单调队列

> 给你输入一个数组nums和一个正整数k，有一个大小为k的窗口在nums上从左至右滑动，请你输出每次窗口中k个元素的最大值。

结论： **在一堆数字中，已知最大值为A，则当加入一个元素B时，很容易就可以得到当前的最大值，比较A和B即可。当删除一个元素时如果删除元素为A时，就无法快速得到最大值，需要重新扫描数组。**

单调队列的实现：

```
#include <list>

// 单调队列的实现
class MonotonicQueue {
public:
  void push(int n) {
    // 删除当前队列中所有小于n的元素
    while (!que_.empty() && que_.back() <= n) {
      que_.pop_back();
    }

    que_.push_back(n);
  }

  int max() const {
    return que_.front();
  }

  void popFirst(int n) {
    if (n == que_.front()) {
      return que_.pop_front();
    }
  }

private:
  std::list<int> que_;
};
```

leetcode 239:

```
class Solution {
public:
  std::vector<int> maxSlidingWindow(std::vector<int>& nums, int k) {
    MonotonicQueue   window;
    std::vector<int> res;
```

```
    for (int i = 0; i < nums.size(); i++) {
      // 填满前k个元素
      if (i < k - 1) {
        window.push(nums[i]);
      } else {
        // 窗口开始向前移动
        window.push(nums[i]);
        //当前窗口中最大元素加入结果集
        res.push_back(window.max());
        // 移出最后的元素
        window.popFirst(nums[i - k + 1]);
      }
    }
    return res;
  }

private:
  // 单调队列的实现
  class MonotonicQueue {
  public:
    void push(int n) {
      // 删除当前队列中所有小于n的元素
      while (!que_.empty() && que_.back() < n) {
        que_.pop_back();
      }

      que_.push_back(n);
    }

    int max() const {
      return que_.front();
    }

    void popFirst(int n) {
      if (n == que_.front()) {
        return que_.pop_front();
      }
    }

  private:
    std::list<int> que_;
  };
};
```

单调队列多用于从0~nums.size()开始，判断队列中的元素和当前的元素的大小对比，如果队列中的元素小于当前元素弹出，直到队尾元素大于等于当前元素。将当前元素入队。

求数组中最大值和最小值差为num

```
#include <deque>
#include <vector>
```

```
class Solution {
public:
  int getNum(std::vector<int> &nums, int k) {
    std::deque<int> minstack;
    std::deque<int> maxstack;
    int             i   = 0;
    int             j   = 0;
    int             ans = 0;
    int             len = nums.size();

    while (i < len) {
      while (j < len) {
        // 求最大值与最小值索引
        while (minstack.size() && nums[j] <= nums[minstack.back()]) {
          minstack.pop_back();
        }
        minstack.push_back(j);  // 当前子数组中的最小值索引

        while (maxstack.size() && nums[j] >= nums[maxstack.back()]) {
          maxstack.pop_back();
        }
        maxstack.push_back(j);  // 当前子数组中最大值索引

        if (nums[maxstack.front()] - nums[minstack.front()] >
            k) {  // 最大值索引与最小值索引之差大于k，说明以满足条件
          break;
        }
        j++;
      }

      if (minstack.front() ==
          i) {  // 如果最小值索引等于i，表示子数组需要最左侧需要右移
        minstack.pop_front();
      }

      if (maxstack.front() == i) {
        maxstack.pop_front();
      }

      ans += j - i;
      i++;
    }

    return ans;
  }
};
```