LRU缓存策略

```
class LRUCache {
public:
  typedef std::list<std::pair<int, int>>::iterator node;

  LRUCache(int capacity)
      : capacity_(capacity), head_(new std::list<std::pair<int, int>>) {
  }

  int get(int key) {
    if (map_.count(key) == 0) {
      return -1;
    }
    node val = map_[key];
    int  res = val->second;
    // 删除老节点
    map_.erase(key);
    head_->erase(val);
    // 构造一个新节点,并插入头部
    head_->push_front(std::make_pair(key, res));
    map_[key] = head_->begin();
    return res;
  }

  void put(int key, int value) {
    // 判断是否已经存在这个key
    if (map_.count(key)) {
      // 已经存在
      // 删除老节点，并插入头部
      node old = map_[key];
      map_.erase(key);
      head_->erase(old);
    } else {
      // 不存在
      if (map_.size() == capacity_) {
        // 已满，移除最后一个
        int key = head_->back().first;
        head_->pop_back();
        map_.erase(key);
      }
    }
    head_->push_front(std::make_pair(key, value));
    map_[key] = head_->begin();
  }

private:
  std::list<std::pair<int, int>> *head_;  // 存储数据
  std::unordered_map<int, node>   map_;   // 存储key,  node*
  const int                       capacity_;
```

```
};
```