本章涵盖

ϒ 等待事件

ϒ 等待带有 future 的一次性事件

ϒ 有时间限制的等待

ϒ 使用操作同步来简化代码

标准 C++ 库提供的不是一种而是两种条件变量的实现：std::condition\_variable 和 std::condition\_variable\_any。

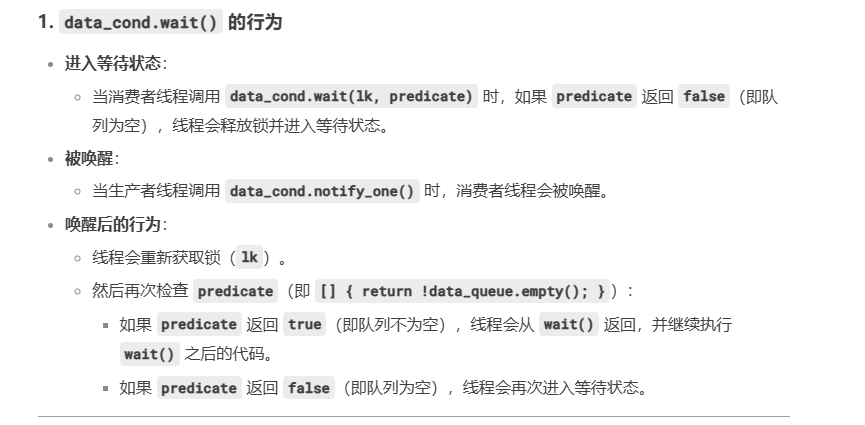


电脑萤幕的截图

描述已自动生成

表格

描述已自动生成



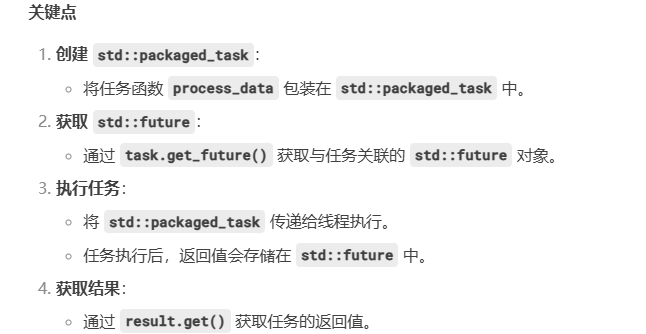
**1. 虚假唤醒的原因**

虚假唤醒通常是由以下原因引起的：

1. **操作系统或线程库的实现**：
   * 某些操作系统或线程库在实现条件变量时，可能会因为性能优化或其他原因导致线程被意外唤醒。
2. **信号或中断**：
   * 系统信号或中断可能会打断线程的等待状态，导致虚假唤醒。

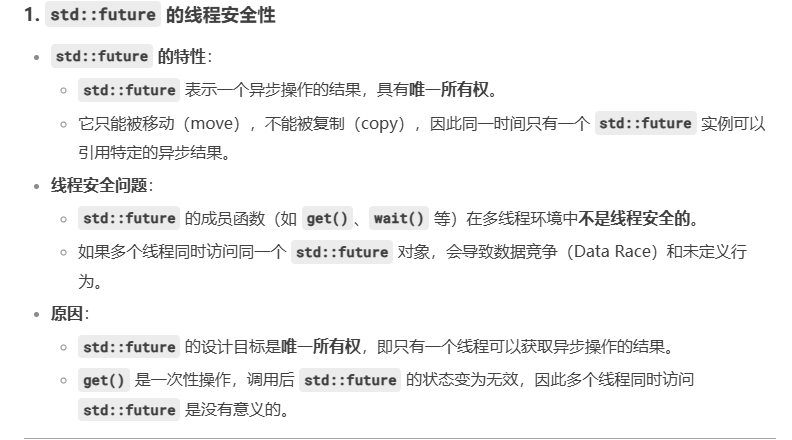
文本

描述已自动生成



手机屏幕截图

描述已自动生成



文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文本

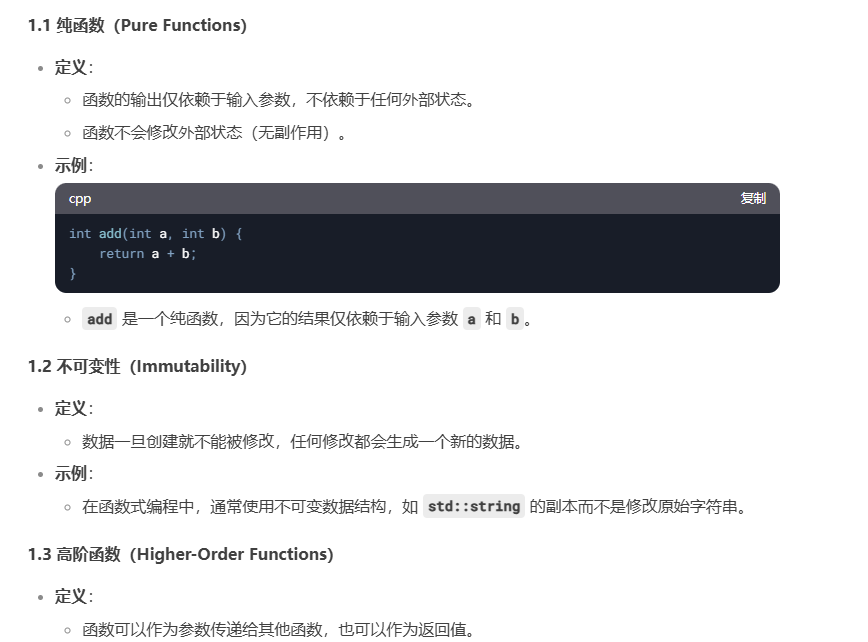
描述已自动生成

图形用户界面, 文本, 电子邮件

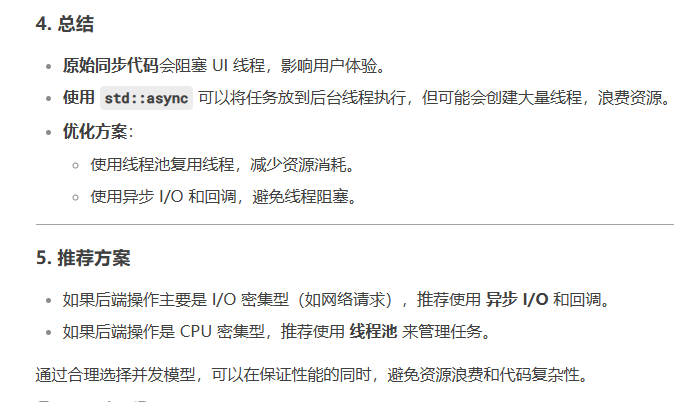
描述已自动生成



**不是在线程之间直接共享数据，而是可以为每个任务提供其所需的数据，并且可以通过使用 future 将结果传播到需要它的任何其他线程。**







文本

描述已自动生成

**方案 2：使用 std::when\_all（C++23 或第三方库）**

* **std::when\_all 可以等待一组 future 完成，并返回一个包含所有结果的 future。**
* **这是最简洁的解决方案，但需要 C++23 或第三方库（如 Boost）支持。**



文本

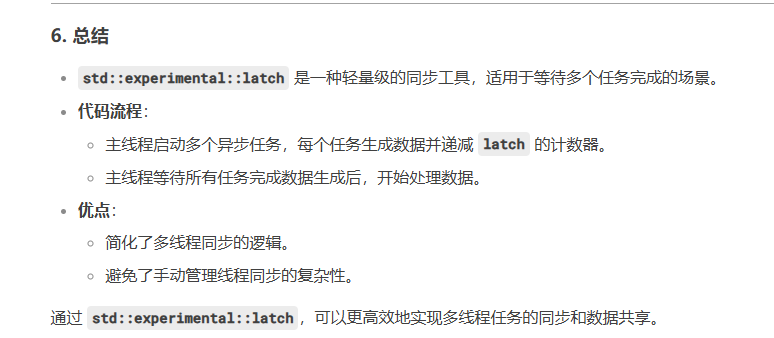
描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

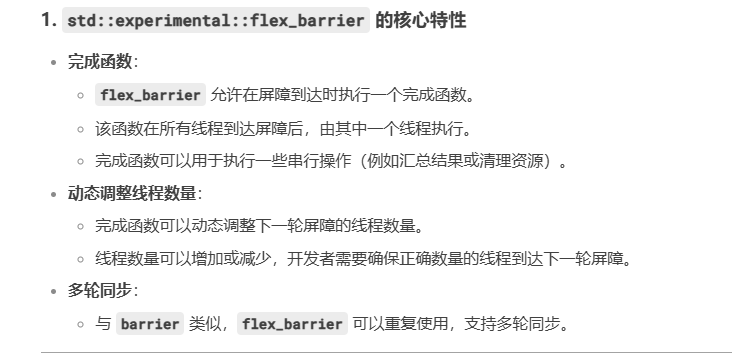
描述已自动生成

手机屏幕截图

描述已自动生成



**闩锁会闩锁，因此一旦准备就绪，它们就会保持就绪状态，而屏障则不会，屏障会释放等待的线程，然后重置，以便它们可以再次使用。**



**线程之间的同步操作是编写使用并发的应用程序的重要部分：如果没有同步，则线程本质上是独立的，并且可能会编写为单独的应用程序，因为它们的相关活动作为一个组运行。 在本章中，我介绍了从基本条件变量到 future、promise、打包任务、锁存器和屏障等同步操作的各种方法。 我还讨论了处理同步问题的方法：函数式编程，其中每个任务产生的结果完全依赖于其输入而不是外部环境； 消息传递，线程之间的通信是通过充当中介的消息子系统发送的异步消息进行的； 延续风格，其中指定每个操作的后续任务，并且系统负责调度。**

**讨论了 C++ 中可用的许多高级设施后，现在是时候看看使这一切正常工作的低级设施了：C++ 内存模型和原子操作。**