

并发

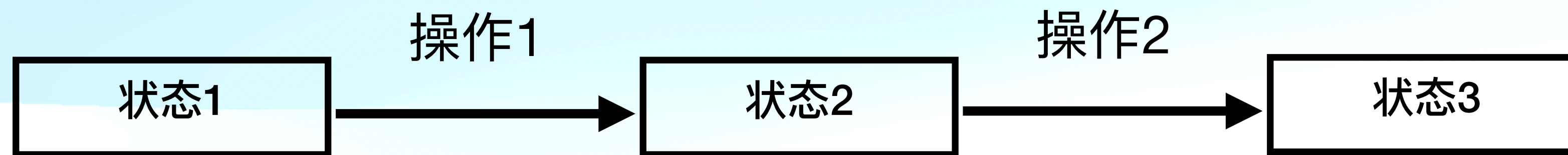
为什么多线程

- 利用cpu多核心进行并发执行——cpu利用率高于100%
- 简化项目构建过程，模块拆分
- 每个线程单独运行，避免阻塞，提高用户使用体验

操作系统帮助我们管理各个模块

状态机

- 状态+状态间转移

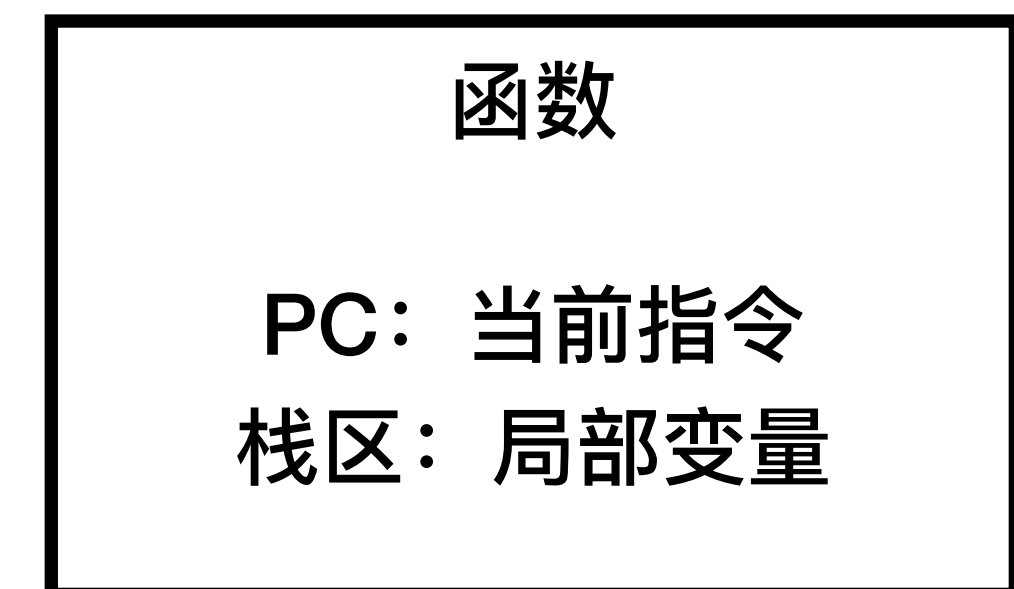
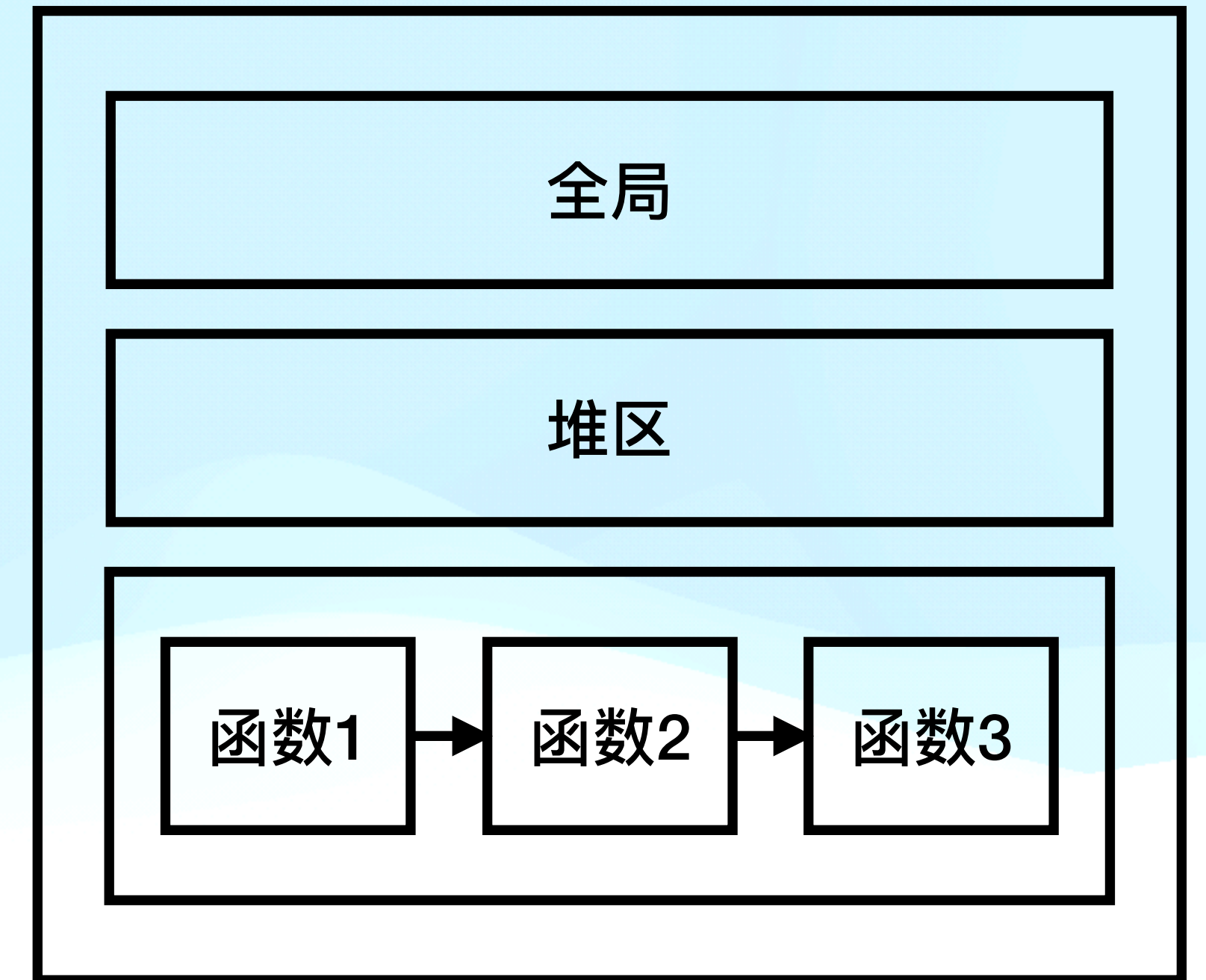


- 定义状态+转移方式

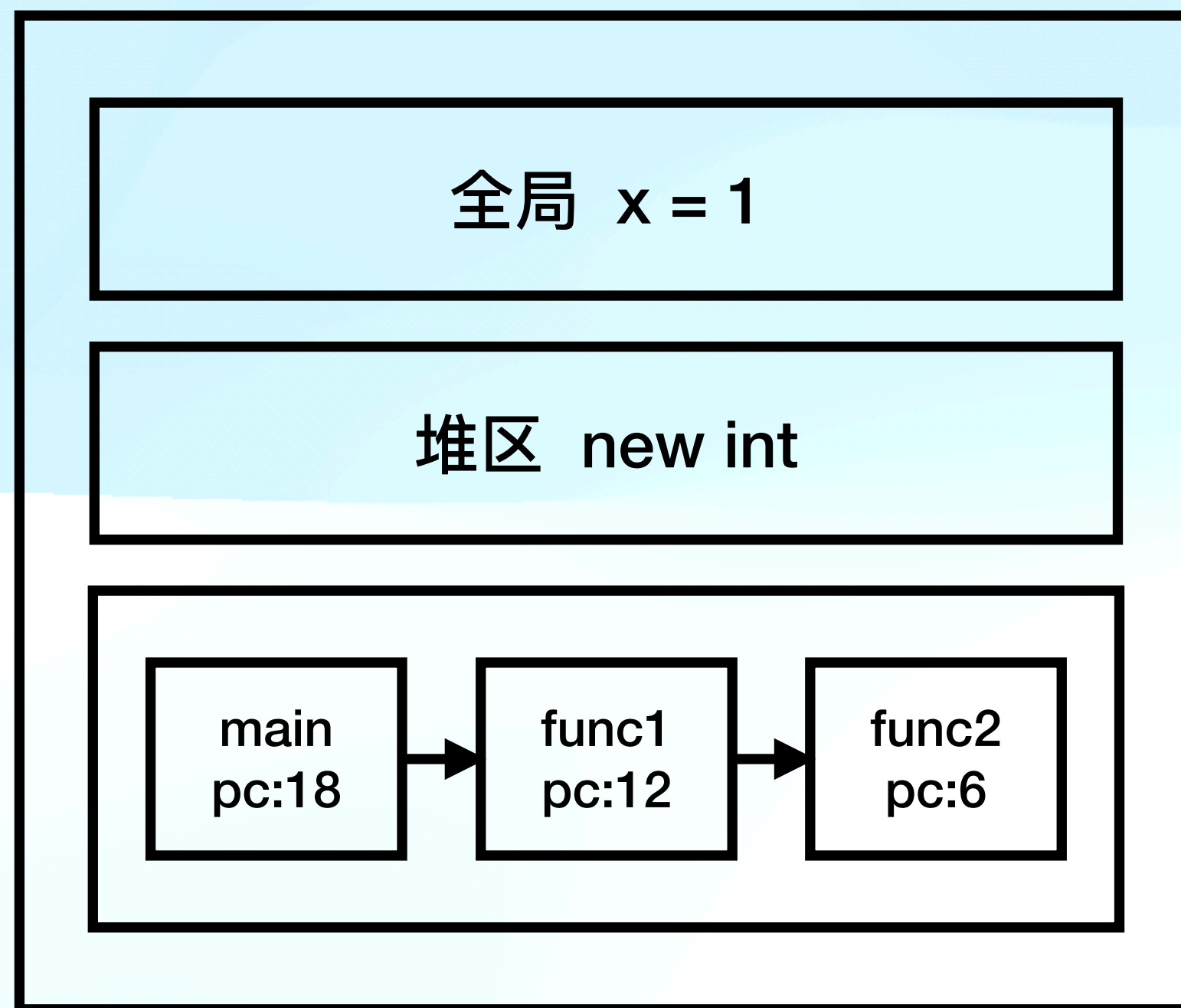
程序——状态机

C++ 程序的状态机模型

- 状态 = stack frame 的列表 (每个 frame 有 PC) + 全局变量
- 初始状态 = main(), 全局变量初始化
- step = 执行 top stack frame PC 的语句;
PC++
- 函数调用 = 创建一个stack frame
- 函数返回 = 弹出一个stack frame



程序——状态机



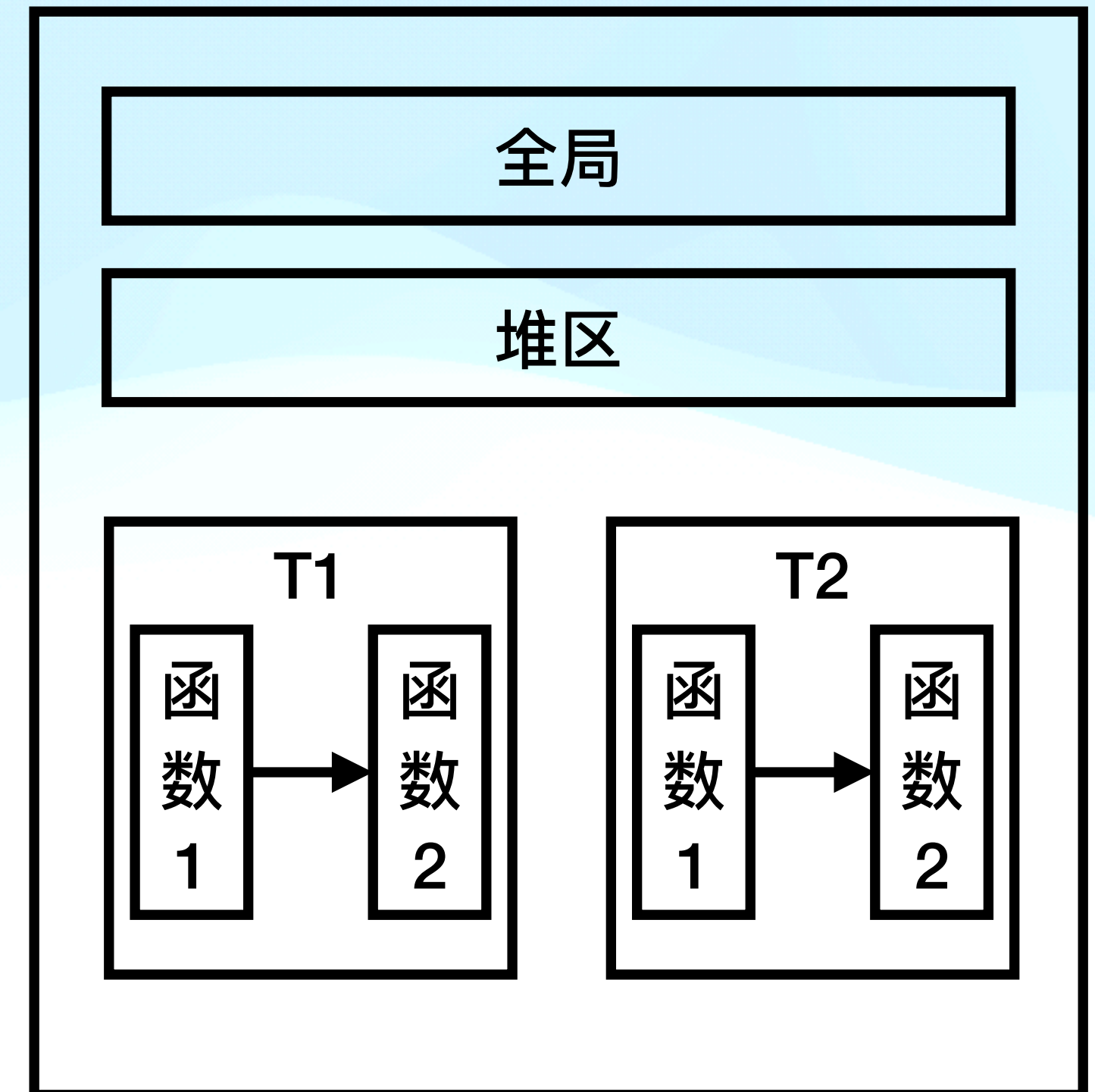
```
1  #include <iostream>
2
3  int x = 1;
4
5  void func2(int* p){
6      → (*p)+=x;
7      return;
8  }
9
10 void func1(int* p){
11     (*p)+=x;
12     → func2(p);
13     return;
14 }
15
16 int main(){
17     int* p = new int(1);
18     → func1(p);
19     std::cout << *p << std::endl;
20     delete p;
21 }
```

线程——并发的基本单位

共享内存的多个执行流

- 独立：执行流拥有独立的堆栈/寄存器
- 共享：共享全部的内存（指针可以互相引用）

```
void Ta(){  
    while(1) printf("a");  
}  
  
void Tb(){  
    while(1) printf("b");  
}
```



线程库

#include <thread>

- std::thread(fn)
 - 创建一个入口函数是fn的线程，立即执行
 - 新增一个栈帧并初始化为fn()
- t1.join()
 - 等待t1返回
 - 为什么要join?
 - RTFM

```
1  #include <thread>
2  #include <iostream>
3
4  void Ta(){
5      while(1) std::cout << "a";
6  }
7
8  void Tb(){
9      while(1) std::cout << "!";
10 }
11
12 int main(){
13     std::thread t1 = std::thread(Ta); // create a thread
14     std::thread t2 = std::thread(Tb);
15     t1.join(); // wait for t1 to exit
16     t2.join();
17     return 0;
18 }
19
20
```

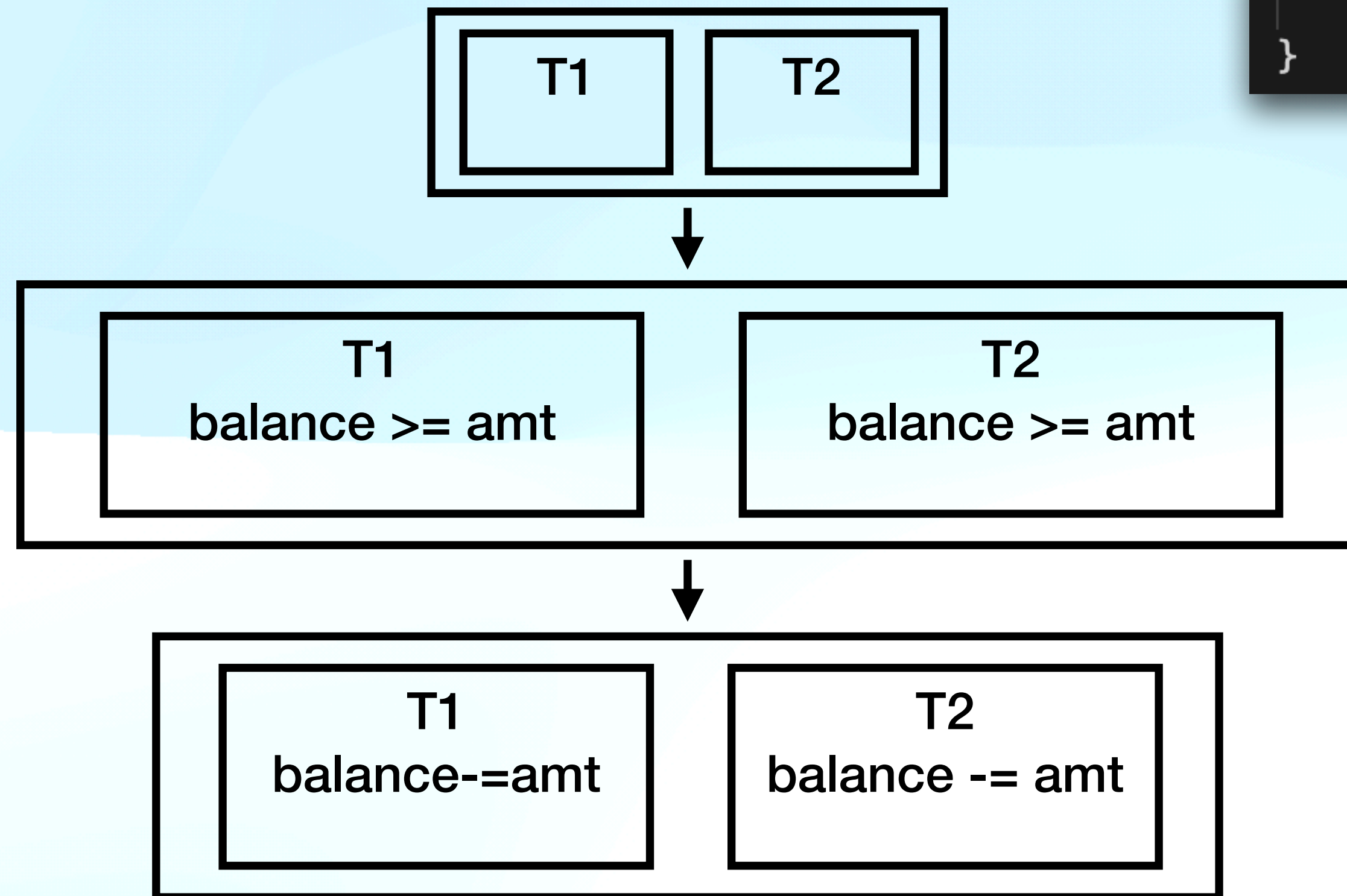
并发——原子性丧失

```
void Alipay_withdraw(int amt){  
    if (balance >= amt){  
        std::this_thread::sleep_for(3ms); // unexpected delays  
        balance -= amt;  
    }  
}
```

两个线程并发支付 ¥100 会发生什么？

用状态机的视角理解

```
void Alipay_withdraw(int amt){  
    if (balance >= amt){  
        std::this_thread::sleep_for(3ms); // unexpected delays  
        balance -= amt;  
    }  
}
```



实现原子性

当我在操作钱包的时候，不允许其他人操作

实现一下、bool?

```
#include <mutex>
```

- `std::mutex my_mutex` 创建一个mutex自旋锁
- `my_mutex.lock()` 上锁
- `my_mutex.unlock()` 解锁

实现原子性

自动解锁

```
#include <mutex>
```

- `auto guard = std::lock_guard(my_mutex)`
- `std::unique_lock<std::mutex> lk(my_mutex)`
 - 声明时自动上锁
 - 析构时自动解锁

生产者-消费者问题

```
void Tproduce() { while (true) std::cout<<"("; }  
void Tconsume() { while (true) std::cout<<""); }
```

打印序列满足：

- 一定是某个合法括号序列的前缀
- 括号嵌套深度不超过n

缺陷——打印机

- 生产者并不随时生产数据
- 消费者不断自旋




```
void Tproduce(){
    while(true){
        int input;
        std::cin >> input;
        q.push(input);
    }
}

void Tconsume(){
    while(true){
        if(q.empty()) continue;
        std::cout << q.front() << std::endl;
    }
}
```

条件变量

把自旋换成睡眠，在可以工作时被唤醒

```
#include <condition_variable>
```

- `std::condition_variable cv` 创建条件变量（全局）
- `cv.wait(lk)` 
 - 先获取互斥锁: `unique_lock<mutex> lk(mtx)`
- `cv.notify_one()`  私信：起床
- `cv.notify_all()`  所有人：起床

DETAIL: zh.cppreference.com

条件变量：实现生产者-消费者问题

- 打印机问题
- 括号匹配问题
- 多生产者多消费者

```
void Tproduce(){
    while(true){
        int input;
        std::cin >> input;
        q.push(input);
    }
}

void Tconsume(){
    while(true){
        if(q.empty()) continue;
        std::cout << q.front() << std::endl;
    }
}
```

总结

- 线程库
- 互斥锁 mutex
- 条件变量
- RTFM — — **C++ reference**
 - 并发有关部分
 - jthread (C++20)