baidu

BGCC Manual

bgcc 使用手册

liuxupeng
2012/10/26

目录

BGCC 使用手册

1 介绍

1.1 定位

BGCC 是百度具有完全知识产权的跨平台、跨语言、面向对象与服务的轻量级高性能 RPC 框架。它支持事务级别的服务端消息主动推送，强大的接口级事务管理功能，包含自定义的通信协议、接口描述语言(bidl)、强大的代码生成引擎(bidl2sl)，Java、C++通信无缝整合。

1.2 内容

本手册涉及的内容包括 BGCC 的使用示例、安装方法及实用功能介绍。

2 术语

- BIDL: Baidu Interface Definition Language。百度接口定义语言。
- BGCC: Baidu General Communication Component。 百度通用通信组件。
- bidl2sl: BIDL to Special Language。BIDL 语言解析器，将接口定义翻译成特定语言的源代码。

3 使用示例

前提：已经获取 BGCC 头文件、lib 库和 bidl2sl 工具。获取方法参见 3.7

本示例演示如何通过 BGCC 快速完成网络服务程序的开发。编写 BGCC 应用程序涉及如下步骤：

(1) 编写 bidl 定义文件；
(2) 使用 bidl2sl 工具完成代码转换；
(3) 编写 server；
(4) 编写 client；
(5) 编译；
(6) 运行。

3.3~3.6 小节将以 C++语言为例，3.7 小节以 Java 语言为例，分别在 Linux 平台下展示功能简单但结构完备的 client 和 server。server 提供加法服务。client 通过 SOCKET 访问 server 提供的服务。对于 Java 使用者，在源码 sample/java 目录下可以找到示例工程。

3.1 编写 bidl

math.bidl 文件如下所示:

```
namespace math { //#1
    enum Result { //#2
        E_SUCCESS,
        E_OVERFLOW,
        E_DOWNFLOW
    }

    class Math { //#3
        Result add(int32 a, int32 b, [out] int32 sum); //#4
    }
}
```

对 math.bidl 的说明：

(1) #1 指定了名称空间 math。bidl2sl 工具在对 math.bidl 进行转换时，将生成的 C++源码置于此名称空间中。

(2) #2 定义了枚举类型 Result。枚举类型的使用与 C 语言中 enum 的使用类似。

(3) #3 定义了服务 Math。服务通过 class 关键字指定。服务包含若干个方法。本例中，Math 服务仅包含一个方法 add。add 方法的功能是计算两个 32 位有符号整数之和，并将和存放于传出参数 sum 中。add 方法的返回类型为 Result，表示是否有溢出。

3.2 编译 bidl

下面使用 bidl2sl 工具，将 math.bidl 文件转换为 C++源码。

```
bidl2sl  -g cpp math.bidl
```

执行命令后，在当前目录生成 output-cpp 目录。目录结构如下：

```
output-cpp/
|-- math.cpp
`-- math.h
```

3.3 编写 server

sever.cpp 如下所示：

```
#include <bgcc.h>
#include "math.h"

using namespace bgcc;
using namespace math;

class MathImpl : public Math {
public:
    virtual ~MathImpl() { }

    virtual Result add(
            int32_t a,
            int32_t b,
            int32_t & sum,
            const std::map<std::string, std::string>& ctx) {
        Result result = Result::E_SUCCESS;
        int32_t tmp = a + b;

        if (a > 0 && b > 0 && tmp <= 0) {
            result = Result::E_OVERFLOW;
        } else if (a < 0 && b < 0 && tmp >= 0) {
            result = Result::E_DOWNFLOW;
        } else {
            sum = tmp;
        }
        return result;
    }
};

int main(int argc, char* argv[]) {
    SharedPointer<IProcessor> processor(new MathProcessor(
                SharedPointer<Math>(new MathImpl)));

    ServiceManager sm;
    sm.add_service(processor);

    ThreadPool tp;
    tp.init(10);

    Server server(&sm, &tp, 8504);
    if (0 != server.serve()) {
        return 0;
    }
    return 0;
```

```
}
```

下面逐行进行分析。

```
#include <bgcc.h>
using namespace bgcc;
```

包含 BGCC 头文件，并引用名称空间 bgcc。像类 SharedPointer, IProcessor, ThreadPool, Server, ServiceManager 都是在 bgcc 名称空间中定义。

```
#include "math.h"
using namespace math;
```

包含 bidl2sl 工具根据 math.bidl 生成的源码头文件，并引用名称空间 math。该头文件中定义了服务接口类 math::Math。

```
class MathImpl : public Math {
public:
    virtual ~MathImpl() { }

    virtual Result add(
            int32_t a,
            int32_t b,
            int32_t & sum,
            const std::map<std::string, std::string>& ctx) {
        Result result = Result::E_SUCCESS;
        int32_t tmp = a + b;

        if (a > 0 && b > 0 && tmp <= 0) {
            result = Result::E_OVERFLOW;
        } else if (a < 0 && b < 0 && tmp >= 0) {
            result = Result::E_DOWNFLOW;
        } else {
            sum = tmp;
        }
        return result;
    }
};
```

以上行定义类 MathImpl，继承 math::Math 接口并实现 add 方法。add 方法的返回值及前三个参数与 math.bidl 文件中意义相同。第 4 个参数提供了若干上下文环境变量，这里不需要使用，故不做详细说明。

```
    SharedPointer<IProcessor> processor(new MathProcessor(
```

```
                SharedPointer<Math>(new MathImpl)));
```

实例化 MathImple 和 MathProcessor。

```
    ServiceManager sm;
    sm.add_service(processor);
```

实例化 ServiceManager sm，并 processor 添加至 sm 中。

```
    ThreadPool tp;
    tp.init(10);
```

实例化线程池，并初始化线程包含 10 个线程。

```
    Server server(&sm, &tp, 8504);
    if (0 != server.serve()) {
        return 0;
    }
```

实例化 Server，监听端口 8504，并对外提供服务。

3.4 编写 client

client.cpp 如下所示：

```
#include <iostream>
#include <bgcc.h>
#include "math.h"

using namespace bgcc;
using namespace math;

int main()
{
    MathProxy proxy(ServerInfo("localhost", 8504));

    int32_t a = 24;
    int32_t b = 8;
    int32_t sum;
    Result result;

    std::cout << "Please input number a: ";
    std::cin >> a;
    std::cout << "Please input number b: ";
    std::cin >> b;
```

```
result = proxy.add(a, b, sum);
if (proxy.get_errno() != 0) {
    std::cout << "Call proxy.add failed" << std::endl;
    return 0;
   }

   if (Result::E_OVERFLOW == result) {
       std::cout << "Error: overflow" << std::endl;
   } else if (Result::E_DOWNFLOW == result) {
       std::cout << "Error: downflow" << std::endl;
    } else {
    std::cout << a << "+" << b << "=" << sum << std::endl;
    }

return 0;
}
```

下面逐行进行分析。

```
#include <iostream>
```

包含 C++输入输出流头文件。

```
#include <bgcc.h>
using namespace bgcc;
```

包含 BGCC 头文件，并引用名称空间 bgcc。类 ServerInfo 是在 bgcc 名称空间中定义。

```
#include "math.h"
using namespace math;
```

包含 bidl2sl 工具根据 math.bidl 生成的源码头文件，并引用名称空间 math。该头文件中定义了服务代理类 math::MathProxy。

```
MathProxy proxy(ServerInfo("localhost", 8504));
```

实例化服务代理对象。

```
int32_t a = 24;
int32_t b = 8;
int32_t sum;
Result result;
```

定义接口调用的传入传出及返回值。

```
    std::cout << "Please input number a: ";
    std::cin >> a;
    std::cout << "Please input number b: ";
    std::cin >> b;
```

读入整数 a 和 b。

```
result = proxy.add(a, b, sum);
```

通过代理对象进行实际的方法调用。

```
if (proxy.get_errno() != 0) {
```

通过代理对象的 get_errno()方法，来判断本次接口调用是否成功或者说有无网络异常。当返回为 0 时表示调用成功。

```
    if (Result::E_OVERFLOW == result) {
    } else if (Result::E_DOWNFLOW == result) {
```

以上两行判断有无和溢出。

3.5 编译

```
g++ -o server server.cpp output-cpp/*.cpp -I output/include -I output-cpp -L output/lib -lbgcc -lpthread
g++ -o client client.cpp output-cpp/*.cpp -I output/include -I output-cpp -L output/lib -lbgcc -lpthread
```

3.6 运行

```
./server&
./client
```

运行结果

```
Please input number a: 1
Please input number b: 2
1+2=3
```

3.7 另一个客户端

前提：bgcc.jar 和 bgcc4j.so。

3.1~3.6 完整展现了 Linux 平台下 C++客户端与 C++服务器端间功能调用过程。本小节，介绍 Linux 平台下如何使用 Java 客户端调用 C++服务器提供的功能。

客户端代码

```
import bgcc.*;
```

```
import math.*;
import java.util.*;
import java.lang.*;

public final class Client {
    public static void main(String[] args) {
        ServerInfo serverInfo = new ServerInfo("localhost", 8504);
        math.Math.Proxy proxy = new math.Math.Proxy(serverInfo);

     int a = 24;
     int b = 7;
     IntHolder sum = new IntHolder(0);
     Result result;
     result = proxy.add(a, b, sum);

     if (0 != proxy.getErrno()) {
         System.err.println("Call add failed");
         return;
     }

     if (Result.E_OVERFLOW == result) {
         System.err.println("Error: overflow");
     } else if (Result.E_DOWNFLOW == result) {
         System.err.println("Error: downflow");
     } else {
         System.out.println(a + "+" + b + "=" + sum.value);
     }
    }
}
```

编译

```
javac -cp bgcc.jar output-java/math/*.java
javac -cp bgcc.jar:output-java Client.java
```

运行客户端程序

```
java -classpath bgcc.jar:output-java:. Client
```

4 安装

4.1 源码获取

BGCC 源码从以下站点获得：

4.2 源码结构

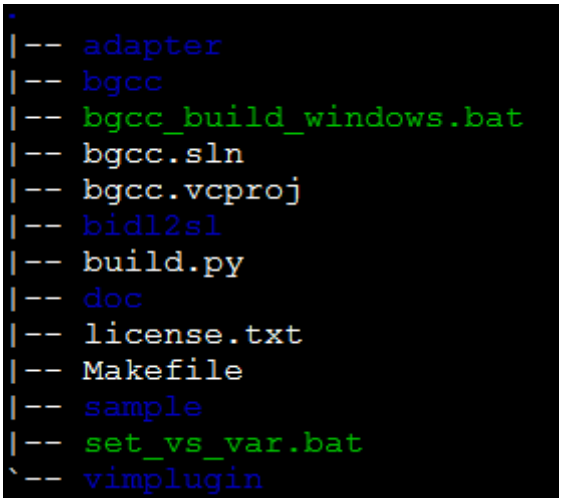

图 1 源码结构

如图 1 所示：

- adapter：目录。存放生成 bgcc.jar 及 libbgcc4j.so 的源码；
- bgcc：目录。存放生成 libbgcc.so 和 libbgcc.a 的源码；
- bgcc_build_windows.bat：文件。BGCC 在 Windows 平台上的自动编译脚本；
- bgcc.sln、bgcc.vcproj：文件。BGCC 在 Windows 平台上的 VS2003 工程文件；
- bidl2sl：目录。用于存入生成工具 bidl2sl 的源码；
- build.py：文件。BGCC 使用 Python 语言的自动编译脚本；
- doc：目录。参考文档目录；
- license.txt：文件。license 文件。
- Makefile：文件。BGCC 在 Linux 平台下自动编译脚本；
- sample：代码使用示例；
- set_vs_var.bat：文件。VS2003 环境变量设置脚本；
- vimplugin：目录。vim bidl 插件目录。

4.3 Linux 平台源码编译

4.3.1 C++

本小节适用于 C++程序员。

下载源码并解压，进入解压后生成的目录后执行如下命令：

```
make bgcc bidl2sl
```

编译成功后，在当前目录中生成 output 目录。output 目录结构如下所示：

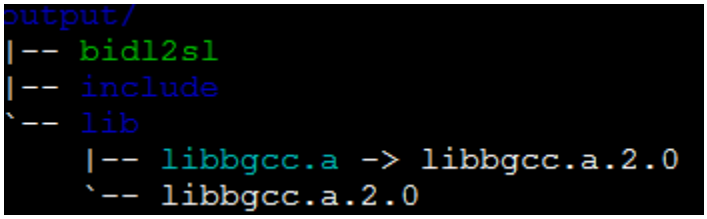

图 2　c++ output 目录结构

如图 2 所示，

- bidl2sl：可执行文件。用于将 bidl 接口定义文件转换为特定语言（如 C++、Java）的源程序；
- include：头文件目录。包含 BGCC 头文件。在使用 BGCC 时，请包含<bgcc.h>头文件，并引用 bgcc 名称空间；
- lib：库目录。包含 BGCC 的 C++静态链接库；
  - libbgcc.a：BGCC 静态链接库的符号链接，链接至当前目录下的 libbgcc.a.2.0；
  - libbgcc.a.2.0：BGCC 的静态链接库；

4.3.2　Java

本小节适用于 Java 程序员。要求使用环境已安装 JDK 和 ant。

下载源码并解压，进入解压后生成的目录后执行如下命令：

```
make adapter
```

编译成功后，在当前目录中生成 output 目录。output 目录结构如下所示：

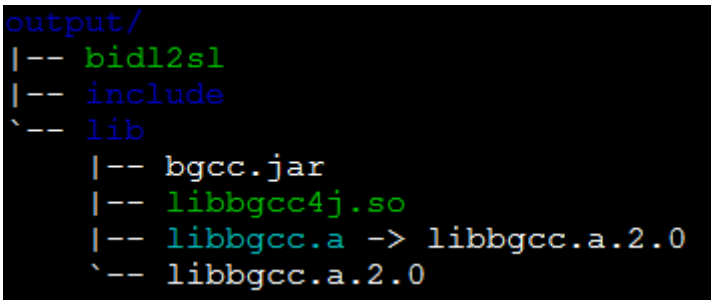

图 3　Java output 目录结构

如图 3 所示，

- bidl2sl：可执行文件。用于将 bidl 接口定义文件转换为特定语言（如 C++、Java）的源程序；

- include：头文件目录。包含 BGCC 头文件。在使用 BGCC 时，请包含<bgcc.h>头文件，并引用 bgcc 名称空间；
- lib：库目录。包含 BGCC 的 C++静态链接库；
  - bgcc.jar：BGCC Java 运行时；
  - libbgcc4j.so：BGCC JNI 动态链接库；
  - libbgcc.a：BGCC 静态链接库的符号链接，链接至当前目录下的 libbgcc.a.2.0；Java 程序员不需要此文件。
  - libbgcc.a.2.0：BGCC 的静态链接库；Java 程序员不需要此文件。

4.4 Windows 平台源码编译

4.4.1 C++

本小节适用于 C++程序员。

进入 bgcc 目录，双击“bgcc.sln”，打开 bgcc 工程文件。编译 bgcc 项目，即可生成 bgcc 的静态链接库 bgcc.lib。

BGCC 头文件可从源码中获取: bgcc/*.h。

4.4.2 Java

本小节适用于 Java 程序员。要求使用环境已安装 JDK 和 ant。

进入 bgcc 目录，双击“bgcc.sln”，打开 bgcc 工程文件。配置 bgcc4j 项目的属性，如图所示，将路径 jdk/include 和 jdk/include/win32 添加进包含目录中。

编译 bgcc4j 项目，即可生成 bgcc4j 的动态链接库 bgcc4j.so。

下面进行 bgcc.jar 的编译。

进入 bgcc/adapter/java/bgcc 目录下，执行如下命令：

```
ant
```

将在 bgcc/output/lib 目录下生成 bgcc.jar。

4.4.3 bidl2sl 在哪里？

在 Windows 平台上，在 VS 中不能直接编译生成 bidl2sl。但可以使用 cygwin 来编译 bidl2sl 源码来生成 bidl2sl 可执行程序。方法：

进入 bidl2sl 目录，然后执行 make。make 成功结束后，在当前目录中将生成可执行程序 bidl2sl。bidl2sl 的使用请参见《bidl2sl manual.docx》。

```
cd bgcc/bidl2sl
make
```

5 回调场景

在特殊场景下，服务器需要回调客户端功能。使用 BGCC，可能方便地实现客户功能回调。

在服务端提供 printer 服务。在客户端，提供 get_age 服务。

printer.bidl

```
namespace printer {
    class Printer {
        void print_message(string message);
    }
}
```

callback.bidl

```
namespace callback {
    class Callback {
        int32 get_age();
    }
}
```

编译生成原码

```
bidl2sl printer.bidl callback.bidl
```

client 如下:

```
#include <bgcc.h>
using namespace bgcc;
#include <printer.h>
using namespace printer;
#include <callback.h>
using namespace callback;
class CallbackImpl : public Callback {
    public:
        ~CallbackImpl() { }
        virtual int32_t get_age(const std::map<std::string, std::string> &ctx) {
            return 100;
```

```
            }
    };
    int main() {
        SharedPointer<IProcessor> xp(
                new CallbackProcessor (
                    SharedPointer<Callback>(
                        new CallbackImpl)));
        ServiceManager sm;
        sm.add_service(xp);
        ServerInfo si("localhost", 8500);
        PrinterProxy printer(si, 1, &sm);     //在代理对象的构造中，添加ServiceManager,以服务回
调
        printer.print_message("hello");
        bgcc::TimeUtil::safe_sleep_s(1);    //等待服务端回调完成
    }
```

server 如下

```
    #include <bgcc.h>
    using namespace bgcc;
    #include <printer.h>
    using namespace printer;
    #include <callback.h>
    using namespace callback;

    class Hurry : public Runnable {
        public:
            Hurry(const std::string& proxy_name) : _proxy_name(proxy_name) {
            }
            int32_t operator()(void* ) {
                CallbackProxy callback (_proxy_name); //从proxyname   构造反向代理
                int32_t age = callback.get_age(); //使用反向代理进行回调
                BGCC_TRACE("LXB", "%d", age);
                return 0;
            }
        private:
            std::string _proxy_name;
    };
    class PrinterImpl : public Printer {
        public:
            virtual void print_message(
                    const std::string& message,
                    const std::map<std::string, std::string> &ctx) {
                BGCC_TRACE("LXB", "%s", message.c_str());
                std::map<std::string, std::string>::const_iterator itr;
```

```
                itr = ctx.find("ProxyName");
                std::string proxy_name;
                if (itr != ctx.end()) {
                        proxy_name = itr->second;//从正向调用中获取proxyname
                        std::cout << proxy_name << std::endl;
                        Thread* t =    new Thread( //新启动线程，将proxyname传入
                                        SharedPointer<Runnable>(
                                                new Hurry(proxy_name)));
                        t->start();
                }
        }
};
int main() {
        log_open("bgcc.cfg");
        SharedPointer<IProcessor> printer(
                        new PrinterProcessor(
                                SharedPointer<Printer>(new PrinterImpl)));
        ServiceManager sm;
        sm.add_service(printer);
        ThreadPool tp;
        tp.init(10);
        Server server(&sm, &tp, 8500);
        server.serve();
}
```

bgcc.cfg

```
version = 1.0

[@log_devices]
level = TRACE
device_name = LXB
filepath = ./LXB.log
split_policy = SIZE
max_size = 5000000
#life_circle = -1
layout = %D [%N] %T {%F(%L)} %C

[@log_devices]
level = TRACE
device_name = bgcc
filepath = ./bgcc.log
split_policy = SIZE
max_size = 5000000
#life_circle = -1
```

```
layout = %D [%N] %T {%F(%L)} %C
```

编译

```
g++ -o server -Ioutput/include -Ioutput-cpp -Loutput/lib -lbgcc -lpthread server.cpp output-cpp/*.cpp
g++ -o client -Ioutput/include -Ioutput-cpp -Loutput/lib -lbgcc -lpthread client.cpp output-cpp/*.cpp
```

运行

```
./server&
./client
```

6 实用功能

本节内容适用于 C++程序员。

BGCC 除了实现核心功能外，还提供了大量 C++实用工具类，包括跨平台日志库，字符串操作函数、时间操作函数、同步机制类以及线程模型类。

6.1 日志库

BGCC 提供了方便和高效的日志库。

6.1.1 使用示例

mai.cpp：

```
#include <iostream>
#include <bgcc.h>
using namespace bgcc;
int main()
{
    log_open();
     //log_open("bgcc.cfg"); //如果有配置文件
   BGCC_TRACE("bgcc", "This is a trace message");
   BGCC_DEBUG("bgcc", "This is a debug message");
   BGCC_NOTICE("bgcc", "This is a notice message");
   BGCC_WARN("bgcc", "This is a warning message");
   BGCC_FATAL("bgcc", "This is a fatal message");
   log_close();
   return 0;
}
```

编译并运行以上程序：

```
g++ -o test_log main.cpp -I output/include -Loutput/lib -lbgcc -lpthread
./test_log
```

运行后，在当前目录下生成文件 bgcc.log。文件内容如下:

```
    2012-10-26 17:37:26.038 [warn ] 182894118112 {client.cpp(12)} This is a warning
message
    2012-10-26 18:43:13.467 [fatal ] 182894118112 {client.cpp(12)} This is a fatal
message
```

可以发现，bgcc.log 中只有两个日志级别的日志记录: warn 和 fatal。其他日志级别日志被丢弃。详见配置文件小节。

6.1.2 从哪里开始

欲使用 BGCC 日志功能，请在 main 函数开始处调用 log_open，并在 main 函数结束处调用 log_close。

6.1.3 日志宏

BGCC 日志库通过 10 个日志宏来提供日志功能。如下：

c 风格	BGCC_TRACE	打印 trace 级别日志
	BGCC_DEBUG	打印 debug 级别日志
	BGCC_NOTICE	打印 notice 级别日志
	BGCC_WARN	打印 warn 级别日志
	BGCC_FATAL	打印 fatal 级别日志
c++风格	BGCC_STREAM_TRACE	打印 trace 级别日志，支持 c++流式语法
	BGCC_STREAM_DEBUG	打印 debug 级别日志，支持 c++流式语法
	BGCC_STREAM_NOTICE	打印 notice 级别日志，支持 c++流式语法
	BGCC_STREAM_WARN	打印 warn 级别日志，支持 c++流式语法
	BGCC_STREAM_FATAL	打印 fatal 级别日志，支持 c++流式语法

以 BGCC_TRACE 为例，c 风格日志宏使用方式如下：

```
BGCC_TRACE("bgcc", "Count: %d, Name: %s", count, name);
```

以 BGCC_STREAM_TRACE 为例，c++风格日志宏使用方式如下：

```
BGCC_STREAM_TRACE("bgcc", "Count: "<< count << ", Name: "<< name);
```

日志宏的第一个参数为日志设备名。在没有配置文件的情况下，BGCC 日志功能默认提供”bgcc”日志设备。关于日志设备的详细信息请参见配置文件小节。

6.1.4 配置文件

BGCC 提供的缺省的日志设备名为“bgcc”，日志级别为“warn”。

BGCC 可以通过配置文件来配置缺省日志设备或添加新日志设备。BGCC 配置文件配置项均大小写敏感。其中，level, device_name, filepath, split_policy, layout 为必选项。max_size 和 life_circle 是否必须则根据 split_policy 的值不同而不同。当 split_policy 取值为 SIZE 时，max_size 为必选项。当 split_policy 为 TIME 时，life_circle 为必选项。格式举例说明如下：

```
[@log_device]
```

```
level = TRACE
device_name = bgcc
filepath = ./bgcc.log
split_policy = SIZE
max_size = 524288000
layout = %D [%N] %T {%F(%L)} %C
[@log_device]
level = NOTICE
device_name = lxb
filepath = ./lxb.log
split_policy = TIME
life_circle = 3000000
layout = %D [%N] %T {%F(%L)} %C
```

level 可取值如下：

TRACE	trace 级别
DEBUG	debug 级别
NOTICE	notice 级别
WARN	warn 级别
FATAL	fatal 级别

split_policy 可取值如下：

SIZE	按大小切分
TIME	按时间切分

layout 中各控制字符的含义：

%D	日志产生的日期和时间。格式“YYYY-MM-DD HH:mm:SS.XXX”
%N	日志级别名称，包括 trace, debug, notice, warn, fatal
%T	打印日志的线程 ID
%F	日志所在的文件名
%L	日志所在的行号
%C	日志内容
%%	%

其他以%引导的字符将与%一同原样输出。

6.2 字符串

toupper_inplace

原型： void toupper_inplace(char* str);

说明： 将 C 风格字符串 str 转换为大写，原字符串被改写

参数： str　待转换的 C 风格字符串

返回值： void

举例：

```
char str[] = "a";
bgcc::StringUtil::toupper_inplace(str);
```

toupper

原型：std::string toupper(const char* str);
说明：将 C 风格字符串 str 转换为大写，原字符串保持不变
参数：str　待转换的 C 风格字符串
返回值：字符串 str 的大写形式
举例：

```
char str[] = "a";
std::string upstr = bgcc::StringUtil::toupper(str);
```

toupper

原型：std::string toupper(const std::string& str);
说明：将字符串 str 转换为大写，原字符串保持不变
参数：str　待转换的字符串
返回值：字符串 str 的大写形式
举例：

```
std::string str("a");
std::string upstr = bgcc::StringUtil::toupper(str);
```

tolower_inplace

原型：void tolower_inplace(char* str);
说明：将 C 风格字符串 str 转换为小写，原字符串被改写
参数：str　待转换的 C 风格字符串
返回值：void
举例：

```
char str[] = "A";
bgcc::StringUtil::tolower_inplace(str);
```

tolower

原型：std::string tolower (const char* str);
说明：将 C 风格字符串 str 转换为小写，原字符串保持不变
参数：str　待转换的 C 风格字符串
返回值：字符串 str 的小写形式
举例：

```
char str[] = "A";
std::string lowstr = bgcc::StringUtil::tolower (str);
```

tolower

原型：std::string tolower (const std::string& str);
说明：将字符串 str 转换为小写，原字符串保持不变
参数：str　字符串 str 的小写形式
返回值：void
举例：

```
std::string str("A");
std::string lowstr = bgcc::StringUtil::tolower (str);
```

ltrim_inplace

原型：void ltrim_inplace(char* str);
说明：去除字符串左端的空白，原地修改
参数：str 原字符串
返回值：void
举例：

```
char hello[] = "    hello";
bgcc::StringUtil::ltrim_inplace(hello);
```

ltrim

原型：std::string ltrim(const char* str);
说明：去除字符串左端的空白，原字符串保持不变
参数：str 原字符串
返回值：去除左端空白的字符串
举例：

```
char hello[] = "    hello";
std::string trimedstr = bgcc::StringUtil::ltrim (hello);
```

ltrim

原型：std::string ltrim(const std::string& str);
说明：去除字符串右端的空白，原地修改
参数：str 原字符串
返回值：去除左端空白的字符串
举例：

```
std::string hello("    hello");
std::string trimedstr = bgcc::StringUtil::ltrim (hello);
```

trim_inplace

原型：void trim_inplace(char* str);
说明：去除 C 风格字符串两端空白，原地修改
参数：str 原字符串
返回值：void

举例：

```
char hello[] = "    hello    ";
bgcc::StringUtil::trim_inplace (hello);
```

trim

原型：std::string trim(const char* str);
说明：去除 C 风格字符串两端空白
参数：str 原字符串
返回值：去除两端空白的字符串
举例：

```
char hello[] = "    hello    ";
std::string trimedstr = bgcc::StringUtil::trim(hello);
```

trim

原型：std::string trim(const std::string& str);
说明：去除字符串两端空白
参数：str 原字符串
返回值：去除两端空白的字符串
举例：

```
std::string hello("    hello    ");
std::string trimedstr = bgcc::StringUtil::trim(hello);
```

split_string

原型：void split_string(const std::string& str, const std::string& separator, std::vector<std::string>& container, bool filter_empty = false);
说明：切分子串
参数：
str 原字符串
separator 分隔串
container 子串结果
filter_empty 是否过滤空子串(length == 0)
返回值：void
举例：

```
std::string hello("3.3");
std::vector<std::string> result;
bgcc::StringUtil::split_string(hello, ".", result, true);
```

replace_string

原型：void replace_string(const std::string& str, const std::string& separator, const std::string& replacement);

说明：替换子串
参数：
str 原字符串
target 目标串
replacement 替换串
返回值：替换完成后的字符串
举例：

```
std::string hello("hallo world");
std::vector<std::string> result;
hello = bgcc::StringUtil::replace_string(hello, "hello", "hello");
```

str2uint32

原型：bool str2uint32(const char* str, uint32_t& number);
说明：将数字字符串转化为 32 位无符号整数
参数：
str 数字字符串
number 结果整数
返回值：true 表示转换成功；否则返回 false
举例：

```
uint32_t num;
bool ret;
ret = bgcc::StringUtil::str2uint32("9999", num);
if (true == ret) {
    std::cout << "转换成功"<< std::endl;
}
else {
    std::cout << "转换失败" << std::endl;
}
```

str2int32

原型：bool str2int32(const char* str, int32_t& number);
说明：将数字字符串转化为 32 位有符号整数
参数：
str 数字字符串
number 结果整数
返回值：true 表示转换成功；否则返回 false
举例：

```
int32_t num;
bool ret;
ret = bgcc::StringUtil::str2int32("9999", num);
if (true == ret) {
    std::cout << "转换成功" << std::endl;
```

```
    }
    else {
        std::cout << "转换失败" << std::endl;
    }
```

str2uint64

原型：bool str2uint64(const char* str, uint64_t& number);
说明：将数字字符串转化为 64 位无符号整数
参数：
str 数字字符串
number 结果整数
返回值：true 表示转换成功；否则返回 false
举例：

```
    uint64_t num;
    bool ret;
    ret = bgcc::StringUtil::str2uint64("9999", num);
    if (true == ret) {
        std::cout << "转换成功" << std::endl;
    }
    else {
        std::cout << "转换失败" << std::endl;
    }
```

str2int64

原型：bool str2int64(const char* str, int64_t& number);
说明：将数字字符串转化为 64 位有符号整数
参数：
str 数字字符串
number 结果整数
返回值：true 表示转换成功；否则返回 false
举例：

```
    int32_t num;
    bool ret;
    ret = bgcc::StringUtil::str2int64 ("9999", num);
    if (true == ret) {
        std::cout << "转换成功" << std::endl;
    }
    else {
        std::cout << "转换失败" << std::endl;
    }
```

generate_uuid

原型：std::string generate_uuid();
说明：生成“1b4e28ba-2fa1-11d2-883f-b9a76”格式的 uuid 字符串
参数：
无
返回值：uuid 字符串
举例：

```
std::string uuid = bgcc::StringUtil::generate_uuid();
```

ipv4_ntoa

原型：bool ipv4_ntoa(uint32_t src, std::string& dest);
说明：整型 ip 地址转换为字符串形式
参数：
src 整型 IP
dest 接收字符串形式的 IP
返回值：成功转换返回 true；否则返回 false
举例：

```
uint32_t src = 0xffffffff;
std::string dest;
bgcc::StringUtil::ipv4_ntoa(src, dest);
```

ipv4_aton

原型：bool ipv4_aton(const char* src, uint32_t& dest);
说明：字符串 IP 地址转换为整型
参数：
src 字符串形式的 IP
dest 整型 IP
返回值：成功转换返回 true；否则返回 false
举例：

```
uint32_t dest;
bool ret;
std::string src = "192.168.1.3";
ret = bgcc::StringUtil::ipv4_aton(src.c_str(), dest);
if (true == ret) {
        std::cout << "转换成功" << std::endl;
} else {
        std::cout << "转换失败" << std::endl;
}
```

xstrncpy

原型：int32_t xstrncpy(char* dest, const char* src, int32_t n);
说明：xstrncpy 功能同 strncpy。区别在于 xstrncpy 返回值为成功复制的字节数
参数：
dest 目的缓冲区
src 源字符串
n 目的缓冲区大小
返回值：成功复制的字节数。错误时返回-1
举例：

```
char buffer[BUFSIZ];
int32_t ret = 0;
ret = xstrncpy(buffer, "hello", BUFSIZ);
```

6.3 时间

get_timestamp_s

原型：uint64_t get_timestamp_s();
说明：获取时间戳（秒）
参数：
无
返回值：返回从 Epoch 所经过的秒数
举例：

```
uint64_t t = bgcc::TimeUtil:: get_timestamp_s();
```

get_timestamp_ms

原型：uint64_t get_timestamp_ms();
说明：获取时间戳（微秒）
参数：
无
返回值：返回从 Epoch 所经过的微秒数
举例：

```
uint64_t t = bgcc::TimeUtil:: get_timestamp_ms();
```

safe_sleep_s

原型：void safe_sleep_s(uint32_t second);
说明：支持中断的 sleep（秒）
参数：
second 睡眠的秒数
返回值：无
举例：

```
bgcc::TimeUtil::safe_sleep_s(4);
```

safe_sleep_ms

原型：void safe_sleep_ms(uint32_t millisecond);
说明：支持中断的 sleep（毫秒）
参数：
second 睡眠的毫秒数
返回值：无
举例：

```
bgcc::TimeUtil::safe_sleep_ms(500);
```

format_datetime_str

原型：std::string format_datetime_str(uint64_t millisecond);
说明：将时间戳格式化
参数：
millisecond 从 Epoch 所经过的毫秒数
返回值：时间戳字符串
举例：

```
uint64_t timestamp = 1339388847250ULL;
std::string str = bgcc::TimeUtil::format_datetime_str(timestamp);
```

6.4 信号量

信号量实现线程间的任务同步。Semaphore 类提供跨平台且统一的接口 API。

```
class Semaphore {
public:
    Semaphore(int32_t ninit = 0);
    ~Semaphore();
    int32_t wait(uint32_t millisecond = BGCC_SEMA_WAIT_INFINITE);
    int32_t signal();
};
```

如上，Semaphore 类提供三个重要的 API，分别是构造函数，wait 和 signal。

构造函数

通过参数 ninit 来指定信号量初值，即在不调用 signal 的情况下，连续成功调用 wait 的次数。默认值为 0。

wait

等待信号量被触发

signal

触发信号量

示例

```
bgcc::Semaphore sema;
sema.signal();
sema.wait();
int32_t ret = sema.wait(100);
if (0 == ret) {
        std::cout << "sema wait success" << std::endl;
}
else if (E_BGCC_TIMEOUT == ret) {
        std::cout << "time out"<< std::endl;
}
else {
        std::cout << "other error" << std::endl;
}
```

6.5 同步容器

同步容器是一个线程安全容器模板，采用先入后出策略。

```
template <typename ElemType>
class SyncVector : public Shareable {
public:
        SyncVector();
        int32_t put(ElemType elem);
        int32_t get(ElemType& elem, int32_t millisecond);
};
```

如上，SyncVector 提供了两个重要的 API：put 和 get。

put

向同步 vector 中添加元素（允许重复）。同时触发信号量

get

从同步 vector 中取元素。如果同步 vector 为空，将阻塞指定时长。

BGCC_SEMA_WAIT_INFINITE 表示无限阻塞。

示例

```
bgcc::SyncVector<int32_t> numbers;
```

线程 1

```
numbers.put(1);
numbers.put(2);
numbers.put(3);
```

线程 2

```
int32_t num;
int32_ ret;
while (true) {
        ret = numbers.get(num, 100);
        if (0 == ret) {
                printf("%d\t", num);
        }
        else if (E_BGCC_TIMEOUT == ret) {
                break;
        }
        else {
                printf("error");
                break;
        }
}
```

6.6 线程

操作系统线程模型的抽象，负责线程的创建、执行、等待以及销毁

```
typedef void* (*run_func_t)(void*);
typedef SharedPointer<Runnable> RunnableSP;

class Thread : public Shareable {
public:
        Thread(const RunnableSP& runner, bool detached = false);
        Thread(run_func_t func, void* arg = NULL, bool detached = false);
        ~Thread();
        bool start();
        bool join();
        bool stop();
};
```

创建

线程的创建有两种方式，一是通过 run_func_t 函数指针及需要传入的函数参数指针进行创建，二是通过一个 Runnable 智能指针进行创建。如下：

```
//方式1
void* coco(void* arg) {
        while (true) {
                std::cout << "coco" << std::endl;
                bgcc::TimeUtil::safe_sleep_ms(100);
        }
```

```
}
Thread t1(coco);

//方式2
class CocoRunner : public Runnable {
public:
	virtual int32_t operator()(void* arg) {
		while (true) {
			std::cout <<"coco" << std::endl;
			bgcc::TimeUtil::safe_sleep_ms(100);
		}
	}
};
Thread t2(SharedPointer<Runnable>(new CocoRunner));
```

执行

调用 Thread 对象的 start 方法，可以启动线程执行体的执行。

```
t1.start();
t2.start();
```

等待

对于 joinable 线程，即在创建时 detached 参数为 false 的线程，可以调用 join 方法来等待线程执行体的执行结束。对于 joinable 线程，当 Thread 对象析构时会自动调用 join 方法。

终止与销毁

调用 Thread 对象的 stop 方法，可强制终止线程执行体的执行。

```
t1.stop();
```

6.7 互斥锁

Mutex 类提供了跨平台的互斥锁操作 API，包括互斥锁的创建，加锁与解锁。

```
class Mutex {
public:
	Mutex();
	int32_t lock(uint32_t millisecond = BGCC_MUTEX_WAIT_INFINITE);
	int32_t unlock();
};
```

创建

Mutex 构造函数负责创建一个互斥锁。初始状态下未加锁。

加锁

调用 lock 对互斥锁加锁。加锁成功返回 0。可以指定一个超时时间。当超时后 lock 返回 E_BGCC_TIMEOUT。

解锁

调用 unlock 对互斥锁解锁。

示例

```
bgcc::Mutex mutex;
mutex.lock();
mutex.unlock();
```

6.8 线程池

线程池维护一组预先创建好的线程及一个任务队列。线程依次执行任务队列中的任务。

```
class ThreadPool : public Shareable {
public:
    int32_t init(int32_t nThreads = DEFAULT_THREADS_NUM);
    bool addTask(RunnableSharedPointer pr);
};
```

ThreadPool 提供了两个重要的 API：init 和 addTask。

初始化

初始化线程池中并发的线程总个数。

添加任务

向线程池任务队列中添加新任务。

示例

```
ThreadPool tp;
tp.init(100);
tp.addTask(SharedPointer<Runnable>(new CocoRunner)); //CocoRunner见线程一节
```