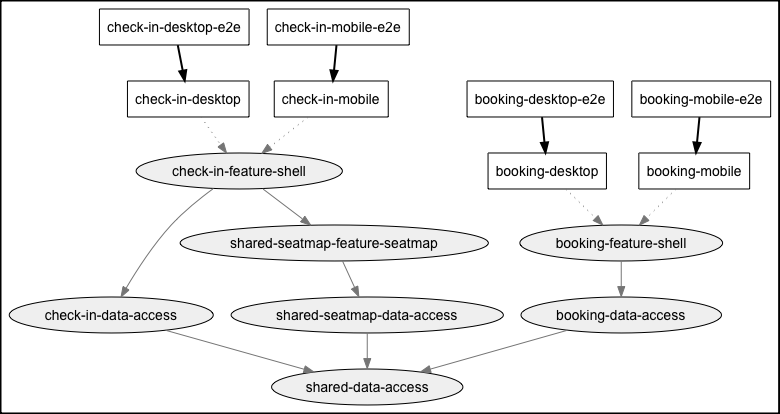
npm run affected:test -- --base=master --parallel

npm run affected:e2e -- --base=master --parallel

我们还可以通过——maxParallel来指定并行进程的数量。.

以下是一些进入Monorepo的共同挑战:  
1. 在同一个存储库中有许多团队和应用程序时，我们如何管理针对不同环境的版本?  
2. 我们如何管理项目之间的依赖关系?许多项目可以同时运行，我们可能需要引用某些代码的旧版本。  
3.我们如何组织共享的代码，以便在不产生太多技术债务和不需要经常维护的情况下可重用?

我们在本书的第2部分(使用库组织代码)中讨论了第3个问题。本节的其余部分将重点介绍如何允许团队在单个存储库中协同工作，同时仍然能够灵活地部署不同的应用程序。  
让我们以回购为例。依赖关系图如下:



我们的Monorepo有4个需要部署的应用程序booking and check-in的桌面和移动版本。所有的应用程序都隐式地依赖于shared-data-access库。

下面列出了一些常见的挑战:

1. booking 团队对共享数据访问进行更改:他们如何将更改告知check-in团队，以确保没有任何问题出现?  
   2. check-in团队在QA检查中失败，需要进行代码修复;然而，booking团队的回购中有新的代码。  
   3.seatmap团队在prod中发现了一个bug: 我们如何修复它并确保修复也是在我们的代码库中 ?
2. 构建和测试在CI中花费了很长时间:我们如何减少时间的数量?  
   5. 我们如何确保主干分支是可部署的?我们甚至应该采用基于集群的开发吗?

如何处理来自其他团队的代码更改

有一些方法可以最小化来自其他团队的代码更改的影响:  
1. 确保存储库设置不允许纯合并，而只允许重基(或快进)合并。这可以确保开发人员在合并PR之前拥有共享分支的所有最新更改(并测试了代码)。

1. 确保pr在范围内非常小。这使得风险大大降低，并且允许更好的审查(通过代码审查和手工测试)。  
   3.使用特性切换，使仍在开发中的特性对最终用户不可见(见下节)。

4注意(并让其他人注意)共享代码的更改，并通过以下方式将风险降到最低:

1. 创建代码的新版本(method/class/library)  
2. 开发这个新版本，直到它准备好为止

3.对设置了过期时间的旧方法/类/库发出弃用警告  
4. 与其他团队合作，帮助他们迁移到新版本  
5. 过期时间一过，删除旧版本

代码的所有者

另一种缓解策略是为Nx工作区中的库分配代码所有者。代码所有者(通常是一组人而不是一个人)负责对lib的所有更改，并编排上面列出的流程，以弃用和迁移到代码的新版本。

与共享代码的代码所有者一起工作可以消除一些猜测，并有助于制定修改共享代码的计划。到了整合的时候，惊喜就少了。

功能切换

有两种类型的特性切换:构建时和运行时。当应用程序加载时，初始化设置(连接/URL设置，布局设置等)建议使用构建时切换，这样我们就不用在呈现应用程序之前等待网络请求完成。

运行时特性切换通常通过对服务器上托管的设置文件的网络调用来实现。这是为不需要重新构建应用程序就可以控制设置的场景准备的。它需要仔细管理来处理缓存行为。

Trunk-based发展  
我们通常鼓励基于集群的开发，因为我们相信基本原则:

分支应该是短暂的，并且应该非常具体地用于完成单个工作  
•我们相信所有的代码都应该与其他团队尽可能的更新，这样就不会有漫长的集成过程  
•所有合并到主干分支的代码都已经在基本级别上进行过测试

实现这一点会遇到一些组织上的挑战:发布期限、需要作为研究或秘密工作的一部分独立工作的团队、在尝试实现单独的特性时在共享代码中发现意外变化等等。

下面是我们使用Monorepo的建议。它需要针对您的组织进行定制，但是可以为大多数组织提供一般工作方式的指导。

推荐的git策略  
我们的建议如下:  
•在合并代码时始终使用拉请求。公关有两个目的:  
•发起对话并获得实施的反馈  
•启动构建并运行测试和lint检查，以确保我们不会破坏构建  
•避免长时间运行的分支，不要在本地合并分支  
•执行git合并策略，确保在合并之前特性分支是最新的。这确保了在合并之前用最新的代码测试这些分支。

网站trunkbaseddevelopment.com包含许多关于基于集群的开发的非常有用的信息，是一个很好的资源。  
以下各节是最相关的:  
•特点标志  
•迁移代码的策略  
•特性分支

总结

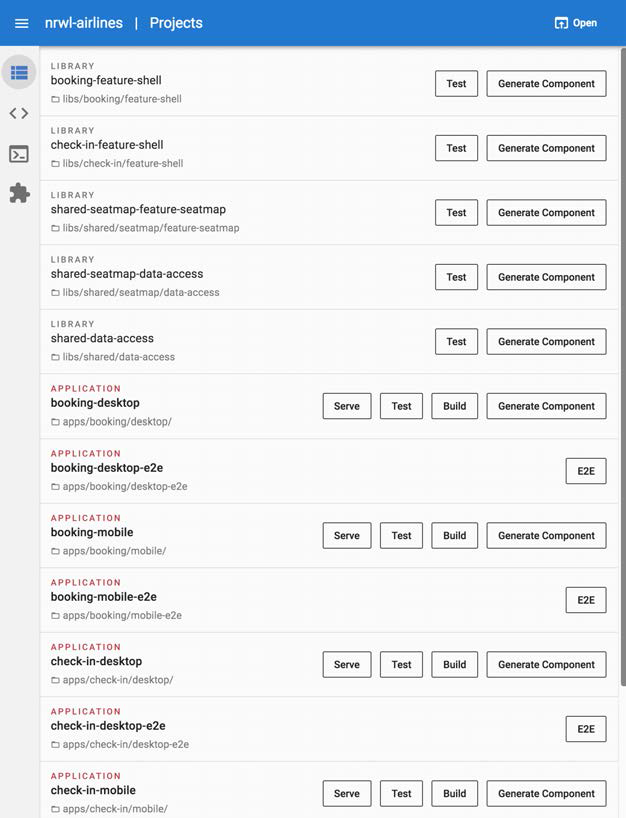
在本节中，我们简要介绍了团队在采用单一orepo工作流时遇到的一些常见问题。

处理团队之间重叠代码的一些策略:更小的PRs，要求PRs更新到目标分支，并且有一种版本代码的方法，这样就有了一个转换计划。  
为各种库设置代码所有者，这样就有责任将代码合并到该库中，并向受影响的团队负责地促进变更。  
特性标志允许团队在不影响或不受其他团队影响的环境中测试他们的代码。运行时国旗avs。讨论了构建时策略。

trunkbaseddevelopment.com是一个有价值的资源

附录A:其他环境  
用户可以选择为Nx和Angular CLI使用图形用户界面。Angular控制台允许开发者以一种可视化的方式与Nx交互。所有的CLI选项都是可见的和交互的，并且命令可以在界面中预览和直接执行(命令是使用Angular CLI执行的)。

下面的屏幕截图是Angular Console中的示例工作区。



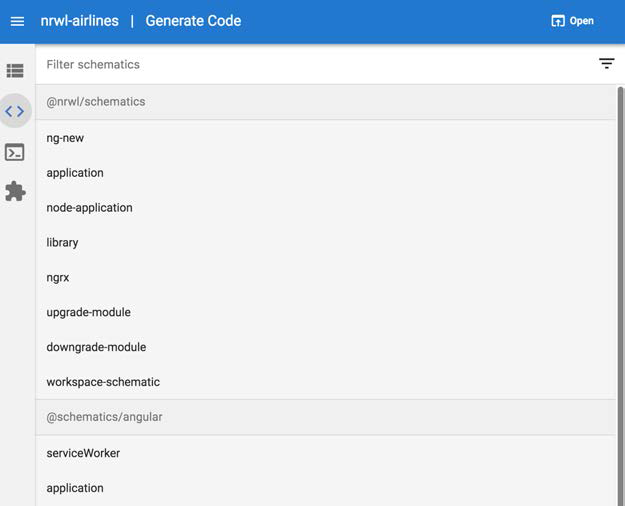
特性  
快速的行动  
工作区视图中列出的应用程序有快速访问按钮来服务、构建、测试和创建组件。在左侧菜单的Run tasks下可以找到进一步的操作。



图5。应用程序快速动作库具有快速测试和创建组件的动作。在左侧菜单的Run tasks下也可以找到进一步的操作。

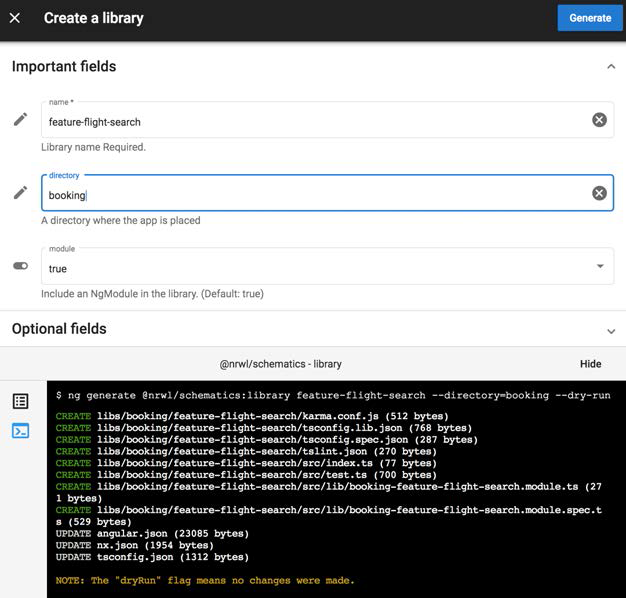


生成图表  
通过从分组列表中选择，我们可以生成工作区中检测到的任何图表(包括自定义图表)。

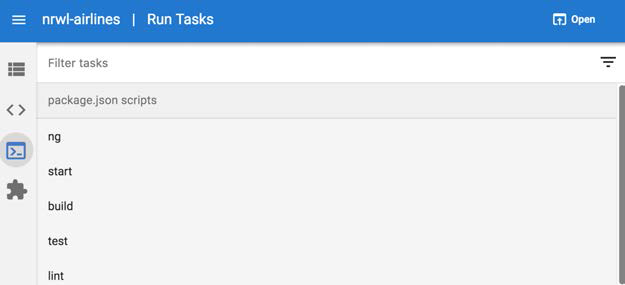


选择完成  
可以从选项列表中进行选择，而不是手动输入。在指定模块的路径时，这非常方便，因为您不需要手动输入路径。此外，切换是可视的，允许你避免在真、假、无选项等之间做出选择。

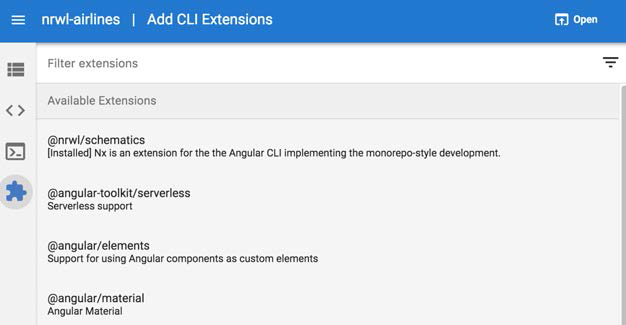
查看所有可用选项  
可以看到所有可用的命令行选项。它们被分为“必需的”和“可选的”。还有一个运行输出，显示运行命令的预期结果，以便在执行命令之前进行更正。



运行npm脚本  
Angular控制台读取所有在scripts in package下指定的命令。并允许执行它们。



第三方扩展  
第三方扩展可以从一个策划列表中添加。



在这本书中我们已经讨论了许多命令。我们可以将命令分成两个更大的组:由CLI提供的和由Nx提供的;我们还可以将后者进一步划分为可以使用影响和不能或不需要针对特定文件的对象。

CLI提供的脚本

build

• test

• lint

• e2e

• workspace-schematic

所有这些命令都可以使用npm运行，例如npm run lint。

脚本提供的Nx格式  
Nx增加了对prettier的支持，可以根据prettier设置格式化文件。有两个命令:.

format:write 修改所有文件

• format:check 在prettier中列出所有不符合规则的文件

这些在本书的第3部分进行了讨论。

Update

Nx允许您用这个命令更新它。check只检查并通知您是否需要更新。

ep-graph  
Nx显示了应用程序和库之间的项目依赖关系的图形视图。这有助于了解哪些项目可能会受到库或应用程序更改的影响。

依赖性图将在本书的第3部分中讨论。

affected的命令  
affected的命令包含一个“affected:”的前缀，并针对给定操作的特定文件。  
支持的行动如下:

affected:apps

• affected:libs

• affected:build

• affected:e2e

• affected:test

• affected:lint

• affected:dep-graph

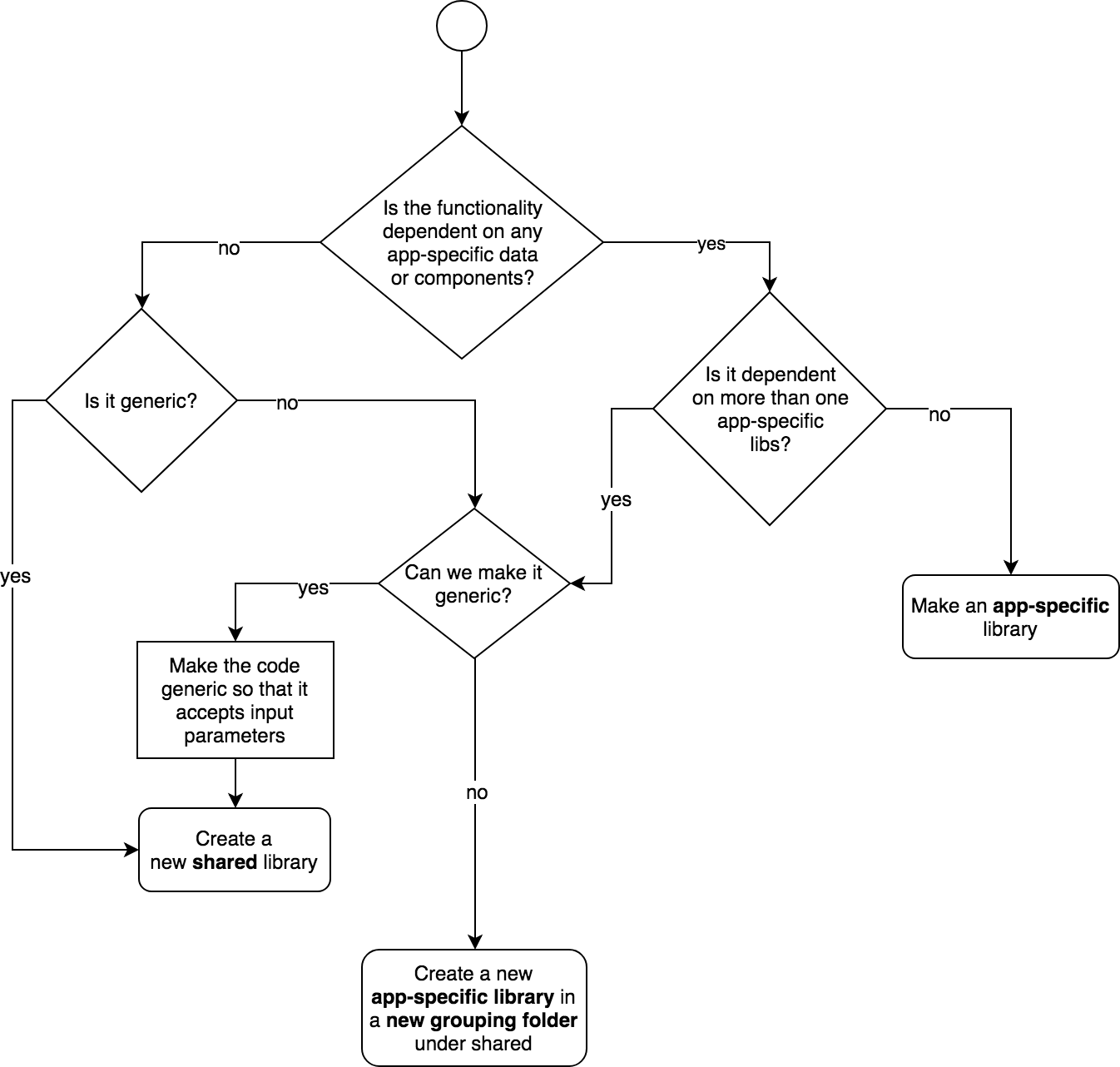
我们可以通过以下方式对文件进行定位:  
比较两个git提交(使用它们的SHAs或使用分支)  
•-未跟踪:当前工作分支上所有未跟踪的文件  
未commited:检出这个分支后被修改的所有文件  
•特定文件:以逗号分隔的文件列表  
所有文件:如果你想显式地使用所有文件  
•仅失败:仅使用上次失败作业的文件

这些命令将在本书的第4部分中讨论

更新Nx  
如果您使用Nx 6.0.0创建了一个Nx工作区，然后决定升级Nx版本，那么Nx update命令可以根据需要修改配置文件和源文件，以使您的工作区配置匹配新版本Nx的需求。

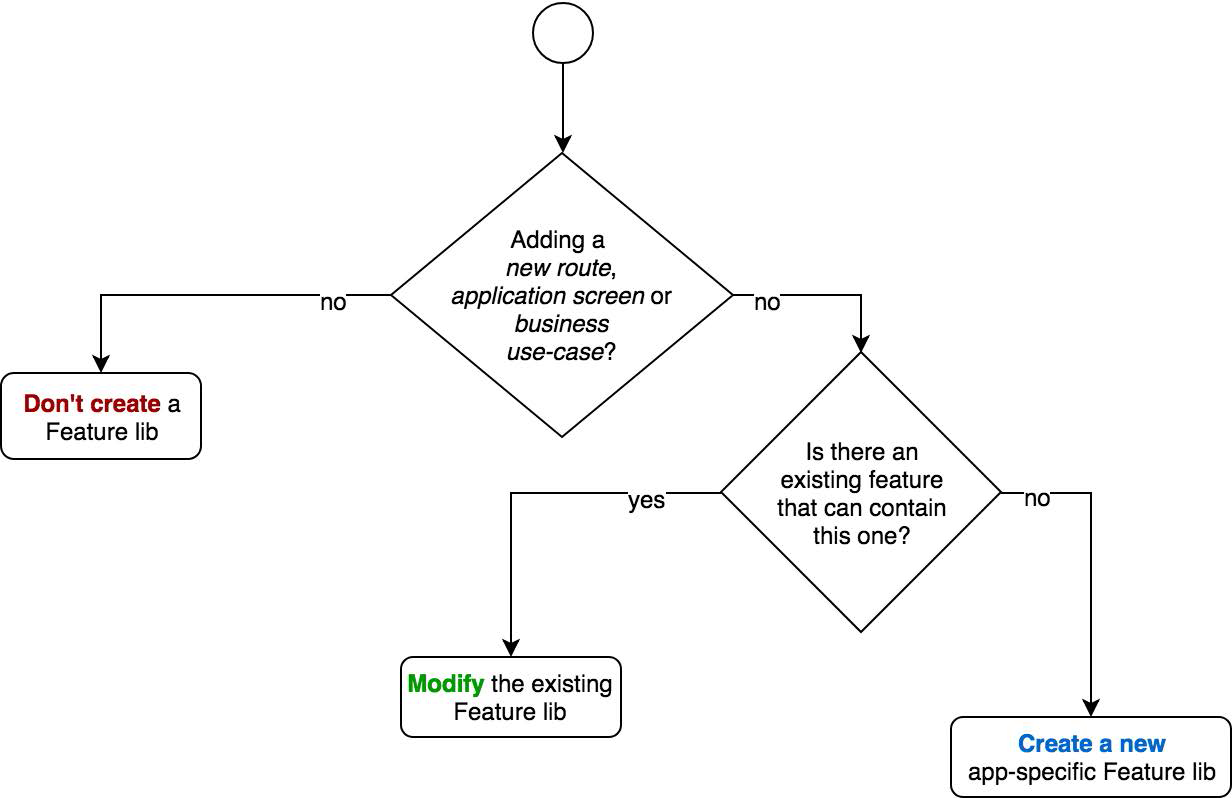
npm run update

我应该在哪里创建新库?



我应该重用还是创建一个特性库?

为了回答这个问题，问问你自己:  
•我添加了一个新路线吗?  
•我添加了一个新的“应用程序屏幕”吗?  
•我是否在处理一个特定于应用程序的用例?  
创建新的特性库或修改现有的特性库。所有(大多数)特性库都是特定于应用程序的，因此您需要选择一个application section目录来放置它。



如何从应用程序中提取特性库?  
提取一个特定于应用程序的库的过程是这样的:  
•运行ng g lib-name -directory=app-name(添加——routing，——lazy，和——parentModule(如果需要))  
•复制应用程序的内容(除了index之外的所有内容)。html,主要。ts, polyfills。ts, app.module。ts、app.component.ts和其他“全局”文件)放入库中。

更新app.module。删除所有不再需要的导入和声明。  
保留forRoot调用，但是我们可能必须更新它们(例如，StoreModule.forRoot({}))。  
•用特性库中的forFeature调用替换所有的forRoot调用。我们必须为存储引入一个前缀，因此我们必须更新测试和生产代码来反映这一点。

将特性库中的BrowserModule替换为CommonModule, BrowserModule只能导入一次，应该在app层进行导入。  
•创建一个集成测试，在不需要包含应用程序的情况下练习新的模块。

自定义材料模块  
在自定义视图中使用Angular材质组件时，最好的做法是鼓励开发人员创建一个自定义材质模块来管理特定Angular的导入

这是一种特殊形式的库模块;通常在path /lib /common/ui/custom-material/下管理。

例子NgModule定制的material.module。下面的ts已配置为仅加载材料按钮，卡片，图标和ProgressBar组件。

import { NgModule } from '@angular/core';

import {

MatButtonModule,

MatCardModule,

MatIconModule,

MatProgressBarModule

} from '@angular/material';

import { BidiModule } from '@angular/cdk/bidi';

@NgModule({

exports: [

MatButtonModule,

MatCardModule,

MatIconModule,

MatProgressBarModule,

BidiModule

]

})

export class CustomMaterialModule {}

任何使用这些组件(在模板中)的库或应用程序只需导入CustomMaterialModule。当需要额外的组件时，只需要修改这个CustomMaterialModule来添加所需的导入和导出。

联系，扩大你的开发能力  
重新设想如何解决最棘手的挑战，并使用最好的开发伙伴、最好的工具和最新的实践来构建一个令人惊叹的应用程序。为未来构建软件  
Nrwl。我们的现代开源开发工具和实践使软件团队能够实现更好的协作，并建立长期的团队能力和成功。请通过hello@nrwl联系我们。关于您的项目的io或填写表格https://nrwl.io/services。期待您的回复!