数据类型

徐枫

清华大学软件学院

feng-xu@tsinghua.edu.cn

QString

管理文本

- 简单的C字符串是方便的，但这仅限于本地字符编码

```
char *text = "Hello world!";
```

  - 从C语言开始，其中一个最基本的，也依旧复杂的，是字符串管理。
- QString类——试图成为现代的字符串类
  - **Unicode** 和 codecs
  - 隐式共享的性能
    - 即实际字符串数据不被复制，除非其中一个副本被更改

Unicode（中文：万国码、国际码、统一码、单一码）是计算机科学领域里的一项业界标准。它对世界上大部分的文字系统进行了整理、编码，使得电脑可以用更为简单的方式来呈现和处理文字。

QString

- 支持存储Unicode字符串，几乎当前在用的所有书写系统都能表示

- 支持从不同的本地编码转换或者转成不同的本地编码

  QString::toAscii – QString::toLatin1 – QString::toLocal8Bit

  - **ASCII:**美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为ASCII码, ASCII码一共规定了128个字符的编码
  - **UTF-8:**UTF-8是Unicode的实现方式之一。UTF-8最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

- 提供了一个方便的字符串检查和修改的API

建立字符串

- 有三种建立字符串的主要方法
  - 运算符‘+’方法

```
QString res = "Hello " + name +
    ", the value is " + QString::number(42);
```

多次构造 QString

  - QStringBuilder 的方法

```
QString res = "Hello " % name %
    ", the value is " % QString::number(42);
```

一次构造 QString

  - arg 方法

```
QString res = QString("Hello %1, the value is %2")
    .arg(name)
    .arg(42);
```

QStringBuilder

- 使用+运算符来连接字符串，这需要多次内存分配和字符串长度检查
- 一个更好的方式是包含QStringBuilder并使用%操作符
- 该字符串生成器在连接之前一次性收集所有字符串的长度，只需执行一次内存分配

```
QString res = "Hello " % name %
    ", the value is %" % QString::number(42);
```

```
QString temp = "Hello ";
temp = temp % name;
temp = temp % ", the value is %"
temp = temp % QString::number(42);
```

分成多个小步骤连接字符串会降低性能

QString::arg

- arg方法用值来替换 %1-99

```
"%1 + %2 = %3, the sum is %3"
```

%n的所有实例都被替换

- 可以处理字符串，字符，整型和浮点型

```
...).arg(qulonglong a)
...).arg(short a)
...).arg(ushort a)
...).arg(QChar a)
...).arg(char a)
...).arg(double a)
...).arg(QString, ... QString)
...).arg(int a)
...).arg(uint a)
...).arg(long a)
...).arg(ulong a)
...).arg(qlonglong a)
```

每次上限为9个参数

- 能在数字基数之间转换

```
...).arg(value, width, base, fillChar);
...).arg(42, 3, 16, QChar('0'));
```

子串

- 使用left, right 和 mid访问子串

```
QString s = "Hello world!";
r = s.left(5); // "Hello"
r = s.right(1); // "!"
r = s.mid(6,5); // "world"
```

- 如果mid不指定长度，则返回字符串的剩余部分

```
r = s.mid(6); // "world!"
```

- 用 replace查找和替代字符串

```
r = s.replace("world", "universe"); // "Hello universe!"
```

打印到控制台

- Qt是一个主要用于可视应用的工具包，即不专注于命令行界面的
- 要打印, 用 qDebug 函数
  - 它总是可用, 但是在建立发布版本时会静默
  - 像printf 函数那样工作 (但加上 “\n”)
  - 使用qPrintable宏很容易打印QString的文本

```
qDebug("Integer value: %d", 42);
qDebug("String value: %s", qPrintable(myQString));
```

  - 当包含QtDebug后，能与流操作符一起使用

```
#include <QtDebug>

qDebug() << "Integer value:" << 42;
qDebug() << "String value:" << myQString;
qDebug() << "Complex value:" << myQColor;
```

与数字间转换

- 把数字转换为字符串

```
QString::number(int value, int base=10);
QString twelve = QString::number(12); // "12"
QString oneTwo = QString::number(0x12, 16); // "12"

QString::number(double value, char format='g', int precision=6);
QString piAuto = QString::number(M_PI); // "3.14159"
QString piScientific = QString::number(M_PI,'e'); // "3.141593e+00"
QString piFixedDecimal = QString::number(M_PI,'f',2); // "3.14"
```

- 把字符串转换为数值

```
bool ok;
QString i = "12";
int value = i.toInt(&ok);
if(ok) {
    // Converted ok
}
```

```
bool ok;
QString d = "12.36e-2";
double value = d.toDouble(&ok);
if(ok) {
    // Converted ok
}
```

不能处理千分位

与std::(w)string一起工作

- 当连接第三方库和其他代码时，与标准库的字符串转换很方便
- 从标准库的字串转换成其他

```
std::string ss = "Hello world!";
std::wstring sws = "Hello world!";

QString qss = QString::fromStdString(ss);
QString qsws = QString::fromStdWString(sws);
```

假定为ASCII

- 转成标准库字符串

```
QString qs = "Hello world!";
std::string ss = qs.toStdString();
std::wstring sws = qs.toStdWString();
```

Wstring是宽字符，unicode编码

Empty字符串和null字符串

- 一个QString可以为null，
  即什么也没有包含

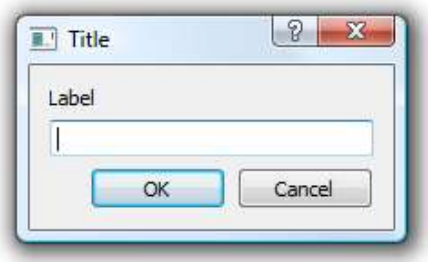
```
QString n = QString();

n.isNull();  // true
n.isEmpty(); // true
```

相对于一个empty字符串来说，传递没有字符的串是非常有用的，

见QInputDialog：gettext –当取消时返回一个null字符串

如果用户用cancel来关闭对话框或关闭对话框窗口的话，返回一个null字符串。



- 它也可是empty，即包含
  一个空字符串

```
QString e =  "";

e.isNull();  // false
e.isEmpty(); // true
```

如果用户不输入任何文本却选择ok的话，一个对话框可以返回一个empty字符串

分割和组合

- 一个QString 能够被分割成子串

```
QString whole = "Stockholm - Copenhagen - Oslo - Helsinki";
QStringList parts = whole.split(" - ");
```

- 由此产生的对象是QstringList,它可被组合成一个QString

```
QString wholeAgain = parts.join(", ");
    // Results in "Stockholm, Copenhagen, Oslo, Helsinki"
```

QStringList

QStringList

- QStringList是一个专门列表类型
- 为存储字符串而设计，提供了一个方便的作用于列表中字符串的API

QStringList

- 这个类使用隐含共享
  - 一个共享类是由一个指向共享数据块的的指针组成的，该数据块包含一个引用计数和实际数据
  - 当一个隐式共享类的对象被创建时，它会引用计数设为1。无论何时，当一个新的对象引用这个共享数据时，引用计数会增加，当一个对象解引用这个共享数据时，引用计数会减少。当引用计数变为0时，共享数据会被删除。
  - 深拷贝意味着复制一个对象；浅拷贝只拷贝引用，即只有指向共享数据的指针被复制
  - 执行一次浅拷贝是非常快的，因为这只牵涉到设置一个指针和增加引用计数。
  - 隐式共享对象的赋值（使用operator=）被实现为浅拷贝
  - 副本在修改时才真的进行拷贝操作

建立和修改字符串列表

- 用<< 操作符把字符串增加到字符串列表

```
QStringList verbs;
verbs = "running" << "walking" << "compiling" << "linking";
```

- replaceInStrings函数能让你在QStringList的所有字符串中进行搜索和替换。

```
qDebug() << verbs; // ("running", "walking", "compiling", "linking")
verbs.replaceInStrings("ing", "er");
qDebug() << verbs; // ("runner", "walker", "compiler", "linker")
```

排序与筛选

- QStringList可以进行排序...

```
qDebug() << capitals; // ("Stockholm", "Oslo", "Helsinki", "Copenhagen")
capitals.sort();
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm")
```

- ...筛选...

默认区分大小写

```
QStringList capitalsWithO = capitals.filter("o");
qDebug() << capitalsWithO; // ("Copenhagen", "Oslo", "Stockholm")
```

- ...及清除重复的条目

```
capitals << capitalsWithO;
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm",
                      //  "Copenhagen", "Oslo", "Stockholm")
capitals.removeDuplicates();
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm")
```

遍历字符串

- 使用操作符 [] 和 length 函数,你可以遍历QStringList的内容

```
QStringList capitals;
for(int i=0; i<capitals.length(); ++i)
    qDebug() << capitals[i];
```

- 另一种方法是使用at() 函数，它提供列表项的只读访问

- 你也可以使用foreach 宏，它扩展到一个基于迭代器的循环

```
QStringList capitals;
foreach(const QString &city, capitals)
    qDebug() << city;
```

Qt中的集合

Qt的集合

- QStringList接口不是唯一的字符串列表。 QStringList是由QList <QString>派生的。
- QList是众多Qt容器模板类中的一个
  - QLinkedList - 在中间快速插入，通过迭代器
  - QVector - 使用连续内存，缓慢插入
  - QStack – LIFO, 后进先出
  - QQueue – FIFO, 先进先出
  - QSet – 唯一值
  - QMap – 关联数组
  - QHash – 关联数组，比QMap快，但需要哈希
  - QMultiMap – 通过每个键的多个值关联数组
  - QMultiHash – 通过每个键的多个值关联数组

关联到Qlist的快慢将是我们的参考和标准

填充

- 你可以用操作符<<填充一个QList

```
QList<int> fibonacci;
fibonacci << 0 << 1 << 1 << 2 << 3 << 5 << 8;
```

- 函数prepend, insert 和append 也可以使用

```
QList<int> list;
list.append(2);
```

index 0: **2**

```
list.append(4);
```

index 0: **2**
index 1: **4**

```
list.insert(1,3);
```

index 0: **2**
index 1: **3**
index 2: **4**

```
list.prepend(1);
```

index 0: **1**
index 1: **2**
index 2: **3**
index 3: **4**

删除

- 使用removeFirst， removeAt， removeLast从Qlist中删除列表项

```
while(list.length())
    list.removeFirst();
```

- 用takeFirst, takeAt, takeLast去得到一个列表项并删除

```
QList<QWidget*> widgets;
widgets << new QWidget << new QWidget;
while(widgets.length())
    delete widgets.takeFirst();
```

- 使用removeAll或removeOne删除特定值的列表项

```
QList<int> list;
list << 1 << 2 << 3 << 1 << 2 << 3;
list.removeAll(2); // Leaves 1, 3, 1, 3
```

访问

- 一个QList 的索引范围是0 -（length-1）
- 单个列表项可以用at 或者[] 操作符来访问。如果你能接受超出界限的情况，可以用value。

```
for(int i=0; i<list.length(); ++i)
    qDebug("At: %d, []: %d", list.at(i), list[i]);

for(int i=0; i<100; ++i)
    qDebug("Value: %d", list.value(i));
```

当索引超出范围时返回默认构造值

- [ ]运算符返回一个可修改的引用

```
for(int i=0; i<list.length(); ++i)
    list[i]++;
```

迭代 - Java风格

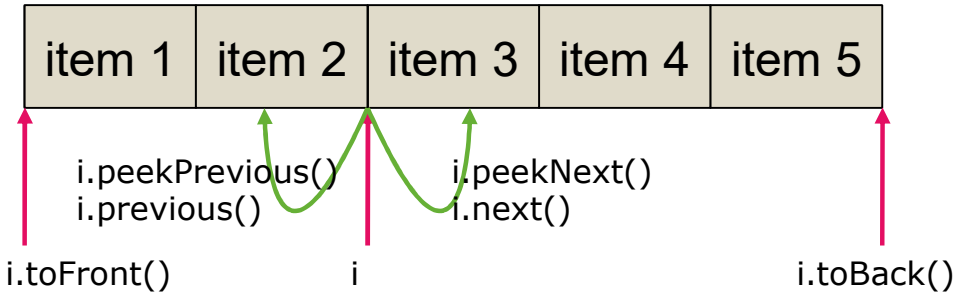
- Qt 支持 Java 风格迭代器

```
QListIterator<int> iter(list);
        while(iter.hasNext())
            qDebug("Item: %d", iter.next());
```

均返回一个值并迈向列表的下一个位置

只读迭代器，如果需要修改列表项，使用QMutableListIterator

- Java风格的迭代器指向条目之间
  - toFront把迭代器置于第一项前面
  - toBack把迭代器置于最后一项后面
  - 用peekNext和peekPrevious
    - 观察对后/前应列表项
  - 用 next 或 previous观察并指向后/前对应列表项



迭代 - STL的风格

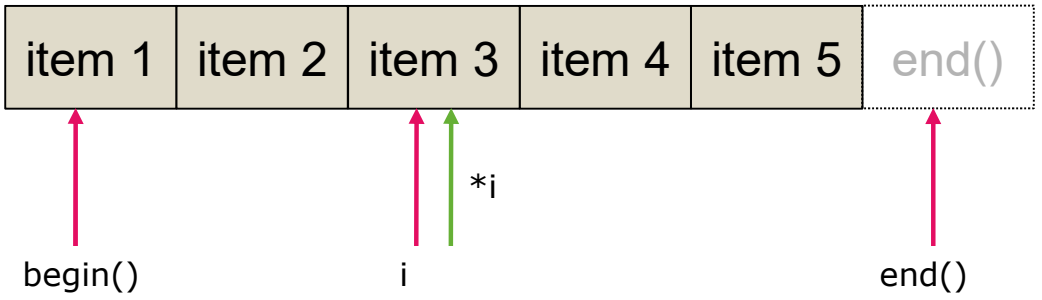
- Qt 支持 STL 风格的迭代器

如果你需要修改列表项，使用迭代器

```
for(QList<int>::ConstIterator iter=list.begin();
    iter!=list.end(); ++iter)
    qDebug("Item: %d", *iter);
```

STL和Qt两者的命名可以使用。Iterator|iterator和ConstIterator | const_iterator

- STL的迭代器指向每个列表项，并以此作为结束标记无效项
  - 第一项用begin来返回
  - 结束标志用end返回
  - * 操作符关联项的值
  - 当你向后遍历访问之前，必须移动操作符



懒惰式的迭代

- 遍历整个集合，使用foreach

```
QStringList texts;
foreach(QString text, texts)
    doSomething(text);
```

```
QStringList texts;
foreach(const QString &text, texts)
    doSomething(text);
```

使用常量的引用有助于提高性能，但无法让你改变列表的内容

```
const QList<int> sizes = splitter->sizes();
QList<int>::const_iterator i;
for(i=sizes.begin(); i!=sizes.end(); ++i)
    processSize(*i);
```

由于隐性共享，复制代价低廉（第一行无需执行拷贝）

与STL的交互

- QList 能与相应的std::list相互转换

```
QList<int> list;
list << 0 << 1 << 1 << 2 << 3 << 5 << 8 << 13;

std::list<int> stlList = list.toStdList();

QList<int> otherList = QList<int>::fromStdList(stlList);
```

从Qt 列表到STL列表

从STL列表到 Qt 列表

- 与STL的相互转换意味着对列表内容进行深度复制 – 不存在隐含共享

其他集合

- 谁能代替QList，它们跟QList有何区别？

- QLinkedList
  - 用索引访问缓慢
  - 使用迭代器很快
  - 快速（恒定时间）在列表的中间插入

- QVector
  - 使用连续的内存空间，随机访问快
  - 插入和置首缓慢

其他集合

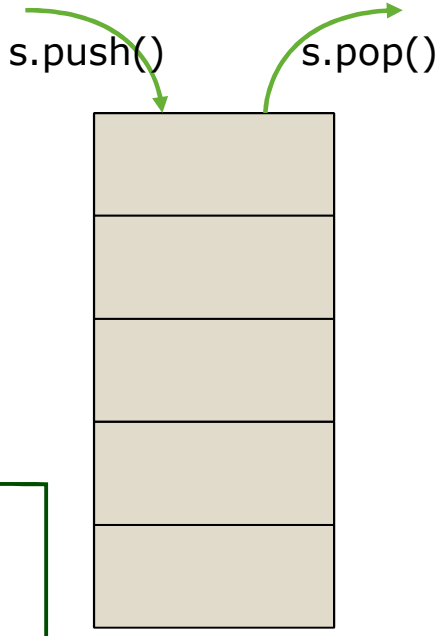
Collection	Index access	Insert	Prepend	Append
QList	O(1)	O(n)	Amort. O(1)	Amort. O(1)
QLinkedList	O(n)	O(1)	O(1)	O(1)
QVector	O(1)	O(n)	O(n)	Amort. O(1)

- 注意，摊销行为（amortized behavior）是指对若干条指令整体进行考虑其时间复杂度（以获得更接近实际情况的时间复杂度）其他集合是以Qlist为基础的
  - QStringList
  - QStack
  - QQueue
  - QSet

特殊情况 - QStack

- 栈，一个后进先出（LIFO）的容器

  后进先出

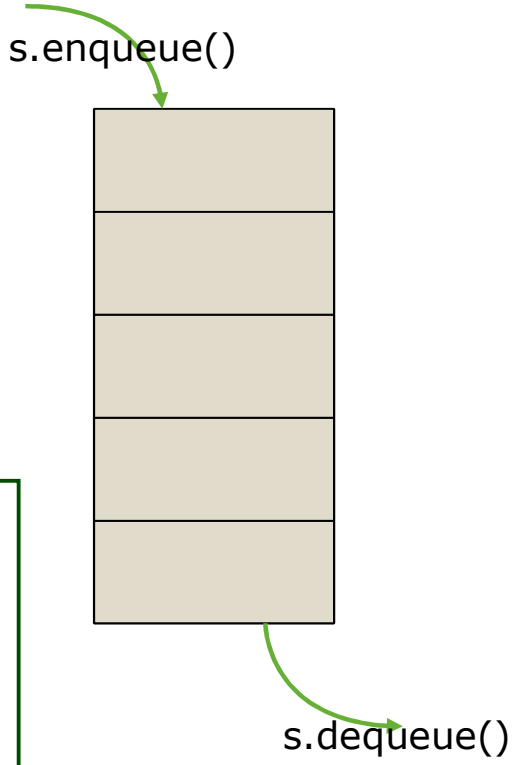


- 列表项压入
- 列表项弹出
- 用top()取出顶项

```
QStack<int> stack;
stack.push(1);
stack.push(2);
stack.push(3);
qDebug("Top: %d", stack.top()); // 3
qDebug("Pop: %d", stack.pop()); // 3
qDebug("Pop: %d", stack.pop()); // 2
qDebug("Pop: %d", stack.pop()); // 1
qDebug("isEmpty? %s", stack.isEmpty()?"yes":"no");
```

特殊情况- QQueue

- 队列，一个先进先出（ FIFO）的 容器
  先进先出



- 项排队进入队列
- 项从队列中出列
- 取第一项可以用head()

```
QQueue<int> queue;
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
qDebug("Head: %d", queue.head());   // 1
qDebug("Pop: %d", queue.dequeue()); // 1
qDebug("Pop: %d", queue.dequeue()); // 2
qDebug("Pop: %d", queue.dequeue()); // 3
qDebug("isEmpty? %s", queue.isEmpty()?"yes":"no");
```

特殊情况- QSet

- 一个集合包含值，但对每个值只有一个实例。
- 可以判断一个值是否是集合的一部分

```
QSet<int> primes;
primes << 2 << 3 << 5 << 7 << 11 << 13;
for(int i=1; i<=10; ++i)
    qDebug("%d is %sprime", i, primes.contains(i)?"":"not ");
```

- 也可以遍历一个集合，看到所有的值

```
foreach(int prime, primes)
    qDebug("Prime: %d", prime);
```

- 可以转换一个QList 到一个QSet

```
QList<int> list;
list << 1 << 1 << 2 << 2 << 2 << 3 << 3 << 5;
QSet<int> set = list.toSet();
qDebug() << list; // (1, 1, 2, 2, 2, 3, 3, 5)
qDebug() << set;  // (1, 2, 3, 5)
```

键 - 值集合

- QMap和QHash类让你创建关联数组

```
QMap<QString, int> map;

map["Helsinki"] = 1310755;
map["Oslo"] = 1403268;
map["Copenhagen"] = 1892233;
map["Stockholm"] = 2011047;

foreach(const QString &key, map.keys())
    qDebug("%s", qPrintable(key));

if(map.contains("Oslo"))
{
    qDebug("Oslo: %d",
                map.value("Oslo"));
}
```

```
QHash<QString, int> hash;

hash["Helsinki"] = 1310755;
hash["Oslo"] = 1403268;
hash["Copenhagen"] = 1892233;
hash["Stockholm"] = 2011047;

foreach(const QString &key, hash.keys())
    qDebug("%s", qPrintable(key));
```

使用QMap

- QMap 类需要操作符< 来定义键类型的大小比较

  此操作符用来保持键的顺序，按照Key的次序存储数据

- 填充使用运算符[]或 insert来完成

```
map["Stockholm"] = 2011047;

map.insert("London", 13945000);
```

- 对于读取，用value 结合contains

```
if(map.contains("Oslo"))
    qDebug("Oslo: %d",
           map.value("Oslo"));

qDebug("Berlin: %d", map.value("Berlin",42));
```

用 value 而不是 [] 以避免误增加项

可选默认值。如果无此键但有默认值，则返回默认值；如果没有指定一个默认值且无此键，将返回构造值（0、空QString…）

哈希

- QMap使用了给定的模板中的类型的键
- QHash 使用 uint 值
- key类型散列到一个uint值

- 运用uint 值可能提高性能
- 哈希值表示键没有排序，根据哈希值存储键
- 哈希函数必须设法避免碰撞，以达到良好的性能

用 QHash

- 键类型必须提供一个qHash 函数和操作符== 到 QHash

```
uint qHash(const Person &p)
{
    return p.age() + qHash(p.name());
}

bool operator==(const Person &first, const Person &second)
{
    return ((first.name() == second.name()) &&
            (first.age() == second.age()));
}
```

- 相对Qmap，填充和读取相同

一键多值

- QMultiMap和QMultiHash提供支持一键多值的关联数组

```
QMultiMap<QString,int> multiMap;

multiMap.insert("primes", 2);
multiMap.insert("primes", 3);
multiMap.insert("primes", 5);
    ...
multiMap.insert("fibonacci", 8);
multiMap.insert("fibonacci", 13);

foreach(const QString &key, multiMap.uniqueKeys())
{
    QList<int> values = multiMap.values(key);
    QStringList temp;
    foreach(int value, values)
        temp << QString::number(value);

    qDebug("%s: %s", qPrintable(key), qPrintable(temp.join(",")));
}
```

没有[], 用insert

QMap 和QHash 也支持这个使用insertMulti

keys 为每个值重复每个键, 使用uniqueKeys一次获取每个键

value 返回每个键的最后插入, values 返回键的所有值的列表

数据类型

Qt的类型定义

- C++中没有定义严格跨平台类型的大小

sizeof(int) = ?

ARM = 4 bytes
x86 = 4 bytes
IA64 = 8 bytes
...

取决于
CPU架构，
操作系统，
编译器等

- 对于跨平台的代码，以严格的方式定义的所有类型是很重要的

跨平台类型

Type	Size	Minimum value	Maximum value
uint8	1 byte	0	255
uint16	2 bytes	0	65 535
uint32	4 bytes	0	4 294 967 295
uint64	8 bytes	0	18 446 744 073 709 551 615
int8	1 byte	-128	127
int16	2 bytes	-32 768	32 767
int32	4 bytes	-2 147 483 648	2 147 483 647
int64	8 bytes	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
quintptr	“pointer sized”	n/a	n/a
qptrdiff	“pointer sized”	n/a	n/a
qreal	fast real values	n/a	n/a

所有类型都定义在头<QtGlobal>中

Qt的复杂类型

- Qt提供了多个复杂的类和类型

QFont

QList

QColor

QString

QRect

QPen

QBrush

QSize

QPoint

QImage

QPixmap

QByteArray

QVariant

- 有时候，希望能够通过一个普通的接口返回任何类型

```
const QVariant &data(int index);
void setData(const QVariant &data, int index);
```

数据可以是一个字符串，
图片，颜色，画刷
一个整数值，等

- QVariant类可以被视为一个联合
  - 创造一个Qt的类型联合是不可能，因为联合需要默认的构造函数
  - 变异类可以包含其他及自定义的复杂类型，例如QColor属于QtGui，QVariant属于QtCore

使用QVariant

- 基本类型用构造函数和 to*Type* 函数处理

```
QVariant v;
int i = 42;
qDebug() << "Before:" << i; // Before: 42
v = i;
i = v.toInt();
qDebug() << "After:" << i;  // After: 42
```

- 非QtCore类型和自定义类型，使用setValue 函数和模板 value<*type*> 函数来处理

```
QVariant v;
QColor c(Qt::red);
qDebug() << "Before:" << c; // Before: QColor(ARGB 1, 1, 0, 0)
v.setValue(c);
c = v.value<QColor>();      // After: QColor(ARGB 1, 1, 0, 0)
qDebug() << "After:" << c;
```

自定义的复杂类型

- 我们实现一个小类，包含一个人的名字和年龄

```
class Person
{
public:
    Person();
    Person(const Person &);
    Person(const QString &, int);

    const QString &name() const;
    int age() const;

    void setName(const QString &);
    void setAge(int);

    bool isValid() const;

private:
    QString m_name;
    int m_age;
};
```

不必是一个 QObject.

```
Person::Person() : m_age(-1) {}

...

void Person::setAge(int a)
{
    m_age = a;
}

bool Person::isValid() const
{
    return (m_age >= 0);
}
```

QVariant与Person对象

- 试图通过一个QVariant对象传递一个Person对象失败

```
qmetatype.h:200: error: 'qt_metatype_id' is not a member of 'QMetaTypeId<Person>'
```

- 在元类型系统中声明这一类型解决了问题

```
class Person
{
  ...
};

Q_DECLARE_METATYPE(Person)

#endif // PERSON_H
```

QVariant与Person对象

- 当类型注册成为一个元类型, Qt能把它存储在一个Qvariant中

```
QVariant var;
var.setValue(Person("Ole", 42));
Person p = var.value<Person>();
qDebug("%s, %d", qPrintable(p.name()), p.age());
```

- 要求声明类型
  - Public default constructor
  - Public copy constructor
  - Public destructor

然后，它中断了...

- 当与信号和槽工作，大部分连接是直接的
  - 直接连接时，类型可工作
  - 排队的连接，即非阻塞，异步的时候，这些类型不能工作（比如跨越线程边界）

```
connect(src, SIGNAL(), dest, SLOT(), Qt::QueuedConnection);

...

QObject::connect: Cannot queue arguments of type 'Person'
(Make sure 'Person' is registered using qRegisterMetaType().)
```

运行时的错误信息

注册类型

- 错误信息告诉我们如何解决问题
- qRegisterMetaType函数必须在连接建立之前被调用 (通常从main开始)

```
int main(int argc, char **argv)
{
    qRegisterMetaType<Person>();
    ...
```

文件和文件系统

- 在跨平台中的文件和目录带来许多问题
  - 系统是否有驱动器，或只是一个根？
  - 路径是否被“/” 或 “\”隔开?
  - 系统在哪里存储临时文件?
  - 用户在哪里存储文档?
  - 应用程序在哪里存储?

路径

- 用QDir 类去处理路径

```
QDir d = QDir("C:\\");
```

- 学会用静态函数去初始化

```
QDir d = QDir::root(); // C:/ on windows

QDir::current() // Current directory
QDir::home()    // Home directory
QDir::temp()    // Temporary directory

// Executable directory path
QDir(QApplication::applicationDirPath())
```

找目录内容

- entryInfoList返回该目录内容的信息列表

```
QFientryInfoList infos = QDir::root().entryInfoList();
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

以任意顺序列出文件和目录

- 你可以添加过滤器以跳过文件或目录

QDir::Dirs
QDir::Files
QDir::NoSymLinks

目录，文件或符号链接？

QDir::Readable
QDir::Writable
QDir::Executable

哪个文件?

QDir::Hidden
QDir::System

隐藏文件?
系统文件?

找目录内容

- 你也可以指定排序顺序

QDir::Name
QDir::Time
QDir::Size
QDir::Type

排序方法 ...

QDir::DirsFirst
QDir::DirsLast

目录在文件的前面还是后面

QDir::Reversed

倒序

- 从主目录根据名字排列所有目录

过滤器

顺序

```
QFileInfoList infos =
    QDir::root().entryInfoList(QDir::Dirs, QDir::Name);
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

找目录内容

- 最后，可以添加名字过滤器

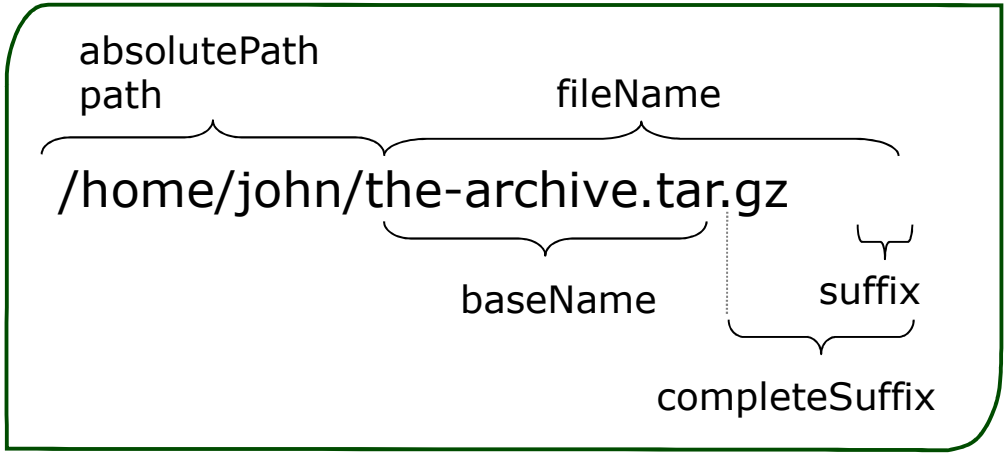
```
QFileInfoList infos =
    dir.entryInfoList(QStringList() << "*.cpp" << "*.h",
                      QDir::Files, QDir::Name);
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

所有cpp文件和头文件

QFileInfo

- 每个 QFileInfo 对象都有许多函数
  - absoluteFilePath – 某项的完整路径
  - isDir / isFile / isRoot –项的类型
  - isWriteable / isReadable / isExecutable – 文件的权限

当遍历时，有利于建立新的Qdir对象



打开和读取文件

- QFile 用来访问文件

```
QFile f("/home/john/input.txt");

if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");

QByteArray data = f.readAll();
processData(data);

f.close();
```

一次性读取所有文件

```
while(!f.atEnd())
{
    QByteArray data = f.read(160);
    processData(data);
}
```

每次读取160字节

写文件

- 写入文件时，用WriteOnly 模式打开文件，用写函数向文件添加数据

```
QFile f("/home/john/input.txt");

if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");


QByteArray data = createData();
f.write(data);

f.close();
```

- 文件也可用ReadWrite 模式打开
- Append或Truncate标志可以与写入模式相结合以追加数据到文件或截断它（即清除文件以前的内容）

```
if (!f.open(QIODevice::WriteOnly|QIODevice::Append))
```

QIODevice

- QFile由QIODevice派生
- QTextStream和QDataStream的构造函数以QIODevice指针为参数，而不是QFile指针
- QIODevice的实现
  - QBuffer – 读写到内存缓冲 区
  - QextSerialPort – 串行(RS232) 通讯 (第三方)
  - QAbstractSocket - TCP，SSL和UDP套接字类的基
  - QProcess –读写进程的标准输入和输出

流操作与文件

- 读取和写入函数在许多情况下显得尴尬 – 如处理复杂类型等
- 一个现代的方案是使用流操作
- Qt的提供两种流操作
  - 用于处理文本文件
  - 用于处理二进制文件格式

QTextStream

- QTextStream 类处理基于文本的文件读写
- 这个类能
  - 感知编解码（默认是使用区域设置，但也可以显式设置）
  - 感知行和字
  - 感知数字

写入文本流

- 使用操作符 << 和修饰符, 跟使用 STL 的流类似

```
QFile f(...);
if(!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QTextStream out(&f);
out << "Primes: " << qSetFieldWidth(3) << 2 << 3 << 5 << 7 << endl;
```

数字可以直接放到流中

修饰符,设置字段最小宽度

向流添加一个换行符

Results in:

```
Primes:   2  3  5  7
```

使用文本流读取

- 可以逐行读文件

```
QTextStream in(&f);
while(!f.atEnd())
    qDebug("line: '%s'", qPrintable(in.readLine()));
```

- 可以提取单词和数字

```
QTextStream in(&f);
QString s;
int i;
in >> s >> i;
```

- 用 atEnd 去判断是否到达文件末尾

处理二进制文件

- QDataStream类用于字节流
  - 保证字节顺序
  - 支持基本类型
  - 支持Qt的复杂类型
  - 支持添加自定义的复杂类型

```
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QDataStream ds(&f);
ds << QString("Unicode string data");
```

简单地传递一个指向QFile对象的指针流，到流构造函数去为指定的文件设置一个流

数据流作为一种文件格式

- 当基于QDataStream的文件格式时有一些细节需要注意
  - 版本控制 – 随着Qt的结构演变，它们的二进制序列化格式也随之改变。使用QDataStream::setVersion，可以显式强制使用特定的序列化格式。
  - 类型信息 - Qt不添加类型信息，所以需要记录下你所存储的类型是以什么样的顺序存储。

数据流作为一种文件格式

```
QFile f("file.fmt");
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QDataStream out(&f);
out.setVersion(QDataStream::Qt_4_6);

quint32 value = ...;
QString text = ...;
QColor color = ...;

out << value;
out << text;
out << color;
```

这里支持低到Qt1.0的版本。

进行流读写时要确保类型和顺序匹配

```
QFile f("file.fmt");
if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");

QDataStream in(&f);
in.setVersion(QDataStream::Qt_4_6);

quint32 value = ...;
QString text = ...;
QColor color = ...;

in >> value;
in >> text;
in >> color;
```

流化自定义类型

- 通过实现流操作符<<和>>，自定义类型可以跟数据流互相转换

```
QDataStream &operator<<(QDataStream &out, const Person &person)
{
    out << person.name();
    out << person.age();
    return out;
}

QDataStream &operator>>(QDataStream &in, Person &person)
{
    QString name;
    int age;
    in >> name;
    in >> age;
    person = Person(name, age);
    return in;
}
```

流化自定义类型

- 为了流化包含在QVariant对象中的自定义类型，流操作符必须注册

```
qRegisterMetaTypeStreamOperators<Person>("Person");
```

- 当变量被流化后，它增加了数据类型的名称，以确保它之后可以从流中恢复

```
00: 0x00 0x00 0x00 0x7f 0x00 0x00 0x00 0x00   ____________
08: 0x07 0x50 0x65 0x72 0x73 0x6f 0x6e 0x00   _Person_
16: 0x00 0x00 0x00 0x06 0x00 0x4f 0x00 0x6c   ______O_l
24: 0x00 0x65 0x00 0x00 0x00 0x2a             e____*
```

使用QVariant而不是你自己进行跟踪的话会增加文件大小和复杂度。

自定义类型的检查清单

- 实现
  - Type::Type() – 公有的缺省构造函数
  - Type::Type(const Type &other) – 公有的拷贝构造函数
  - Type::~Type() – 公有的析构函数
  - QDebug operator<< – 方便的调试
  - QDataStream operator<< 和>> – 流化
- 注册
  - Q_DECLARE_METATYPE – 在头文件中
  - qRegisterMetaType – 在主函数main中
  - qRegisterMetaTypeStreamOperators – 在主函数main中