جلوگیری از Exceptions در برنامه‌های نرم‌افزاری

علیرضا سلطانی نشان

۲۴ آبان ۱۴۰۳

۱ چکیده

در این روش که یک نوع تاکتیک برای افزایش ویژگی کیفی Availability مورد استفاده قرار می‌گیرد هدف آن است که نرم‌افزار نوشته شده عاری از هر گونه Fault باشد که توسط توسعه دهنده در هنگام توسعه مورد بررسی و آزمون نرم‌افزاری قرار نگرفته است. این کار باعث می‌شود که نرم‌افزار با کمترین مشکل به دست استفاده کننده برسد و در صورتی که Fault در نرم‌افزار رخ داد،این Fault باعث از بین رفتن چرخه عمر نرم‌افزار نشود و کاربر کمافی‌سابق بتواند از نرم‌افزار مورد نظرش استفاده کند و نیازهایش را بر طرف سازد.
ما به عنوان توسعه‌دهنده می‌توانیم با استفاده از استراتژی‌هایی از هر گونه Exception در نرم‌افزار جلوگیری کنیم. این استراتژی‌ها عبارت‌اند از:

۲ Smart Pointers

در زبان C++ برای مدیریت حافظه به شکل هوشمند و خودکار طراحی شده‌اند نقش مهمی برای جلوگیری از نشتی حافظه [۱] دارند. سه نوع رایج از Smart pointerها وجود دارد که هر کدام ویژگی منحصر به فرد خود را دارا هستند:

۱.۲ `std::unique_ptr`

این نوع از پوینترها به شکل اختصاصی مالکیت یک منبع را بر عهده می‌گیرند و هیچ وقت اجازه نمی‌دهد که به صورت همزمان دو پوینتر متفاوت به یک منبع دسترسی داشته باشند که به Deadlock برخورد کند.

```
std::unique_ptr<int> aPointer = std::make_unique<int>(10);
```

در این حالت `aPointer` تنها حافظه را در اختیار دارد و پس از اتمام استفاده از حافظه، حافظه را به صورت خودکار آزاد می‌کند.

۲.۲ `std::shared_ptr`

از این پوینتر زمانی استفاده می‌کنیم که چند پوینتر می‌خواهند به یک منبع دسترسی داشته باشند. در این پوینتر شمارنده‌ای وجود دارد که تعداد پوینترهایی که در حال استفاده از منبع هستند را نشان می‌دهد و به محض اینکه حافظه آزادسازی شود، منابع آزاد می‌شوند و شمارنده نیز به روز می‌شود.

```
std::shared_ptr ptrA = std::make_shared<int>(20); std::shared_ptr ptrB = ptrA;
```

در مثال بالا پوینتر `ptrB` به منبع‌ای اشاره دارد که `ptrA` از آن استفاده می‌کند.

---

[۱] Memory Leak

std::weak_ptr ۳.۲

نسخه ضعیف شده‌ای از پوینترهای shared_ptr هستند که مانع بروز Circular refs می‌شوند. اگرچه به منبع مورد استفاده اشاره می‌کند اما هیچ وقت مالک منبع نیست و در سمت شمارش منابع در پوینتر افزایش رخ نمی‌دهد.

```
std::shared_ptr<int> ptrA = std::make_shared<int>(30); std::weak_ptr<int> ptrC = ptrA
```

به طور کلی استفاده از پوینترها باعث مدیریت حافظه می‌شوند که در نهایت از هر نشت حافظه جلوگیری به عمل می‌آورد. از طرفی دیگر کد را ایمن‌تر و مقاوم‌تر در برابر Faultها می‌کند. بیشتر از Smart pointerها برای مدیریت پویا و امن حافظه استفاده می‌شود.

۳ Abstract data type ADT یا نوع داده انتزاعی

یک نوع ساختار داده‌است که در آن تعریف می‌شود چه عملیاتی می‌توان روی داده‌ها انجام داد. برای توضیحی ساده‌تر رفتار کلاس‌ها را می‌توانیم به صورت چکیده و به دور از جزئیات در یک ساختار داده تعریف کنیم.

مهم‌ترین مزیت استفاده از آن در جلوگیری از Exceptionها این است که به دلیل انتزاع بالا برنامه نویس نیازی به دانستن جزئیات پیاده‌سازی در برخی قسمت‌ها ندارد و آن را با تعریف در ADT می‌تواند در قسمت‌های بعدی مدیریت کند و سپس وارد جزئیات شود. مزیت دیگر آن فهمیدن بهتر کد است که پیاده‌سازی رفتار کلاس‌ها را آسان‌تر می‌کند.

```
۱        class Value {
۲            public int value;
۳            public increase() int;
۴            public decrease() int;
۵            private initialized(v: int) void;
۶        }
```

۴ Wrapperها

رپرها نوعی ساختار یا کلاس هستند که به دور یک Api یا ساختمان داده پیچیده‌تر قرار می‌گیرند و رابطی ساده‌تر و ایمن‌تر را تشکیل می‌دهند. رپرها معمولاً برای ساده‌سازی و محافظت از منطق پیچیده‌ای استفاده می‌شوند که در داخل یک Api یا کتابخانه‌ای وجود دارند.

برای مثال در Apiهایی که سطح پایین هستند و منابع سیستمی به صورت دستی مدیریت می‌شوند می‌توان با استفاده از یک رپر این منابع را به صورت خودکار در اختیار گرفته و در پایان آزاد کند و از بروز استثنا و خطاهای حافظه جلوگیری کند.