Techniques

This page describes some commonly-used design patterns for dealing with Protocol Buffers. You can also send design and usage questions to the Protocol Buffers discussion group.

Streaming Multiple Messages

If you want to write multiple messages to a single file or stream, it is up to you to keep track of where one message ends and the next begins. The Protocol Buffer wire format is not self-delimiting, so protocol buffer parsers cannot determine where a message ends on their own. The easiest way to solve this problem is to write the size of each message before you write the message itself. When you read the messages back in, you read the size, then read the bytes into a separate buffer, then parse from that buffer. (If you want to avoid copying bytes to a separate buffer, check out the `CodedInputStream` class (in both C++ and Java) which can be told to limit reads to a certain number of bytes.)

Large Data Sets

Protocol Buffers are not designed to handle large messages. As a general rule of thumb, if you are dealing in messages larger than a megabyte each, it may be time to consider an alternate strategy.

That said, Protocol Buffers are great for handling individual messages *within* a large data set. Usually, large data sets are really just a collection of small pieces, where each small piece may be a structured piece of data. Even though Protocol Buffers cannot handle the entire set at once, using Protocol Buffers to encode each piece greatly simplifies your problem: now all you need is to handle a set of byte strings rather than a set of structures.

Protocol Buffers do not include any built-in support for large data sets because different situations call for different solutions. Sometimes a simple list of records will do while other times you may want something more like a database. Each solution should be developed as a separate library, so that only those who need it need to pay the costs.

Union Types

You may sometimes want to send a message that could be one of several different types. However, protocol buffer parsers cannot necessarily determine the type of a message based on the contents alone. So how do you make sure that the recipient application knows how to decode your message? One solution is to create a wrapper message that has one optional field for each possible message type.

For example, if you have message types `Foo`, `Bar`, and `Baz`, you can combine them with a type like:

```
message OneMessage {
  // One of the following will be filled in.
  optional Foo foo = 1;
  optional Bar bar = 2;
  optional Baz baz = 3;
}
```

You may also want to have an enum field that identifies which message is filled in, so that you can `switch` on it:

```
message OneMessage {
  enum Type { FOO = 1; BAR = 2; BAZ = 3; }

  // Identifies which field is filled in.
  required Type type = 1;

  // One of the following will be filled in.
  optional Foo foo = 2;
  optional Bar bar = 3;
  optional Baz baz = 4;
}
```

If you have a very large number of possible types, listing every one of them in your container type may be unwieldy. Instead, you should consider using extensions:

```
message OneMessage {
  extensions 100 to max;
}

// Elsewhere...
extend OneMessage {
  optional Foo foo_ext = 100;
  optional Bar bar_ext = 101;
  optional Baz baz_ext = 102;
}
```

Note that you can use the `ListFields` reflection method (in C++, Java, and Python) to get a list of all fields present in the message, including extensions. You might use this as part of a scheme for registering handlers for diverse message types.

Self-describing Messages

Protocol Buffers do not contain descriptions of their own types. Thus, given only a raw message without the corresponding `.proto` file defining its type, it is difficult to extract any useful data.

However, note that the contents of a .proto file can itself be represented using protocol buffers. The file `src/google/protobuf/descriptor.proto` in the source code package defines the message types involved. `protoc` can output a `FileDescriptorSet` – which represents a set of .proto files – using the `--descriptor_set_out` option. With this, you could define a self-describing protocol message like so:

```
message SelfDescribingMessage {
  // Set of .proto files which define the type.
  required FileDescriptorSet proto_files = 1;
```

```
  // Name of the message type.  Must be defined by one of the files in
  // proto_files.
  required string type_name = 2;

  // The message data.
  required bytes message_data = 3;
}
```

By using classes like `DynamicMessage` (available in C++ and Java), you can then write tools which can manipulate `SelfDescribingMessage`s.

All that said, the reason that this functionality is not included in the Protocol Buffer library is because we have never had a use for it inside Google.

Last updated 四月 *2, 2012.*