TypeScript

多场景开发实践

Best practices of TypeScript and Dev in Alibaba

陈仲寅（花名：张挺）

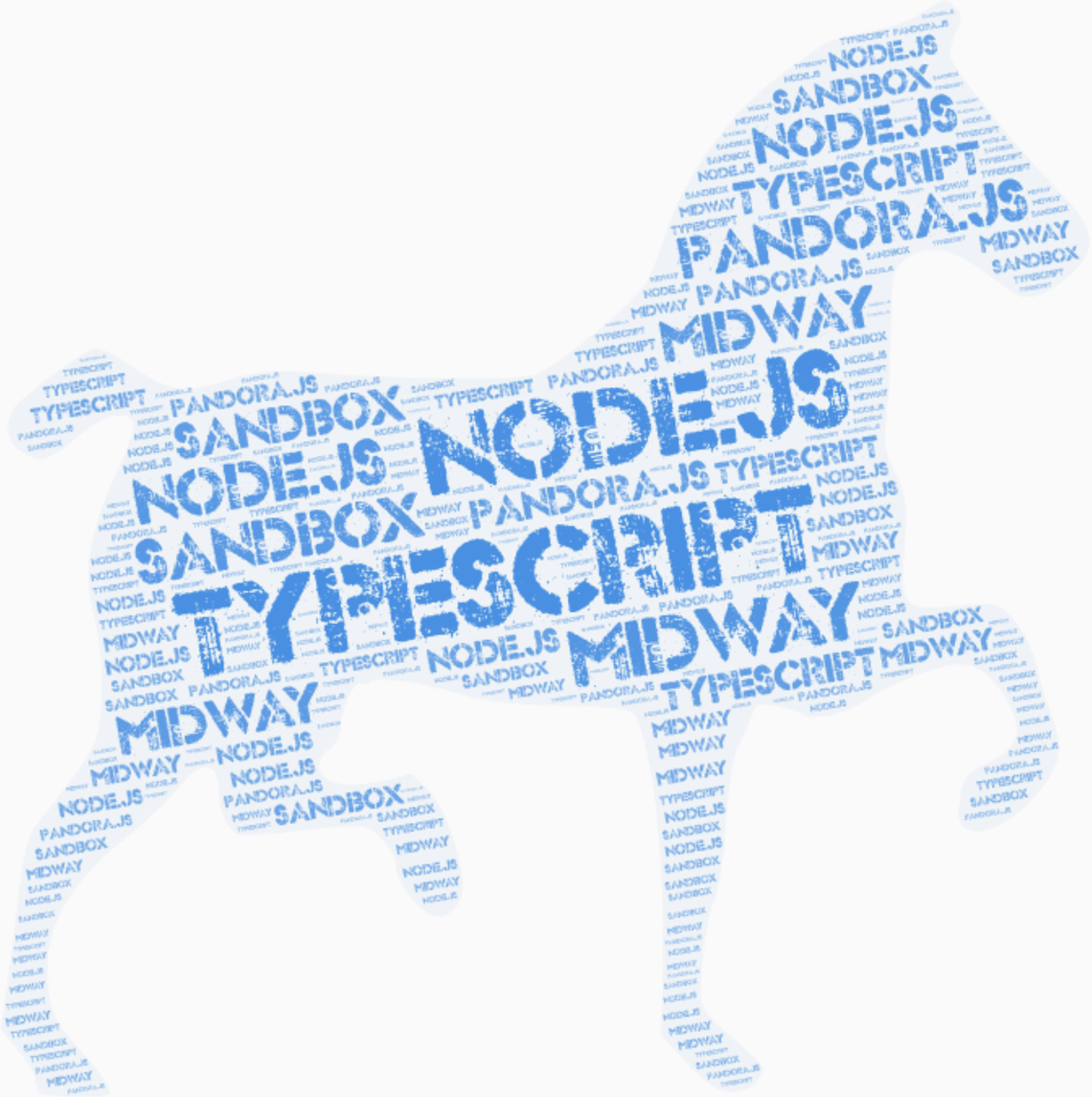
就职于 阿里巴巴淘宝技术部 MidwayJS 团队

zhangting@taobao.com

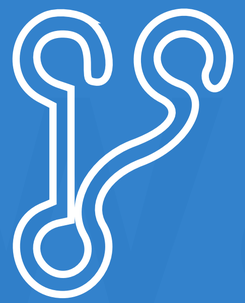
@czy88840616

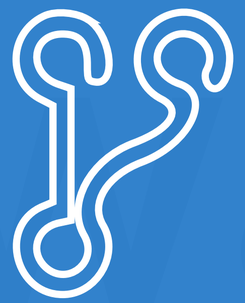
@czy88840616

https://github.com/czy88840616



MidwayJS





Midway

Pandora.js

Sandbox

Midway

面向未来的全栈开发框架

Review

面向过去，接受历史

Solve

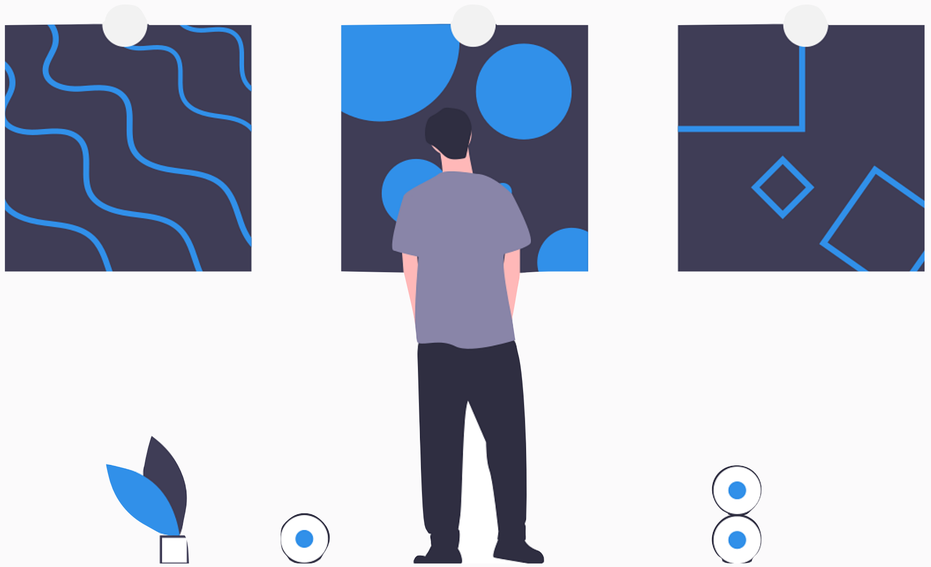
面向现在，解决问题

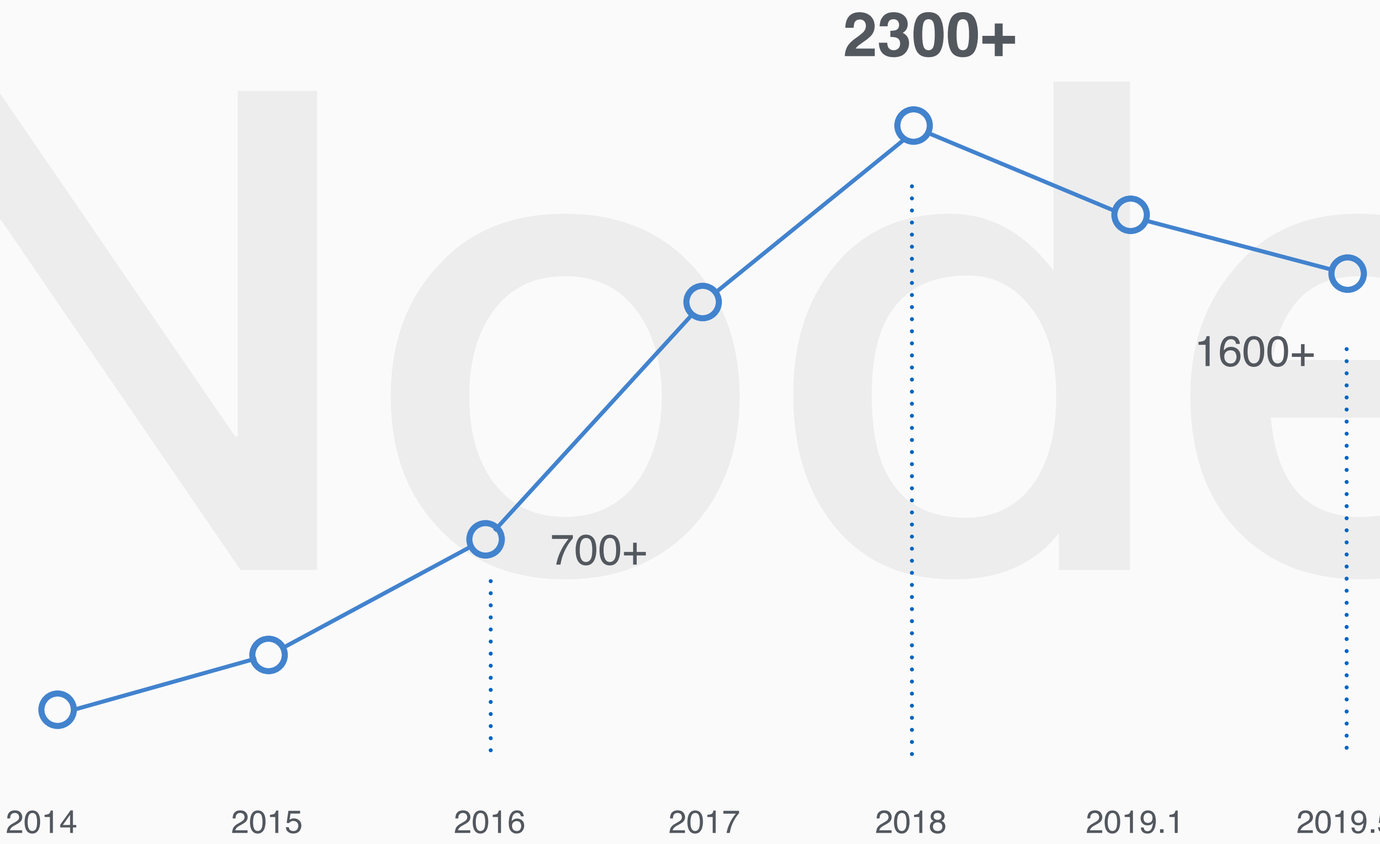
Explore

面向未来，探索未知

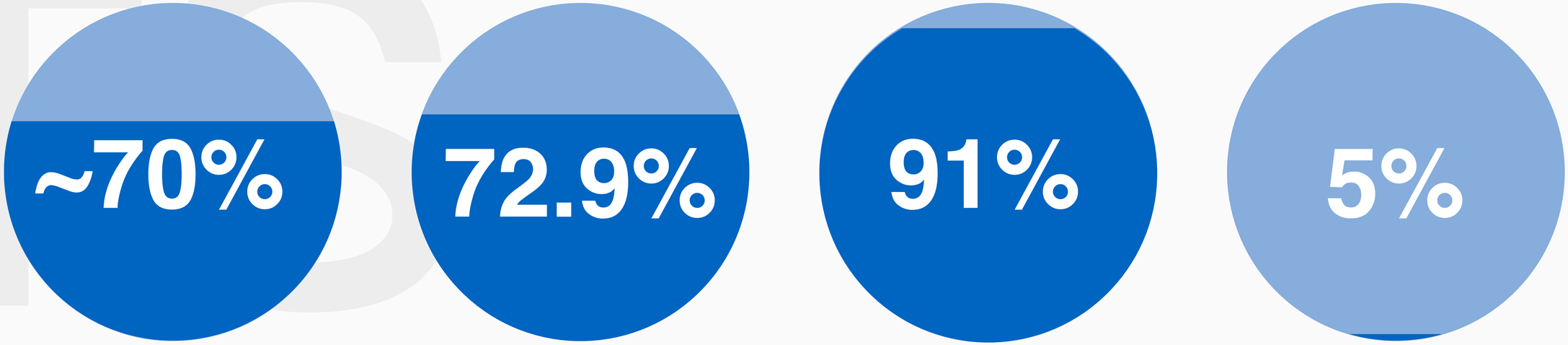
Ready ?

来看看数据





如今集团总约 **1600** 个应用



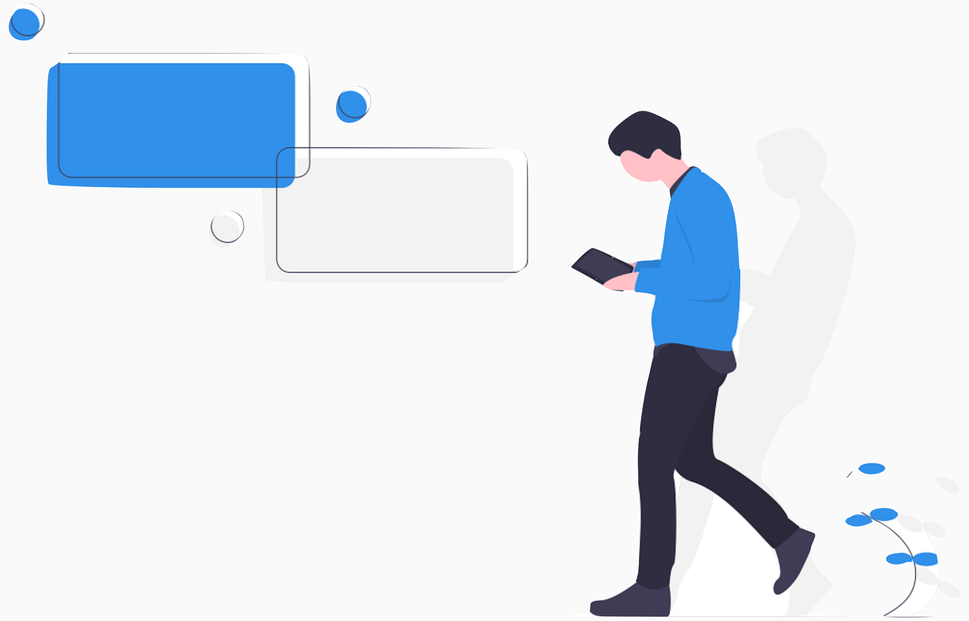
BFF

使用框架

接入治理

使用 TS

来看看问题



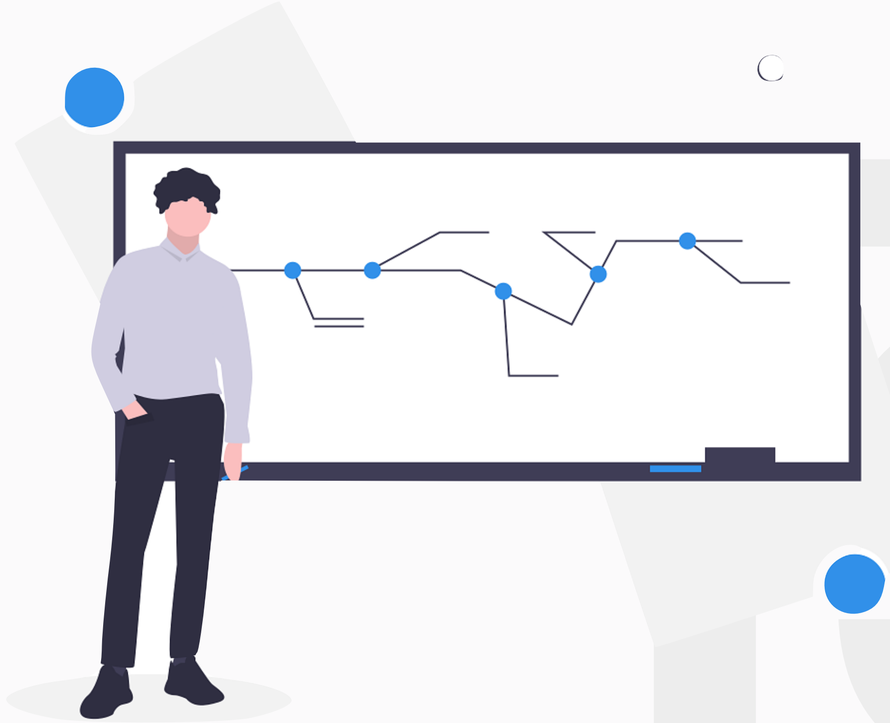
复杂度逐步增加 - **全栈应用**

面向外部用户 - **大流量**

成为中流砥柱 - **核心应用**

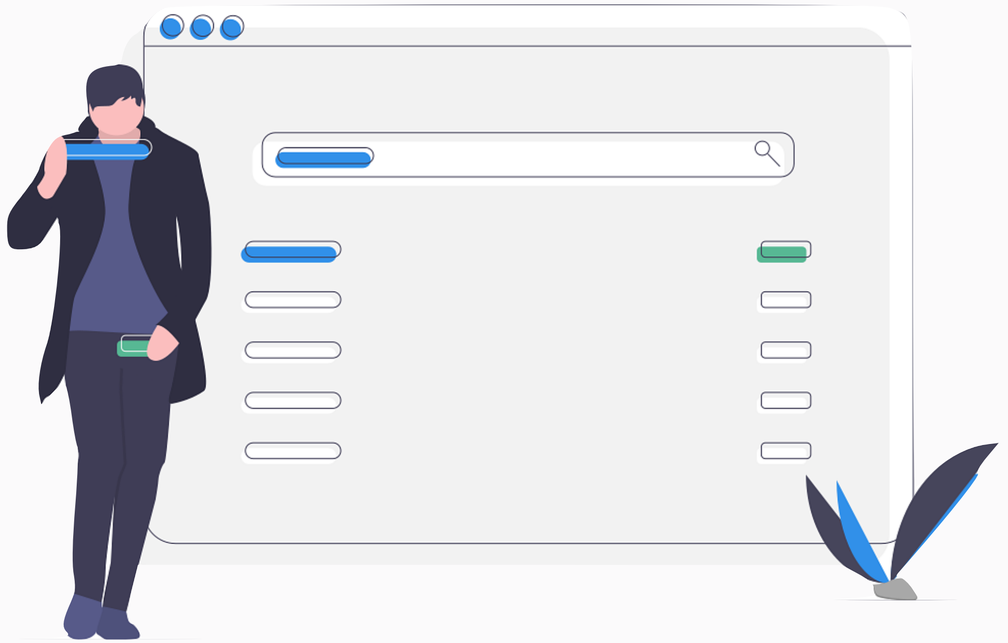
调用 http 服务，没有调用定义

提供 RPC 接口时，需要写 JSDoc



TS

Node.js 测试靠人肉



Import TypeScript

我们都知道 TypeScript 的优势

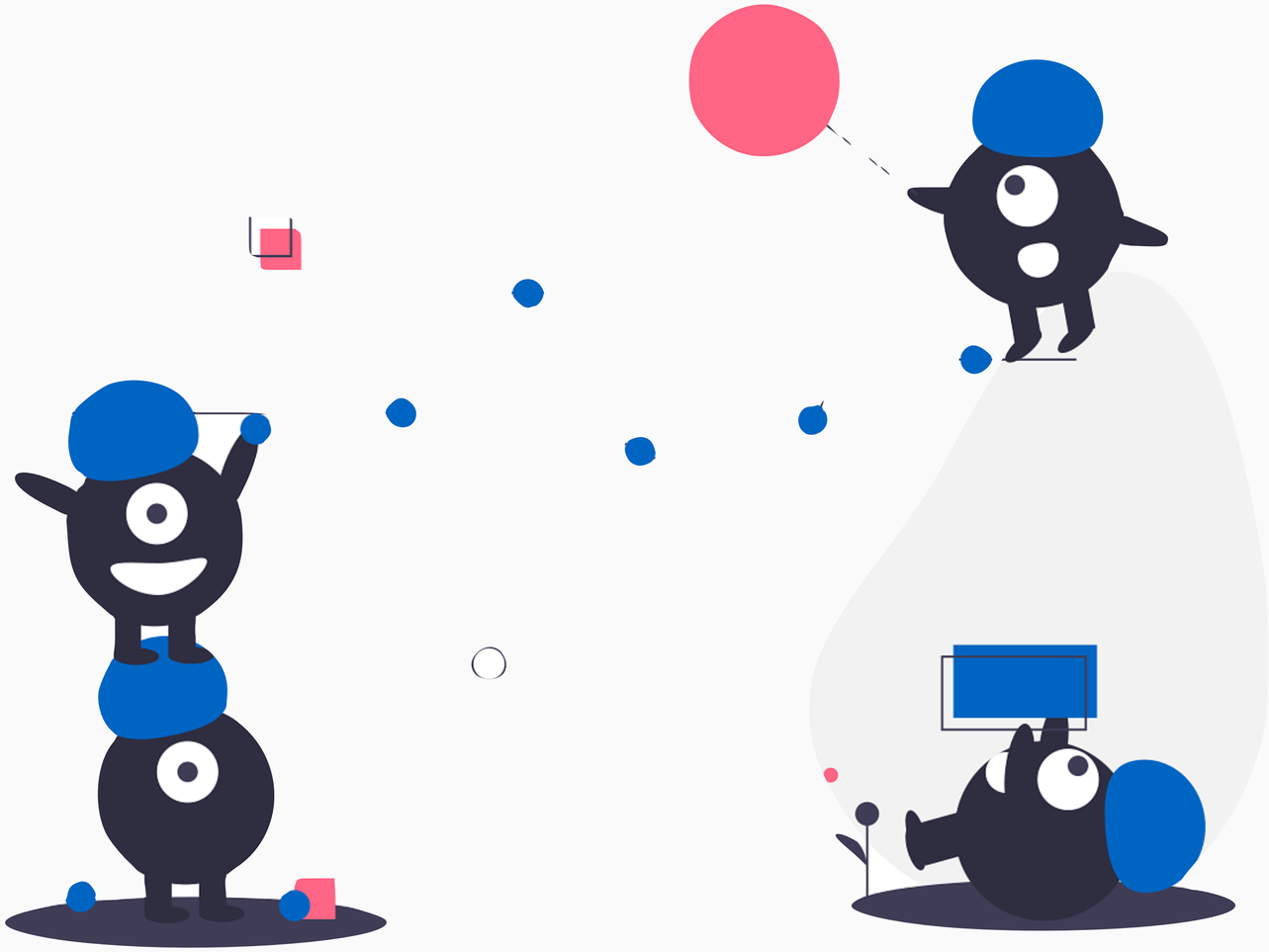
1 类型描述

2 更多的 Feature 支持

3 面向接口编程

TS

个人开发面向类型编码，
协作时面向接口编程

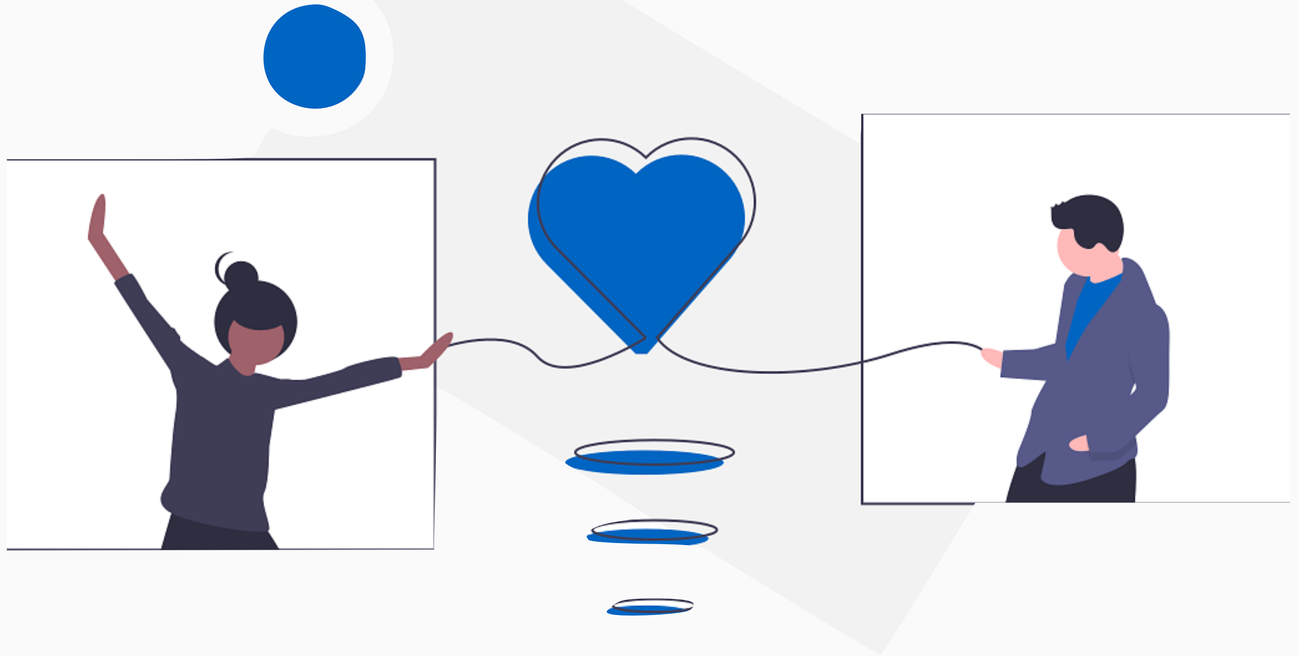


TS

开发时增加更多接口定义，
数据定义，参数定义

TS

跨协议转换



进入正题

我们是来解决问题的

Why is Midway

by TypeScript

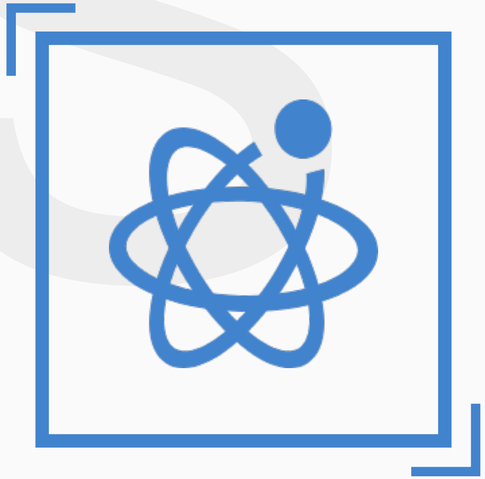
Egg 是个好框架

Egg 有自己解决的东西

Midway 解决的痛点不同，不是非常适合我们的情况

定位不同

在内部体系中，Egg作为底层框架，不直接使用



场景不同

Egg 解决的是 BFF 场景，而淘宝有不少全栈场景

场景不同

Egg 解决的是 BFF 场景，而淘宝有不少全栈场景

除了明确意义的 controller service 承载了太多的职能。

场景不同

子目录缺乏支持

Egg 解决的是 BFF 场景，而淘宝有不少全栈场景

体验不同

我们希望引入 TypeScript 原生的体验

体验不同

Egg 解决的是 BFF 场景，而淘宝有不少全栈场景

js/ts 目录混合

体验不同

Egg 解决的是 BFF 场景，而淘宝有不少全栈场景

```
const Controller = require('egg').Controller;
class PostController extends Controller {
  async create() {
    // TODO
  }
}
module.exports = PostController;
```

class 用法，无法多继承

体验不同

Egg 解决的是 BFF 场景，而淘宝有不少全栈场景

杂糅的 app/ctx 合并机制

第一代设计

解决复杂度问题

尝试引入 IoC 解决复杂业务的问题

配置

使用描述文件创建实例（xml）

很早就开始使用 ioc 注入的方式，苦于 js 一直没有很好的实践产品。

```
<!-- directly require -->
<object id="ctor:obj3" path="obj3" direct="true">
</object>

<object id="ctor:obj3" path="obj3" async="true" construct-method="getStatic">
</object>

<object id="ctor:obj3" path="obj3" external="true" autowire="false">
</object>

<!-- plain object -->
<object id="ctor:obj4" path="obj4">
  <property name="k1" value="v1"></property>
  <property name="num" value="1" type="number" />
  <property name="things">
    <list>
      <value>asdfa</value>
      <value>123</value>
    </list>
  </property>

  <property name="things1">
    <list>
      <map>
        <entry key="foo" value="bar" />
        <entry key="foo">
```

配置

使用描述文件创建实例（xml）

升级

自动化装配

使用 inversify 的自动化

优势：简单，功能精简

劣势：满足不了自定义的需求(xml)

```ts
import { injectable, inject } from "inversify";
import "reflect-metadata";
import TYPES from "./types";

@injectable()
class Katana implements Weapon {
    public hit() {
        return "cut!";
    }
}

@injectable()
class Shuriken implements ThrowableWeapon {
    public throw() {
        return "hit!";
    }
}

@injectable()
class Ninja implements Warrior {

    private _katana: Weapon;
    private _shuriken: ThrowableWeapon;

    public constructor(
        @inject(TYPES.Weapon) katana: Weapon,
        @inject(TYPES.ThrowableWeapon) shuriken: ThrowableWeapon
    ) {
        this._katana = katana;
        this._shuriken = shuriken;
    }

    public fight() { return this._katana.hit(); };
    public sneak() { return this._shuriken.throw(); };

}

export { Ninja, Katana, Shuriken };
```


import "reflect-metadata";
import { injectable, inject } from "inversify";
import TYPES from "./types";
@injectable()
class Katana implements Weapon {
public hit() {
return "cut!";
}
}
@injectable()
class Shuriken implements ThrowableWeapon {
public throw() {
return "hit!";
}
}
@injectable()
class Ninja implements Warrior {
private _katana: Weapon;
private _shuriken: ThrowableWeapon;
public constructor(

```
@injectable()
class Shuriken implements ThrowableWeapon {
    public throw() {
        return "hit!";
    }
}

@injectable()
class Ninja implements Warrior {

    private _shur  Weapon;
                   ThrowableWeapon;

    public constructo
        @inject(TYPES on) katana: Weapon,
        @inject(TYPES owableWeapon) shuriken: ThrowableWeapon
    {
                   atana;
        this._katan = shuriken;
    }

    public fight() { return this._katana.hit(); };
    public sneak() { return this._shuriken.throw(); };

}

export { Ninja, Katana, Shuriken };
```

自研

自研 injection

100%满足自己的需求，整体代码自己能掌握，同时更简单

```
@provide()
export class UserService {

  @inject()
  userManager: UserManager = new UserManager();

  async getUser(id: number) {
    return this.userManager.get(id);
  }

}
```

自研

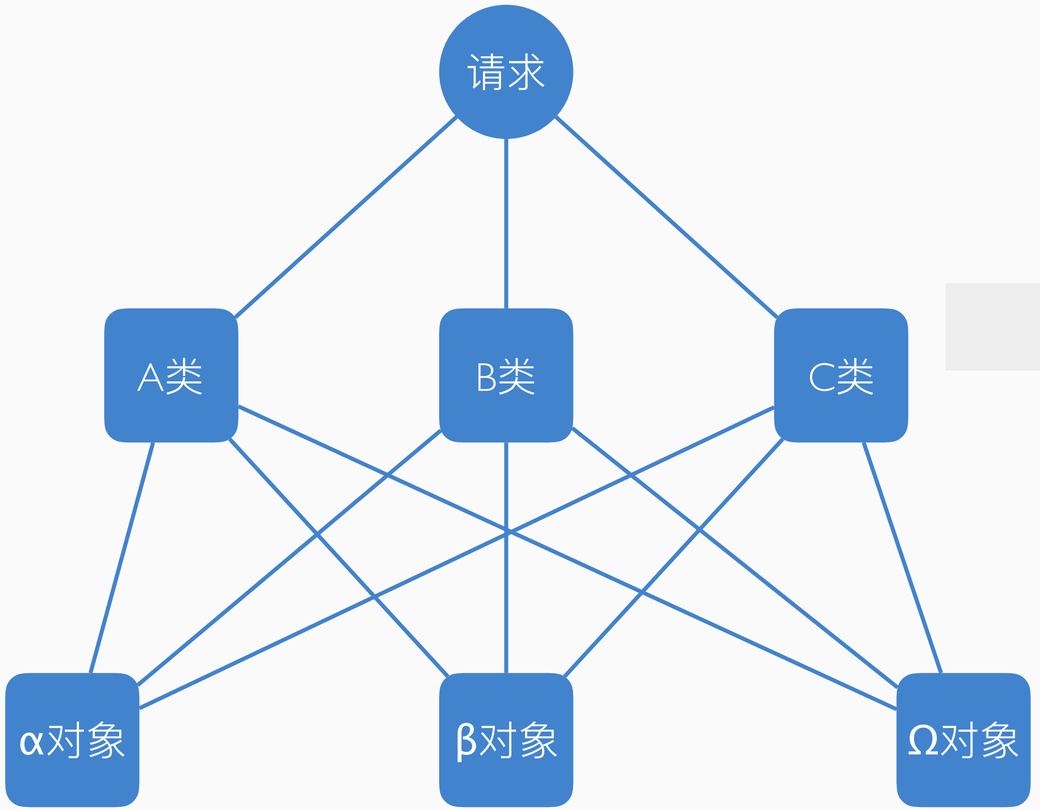
自研 injection

```
@provide()
export class UserService {

  @inject()
  userManager: UserManager;

  async getUser(id: number) {
    return this.userManager.get(id);
  }

}
```

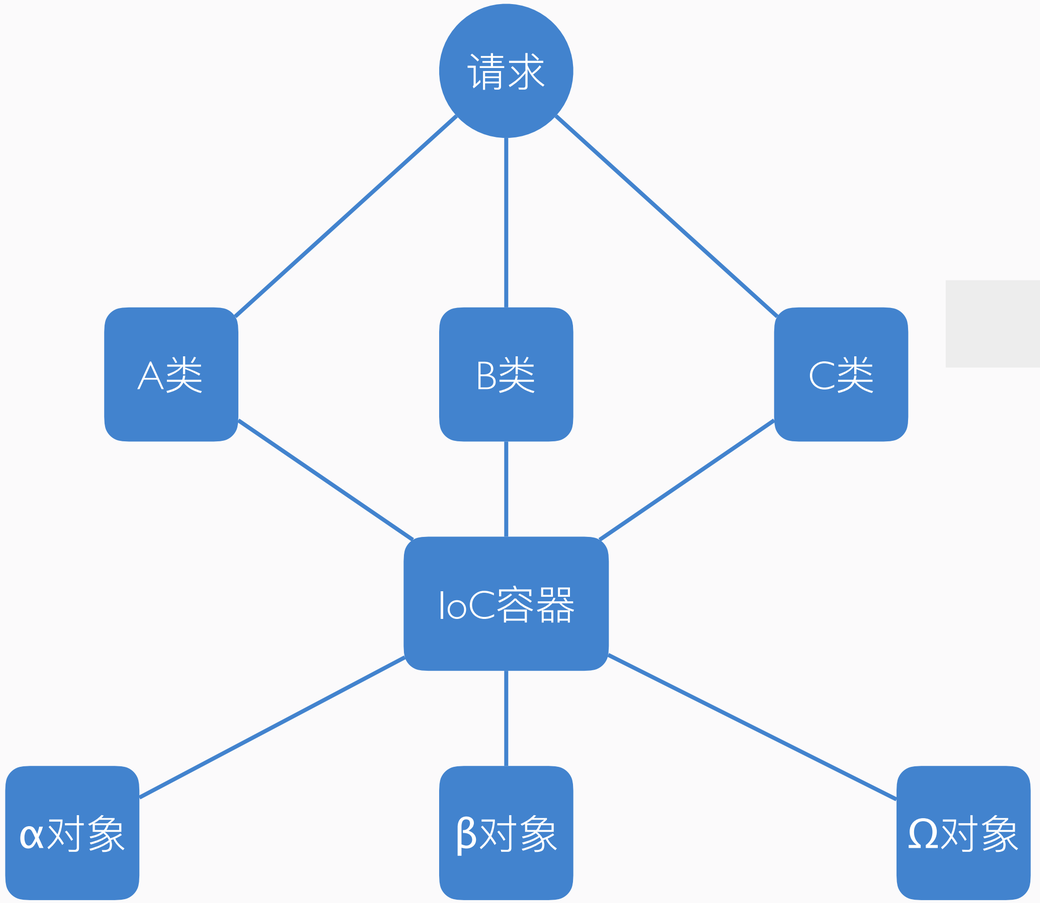
关键字

自研

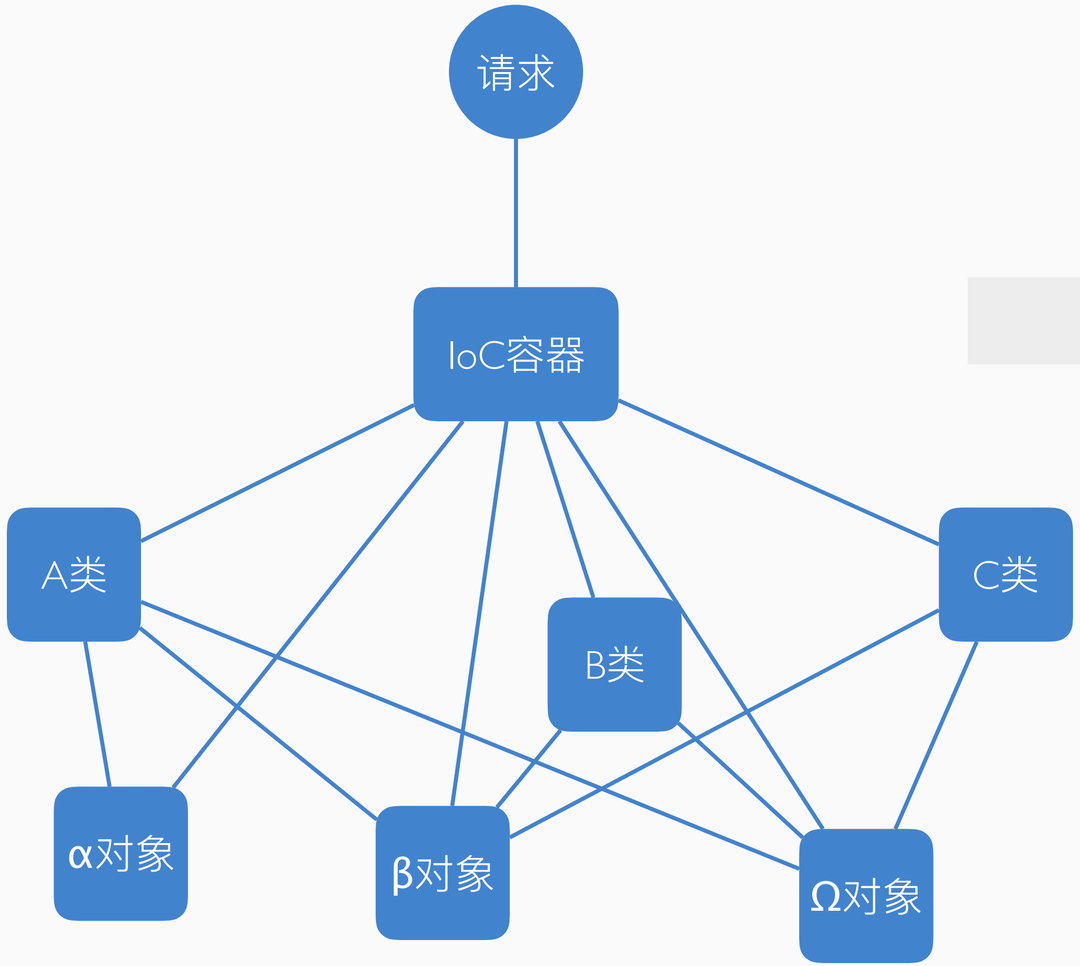
自研 injection



TS



TS



TS

解决体验问题

尝试让用户体感一致，不再考虑写法

服务层写法一致，体验一致

体验上的问题

1、写法上的不一致 - Class / Function

2、多实现上的不一致 - 无法方便的继承

3、代码洁癖上的问题 - 编译目录分离

体验

统一使用 class/interface

为了良好的使用 IoC，我们将整个 Midway 修改为了 OO 的模型，所有的东西都通过 class 来编码，这样也可以更好的借鉴 java 的思想，另一方面可以通过接口来解决多实现的架构。

```
@provide()
return class UserService {

  @inject()
  userManager: UserManager;

  async getUser(id) {
    return this.userManager.getInfo(id);
  }
}
```

体验

统一使用 class/interface

体验

增加接口描述

集团中间件缺少描述，补充了大概 10 几个类的定义，
这样通过注入就不需要额外再摸索。

```ts
interface HsfClient {
}

interface DiamondClient {
}

interface TairClient {
}

interface OSSClient {
}
```

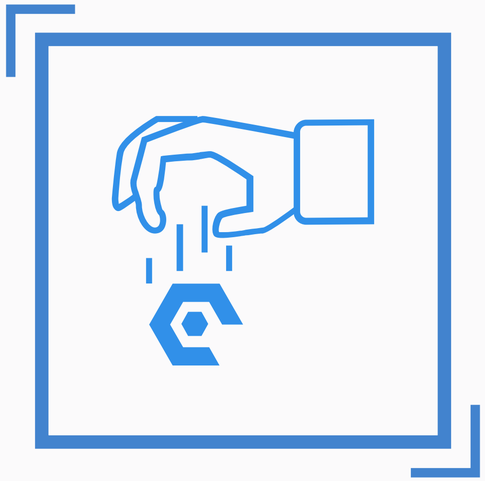
体验

增加接口描述

第二代设计

解决 Egg.js 体验

虽然服务层的使用已经解决了，但是和 egg 耦合的部分还是沿用了 egg 的写法，虽然有变通的办法，但是需要在体验上更进一步。



和 Egg.js 解耦

保留 Egg.js 的能力同时做出提升

1、保留原有能力，可以快速迭代升级（继承 egg-loader)

2、实现装饰器的方式，利用现有的 API，附加能力，比如 loader 的扩展，loadController，load系列的

```ts
// config.default.ts
export const myConfig = {
  defaultUser: 'harry'
}


// user.ts
@provide()
export class UserService {

  @config('myConfig')
  myConfig;


  async getUser() {
    const defaultUser = myConfig['defaultUser'];
    // TODO
  }

}
```

和 Egg.js 解耦

保留 Egg.js 的能力同时做出提升

```ts
// user.ts
@provide()
export class UserService {

  @plugin('mysql')
  mysql;


  async getUser() {
    const user = mysql.query('select *');
    // TODO
  }

}
```

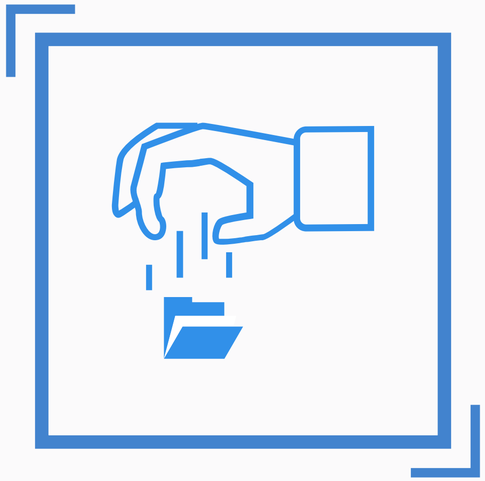
和 Egg.js 解耦

保留 Egg.js 的能力同时做出提升

```ts
// user.ts
@provide()
export class UserService {

  @inject()
  ctx;


  async getUser() {
    const user = this.ctx.mysql('select *');
    // TODO
  }

}
```

和 Egg.js 解耦

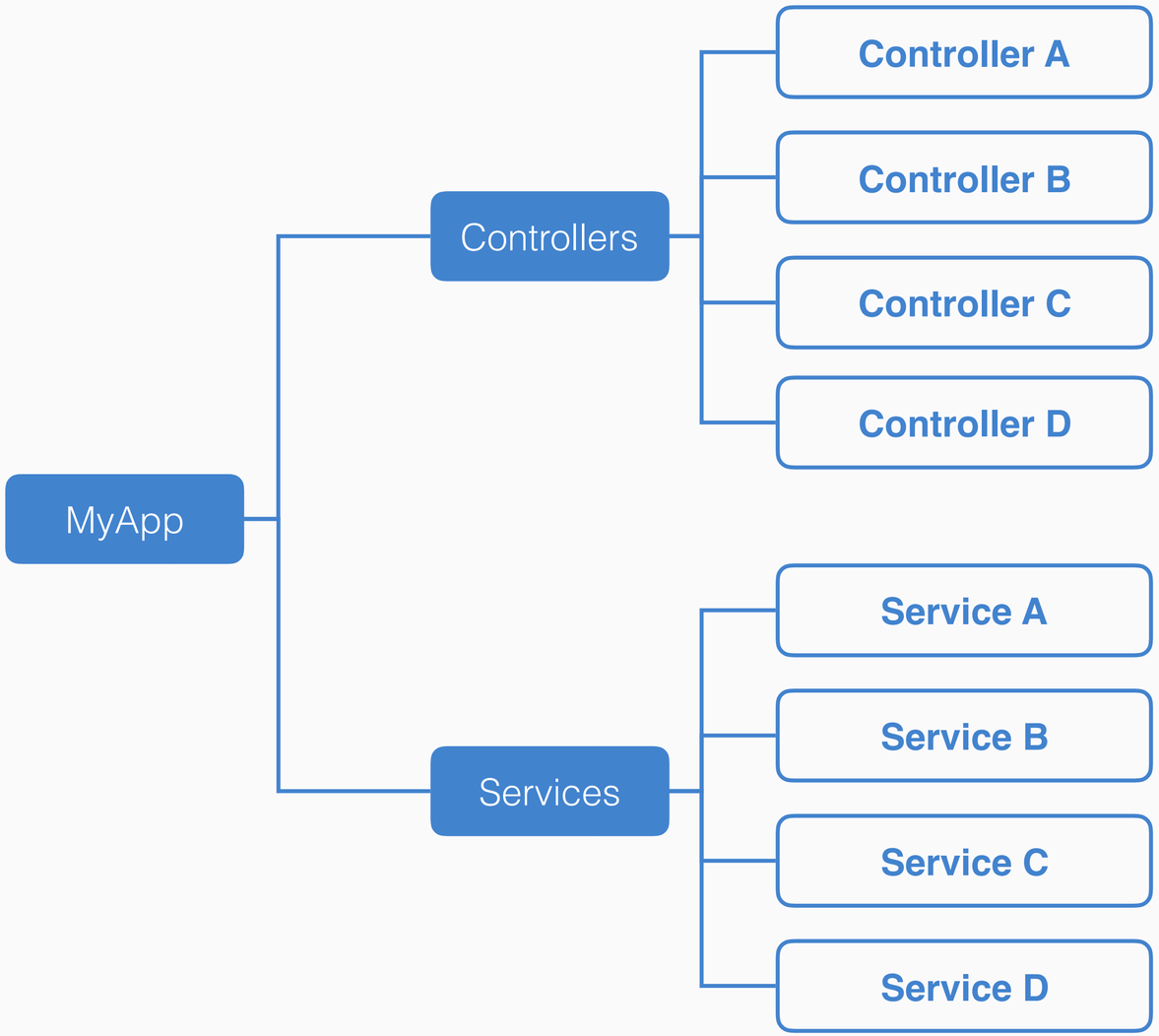
保留 Egg.js 的能力同时做出提升



和目录解耦

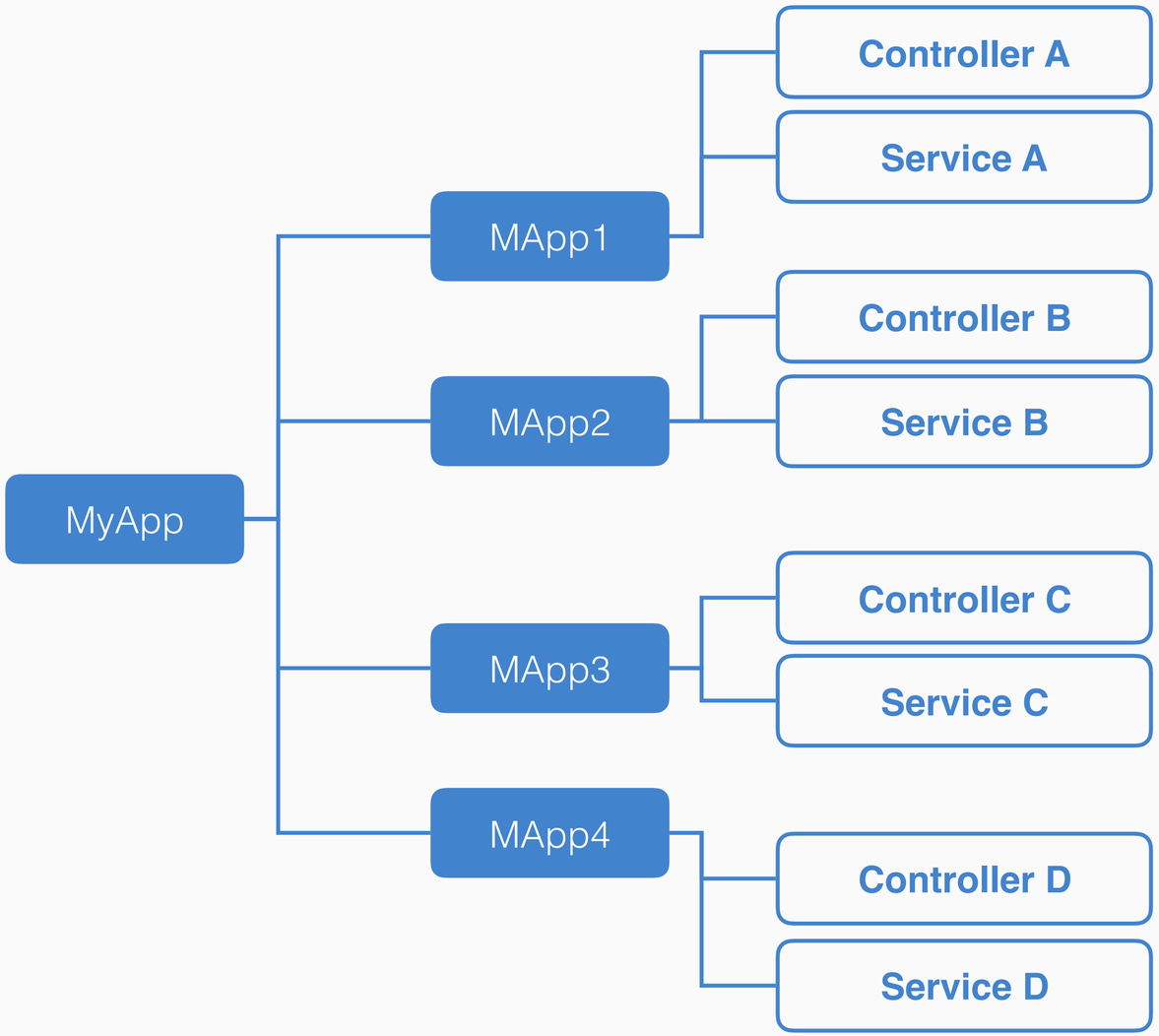
弱化目录约定

做完了之后，我们发现了，加载模式的自动化，通过扫描完全不需要目录约定。



和目录解耦

弱化目录约定



和目录解耦

弱化目录约定

和自己解耦

基于新的装饰器开发模型

实现和定义分离

和自己解耦

基于新的装饰器开发模型

@controller	@get/post	@priority
@plugin	@config	@logger
@hsf	@schedule	@func

Import { controller } from '**@midwayjs/decorator**'

休息一下 。

面向未来的设计

跨场景的设想

定义与实现分离

Web场景/自启动场景/更多自定义场景。

正好碰上了 Serverless 的浪潮，作为集团唯一的 Node.js 架构团队，集团的领航者当仁不让的投奔进去。

代码重构实践

分离通用层

分离 midway-core，将通用的能力都放在这层

代码重构实践

分离通用层

1、自扫描注入 ioc 的能力

2、适配 midway 的请求作用域能力

3、兼容原有装饰器的能力

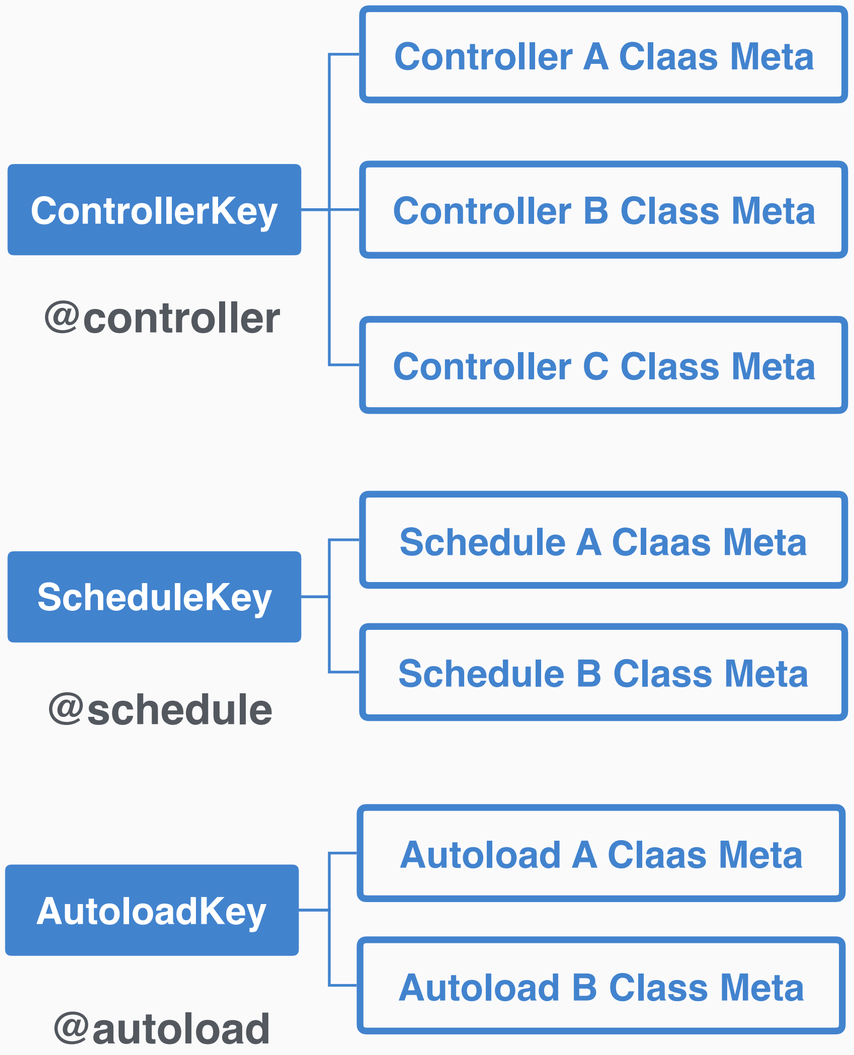
代码重构实践

分离通用层

1、自扫描注入 ioc 的能力

2、适配 midway 的请求作用域能力

3、兼容原有装饰器的能力



代码重构实践

分离通用层

Decorator Manager 承载着自定义装饰器的核心

First: saveModule

```
function autoload() {
    return function (target, propertyKey, descriptor) {
        saveModule('autoload');
    }
}
```

Second: listModule

```
const modules = listModule('autoload');
// TODO something
```

代码重构实践

分离通用层

Decorator Manager 承载着自定义装饰器的核心

Express/Koa

传统 Web 场景初试

我们花了大概 200 行代码，简单实现了在 express ，相同的代码，但是不同的场景。同样的，我们尝试在 typeorm 领域，扩充我们的装饰器。

```ts
import { provide, inject, controller } from 'midway-koa';

@provide()
@controller('/home')
export class UserController {

  @inject()
  ctx;

  @get('/')
  async index() {
    this.ctx.body = 'hello world';
  }

}
```

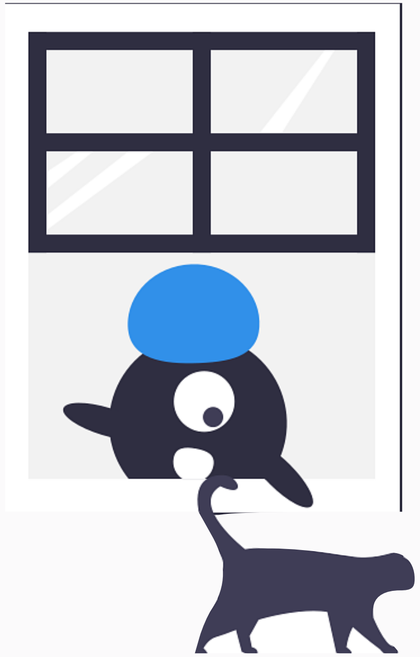
Express/Koa

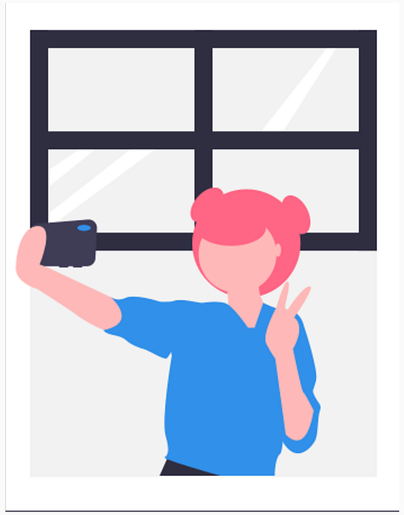
传统 Web 场景初试

```ts
import { provide, inject, controller } from 'midway-express';

@provide()
@controller('/home')
export class UserController {

  @inject()
  ctx;

  @get('/')
  async index(req, res, next) {
    res.send('hello world');
  }

}
```

Express/Koa

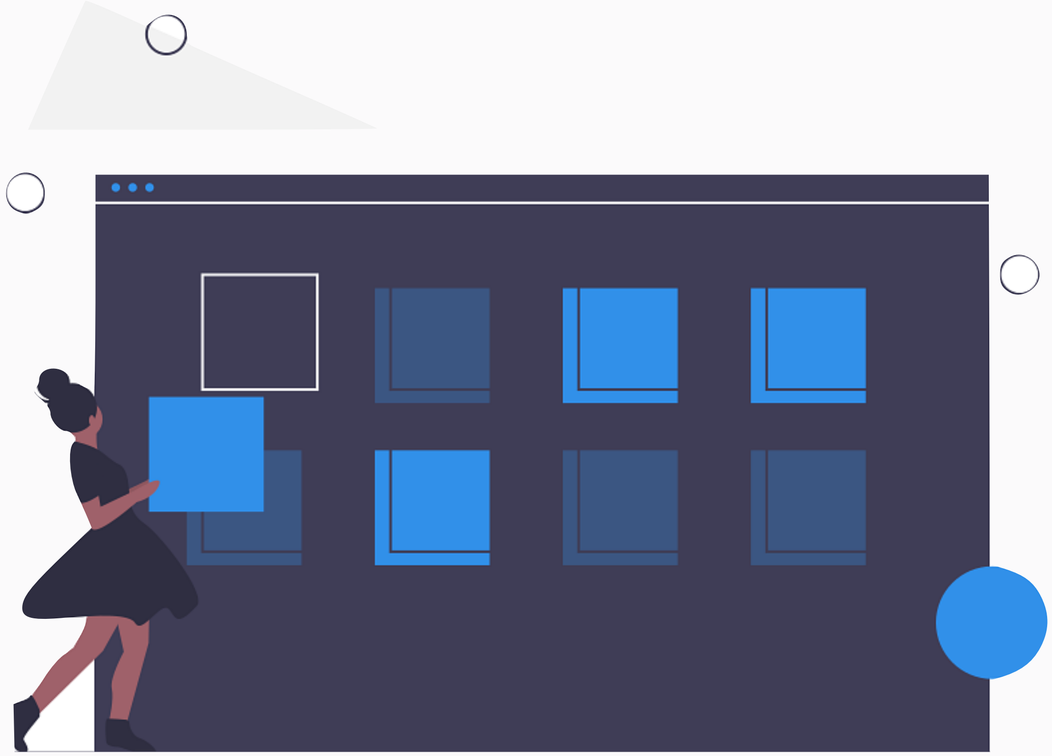
传统 Web 场景初试





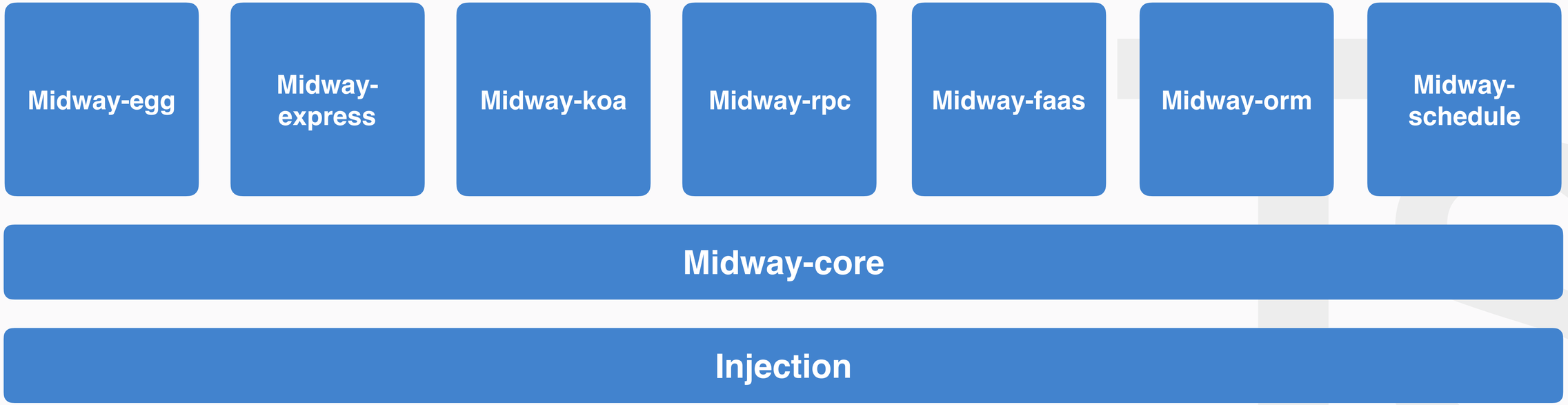
跨场景

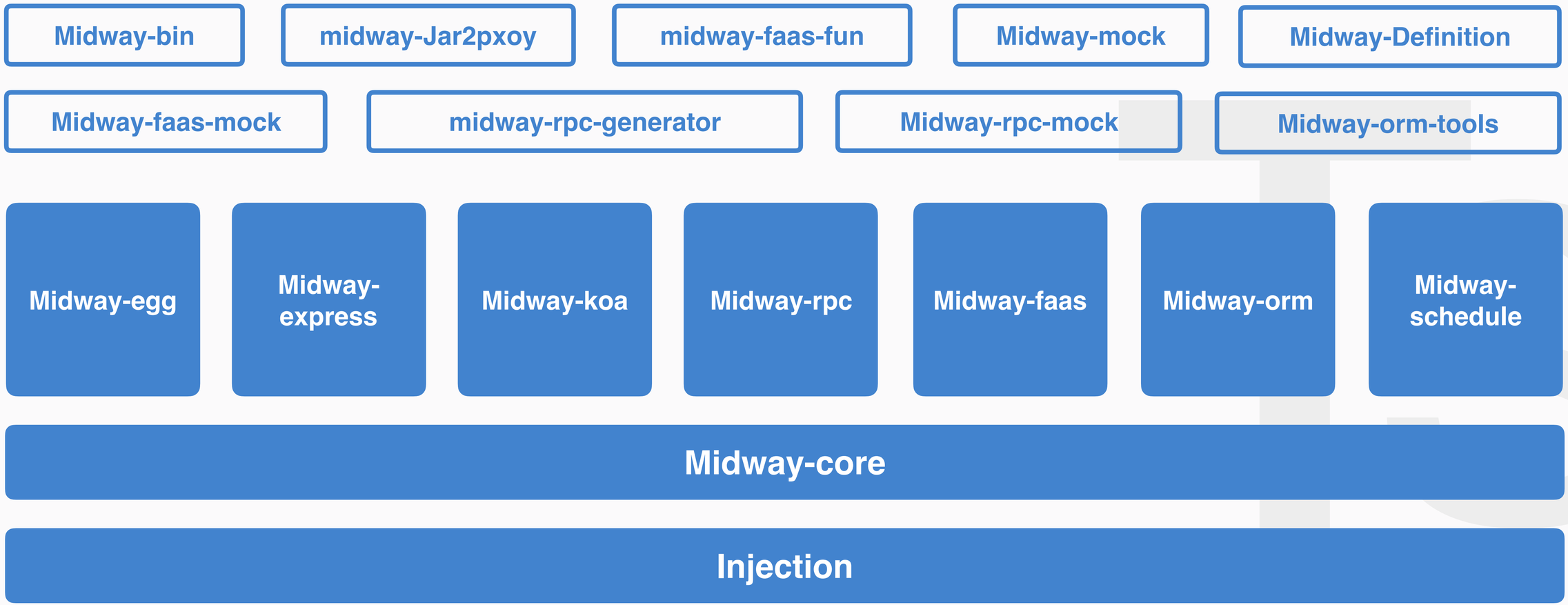
不同场景公用大部分装饰器



同场景跨类型

类似场景复用装饰器


Midway-egg
Midway-express
Midway-koa
Midway-rpc
Midway-faas
Midway-orm
Midway-schedule
Midway-core
Injection


Midway-bin
midway-Jar2pxoy
midway-faas-fun
Midway-mock
Midway-Definition
Midway-faas-mock
midway-rpc-generator
Midway-rpc-mock
Midway-orm-tools
Midway-egg
Midway-express
Midway-koa
Midway-rpc
Midway-faas
Midway-orm
Midway-schedule
Midway-core
Injection

Serverless 场景

前端绝佳的机会

FaaS 初探

一些概念

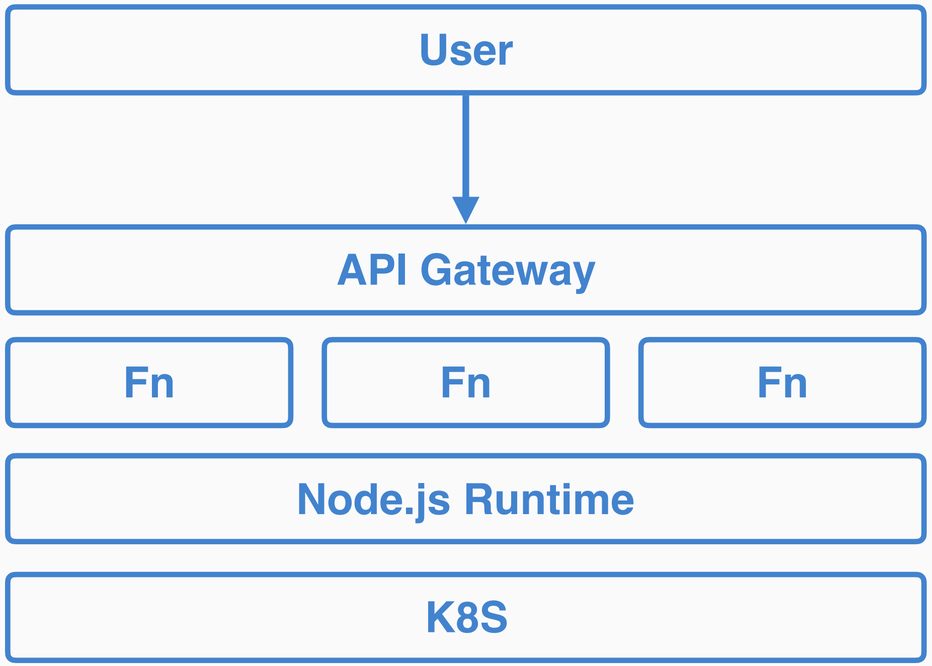
平台

网关

函数

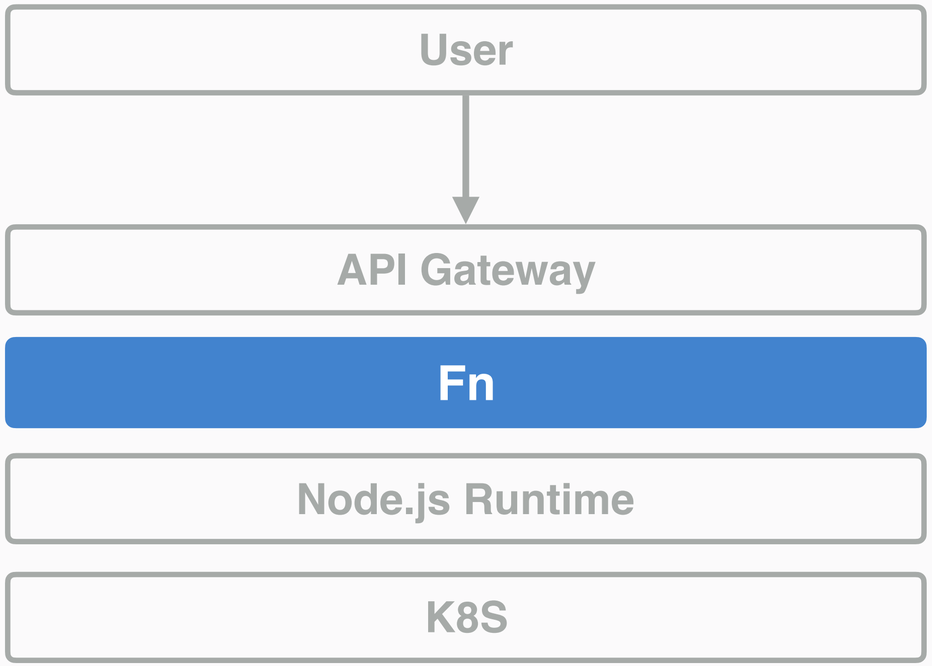
触发器（事件）

运行时



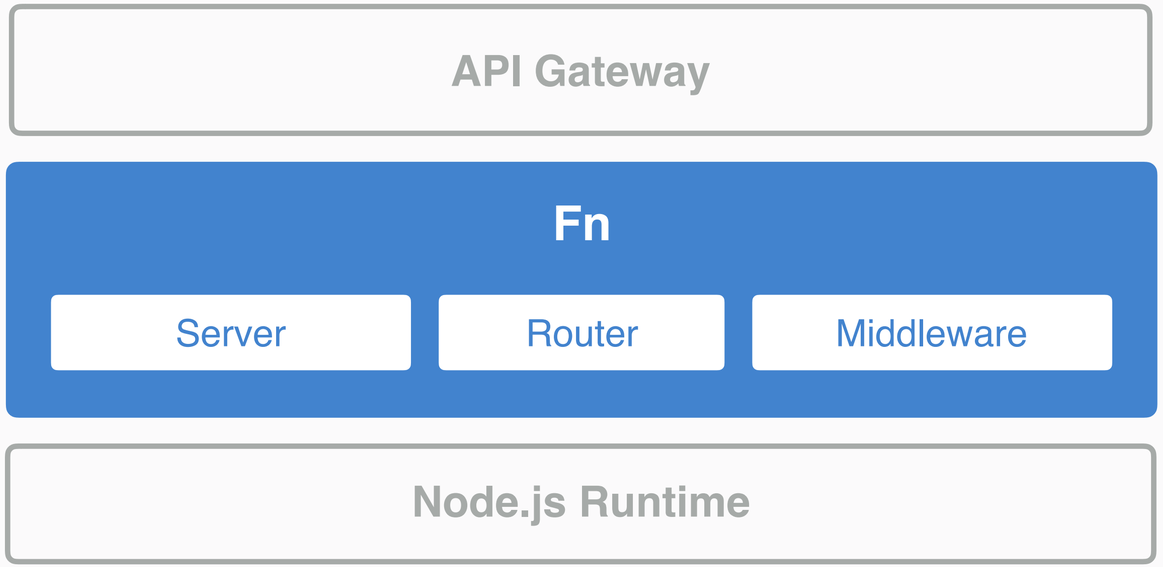
FaaS 初探

一些概念



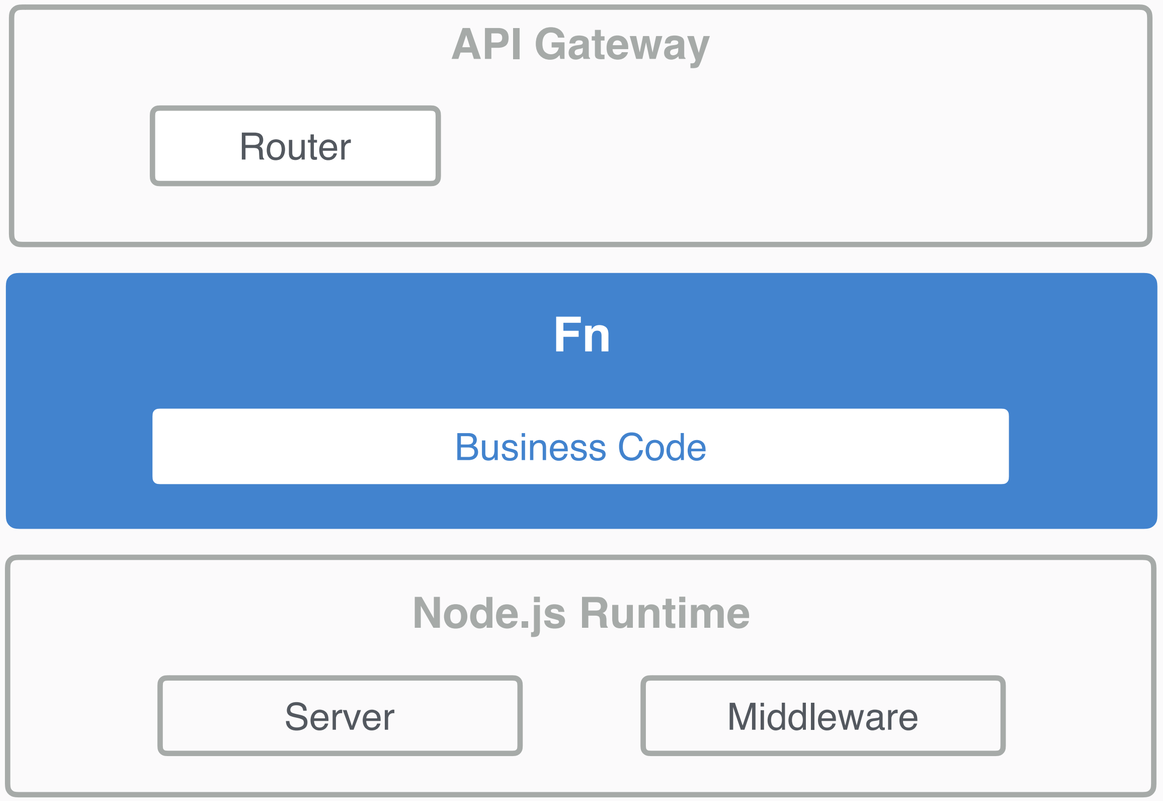
FaaS 初探

一些概念



FaaS 初探

一些概念


API Gateway
Router
Fn
Business Code
Node.js Runtime
Server
Middleware

FaaS 初探

一些概念

FaaS 框架

为什么需要框架

我们调研了业界的大部分 Serverless 的开发平台和开发的私用习惯，虽然业务不同，但是业务的代码量依旧可观。

Midway-FaaS

实现小而美

目标：

1、体验一致，方便代码模块迁移

2、跨平台发布

Midway-FaaS

实现小而美

Midway For Egg	Midway For Koa	Midway For Express	Midway For FaaS
600 line	220 line	240 line	120 line
符合全栈场景的框架，支持 egg 插件体系的完整版本。	支持 koa 作为基础框架，包含最简单单进程场景的 midway 适配版本。	支持 express 作为基础框架，包含最简单单进程场景的 midway 适配版本。	极简的 for FaaS 场景的支持 IoC 的最小框架版本。
			Selected

从 Web 到 FaaS

场景切换，代码一致

首先是入口的变化，装饰器的不同 @func，用老办法处理。

从 Web 到 FaaS

场景切换，代码一致

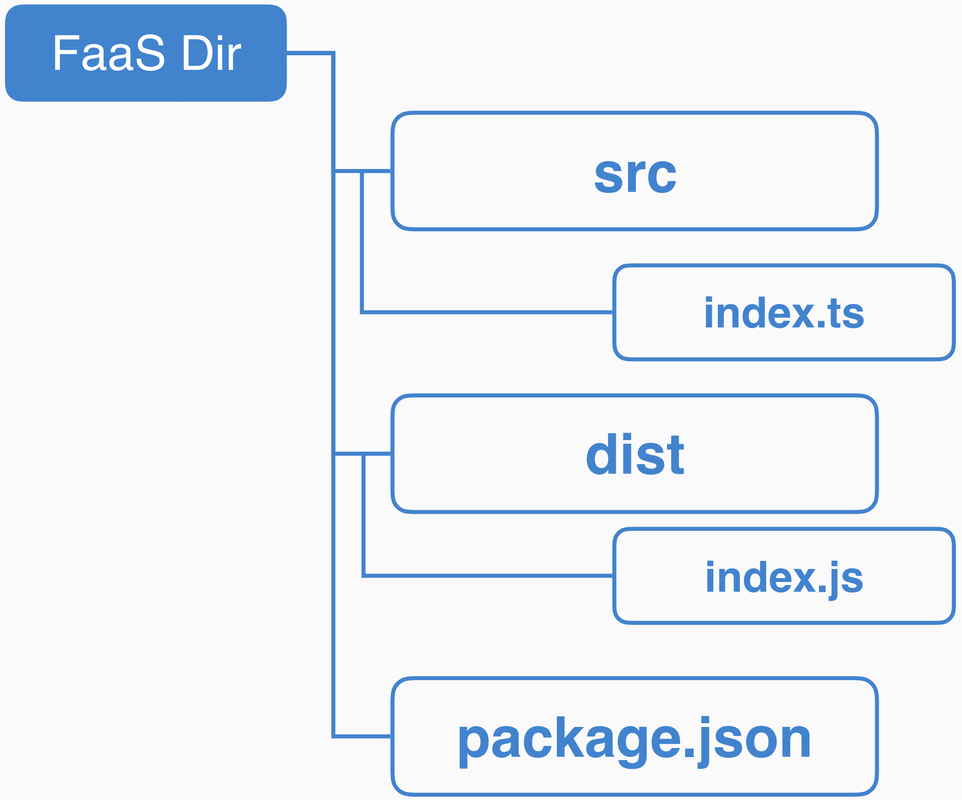
```ts
import { provide, inject, func } from 'midway-faas';

@provide()
@func('index.handler')
export class UserController {

  @inject()
  ctx;

  async handler() {

  }

}
```

跨平台的考虑

多个平台的一致性考虑

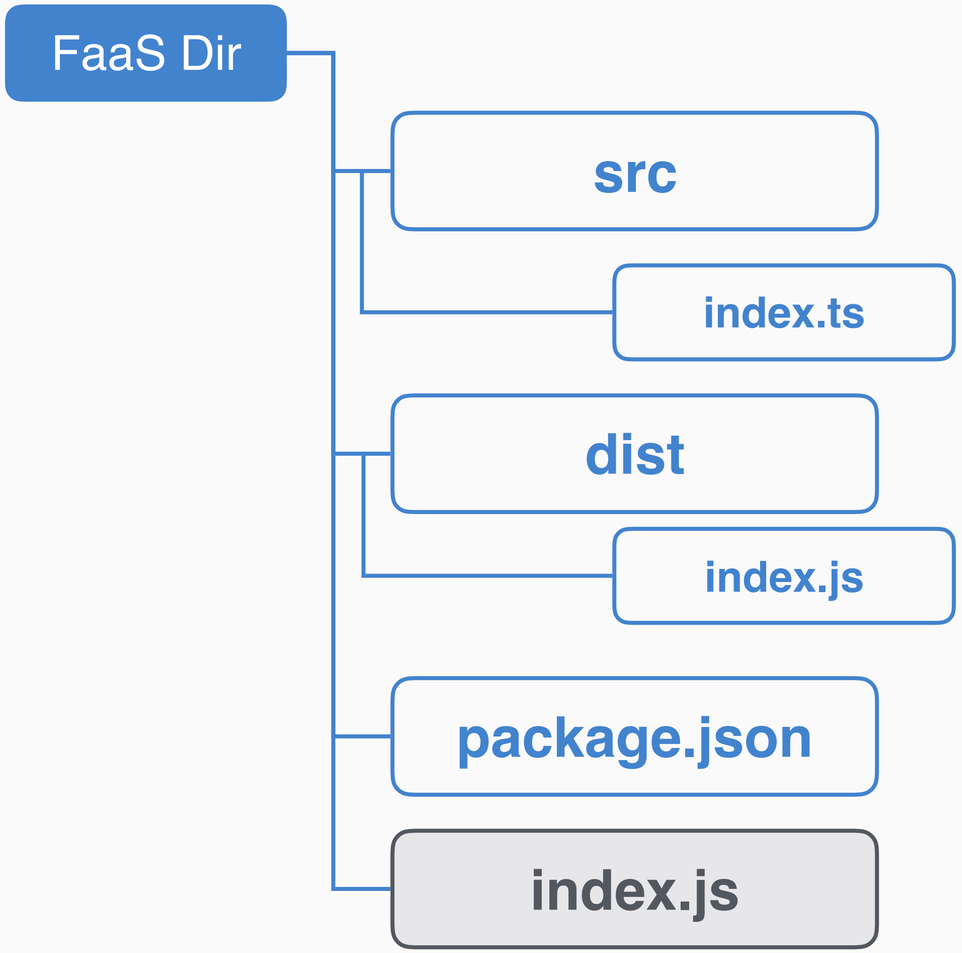
1、调用方式的不同
2、数据参数的不同
3、描述文件的不同

不同的平台需要有不同的的调用方式，由于 typescript 的编译目录不同，可以通过构建的阶段将入口文件动态创建进去。



跨平台的考虑

多个平台的一致性考虑



跨平台的考虑

多个平台的一致性考虑

总结一下

我们通过 IoC，解决了困扰我们多年的全栈开发问题。

我们通过装饰器，解决了和某个框架依赖过深的问题

我们通过多场景，拓宽了 Node.js 的开发职能，也创造了前端的新场景

THANKS

GMTC
全球大前端技术大会