17.递归数据类型

不可变列表

```
empty() → [ ]
cons(0, empty()) → [ 0 ]
cons(0, cons(1, cons(2, empty()))) → [ 0, 1, 2 ]

x = cons(0, cons(1, cons(2, empty())))  → [ 0, 1, 2 ]
first(x) → 0
rest(x) → [ 1, 2 ]

first(rest(x)) → 1
rest(rest(x)) → [ 2 ]
first(rest(rest(x)) → 2
rest(rest(rest(x))) → [ ]
```

在Typescript中

```
// todo: empty() returning ImList<E>
interface ImList<E> {
    cons(first: E): ImList<E>;
    readonly first: E;
    readonly rest: ImList<E>;
}
```

```
class Empty<E> implements ImList<E> {
  public constructor() {
  }
  public cons(first: E): ImList<E> {
    return new Cons<E>(first, this);
  }
  public get first(): E {
    throw new Error("unsupported operation");
  }
  public get rest(): ImList<E> {
    throw new Error("unsupported operation");
  }
}
```

Note the use of getter methods for the `first` and `rest` properties, which allow these operations to fail fast when called on an empty list.

```
class Cons<E> implements ImList<E> {
  public readonly first: E;
  public readonly rest: ImList<E>;

  public constructor(first: E, rest: ImList<E>) {
    this.first = first;
    this.rest = rest;
  }
  public cons(first: E): ImList<E> {
    return new Cons<E>(first, this);
  }
}
```

这个数据类型的函数

isEmpty : ImList → boolean

```
isEmpty(Empty) = true
isEmpty(Cons(first:E, rest:ImList)) = false
```

contains : ImList × E → boolean

```
contains(Empty, e:E) = false
contains(Cons(first:E, rest:ImList), e:E) = (first=e) or contains(rest, e)
```

get: ImList × number → E

```
get(Empty, n:number) = undefined
get(Cons(first:E, rest:ImList), n:number) = if n=0 then first else get(rest, n-1)
```

append: ImList × ImList → ImList

```
append(Empty, list2:ImList) = list2
append(Cons(first:E, rest:ImList), list2:ImList) = cons(first, append(rest, list2))
```

reverse: ImList → ImList

```
reverse(Empty) = empty()
reverse(Cons(first:E, rest:ImList)) = append(reverse(rest), cons(first, empty()))
```

静态类型和动态类型，静态分派和动态分派

这里插叙一个知识点：TypeScript的类型分为静态类型和动态类型
静态类型是编译阶段·的类型，根据string int 等关键字和相关的推断来得出到底是什么类型

动态类型是运行阶段的类型，比如new String() new Empty()等

静态分派，最重要的是函数重载，
比如：int add(int a,int b)
再写一个：int add(double a,double b)

动态分派，最重要的是子类重写父类的函数

相等性

递归数据类型的相等性的判定函数：

```
interface ImList<E> {
    //... 包括 size() 和 get(...)...
    equalValue(that: ImList<E>): boolean;
}
class Empty<E> implements ImList<E> {
    public equalValue(that: ImList<E>): boolean {
        return that.size() === 0;
    }
}
class Cons<E> implements ImList<E> {
    public equalValue(that: ImList<E>): boolean {
        if (this.size()!== that.size()) { return false; }
        for (let ii = 0; ii < this.size(); ii++) {
            if (this.get(ii)!== that.get(ii)) { return false; }
        }
        return true;
    }
}
```

这种方法相当繁琐（更难理解），但很有效，并且能将所有内容都保持在抽象屏障之上（如果所有这些操作都经过了充分测试，那么可以更好地防止错误）。

```
interface ImList<E> {
    //...
    equalValue(that: ImList<E>): boolean;
}
class Empty<E> implements ImList<E> {
    public equalValue(that: ImList<E>): boolean {
        return that instanceof Empty;
    }
}
class Cons<E> implements ImList<E> {
    public equalValue(that: ImList<E>): boolean {
        if (! that instanceof Cons) { return false; }
        return this.first === that.first && this.rest.equalValue(that.rest);
    }
}
```

这种方法带有运行时类型检查的风险（防止错误的安全性较低），但很简洁，并且具有与数据匹配的递归结构（更容易理解）。

请注意，这种实现依赖于一个事实，即给定的抽象不可变列表只能由一个确切的表示值来表示：如果存在其他表示形式，且具体变体实例的结构不同，我们就需要正确考虑这些替代情况。