

Implementar container Inversão Controle

 chat.openai.com/chat/a5ca9eac-1d79-4f75-8eff-2702736a9ec5

 Paulo_CH8

como implementar a inversão de controle em um sistema em camadas?



A implementação da inversão de controle (IoC) em um sistema em camadas pode ser feita de várias maneiras, mas uma forma comum é usando a biblioteca de Inversão de Controle (IoC container) .

1. Escolha uma biblioteca de IoC container: existem várias bibliotecas disponíveis, como Spring Framework, Guice, e PicoContainer.
2. Configure as dependências: as dependências entre os módulos do sistema devem ser definidas no arquivo de configuração do IoC container.
3. Crie os objetos: os objetos do sistema devem ser instanciados pelo IoC container, e não pelo código do módulo.
4. Injete as dependências: as dependências dos módulos devem ser injetadas pelo IoC container, e não criadas pelo código do módulo.
5. Use os objetos: os módulos do sistema devem usar os objetos criados pelo IoC container para acessar as dependências.

Com isso, os módulos do sistema são desacoplados, e a estrutura do sistema é gerenciada pelo IoC container, tornando mais fácil manter e estender o sistema.

 Paulo_CH8

tem um exemplo em typescript?



Sim, aqui está um exemplo simples de como a inversão de controle pode ser implementada em um aplicação Typescript usando a biblioteca inversão de controle `inversifyjs` :

```

import { Container } from "inversify";

// Criação das interfaces
interface Warrior {
    fight(): string;
    sneak(): string;
}
interface Weapon {
    hit(): string;
}
interface ThrowableWeapon {
    throw(): string;
}

// Criação das classes
@injectable()
class Ninja implements Warrior {
    private _katana: Weapon;
    private _shuriken: ThrowableWeapon;

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

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

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

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

// Criação do container
let container = new Container();
container.bind<Warrior>("Warrior").to(Ninja);
container.bind<Weapon>("Weapon").to(Katana);
container.bind<ThrowableWeapon>("ThrowableWeapon").to(Shuriken);

// Resolve o objeto e usa-o
let ninja = container.get<Warrior>("Warrior");
console.log(ninja.fight()); // Exibe "cut!"
console.log(ninja.sneak()); // Exibe "hit!"

```

Neste exemplo, a classe `Ninja` é uma implementação da interface `Warrior` e depende de duas outras interfaces, `Weapon` e `ThrowableWeapon`, que são fornecidas pelo container através da anotação `@inject`. O container é criado usando a classe `Container` da biblioteca `inversifyjs` e as dependências são mapeadas usando os métodos `bind` e `to`. No final, o objeto é resolvido usando o método `get` e pode ser usado para chamar seus métodos.



e como fazer sem usar framework?



Você pode implementar a inversão de controle sem usar um framework específico, usando uma abordagem de programação orientada a interfaces.

Aqui está um exemplo simples de como a inversão de controle pode ser implementada em Typescript sem usar um framework:

```

interface Warrior {
    fight(): string;
    sneak(): string;
}
interface Weapon {
    hit(): string;
}
interface ThrowableWeapon {
    throw(): string;
}

class Ninja implements Warrior {
    private _katana: Weapon;
    private _shuriken: ThrowableWeapon;

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

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

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

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

// Cria os objetos e injeta as dependencias
let katana = new Katana();
let shuriken = new Shuriken();
let ninja = new Ninja(katana, shuriken);

console.log(ninja.fight()); // Exibe "cut!"
console.log(ninja.sneak()); // Exibe "hit!"

```

Neste exemplo, a classe **Ninja** é uma implementação da interface **Warrior** e depende de duas outras interfaces, **Weapon** e **ThrowableWeapon**. As dependências são passadas para o construtor da classe **Ninja** e são armazenadas como propriedades privadas. Desta forma, a classe **Ninja** não precisa se preocupar com a criação das dependências, elas são fornecidas externamente. Isso desacopla a classe **Ninja** das dependências.

 Paulo_CH8

Mas onde está o IoC container nesse exemplo?



No exemplo que eu dei anteriormente, não há uma implementação explícita de um container de inversão de controle (IoC container), porque as dependências são passadas para o construtor da classe `Ninja` manualmente e as classes são instanciadas manualmente. Em vez disso, as dependências são gerenciadas manualmente pelo código.

Um IoC container geralmente é responsável por gerenciar as dependências do sistema, isso inclui a criação de objetos e a injeção de dependências. Com um container, as dependências são configuradas uma vez e os objetos são criados e gerenciados pelo container. Isso torna mais fácil mudar as dependências do sistema sem alterar o código dos módulos que as usam.

Se você deseja usar um container, você pode usar bibliotecas como o `inversifyjs` ou o `Awilix` que oferecem recursos de container IoC.

 Paulo_CH8

pode me dar um exemplo em `Awilix`?



Claro, aqui está um exemplo simples de como a inversão de controle pode ser implementada em um aplicação Typescript usando a biblioteca `awilix` :

```

import { createContainer, asValue, asClass } from "awilix";

interface Warrior {
    fight(): string;
    sneak(): string;
}
interface Weapon {
    hit(): string;
}
interface ThrowableWeapon {
    throw(): string;
}

class Ninja implements Warrior {
    private _katana: Weapon;
    private _shuriken: ThrowableWeapon;

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

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

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

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


const container = createContainer();

container
    .register({
        katana: asClass(Katana),
        shuriken: asClass(Shuriken),
        ninja: asClass(Ninja).singleton(),
    });

const ninja = container.resolve<Warrior>("ninja");
console.log(ninja.fight()); // Exibe "cut!"
console.log(ninja.sneak()); // Exibe "hit!"

```

No exemplo acima, a biblioteca `awilix` é usada para criar um container e gerenciar as dependências da aplicação. O método `createContainer` é usado para criar um container vazio, e os métodos `register`, `asClass`, `asValue` são usados para registrar as dependências no container. As dependências são registradas usando o nome delas, e podem ser resolvidas usando o método `resolve` passando o nome. Desta forma, as dependências são gerenciadas pelo container e podem ser injetadas nas classes que precisam delas.

 Paulo_CH8

Como poderia criar um IoC container sem framework?



Você pode criar um IoC container sem usar um framework específico, usando uma abordagem de programação orientada a objetos.

Aqui está um exemplo simples de como um IoC container pode ser implementado em Typescript:


```

interface Container {
    register<T>(name: string, factory: () => T): void;
    resolve<T>(name: string): T;
}

class SimpleContainer implements Container {
    private factories = new Map<string, () => any>();

    public register<T>(name: string, factory: () => T): void {
        this.factories.set(name, factory);
    }

    public resolve<T>(name: string): T {
        const factory = this.factories.get(name);
        if (!factory) {
            throw new Error(`No factory registered for ${name}`);
        }
        return factory();
    }
}

const container = new SimpleContainer();

container.register("katana", () => new Katana());
container.register("shuriken", () => new Shuriken());
container.register("ninja", () => new
Ninja(container.resolve("katana"), container.resolve("shuriken")));

const ninja = container.resolve<Warrior>("ninja");
console.log(ninja.fight()); // Exibe "cut!"
console.log(ninja.sneak()); // Exibe "hit!"

```

Neste exemplo, a classe `SimpleContainer` implementa a interface `Container` e gerencia as dependências da aplicação. O método `register` é usado para registrar as dependências no container, passando o nome dela e uma função `factory` que será usada para criar a dependência quando ela for resolvida. O método `resolve` é usado para obter uma dependência pelo nome e criá-la usando a `factory` registrada. Desta forma, as dependências são gerenciadas pelo container e podem ser injetadas nas classes que precisam delas.

Esse é apenas um exemplo básico, e você pode adicionar mais recursos e melhorias ao seu container como por exemplo, cache de objetos já criados, gerenciamento de vida útil, verificação de ciclos de dependência e etc.