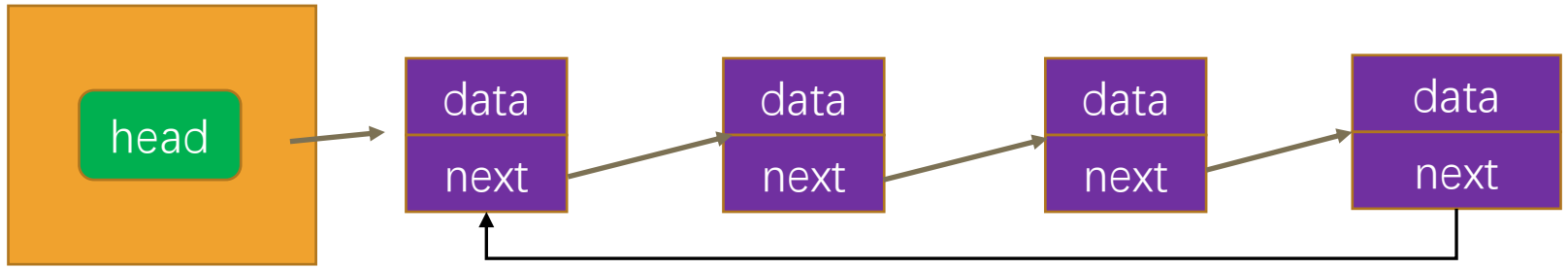
循环链表 – 双向链表

王红元 coderwhy

目录
content

认识循环链表

- **前面我们已经从零去封装了一个链表结构，其实我们还可以封装更灵活的链表结构：循环链表和双向链表。**
- **循环链表（Circular LinkedList）是一种特殊的链表数据结构：**
  - 在普通链表的基础上，最后一个节点的下一个节点不再是 null，而是指向链表的第一个节点。
  - 这样形成了一个环，使得链表能够被无限遍历。
  - 这样，我们就可以在单向循环链表中从任意一个节点出发，不断地遍历下一个节点，直到回到起点。



- **单向循环链表我们有两种实现方式：**
  - 方式一：从零去实现一个新的链表，包括其中所有的属性和方法；
  - 方式二：继承自之前封装的LinkedList，只实现差异化的部分；

重构LinkedList（方便继承）

- 1.修饰符改成protected
- 2.添加tail指向尾部节点
  - append方法：
    - ✓ this.tail.next = newNode
    - ✓ this.tail = newNode
  - insert方法：判断是否是插入最后一个节点
  - removeAt方法：
    - ✓ this.length === 1
      - ➢ this.tail = null
      - ➢ this.head = null
    - ✓ position === length – 1
      - ➢ this.tail = previous

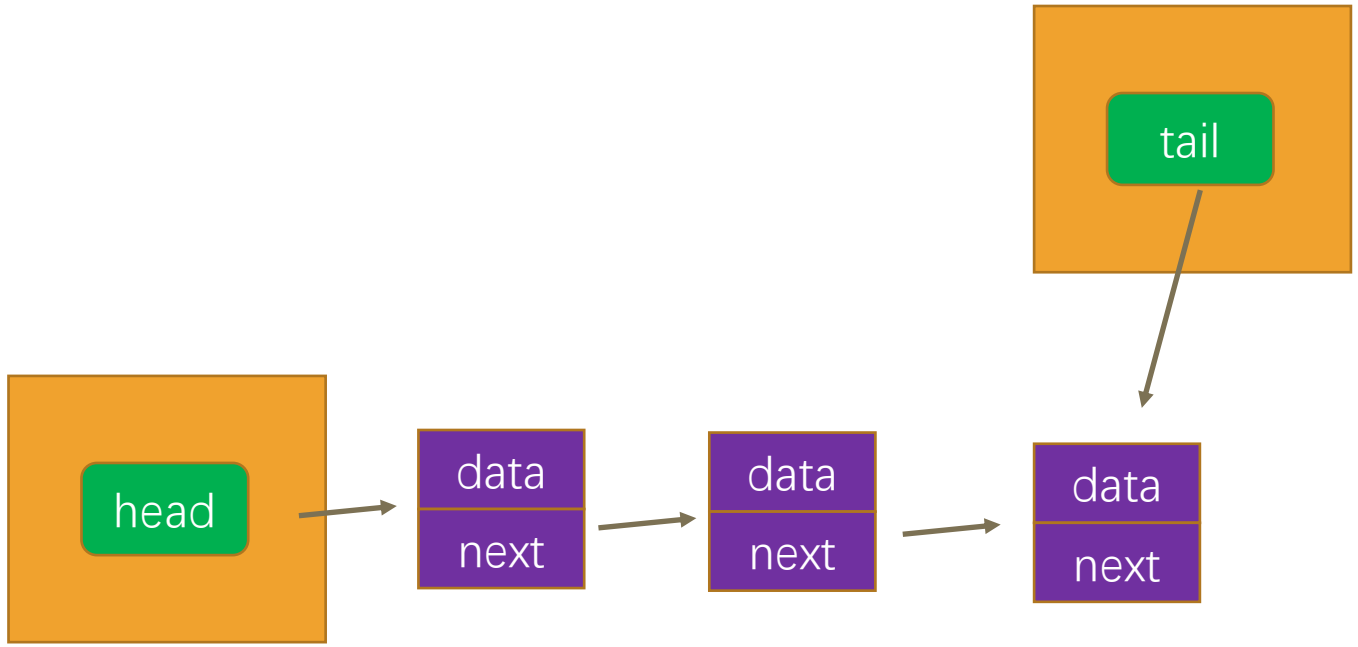
```
} else {
  this.tail!.next = newNode
}
```

```
this.tail = newNode
```

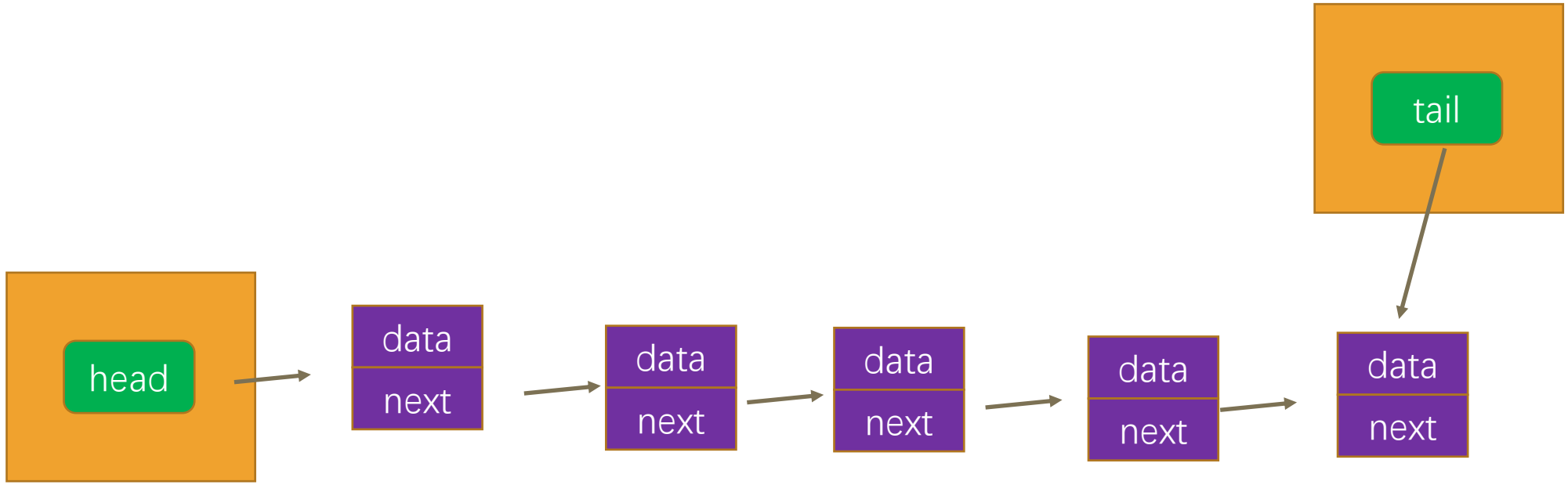
```
const previous = this.getNode(position - 1)
newNode.next = previous!.next
previous!.next = newNode

if (position === this.length) {
  this.tail = newNode
}
```

```
if (position === 0) {
  this.head = current?.next ?? null

  if (this.length === 1) {
    this.tail = null
    this.head = null
  }
} else {
  // 重构成如下代码
  const previous = this.getNode(position - 1)
  current = previous!.next
  // 找到需要的节点
  previous!.next = previous?.next?.next ?? null

  if (position === this.length - 1) {
    this.tail = previous
  }
```

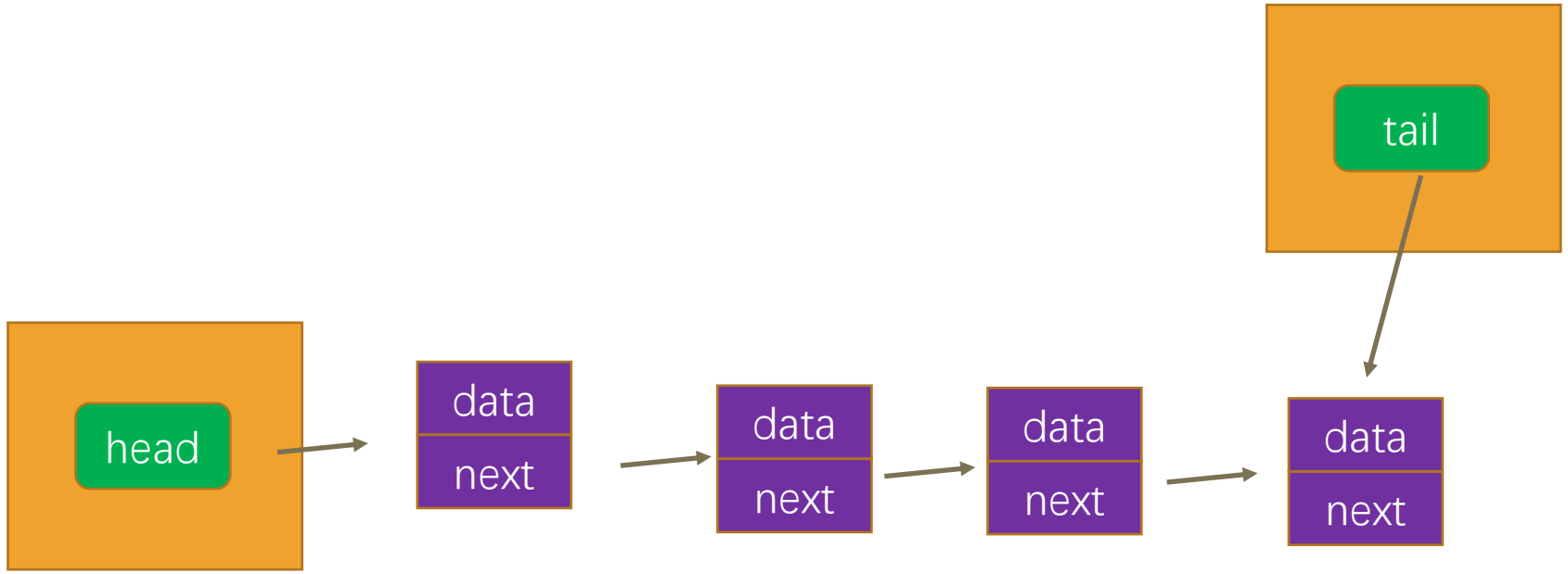
append重构



insert重构



removeAt重构

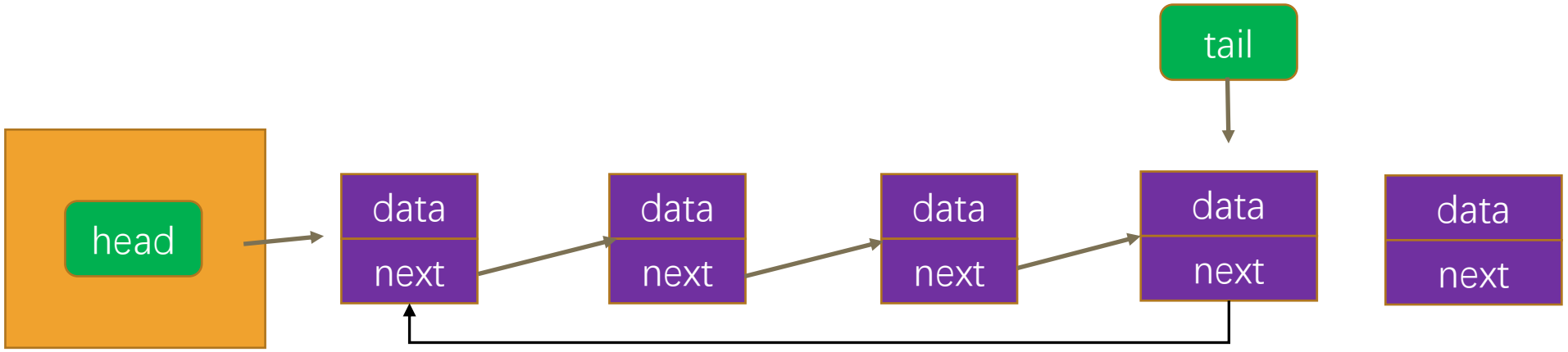


新增判断最后节点方法

- 添加判断是否是最后一个节点

```
private isLastNode(node: LinkedNode<T> | null): boolean {
  return node === this.tail
}
```

实现append方法

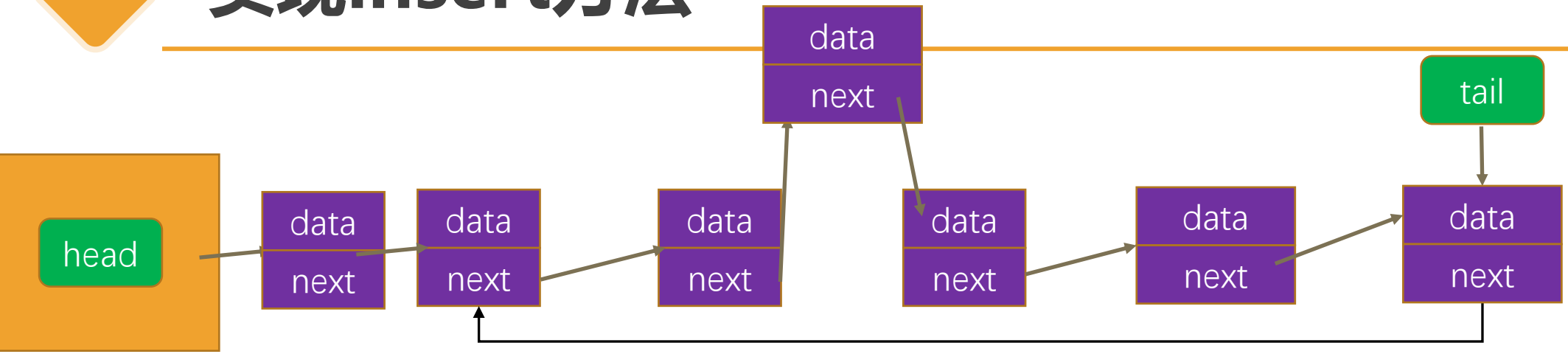



```
append(value: T): void {
  super.append(value)
  this.tail!.next = this.head
}
```

重构traverse方法

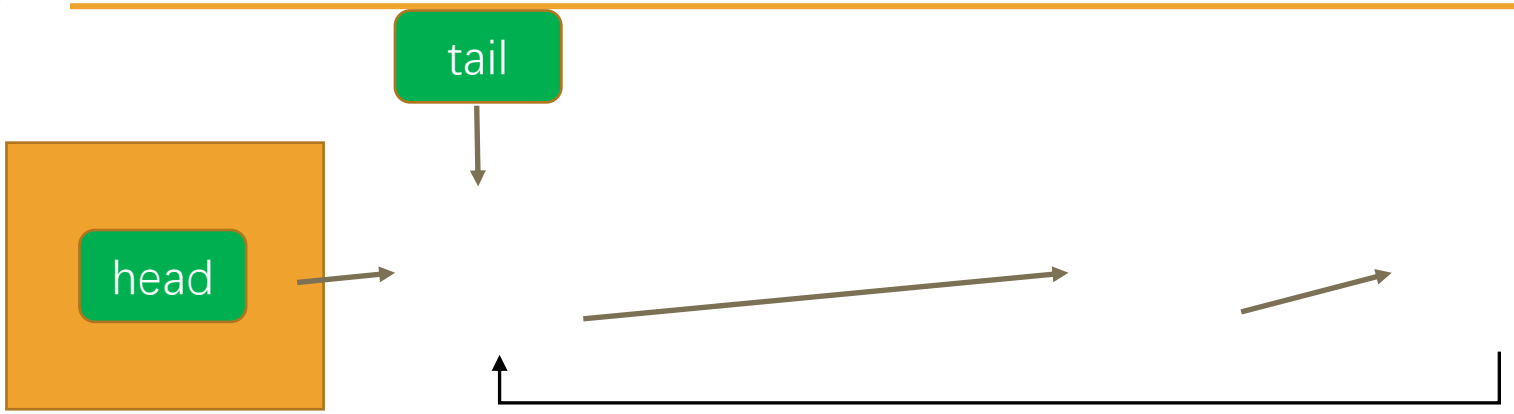
```
traverse() {
  const values: T[] = []

  let current = this.head
  while (current) {
    values.push(current.value)

    if (!this.isLastNode(current)) {
      current = current.next
    } else {
      current = null
    }
  }

  if (this.tail?.next === this.head) {
    values.push(this.head!.value)
  }

  console.log(values.join("->"))
```

实现insert方法




```
insert(value: T, position: number): boolean {
  const isSuccess = super.insert(value, position)
  if (isSuccess && (position === 0 || position === this.length - 1)) {
    this.tail!.next = this.head
  }

  return isSuccess
}
```

实现removeAt方法

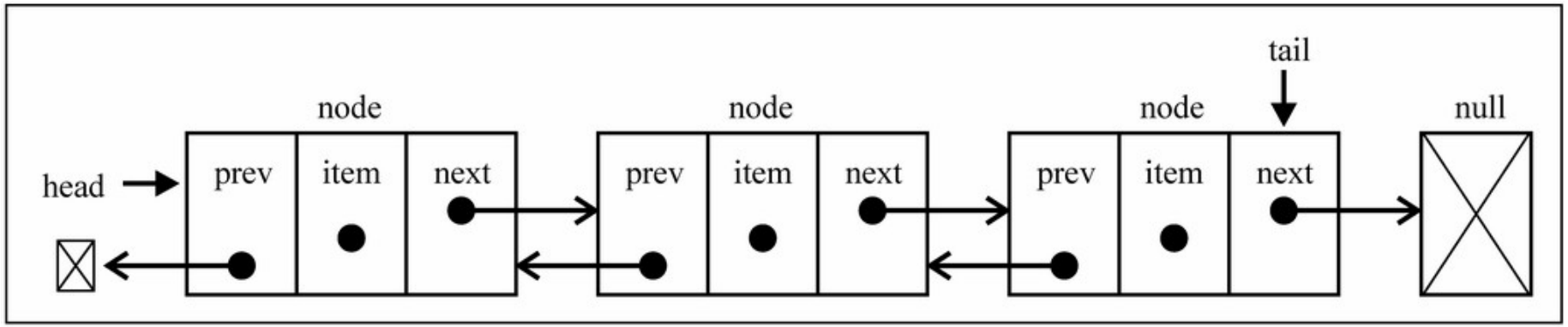


重构indexOf方法

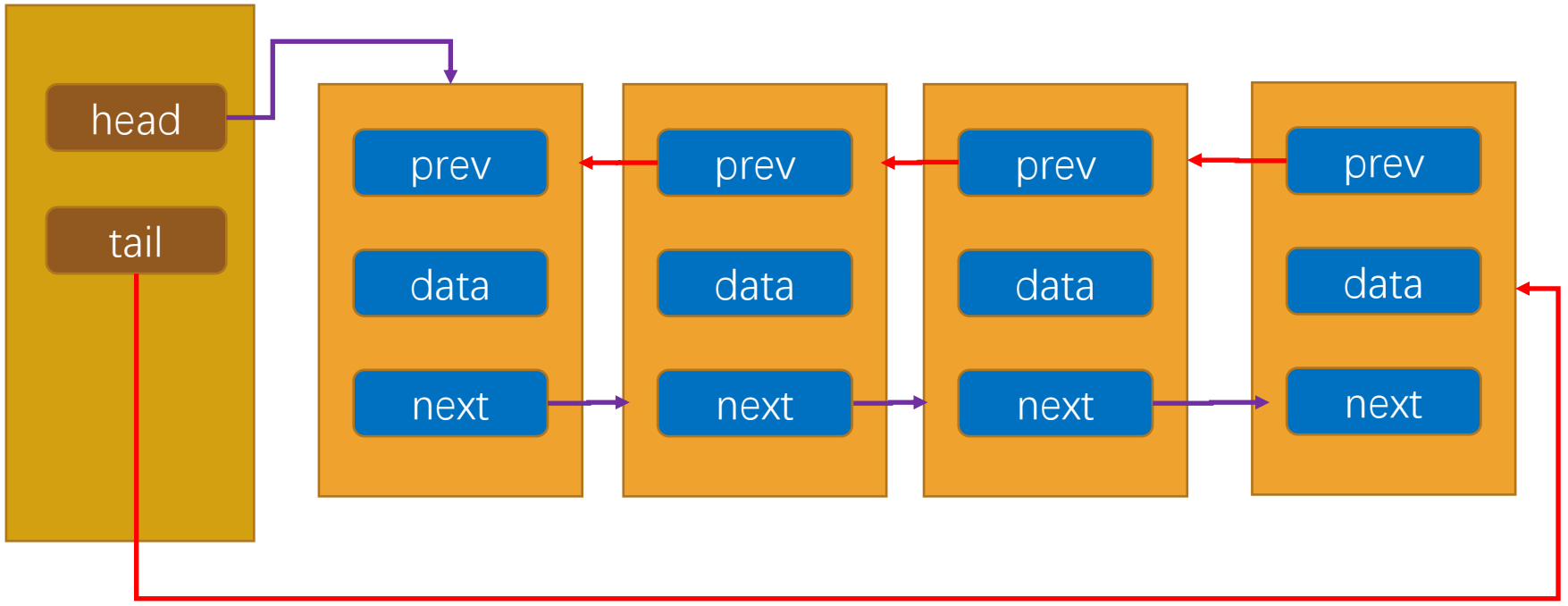
```
// 根据值，获取对应位置的索引
indexOf(value: T): number {
  // 从第一个节点开始，向后遍历
  let current = this.head
  let index = 0
  while (current) {
    if (current.value === value) {
      return index
    }

    if (!this.isLastNode(current)) {
      current = current.next
    } else {
      current = null
    }

    index++
  }

  return -1
}
```

双向链表的结构

- **双向链表:**
  - 既可以从头遍历到尾, 又可以从尾遍历到头.
  - 也就是链表相连的过程是双向的. 那么它的实现原理, 你能猜到吗?
  - 一个节点既有向前连接的引用prev, 也有一个向后连接的引用next.
- **双向链表有什么缺点呢?**
  - 每次在插入或删除某个节点时, 需要处理四个引用, 而不是两个. 也就是实现起来要困难一些
  - 并且相当于单向链表, 必然占用内存空间更大一些.
  - 但是这些缺点和我们使用起来的方便程度相比, 是微不足道的.



双向链表的画图


head
tail
prev
data
next
prev
data
next
prev
data
next
prev
data
next

双向链表的节点封装

- 双向链表的节点，需要进一步添加一个prev属性，用于指向前一个节点：

```
class LinkedNode<T> extends Node<T> {
  next: LinkedNode<T> | null = null
}

class DoublyLinkedNode<T> extends LinkedNode<T> {
  next: DoublyLinkedNode<T> | null = null
  prev: DoublyLinkedNode<T> | null = null
}
```

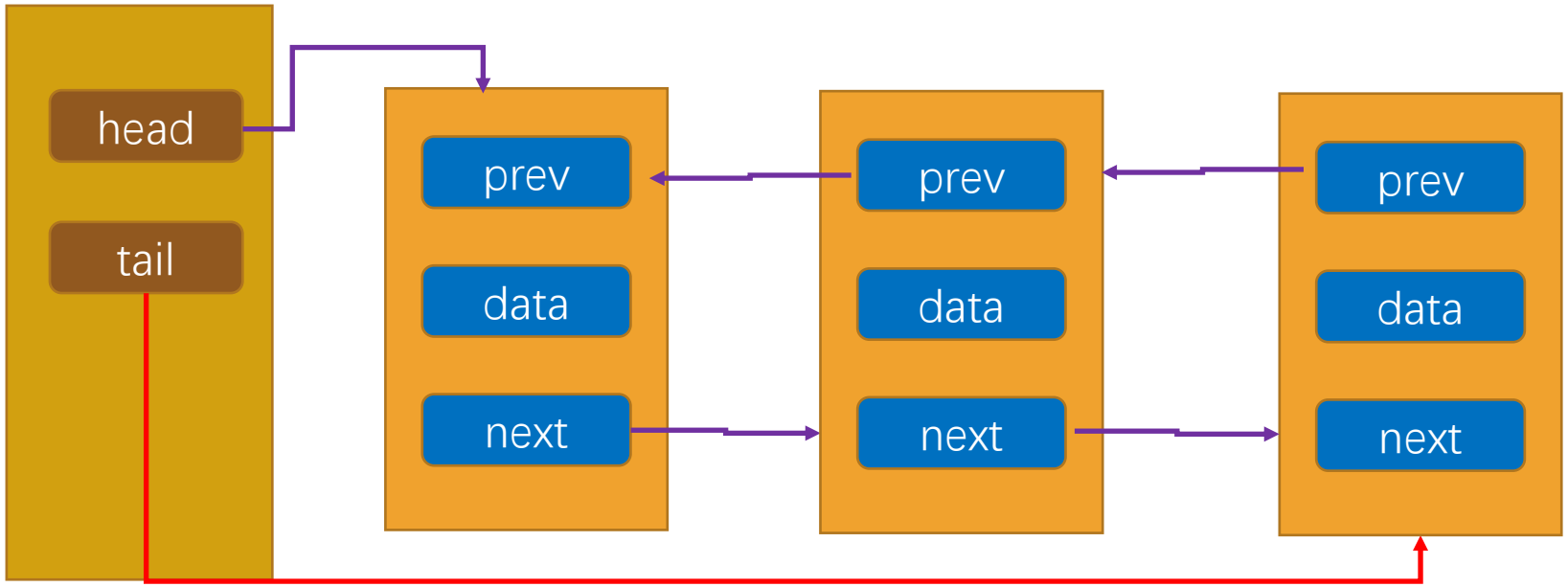
双向链表的实现

- **双向链表中添加、删除方法的实现和单向链表有较大的区别，所以我们可以对其方法进行重新实现：**
  - append方法：在尾部追加元素
  - prepend方法：在头部添加元素
  - postTraverse方法：从尾部遍历所有节点
  - insert方法：根据索引插入元素
  - removeAt方法：根据索引删除元素
- **那么接下来我们就一个个实现这些方法，其他方法都是可以继承的。**

append方法

```
/** 添加节点 */
append(value: T): void {
  const newNode = new DoublyLinkedNode(value)

  if (this.head === null) {
    this.head = newNode
    this.tail = newNode
  } else {
    this.tail!.next = newNode
    newNode.prev = this.tail
    this.tail = newNode
  }

  this.length++
}
```

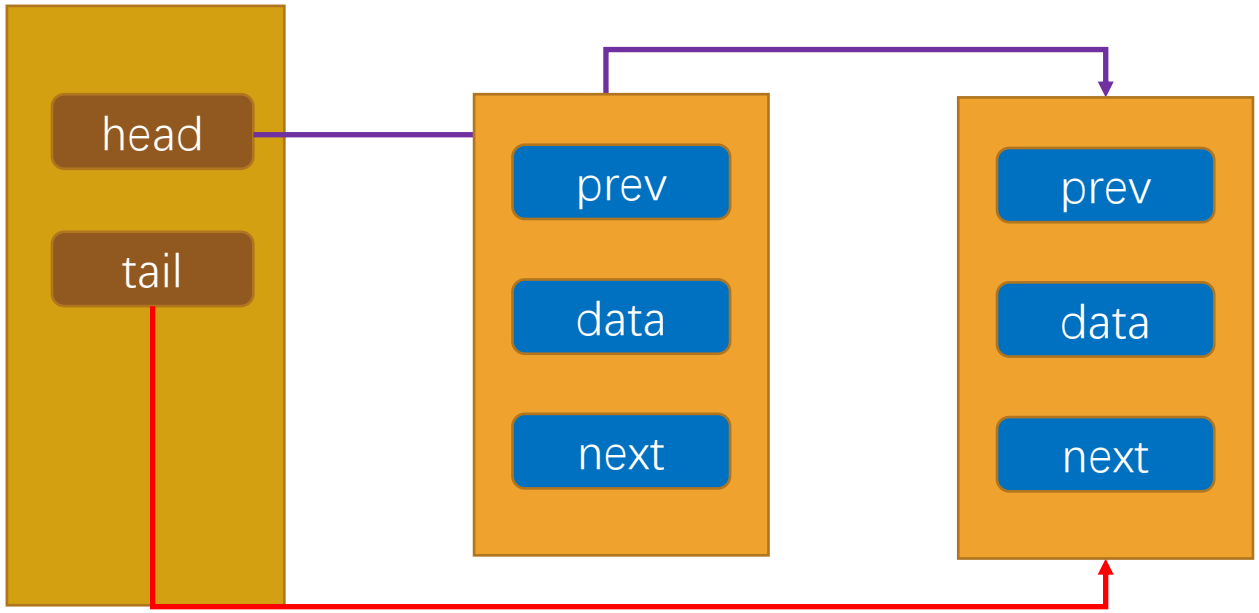
append方法



prepend方法

```
prepend(value: T): void {
  const newNode = new DoublyLinkedNode(value)

  if (!this.head) {
    this.tail = newNode
    this.head = newNode
  } else {
    newNode.next = this.head
    this.head.prev = newNode
    this.head = newNode
  }

  this.length++
}
```

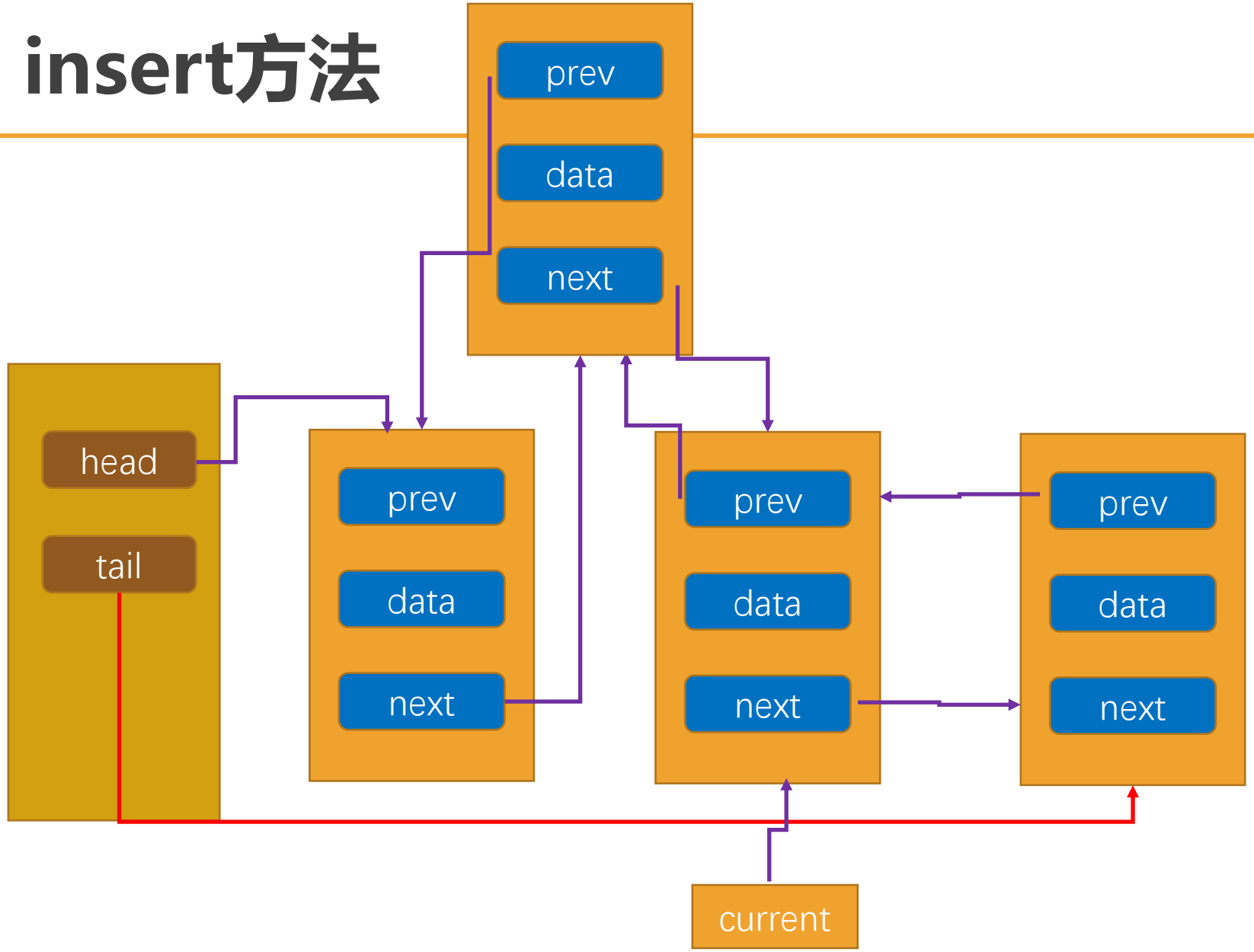
prepend方法



insert方法

```
insert(value: T, position: number): boolean {
  if (position < 0 && position > this.length) return false
  if (position === 0) {
    this.prepend(value)
  } else if (position === this.length) {
    this.append(value)
  } else {
    const newNode = new DoublyLinkedNode(value)
    const current = this.getNode(position) as DoublyLinkedNode<T>
    current.prev!.next = newNode
    newNode.prev = current.prev
    current.prev = newNode
    newNode.next = current

    this.length++ // 注意length++的位置
  }
  return true
}
```

insert方法

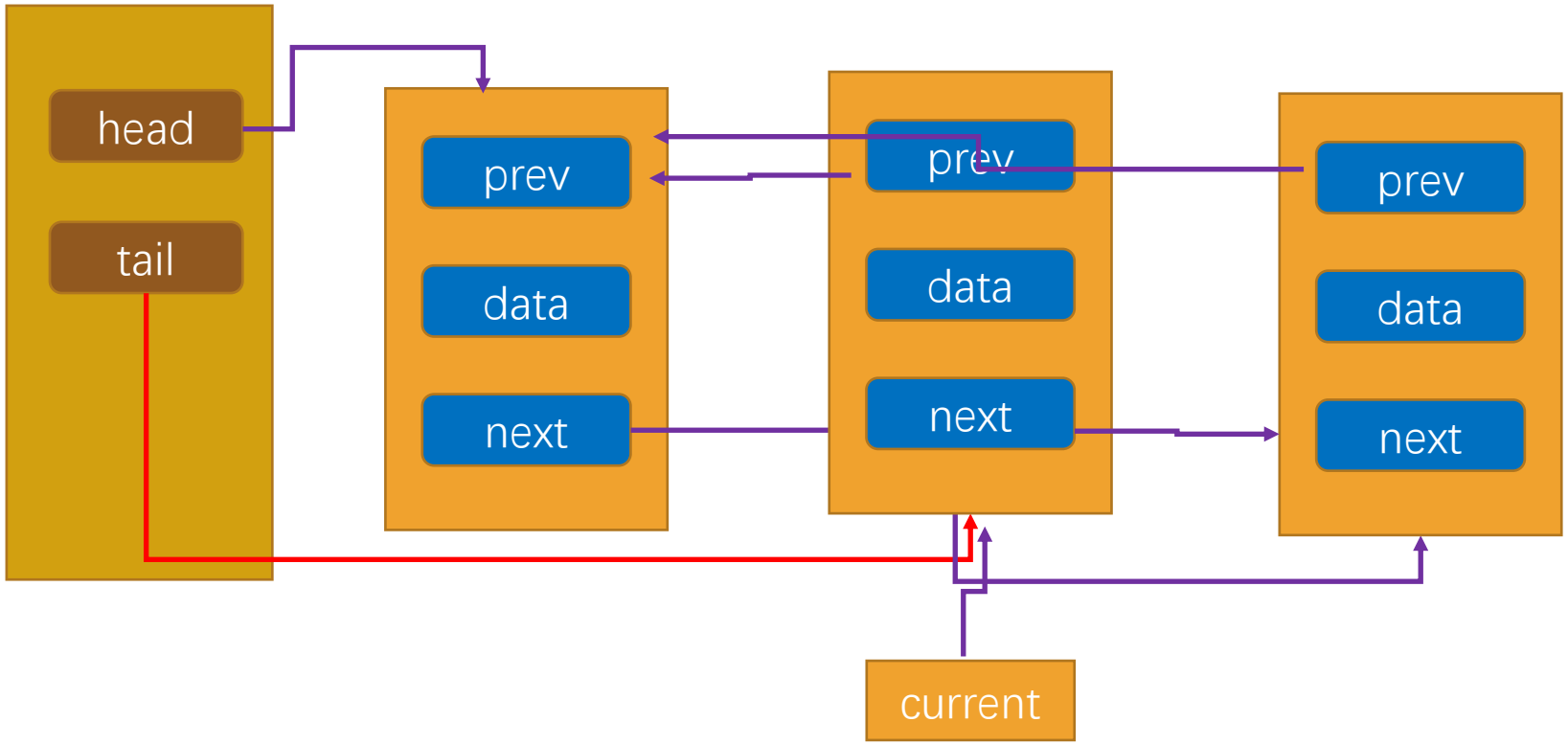


current.prev.next = newNode
newNode.next = current
newNode.prev = current.prev
current.prev = newNode

removeAt

```
removeAt(position: number): T | null {
  if (position < 0 || position >= this.length) return null

  let current = this.head
  if (position === 0) {
    if (this.length === 1) {
      this.head = null
      this.tail = null
    } else {
      this.head = this.head!.next
      this.head!.prev = null
    }
  } else if (position === this.length - 1) {
    current = this.tail
    this.tail = this.tail!.prev
    this.tail!.next = null
  } else {
    current = this.getNode(position) as DoublyLinkedNode<T>
    current.prev!.next = current.next
    current.next!.prev = current.prev
  }

  this.length--
  return current?.value ?? null
}
```

removeAt方法



current.next.prev = curren
current.prev.next = curren