



3장-제네릭

💠 제네릭 사용법

함수의 제네릭

어떤 함수의 매개변수나 반환 값에 다양한 타입을 넣고 싶을 때 제네릭을 사용한다.

```
// target: 세 가지 타입 중 하나를 받을 수 있음
function ReadOnlyRepository<T>(target: ObjectType<T> | EntityS
Repository<T> {
    return getConnection("ro").getRepository(target);
}
```

호출 시그니처의 제네릭

호출 시그니처 : TS의 함수 타입 문법. 함수의 매개변수와 반환 타입을 미리 선언하는 것

- 이를 통해 개발자는 함수 호출 시 필요한 타입을 별도로 지정할 수 있게 된다.
- 호출 시그니처 + 제네릭 타입 → 타입의 범위, 구체 타입 한정 결정한다.

✓ 예제1

```
// T : 페이지네이션될 데이터의 타입
// 이 인터페이스를 사용할 때마다 구체적인 타입을 지정해야 한다.
interface useSelectPaginationProps<T> {
    // Recoil 상태 관리를 위한 atom들
    categoryAtom: RecoilState<number>; // 카테고리 ID 저장
    filterAtom: RecoilState<string[]>; // 필터 조건 배열 저장
    sortAtom: RecoilState<SortType>; // 정렬 타입 저장
    // 데이터 fetch 함수
    fetcherFunc: (props: CommonListRequest) => Promise<Default
}
```


▼ useSelectPaginationProps의 실제 사용 예시


```
// 상품 타입 정의
interface Product {
  id: number;
  name: string;
  price: number;
}

// useSelectPaginationProps를 Product 타입으로 구체화
const props: useSelectPaginationProps<Product> = {
  categoryAtom: productCategoryAtom,
  filterAtom: productFilterAtom,
  sortAtom: productSortAtom,
  fetcherFunc: async (request) => {
    // 반환값이 Promise<DefaultResponse<ContentListResponse>>
    return await fetchProducts(request);
  }
};

// 혹은 사용
const {
  intersectionRef,
  data, // CardListContent[] 타입
  isLoading,
  categoryId,
  isEmpty
} = useSelectPagination(props);
// useSelectPaginationProps가 사용되는 useSelectPagination 혹은
// 인자에서 쓰는 제네릭 타입인 T와 연관이 있다.
// 즉, 혹은의 반환값에도 T 타입이 사용된다.
```

프로젝트 구조에 따른 useSelectPagination 함수

```
//  매개변수
// 객체 구조분해 할당을 사용
// useSelectPaginationProps<T> 타입의 객체를 받음

//  반환 타입
```

```

function useSelectPagination<T extends CardListContent | Comm
    categoryAtom,
    filterAtom,
    sortAtom,
    fetcherFunc,
  >: useSelectPaginationProps<T>): {
  intersectionRef: RefObject<HTMLDivElement>; // 무한 스크롤을
  data: T[]; // 페이지네이션된 데이터 배열
  categoryId: number; // 현재 선택된 카테고리
  isLoading: boolean; // 로딩 상태
  isEmpty: boolean; // 데이터 존재 여부
} {
  // ...

  return {
    intersectionRef,
    data: swappedData ?? [],
    isLoading,
    categoryId,
    isEmpty,
  };
}

```

✓ 예제2

```

// <API 요청을 처리하는 커스텀 hook의 타입을 정의>

// ✓ 제네릭 타입 파라미터
// RequestData - 요청 데이터의 타입 (기본값은 void)
// ResponseData - 응답 데이터의 타입 (기본값은 void)

// ✓ 함수 파라미터
// baseUrl - API의 기본 URL 또는 헤더 (선택적)
// defaultHeader - 기본 헤더 설정 (선택적)

// ✓ 반환 타입 : 튜플
// RequestStatus - 요청의 상태를 나타내는 타입
// Reuqester<RequestData, ResponseData> - 실제 요청을 수행하는 함수

```

```
// 요청 상태를 나타내는 타입
type RequestStatus = 'idle' | 'loading' | 'success' | 'error';

// 실제 요청 함수의 타입
type Requester<RequestData, ResponseData> = (
  data: RequestData
) => Promise<ResponseData>;

// 커스텀 훅의 타입
export type UseRequesterHookType = <RequestData = void, ResponseData = void,
  baseUrl?:
    string | Headers,
    defaultHeader?: Headers
> => [RequestStatus, Requester<RequestData, ResponseData>];
```

<RequestData, ResponseData>는 괄호 () 앞에 선언했기 때문에 UseRequesterHookType 타입의 함수를 실제 호출할 때 제네릭 타입을 구체 타입으로 한정한다.

▼ UseRequesterHookType 실제 사용 예시

```
// 1. 타입 정의
// 요청할 때 보낼 데이터의 형태
interface UserRequest {
  name: string;
  age: number;
}
// 응답으로 받을 데이터의 형태
interface UserResponse {
  id: number;
  name: string;
  age: number;
}

// 2. 실제 훅 구현 - 이때 UseRequesterHookType 타입을 사용
const useRequester: UseRequesterHookType = (baseUrl, defaultHeader) => {
  // ... 구현 예시
```

```

// 상태 관리
const [status, setStatus] = useState<RequestStatus>('idle');

// 실제 요청 함수 생성 : Requester 타입에 맞는 함수 구현
const requester: Requester<RequestData, ResponseData> =
  setStatus('loading');
  try {
    // API 요청 로직
    const response = await fetch(/*...*/);
    setStatus('success');
    return response;
  } catch (error) {
    setStatus('error');
    throw error;
  }
};

// UseRequesterHookType에서 정의한 대로
// [RequestStatus, Requester<RequestData, ResponseData>]
return [status, requester];
};

// 3. 혹은 사용 - 이때는 구체적인 타입(UserRequest, UserResponse)을
function UserComponent() {
  // UserRequest: 요청할 데이터 타입
  // UserResponse: 받을 데이터 타입
  const [status, request] = useRequester<UserRequest, UserResponse>();

  const handleSubmit = async () => {
    const userData = { name: "John", age: 30 };
    const response = await request(userData);
    // response는 UserResponse 타입
  };
}

```



타입 흐름

1. UserComponent에서 `useRequester<UserRequest, UserResponse>` 를 호출하면
2. `UseRequesterHookType` 의 `RequestData`가 `UserRequest`로, `ResponseData`가 `UserResponse`로 대체됨
3. 그 결과로 반환되는 `request` 함수는 `(data: UserRequest) => Promise<UserResponse>` 타입을 가짐

→ 이렇게 제네릭을 통해 타입 정보가 흘러가면서 타입 안전성이 보장된다.

제네릭 클래스

외부에서 입력된 타입을 클래스 내부에 적용할 수 있는 클래스

- 클래스 이름 뒤에 타입 매개변수 `<T>` 를 선언해준다.



클래스 내부의 메서드를 정의할 때는 `function` 키워드를 붙이지 않는다.

```
// LocalDB 클래스는 외부에서 { key: string; value: Promise<Record>
// 타입을 받아들여 클래스 내부에 사용될 제네릭 타입으로 결정된다.
```

```
class LocalDB<T> {
  // ...

  async put(table: string, row: T): Promise<T> {
    return new Promise<T>((resolved, rejected) => {
      /*T타입의 데이터를 DB에 저장*/
    })
  }

  async get(table: string, key: any): Promise<T> {
    return new Promise<T>((resolved, rejected) => {
      /*T 타입의데이터를 DB에서 가져옴*/
    });
  }
}
```

```

    }

    async getTable(table: string): Promise<T[]> {
        return new Promise<T[]>((resolved, rejected)=> {
            /* T 타입의 데이터를 DB 에서 가져옴*/
        });
    }
}

export default class IndexedDB implements ICacheStore {
    private _DB?: LocalDB<{ key: string; value: Promise<Record<string, T>>>

    private DB() {
        if(!this._DB) {
            this._DB = new LocalDB("localCache", { ver: 6, tableName: "data" });
            // 이걸 LocalDB의 constructor에 들어갈 매개변수이다.
        }
        return this._DB;
    }
    // ...
}

```

제한된 제네릭

타입 매개변수에 대한 제약조건 설정 기능

- string 타입 제약 방법 : 타입 매개변수는 특정 타입을 상속 `extends` 해야 한다.
- `바운드 타입 매개변수` : 타입 매개변수가 특정 타입으로 묶였을 때 `bind` 의 키를 지칭하는 용어
- `키의 상한 한계` : 이때 특정 타입 `string` 을 지칭하는 용어
- 인터페이스나 클래스도 사용 가능
- 유니온 타입 상속해서 선언 가능

```

// CardListContent 또는 CommonProductResponse 타입만 가능
function useSelectPagination<T extends CardListContent | CommonProductResponse>() {
    // ...
}

```

```
}
```

```
// 사용하는 쪽 코드
```

```
const {intersectionRef, data, isLoading, isEmpty} = useSele
```

```
type ErrorRecord<Key extends string> =
```

```
  Exclude<Key, ErrorCodeType> extends never // Key 타입에서 Er
```

```
    ? Partial<Record<Key, boolean>> // 만약 Exclude의 결과가
```

```
    : never; // 그렇지 않다면 never타입 반환.
```


?

`type ErrorRecord<T=string> =` 이런 식으로 그냥 `string` 값을 넣어서 제한하면 안되는걸까?

- `<T = string>`
T의 기본값을 설정한 것뿐이다!
- `<Key extends string>`
제한된 제네릭 - T는 반드시 `string` 타입이어야 한다.

```
// 1. <T = string>
type ErrorRecord1<T = string> = {
  value: T;
  value2: number;
}
type Test1 = ErrorRecord1<number>; // 에러 X

// 2. <T extends string>
type ErrorRecord2<T extends string> = {
  value: T; // number X
  value2: number;
}
type Test2 = ErrorRecord2<number>; // ❌ 에러 발생
type Test3 = ErrorRecord2<'error1' | 'error2'>; // OK
type Test4 = ErrorRecord2<string>; // OK
```


이거는 타입을 지정하는 상황이 아니라 **새로운 타입을 정의하는** 상황이므로 `:` 가 아니라 `=` 을 쓴다.

확장된 제네릭

제네릭 타입은 여러 타입을 상속받을 수 있으며 타입 매개변수를 여러 개 둘 수도 있다.

! 제네릭의 유연성을 잃어버리게 된다.

```
<Key extends string>
```

 제네릭의 유연성을 잃지 않으면서 타입을 제약을 해야 할 때, 타입 매개변수에 유니온 타입을 상속해서 선언한다.

```
<Key extends string | number>
```

```
// <OK 타입이나 Err 타입을 매개변수 인자로 받아 사용하는 예시>
```

```
// Ok: 첫 번째 타입 매개변수, 아무 제약 없음(any)
```

```
// Err: 두 번째 타입 매개변수, 기본값이 string
```

```
export class APIResponse<Ok, Err = string> {
  private readonly data: Ok | Err | null;
  private readonly status: ResponseStatus;
  private readonly statusCode: number | null;

  constructor(
    data: Ok | Err | null,
    statusCode: number | null,
    status: ResponseStatus
  ) {
    this.data = data;
    this.status = status;
    this.statusCode = statusCode;
  }

  public static Success<T, E = string>(data: T): APIResponse {
    return new this<T, E>(data, 200, ResponseStatus.SUCCESS);
  }

  public static Error<T, E = unknown>(init: AxiosError): APIResponse {
    // ...
  }
  // ...
}

// 사용하는 쪽 코드
```

```
// 첫 번째 제네릭 타입 Ok가 IShopResponse | null로 지정되고,
// 두 번째 제네릭 타입 Err는 지정하지 않았으므로 기본값인 string이 사용됨
const fetchShopStatus = async (): Promise<APIResponse<IShopRes
    // ...

    return (await API.get<IShopResponse | null>("/v1/main/shop
        (it) => it.result
    );
}
```

제네릭 예시

🌟 가장 많이 제네릭이 활용될 때 : API 응답 값의 타입을 지정할 때

API 응답 값의 타입을 지정할 때 제네릭을 활용하여 적절한 타입 추론과 코드의 재사용성을 높이고 있다.

```
// API 응답 값에 따라 달라지는 data를 제네릭 타입 Data로 선언하고 있다.
export interface MobileApiResponse<Data> {
    data: Data;
    statusCode: string;
    statusMessage?: string;
}
```

```
// 실제 API 응답 값의 타입을 지정할 때 MobileApiResponse 사용 예시
export const fetchPriceInfo = (): Promise<MobileApiResponse<Pri
    const priceUrl = "https: ~~~"; // url 주소

    return request({
        method: "GET",
        url: priceUrl,
    })
}

export const fetchOrderInfo = (): Promise<MobileApiResponse<Ord
    const orderUrl = "https: ~~~"; // url 주소
```

```

        return request({
            method: "GET",
            url: orderUrl,
        })
    }
}

```

제네릭을 굳이 사용하지 않아도 되는 타입

코드 길이만 늘어나고 가독성을 해칠 수 있다.

```

// GType이 getRequirement 함수의 반환값 타입으로 사용되고 있다.
type GType<T> = T; // 제네릭 타입 별칭 문법의 가장 단순한 형태
type RequirementType = "USE" | "UN_USE" | "NON_SELECT";
interface Order {
    getRequirement(): GType<RequirementType>;
}

```

```

// 굳이 제네릭을 사용하지 않고,
// 타입 매개변수를 그대로 선언하는 것과 같은 기능이다.
type RequirementType = "USE" | "UN_USE" | "NON_SELECT";
interface Order {
    getRequirement(): RequirementType;
}

```

any 사용하기

any를 사용하면 제네릭을 포함해 타입을 지정하는 의미가 사라지게 된다.

```

type ReturnType<T = any> = {
    // ...
}

```

가독성을 고려하지 X은 사용

제네릭을 과하게 사용하면 가독성을 해치기 때문에 코드를 읽고 타입을 이해하기 어려워진다.

따라서 부득이한 상황을 제외하고 복잡한 제네릭은 의미 단위로 분할해서 사용하는 게 좋다!

```
ReturnType<Record<OrderType, Partial<Record<CommonOrderStatus
```

```
// 1. 주문상태와 반품상태를 하나로 묶음
```

```
type CommonStatus = CommonOrderStatus | CommonReturnStatus;
```

```
// 2. 주문 역할(구매자/판매자)에 대한 권한 정의
```

```
type PartialOrderRole = Partial<Record<OrderRoleType, string[]
```

```
// 3. 주문상태별 역할 권한 매핑
```

```
type RecordCommonOrder = Record<CommonStatus, PartialOrderRole
```

```
// 4. 주문타입별 상태-역할 권한 매핑
```

```
type RecordOrder = Record<OrderType, Partial<RecordCommonOrder
```

```
// 5. 최종 타입
```

```
ReturnType<RecordOrder>;
```



`type PartialOrderRole = Partial<Record<OrderRoleType, string[]>>;` 단계별 분석

- `Record<OrderRoleType, string[]>`

```
// OrderRoleType이 'BUYER' | 'SELLER'라고 가정하면:
{
  BUYER: string[];
  SELLER: string[];
}
```

- `Partial`

```
{
  BUYER?: string[];    // optional (선택적)
  SELLER?: string[];   // optional (선택적)
}
```

▼ 사용 예시

```
const rolePermissions: PartialOrderRole = {
  BUYER: ["read", "write"],    // OK
  SELLER: ["delete", "update"], // OK
  // 둘 다 생략해도 OK
};
```