线程的创建与执行

POSIX 线程

创建线程并执行线程入口函数

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
```

- void *(*start)(void *)
  - 线程入口函数形参为 void *，并且返回值为 void *
  - 例：void *my_thread_entrance(void *) {/*do something*/}
- void *arg
  - 通常情况下，arg 指向全局或者是堆变量，使用栈指针将会有较大风险
  - 在谨慎的强制类型转换下，我们也可以将 int 类型作为实参传入，但是这与编译器的实现有关，C 标准并未定义该行为

连接已经终止的线程 (join)

```
int pthread_join(pthread_t thread, void **retval);
```

retval 即为 thread 线程所返回的结果

通俗的理解就是当前线程等待 thread 线程的返回，并将该线程的返回结果保存至 retval 中

当一个线程未被设置为分离，则必须使用 pthread_join 进行连接。否则当线程运行结束时将产生僵尸线程，无法在系统线程记录表中清除，除了占用系统资源外，僵尸线程积累过多将导致无法创建新线程

线程的分离 (detach)

有时候我们并不关心线程的执行结果，只想要系统能够创建这个线程运行其任务，并且在运行结束后自动地进行一些清理工作

此时我们就可以将该线程标记为"分离"状态，变为后台任务，清理工作将交给 OS，程序不需要再去连接该线程了

```
int pthread_detch(pthread_t thread);
```

注意，一旦线程出于分离状态，那么我们就再也不可以使用 pthread_join 再进行连接了

C++ 线程

创建线程并执行线程入口函数

普通函数作为入口函数

```
std::thread t {entrance, args ...};
```

C++ 创建线程并执行非常的简单，实例化一个 thread 对象即可

若入口函数形参为引用类型，那么必须使用 std::ref 对其进行包装，否则函数调用将按值传递，对 args 参数进行拷贝

```
class Buz {
public:
    Buz() {};
    Buz(const Buz& buz) {puts("拷贝构造");}
};

void entrance(Buz& buz) {puts("func");}    // 函数形参为引用类型

int main() {
    Buz buz;
    std::thread t(entrance, std::ref(buz));    // 此时必须使用 std::ref 进行包装
    t.join();
    return 0;
}
```

成员函数作为入口函数

```
std::thread t {&class::funcname, &instance, args ...}
```

```
class Buz {
public:
    Buz() {};
    Buz(const Buz& buz) {puts("拷贝构造");}
    void entrance(int value) {puts("entrance");}
};

int main() {
    Buz buz;
    std::thread t(&Buz::entrance, std::ref(buz), 10);    // 还需把对象作为参数传入
    t.join();
    return 0;
}
```

线程的连接

和 POSIX 线程连接功能一样: 当前线程等待某一个线程运行结束，只不过通过 std::thread 创建出来的线程无法获取线程运行结果

线程的分离

线程分离和 POSIX 线程分离功能一样: 某个线程与当前线程分离，不可再进行连接

当线程分离时，线程入口函数不可使用引用、指针类型的形参，否则会出现所引用的对象、指针所指向的对象已被销毁的情况。当然，join() 其实也会出现这种情况，但是它是可控的，而一旦调用 detch()，行为将不受主线程控制

另外一点需要注意的是，如果线程被设置为分离的，那么当主线程运行结束后，守护线程也会被终止